Easy Instances for Model Checking

Markus Frick

Dissertation
zur Erlangung des Doktorgrades
der Mathematischen Fakultät
der Albert Ludwigs Universität
Freiburg im Breisgau

16. August 2001

Dekan der mathematischen Fakultät:
Prof. Dr. Wolfgang Soergel

Erster Referent:
Prof. Dr. Martin Grohe

Zweiter Referent:
Prof. Dr. Heinz-Dieter Ebbinghaus

Datum der Promotion: 20. Juni 2001

Zusammenfassung

Das zentrale Thema der vorliegenden Dissertation ist das sogenannte Model-Checking Problem. In seiner einfachsten Form haben wir eine endliche relationale Struktur $\mathfrak{A}$ und einen logischen Satz φ gegeben und fragen ob φ in $\mathfrak{A}$ wahr ist. Fragen dieser Art treten in den verschiedensten Bereichen der Informatik auf, unter anderem in der Datenbanktheorie und der künstlichen Intelligenz.

Neben diesem Entscheidungsproblem untersuchen wir bestimmte Variationen des Model-Checking Problems. Für eine Formel mit freien Variablen stellen sich die Fragen des Aufzählens aller erfüllenden Belegungen, des Findens einer erfüllenden Belegung, sowie die Anzahl aller erfüllenden Belegungen zu berechnen.

Unser Hauptinteresse gilt der Komplexität dieser Probleme, wobei wir auf Algorithmen abzielen, deren Laufzeit beliebig von der Formel, jedoch nur linear von der Größe der Eingabestruktur abhängen. Offensichtlich hängt die Komplexität der Model-Checking Probleme von der Ausdruckskraft der Sprache sowie der zugelassenen Strukturen ab. So ist z.B. Model-Checking für die monadische Logik der zweiten Stufe (MSO) PSPACE-vollständig. Andererseits hat Courcelle einen Algorithmus gefunden, der die Modellbeziehung in Zeit $O(f(||\varphi||) \cdot ||\mathfrak{A}||)$ für Strukturen mit Baumbreite $\leq$ eine fest vorgegebene Zahl w berechnet.

Die Baumbreite ist ein Maß für die Ähnlichkeit mit einem Baum. In der Algorithmentheorie ist diese numerische Invariante die vielleicht erfolgreichste Eigenschaft bei der Suche nach einfachen Instanzen für normalerweise harte Probleme.

In der vorliegenden Arbeit untersuchen wir im wesentlichen zwei Problemarten. In Kapitel 2 verallgemeinern wir Courcelle's Theorem und beschreiben Algorithmen, die für Formeln der monadischen zweiten Stufe und Strukturen beschränkter Baumbreite alle erfüllenden Belegungen, bzw. eine einzelne erfüllende Belegung finden. Beide Algorithmen arbeiten in Zeit linear in der Größe der Eingabe plus der Größe der Ausgabe (das ist für den Aufzählungfall wichtig). Da das Zählproblem bereits von Arnborg, Lagergren und Seese im Jahre 1991 gelöst wurde, kann damit MSO-Model-Checking auf Strukturen beschänkter Baumbreite als vollständing bearbeitet betrachtet werden.

Die vorgestellten Algorithmen basieren auf einem automatentheoretischen Ansatz und stellen im wesentlichen nur eine Verfeinerung (und Optimierung) bereits bekannter Methoden dar.

Im zweiten Teil der Arbeit (Kapitel 3 und 4), untersuchen wir das Problem der ersten Stufe Logik auf Strukturen, die lokal beschränke Baumbreite besitzen, oder etwas formaler: für eine Funktion $f : \mathbb{N} \to \mathbb{N}$ lassen wir Strukturen $\mathfrak{A}$ zu, deren r-Nachbarschaften höchstens die Baumbreite $f(r)$ haben.

Unter diesen Begriff fallen so wichtige Klassen wie planare Graphen, Strukturen beschränkter Valenz und Graphen mit beschränkter Kreuzungszahl.

Der Ansatz zur Lösung von Model-Checking Problemen dieser Art basiert dabei auf der Beobachtung daß alle erste Stufe Eigenschaften "lokal" sind. Durch die Ergebnisse aus dem 2. Kapitel können nun lokale Lösungsmengen in optimaler Zeit berechnet werden (denn Nachbarschaften haben nach Voraussetzung beschränkte Baumbreite). Danach müssen zur Lösung des jeweiligen Model-Checking Problems "nur" noch die Teilergebnisse zusammengesetzt werden. Auf diese Weise lösen wir die vier angesprochenen Model-Checking Probleme für die Logik der ersten Stufe auf lokal baumartigen Strukturen (nicely locally tree-decomposable classes). Diese Algorithmen arbeiten alle in Zeit linear in der Größe der Eingabestruktur (plus der Größe der Ausgabe beim Aufzählungsproblem). Damit ist auch dieser Problembereich vollständig gelöst.

Contents

Chapter 1

Introduction

1.1 Background: The model-checking problem

It is an important task in complexity theory to find feasible instances for otherwise intractable problems. Apart from delivering algorithms for practical use, feasibility results may shed light on the structural properties of the problem instances. The main subject of this thesis is the so called model-checking problem and to exhibit instances for which this generally intractable problem is easy to solve. In its basic version the problem looks as follows: we have given a structure $\mathfrak{A}$ and a sentence φ and we ask whether $\mathfrak{A}$ satisfies φ. If $\varphi \in \mathcal{L}$ for some logic $\mathcal{L}$, we call this the *$\mathcal{L}$-model-checking problem.*

This problem and its variations appear in various areas of computer science, most notably in database theory [CH82] and in the realm of constraint satisfaction [FV93]. An important feature is its meta-character with respect to algorithms and complexity theory, i.e. standard problems from algorithm theory can be expressed as certain instances of the model-checking problem. For instance, to decide if a graph $\mathcal{G}$ contains a k-clique, we may alternatively decide whether the first-order (FO) sentence φ_k, saying that there are k pairwise adjacent vertices, holds in $\mathcal{G}$. In a similar way colorability can be reduced to monadic second-order (MSO) model-checking.

Essentially there are three different generalizations of the basic model-checking problem. Assume that we are given a formula φ with free variables and a structure $\mathfrak{A}$. The *evaluation problem* asks for all assignments of the free variables such that φ with these assignments holds in $\mathfrak{A}$. The case where we just want to find a single satisfying assignment is called the *witness problem*, and finally, the *counting problem* refers to the task of computing the number of satisfying assignments.

Our interest is focused on the *complexity* of the model-checking problem and its generalizations. This question is intimately related to the expressibility of the logical language in question. First attempts of measuring the complexity of model-checking have been done with respect to the size $\|\varphi\| + \|\mathfrak{A}\|$ of the input. Unfortunately, this complexity turned out to be very high (and certainly depends on the chosen language); already the very basic conjunctive queries are NP-complete [CM77]. Full first-order logic and monadic second-order logic have been shown to be PSPACE-complete [Var82].

To prove PSPACE-hardness for FO-model-checking, we reduce the problem to QBF (quantified Boolean formula, see [BDG95]), which essentially is model-checking over a two element structure with an unary relation. One conceivable interpretation of this fact is that the query carries the bulk of complexity. Since at the same time, the queries are assumed to be small compared to the size of the structure, Vardi [Var82] introduced the notion of *Data complexity*. Herein, the complexity is expressed in terms of the size of the structure, whereas the dependency on the query is completely neglected. In this framework, we get PTIME algorithms for FO-model-checking and its generalizations, e.g. for an FO-sentence with k variables we can decide $\mathfrak{A} \models \varphi$ in time $O(\|\mathfrak{A}\| + |A|^k)$ [Var95].

Although this is considered to be "efficient" (well, it's in PTIME), it is not satisfying. Already for a query involving 5 variables and a structure of moderate size 1000, the algorithm needs at least 10^{15} steps to decide the model relation, what is far from feasible.

Parameterized complexity is a novel approach presented by Downey and Fellows [DF99]. They not only introduced new ways to measure complexity, but developed an entire complexity theory including reductions and notions of completeness. In this framework, one or more parts of the input are considered to be the parameter of the problem instance. Under these conditions the problem's complexity is measured by separate functions on the parameter and the rest of the instance, respectively.

In our context, we consider the formula φ as the parameter and say that model-checking is fixed parameter tractable (in FPT), if there is a function $f : \mathbb{N} \to \mathbb{N}$, a constant $c \geq 1$ and an algorithm that solves the problem in time $f(\|\varphi\|) \cdot \|\mathfrak{A}\|^c$. It was proved that conjunctive (first-order) query evaluation is W[1]-complete [PY97] (AW[1]-complete, respectively [DFT96]). These two classes correspond to the traditional classes NP and PSPACE and both results entail that, unless FPT = W[1], the corresponding problems are not in FPT.[1]

We employ two strategies to bypass these hardness results. The first one is exhibiting numerical invariants of the input instances and examining the complexity with respect to these new parameters. The maybe most successful parameter is the so called *tree-width* of a structure. Intuitively it measures the tree-likeness of a structure, and it has been applied to a huge variety of algorithmic problems (cf. [Bod97] for a survey). For instance, a tree has tree-width 1, and an n-clique has tree-width $n-1$. Of particular interest for model-checking is Courcelle's well known theorem:

> Let $w \geq 1$ and φ be a monadic second-order sentence. Then there is an algorithm that, given a structure of tree-width at most w, decides in linear time whether φ holds in $\mathfrak{A}$.

In our framework, the tree-width $w := \mathrm{tw}(\mathfrak{A})$ of $\mathfrak{A}$ and the sentence φ are considered as parameters (actually, w is a numerical invariant of the input structure). Now this theorem looks as follows:

> There is a function $f : \mathbb{N}^2 \to \mathbb{N}$ and an algorithm that, given a sentence $\varphi \in \mathrm{MSO}$ and a structure $\mathfrak{A}$, decides whether φ holds in $\mathfrak{A}$ in time
> $$f(||\varphi||, \mathrm{tw}(\mathfrak{A})) \cdot ||\mathfrak{A}||.$$

The second, less general approach we follow in this thesis, is to apriori restrict the class of admitted structures. For a long time it has been known that properties expressible in first-order logic are local [Gai82]. Hence, motivated by Courcelle's theorem, it is conceivable that local tree-likeness of structures can be exploited to get efficient algorithms. In this thesis, we introduce the notion of *locally tree-decomposable* classes $\mathcal{C}$ and show the following:

> Let $\mathcal{C}$ be a locally tree-decomposable class of structures. Then there is an $f : \mathbb{N} \to \mathbb{N}$ and an algorithm that, given an FO-sentence φ and a structure $\mathfrak{A} \in \mathcal{C}$, decides whether φ holds in $\mathfrak{A}$ in time
> $$f(||\varphi||) \cdot ||\mathfrak{A}||.$$

Many important classes are locally tree-decomposable, for example, the class of graphs of bounded genus and graphs with bounded valence. Note

[1] Like NP $\neq$ P, it is usually assumed that FPT $\neq$ W[1].

here that Courcelle's theorem is a uniform result, in the sense that there is one algorithm that works for all input structures. In contrast to that, this second approach gives for each class $\mathcal{C}$ an algorithm solving the model-checking problem for structures from $\mathcal{C}$.

All results presented in this thesis are essentially of the two types above. For the decision, witness and counting problems, we aim at algorithms that work within time linear in the size of the input structure. For the evaluation cases we admit time linear in the size of the input structure *plus* the size of the output. In particular, we will settle the MSO-evaluation problem and the MSO-witness problem in linear time parameterized by the tree-width of the input structure. Furthermore, we are able to extend the just mentioned optimal model-checking result for locally tree-decomposable classes to the evaluation, witness and counting cases.

Before we give a survey on the thesis and an exact formulation of the proved results, we have to fix notation and the computation model.

1.2 Preliminaries

For $n \geq 1$, we set $[n] := \{1, \ldots, n\}$. By $\mathrm{Pow}(A)$ we denote the power set of A and $\mathrm{Pow}^{\leq l}(A)$ is the set of elements of $\mathrm{Pow}(A)$ that have cardinality $\leq l$.

A *vocabulary* is a finite set of relation symbols. Each relation symbol is associated with a natural number, its *arity*. Vocabularies are always denoted by the letter τ.

A τ*-structure* $\mathfrak{A}$ consists of a non-empty set A, called the *universe* of $\mathfrak{A}$, and a relation $R^{\mathfrak{A}} \subseteq A^r$ for each r-ary $R \in \tau$. Let $\mathfrak{A}$ be a structure and $B \subseteq A$ non-empty. Then $\langle B \rangle^{\mathfrak{A}}$ denotes the *substructure* of $\mathfrak{A}$ induced by B. If the context is clear we omit the superscript.

We only consider finite structures. A *graph* is an $\{E\}$-structure $(G, E^{\mathcal{G}})$ where $E^{\mathcal{G}}$ is an anti-reflexive and symmetric binary relation (in other words: we consider simple undirected graphs without loops). The elements of the universe of a graph are called *vertices* and sometimes we refer to elements of E^G by the term *edges*. A *colored graph* is a structure $\mathfrak{B} = (B, E^{\mathfrak{B}}, P_1^{\mathfrak{B}}, \ldots, P_m^{\mathfrak{B}})$, where $(B, E^{\mathfrak{B}})$ is a graph and the unary relations $P_1^{\mathfrak{B}}, \ldots, P_m^{\mathfrak{B}}$ form a partition of the universe B.

For a vocabulary τ, the set of *first-order* formulas is built up from an infinite supply of variables $x, y, x_1, x_2, \ldots$, the relation symbols $R \in \tau$ and $=$, the connectives $\vee, \wedge, \neg, \rightarrow$ and the quantifiers $\forall x, \exists x$ ranging over elements of the universe of the structure. This logic is denoted by FO.

We extend this calculus by unary *second-order* variables $X, Y, X_1, X_2, \ldots$ which range over sets of elements of the universe. For a vocabulary τ, the *monadic second-order logic* (MSO) is the closure of FO under the adjunction of the new atoms built from new unary relation variables, and unary second-order quantification $\exists X, \forall X$.

A *free variable* in a formula φ is a variable x (or X) occurring in φ that is not in the scope of a quantifier $\exists x, \forall x$ (or $\exists X, \forall X$, respectively). We write $\varphi(X_1, \ldots, X_l, x_1, \ldots, x_m)$ to indicate that the free variables in φ are exactly $X_1, \ldots, X_l, x_1, \ldots, x_m$. A *sentence* is a formula that contains no free variables. The semantics of FO, MSO should be clear. For a τ-structure $\mathfrak{A}$, $A_1, \ldots, A_l \subseteq A, a_1, \ldots, a_m \in A$ and a τ-formula $\varphi(X_1, \ldots, X_l, x_1, \ldots, x_m)$ we write $\mathfrak{A} \models \varphi(A_1, \ldots, A_l, a_1, \ldots, a_m)$ to say that $\mathfrak{A}$ satisfies φ if the variables $X_1, \ldots, X_l$ are interpreted by the subsets $A_1, \ldots, A_l$ and $x_1, \ldots, x_m$ are interpreted by the elements $a_1, \ldots, a_m$, respectively. The rank $\mathrm{rk}(\varphi)$ of a formula φ is the maximal number of nested quantifiers in φ.

For a structure $\mathfrak{A}$ and a formula $\varphi(X_1, \ldots, X_l, x_1, \ldots, x_m)$ we let

$$\varphi(\mathfrak{A}) := \{(A_1, \ldots, A_l, a_1, \ldots, a_m) \mid \mathfrak{A} \models \varphi(A_1, \ldots, A_l, a_1, \ldots, a_m)\}.$$

For a sentence φ we set $\varphi(\mathfrak{A}) = \text{TRUE}$ if $\mathfrak{A}$ satisfies φ, and $= \text{FALSE}$, otherwise[2]. If the vocabularies of $\mathfrak{A}$ and φ do not agree, we set $\varphi(\mathfrak{A}) = \text{FALSE}$ (or $= \emptyset$ for φ with free variables). Here the convention that all variables of φ enclosed within the brackets actually occur freely in φ is crucial for this definition to be well defined.

Let $\mathcal{L}$ be a logic and $\mathcal{C}$ a class of structures ($\mathcal{C}$ may be the class of all structures). The *$\mathcal{L}$-model-checking* problems over $\mathcal{C}$ are the following:

- the *decision problem*

Input:	Structure $\mathfrak{A} \in \mathcal{C}$, sentence $\varphi \in \mathcal{L}$.
Problem:	Decide, if $\varphi(\mathfrak{A}) = \text{TRUE}$.

- the *evaluation problem*

[2] To be more consistent, we could define $\text{TRUE} = \{\emptyset\}$ and $\text{FALSE} = \{\}$.

Input:	Structure $\mathfrak{A} \in \mathcal{C}$, formula $\varphi \in \mathcal{L}$.
Problem:	Compute $\varphi(\mathfrak{A})$.

- the *witness problem*

Input:	Structure $\mathfrak{A} \in \mathcal{C}$, formula $\varphi \in \mathcal{L}$.
Problem:	Compute an element of $\varphi(\mathfrak{A})$.

- the *counting problem*

Input:	Structure $\mathfrak{A} \in \mathcal{C}$, formula $\varphi \in \mathcal{L}$.
Problem:	Compute $|\varphi(\mathfrak{A})|$.

Algorithms: Since we are looking for linear time algorithms for the above query evaluation problems, the questions of how the input is coded and what model of computation we employ, become a central issue. Our underlying model of computation is the standard RAM-model with addition and subtraction as arithmetic operations [AHU74]. We assume the *logarithmic cost measure*. For an exact description of the model, and why we have chosen it, we refer the reader to the appendix (section A). There we also give a description of the pseudo-code we use in the algorithms. Nevertheless, it should be possible to understand the programs without reading the appendix.

In our main results, we mostly omit the O-notation. This improves readability of the main results (and often we can think of the constants as part of some unspecified function bounding the running time: e.g. f in $f(\|\varphi\|)\cdot\|\mathfrak{A}\|$). But it should be clear that all results involve constants depending on the used hardware and data-structures.

A relational structure $\mathfrak{A}$ is coded by a word w consisting of the vocabulary, the elements of the universe A, and the relations $R^{\mathfrak{A}}$ for $R \in \tau$. Observe that we do not encode relations by the corresponding Boolean matrices, but simply by a list of its tuples. This is considered the standard coding for algorithms. It becomes particular important due to the fact that we develop algorithms running in time linear in the size of the input structure. In that case coding relations by their Boolean matrices would waste space and grant too much time, if the relations contained few tuples. Again, we refer the reader to the appendix (section A.2).

Formulas are coded in some reasonable way, e.g. by a string or its syntactic tree.

We will carefully distinguish between the *size* $\|o\|$ of an object o (which is the length of the used encoding), and, if o is a set, its *cardinality* $|o|$. For instance, if R is an r-ary relation, then $\|R^{\mathfrak{A}}\| = O(r \cdot |R^{\mathfrak{A}}|+1)$. The difference between $|\cdot|$ and $\|\cdot\|$ becomes crucial when we consider sets of the form $\varphi(\mathfrak{A})$ where φ has free second-order variables.

As an example let us present an easy observation concerning the decision problem for first-order logic over the class of all structures.

Example 1 ([Var95]). Let $\varphi(\bar{x}) \in \mathrm{FO}$ with k variables. Then $\varphi(\mathfrak{A})$ can be computed in time

$$O(\|\varphi\|(\|\mathfrak{A}\| + |A|^k))$$

Proof: First, check if $\varphi(\bar{x})$ and $\mathfrak{A}$ are of the same vocabulary and, if not, return $\emptyset$.

Let all variables in φ be among $x_1, \ldots, x_k$. As a matter of fact, there are $O(\|\varphi\|)$ many subformulas $\psi_1(\bar{x}), \ldots, \psi_m(\bar{x})$ of $\varphi(\bar{x})$. We compute $\psi_i(\mathfrak{A})$ in a bottom-up manner. We start with the atoms, i.e. if $\psi_i(\bar{x})$ is of the form $R\bar{x}$ for some $R \in \tau$ then $\psi_i(\mathfrak{A})$ can easily be computed by a loop over the elements of $R^{\mathfrak{A}}$. If ψ_i is a conjunction or negation, then simply take the join or complement of the corresponding intermediate result(s).

Let e.g. $(x_1, \ldots, x_l)$ be the tuple of variables free in ψ_j and $\psi_i(\bar{x}) = \exists x_1 \psi_j(\bar{x})$. Now compute the projection of $\psi_j(\mathfrak{A})$ without the coordinate corresponding to x_1. The result is $\psi_i(\mathfrak{A})$.

Since for all i $|\psi_i(\mathfrak{A})| \leq |A|^k$ it is easy to see that the algorithm, suitably implemented, works within the claimed time bound. $\square$

1.3 The road map

So let us give a survey on the thesis. As mentioned before, our main topic is the model-checking problem and its generalizations. We start with an overview of the most relevant results and give some hints about how these results have been achieved.

In the second chapter, we examine the MSO-model-checking problems over structures of bounded tree-width. The standard way to handle this kind of problems, is to reduce the problem stated for arbitrary structures to a problem on binary trees. This linear time reduction was first employed in [ALS91]. The resulting problem instance is well suited to apply, the algorithmically well understood, automata theoretic methods.

We proceed in a two-fold way. First, we re-prove Courcelle's theorem, which coincides with the decision problem for MSO-logic parameterized by the tree-width of the input structures. This is done without the abovementioned reduction using merely logical methods. Using dynamic programming this approach is extended to solve the evaluation problem for formulas with a single free first-order variable.

After that, we turn to the standard approach. We develop algorithms for the evaluation and witness problem, i.e. algorithms that, given a structure $\mathfrak{A}$ and an MSO-formula $\varphi(\bar{X}, \bar{y})$, compute $\varphi(\mathfrak{A})$ in time $f(\|\varphi\|, w) \cdot (\|\varphi(\mathfrak{A})\| + \|\mathfrak{A}\|)$, for some function $f : \mathbb{N}^2 \to \mathbb{N}$ and w the tree-width of $\mathfrak{A}$ (and in time $f(\|\varphi\|, w) \cdot \|\mathfrak{A}\|$ for the witness problem).

The proofs do not contain essentially new ideas. Some ideas concerning automata can already be found in [CM93] (although no effort was made in optimization). So the main technical problem remained to achieve the tight time bounds.

Observe that the counting problem for MSO over structures of bounded tree-width has already been settled by Arnborg, Lagergren and Seese [ALS91].

The third chapter is dedicated to the problem of deciding first-order properties over *locally tree-decomposable* classes of structures. Essentially, a class $\mathcal{C}$ of structures is locally tree-decomposable, if for each $\mathfrak{A} \in \mathcal{C}$ the universe A can be covered suitably by a family $\mathcal{T}$ of subsets of A, such that for each $T \in \mathcal{T}$, $\langle T \rangle^{\mathfrak{A}}$ has tree-width bounded by a value independent from the size of $\mathfrak{A}$. After giving the definition, we prove some lemmas about the structure of such classes.

Several natural classes of structures happen to be locally tree-decomposable. Most notably structures of bounded degree, structures of bounded tree-width, planar graphs, and graphs embeddable into some surface.

Then we prove the main theorem saying that over classes $\mathcal{C}$ of locally tree-decomposable classes, the FO-decision problem is solvable in linear time. More precisely, there is a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm that, given a structure $\mathfrak{A} \in \mathcal{C}$ and an FO-sentence φ, decides if φ holds in $\mathfrak{A}$ in time $f(\|\varphi\|) \cdot \|\mathfrak{A}\|$. Roughly, we use the normal form theorem from Gaifman [Gai82] that allows to express first-order definable queries by "local" queries. These local queries can be evaluated in the substructures induced by the cover sets (the T's in the above definition). In a last step, we re-assemble the local results and decide whether the initial first-order sentence holds in $\mathfrak{A}$.

A natural continuation of the third chapter is to extend the decision result to the cases with free variables. This is what the forth chapter intends. But unfortunately, to solve these problems in the affirmative, the class characterization developed in the preceding section does not suffice. We will call classes satisfying a more restricted definition *nicely locally tree-decomposable.* A pleasant property of this new, stricter definition is that it still comprises all examples mentioned above.

For such classes of structures we develop algorithms that solve the evaluation and witness problem. Given an input structure $\mathfrak{A}$ and a formula $\varphi(\bar{x})$, the running time is bounded by $f(||\varphi||) \cdot (|\varphi(\mathfrak{A})| + ||\mathfrak{A}||)$ for the evaluation problem and by $f(||\varphi||) \cdot ||\mathfrak{A}||$ for the witness problem.

The details are too involved to be presented here, but essentially we again exploit locality and use the fact that local computations can be done in optimal time (by the results of the second chapter for structures of bounded tree-width).

Finally, we attack the FO-counting problem for nicely locally tree-decomposable classes. At the end of the forth chapter, this counting problem is reduced to the calculation of a sum over labels attached to "independent" sets of a graph of bounded degree (hence we have a polynomial number of addends). Nevertheless, we get an algorithm that solves the FO-counting problem over such classes in time $f(||\varphi||) \cdot ||\mathfrak{A}||$ for some suitable function $f : \mathbb{N} \to \mathbb{N}$ and input $\mathfrak{A}, \varphi$.

In the appendix we will present the machine model and some basic algorithms that are used in the thesis.

Thanks to my supervisor Martin Grohe for guidance and advice. I would also like to thank all people giving useful comments on drafts of this thesis (they know who they are). Particular thanks to Bernhard Herwig who in the last minute pointed out an easy proof for theorem 67.

Chapter 2

Evaluation on Tree-like Structures

Intuitively, the tree-width of a graph measures its similarity with a tree. In the known form, this notion was introduced by Robertson and Seymour [RS86], and the straightforward generalization to arbitrary structures is due to Feder and Vardi [FV93].

We start with the introduction of the notion of a *tree-decomposition* for arbitrary structures. To explain the handling of such decompositions, we develop the adequate data structures and procedures that are essential for linear time algorithms. As a first application, we re-prove Courcelle's theorem [Cou90] by purely logical techniques. In a further step, we extend this result by a Dynamic programming approach to a linear time algorithm that associates to each element of the input structure its logical type.

In the subsequent section we state and prove the main result concerning structures of bounded tree-width. It claims that there is an algorithm that evaluates MSO-queries in time linear in the size of the input plus the size of the output. This is done by an automata theoretic approach, essentially based on ideas from Arnborg, Lagergren and Seese [ALS91]. For that, the question about tree-decompositions is reduced to colored binary trees, on which (by [TW68]) MSO and deterministic tree automata coincide. Doing a suitable prescan, we will be able to avoid unnecessary computation, and thus obtain the tight time bound. In a paper of Courcelle and Mosbah about evaluating MSO on tree-like graphs [CM93] it was already noted that the usual automata theoretic methods can be improved by sorting out unreachable states. But neither the exact idea was made explicit nor an implementation was given to obtain optimal time bounds. In fact, these details will be the technically most involved part of the section.

2.1 Tree-decompositions and linear time

We start with the definition of a *tree-decomposition* of a relational structure. Recall that a *tree* is a connected acyclic graph. Let $\mathfrak{A}$ be a τ-structure for some relational vocabulary τ.

Definition 2. *A* tree-decomposition *of $\mathfrak{A}$ is a pair $(\mathcal{T}, (B_t)_{t\in T})$, where $\mathcal{T}$ is a tree and $(B_t)_{t\in T}$ is a family of subsets of A such that*

(1) For every $a \in A$, the set $\{t \in T \mid a \in B_t\}$ is non-empty and connected in $\mathcal{T}$.

(2) For every $R \in \tau$ and every $\bar{a} \in R^{\mathfrak{A}}$ there is a $t \in T$ such that $\bar{a} \in B_t$.

The sets B_t are called the *blocks* of the tree-decomposition, and the *width* of $(\mathcal{T}, (B_t)_{t\in T})$ is $\max\{|B_t| \mid t \in T\} - 1$. The *tree-width* $\mathrm{tw}(\mathfrak{A})$ of $\mathfrak{A}$ is the minimal width of a tree-decomposition of $\mathfrak{A}$. From property (1) we immediately derive the following important property of tree-decompositions: Let $t, t' \in T$ with common ancestor s. Then $B_t \cap B_{t'} \subseteq B_s$.

Sometimes we use the easy fact that the class of structures of tree-width at most w is *sparse*:

Lemma 3 ([Bod98]). *Let $w \geq 1$ and τ a vocabulary. Then there is a constant k (depending on τ and w) such that for every τ-structure $\mathfrak{A}$ of tree-width at most w we have $\|\mathfrak{A}\| \leq k \cdot |A|$.*

The main application of this fact is that if $\mathcal{C}$ is a class of structures of tree-width $\leq w$, then each relation of a structure $\mathfrak{A} \in \mathcal{C}$ contains at most $f(w, \tau) \cdot |A|$ many tuples for some suitable $f : \mathbb{N}^2 \to \mathbb{N}$. Hence, we can pass through the relation in time linear in the cardinality of the universe.

Definition 4 (Gaifman graph). *Let $\mathfrak{A}$ be a τ-structure. Then the Gaifman graph of $\mathfrak{A}$ is the graph $\mathcal{G}(\mathfrak{A}) = (G, E^{\mathcal{G}(\mathfrak{A})})$, with universe $G := A$ and an edge between $a, b \in A$, $a \neq b$, if there is an $R \in \tau$ and a tuple $(a_1, \ldots, a_k) \in R^{\mathfrak{A}}$ such that $a, b \in \{a_1, \ldots, a_k\}$.*

It is easy to see that, given a structure $\mathfrak{A}$, its Gaifman graph $\mathcal{G}(\mathfrak{A})$ can be calculated in time $O(\|\mathfrak{A}\|)$: For each k-ary relation $R \in \tau$ perform a loop over all $(a_1, \ldots, a_k) \in R^{\mathfrak{A}}$ doing the following: for each $i = 1, \ldots, k$ add a_i to the adjacency list of $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k$, respectively. Afterwards, we remove multiple vertices from every adjacency list (cf. end of section A.3). The time bound is easily verified. Observe here that the constant hidden behind the O-notation depends on the vocabulary τ. If r denotes the maximal arity of relations of τ then the constant is roughly $|\tau| \cdot r^2$ (each $(a_1, \ldots, a_k) \in R^{\mathfrak{A}}$ leads to at most $k(k-1)$ edges in the Gaifman graph).

The next property of Gaifman graphs is particularly useful:

Lemma 5. *Let $\mathfrak{A}$ be a τ-structure. Then $(\mathcal{T}, (B_t)_{t\in T})$ is a tree-decomposition of $\mathfrak{A}$ if, and only if, it is a tree-decomposition of $\mathcal{G}(\mathfrak{A})$.*

Proof: Assume that $(\mathcal{T}, (B_t)_{t\in T})$ is a tree-decomposition of $\mathcal{G}(\mathfrak{A})$. Now take an $\bar{a} \in R^A$, for some $R \in \tau$. Since in $\mathcal{G}(\mathfrak{A})$ the set $\{a_1, \ldots, a_k\}$ is a clique, it must appear in a single block B_t, hence $\bar{a} \in B_t$. The connectedness (1) condition is equal for both, the structure and its Gaifman graph.

For the other direction, assume that $(\mathcal{T}, (B_t)_{t\in T})$ is a tree-decomposition of $\mathfrak{A}$ and $(a, b) \in E^{\mathcal{G}(\mathfrak{A})}$. By definition of the Gaifman graph, there is an $R \in \tau$ and an $\bar{a} \in R^{\mathfrak{A}}$ such that $(a, b) \in \{a_1, \ldots, a_k\}$. Thus, there is a $t \in T$ such that $\bar{a} \in B_t$, hence $(a, b) \in B_t$. $\square$

By this lemma, most results about tree-decompositions of graphs extend to structures. In particular, the outstanding result that tree-decompositions can be calculated in linear time was originally stated for graphs.

Theorem 6 (Bodlaender[Bod96]). *There is a polynomial $p(x)$ and an algorithm that, given a τ-structure $\mathfrak{A}$, computes a tree-decomposition of $\mathfrak{A}$ of width $w := tw(\mathfrak{A})$ in time*

$$O(2^{p(w)}|A|).$$

The first problem with this algorithm is that it is technically so involved that an implementation appears to be hopeless. On the other hand the factor depending on w grows unreasonably high, even small values. This makes the algorithm almost worthless for practice. A Dynamic programming approach to the problem has been developed by Arnborg et al [ACP87]. This, at least implementable, algorithm needs $O(|A|^{w+1})$ time to compute a tree-decomposition of width $w := \mathrm{tw}(\mathfrak{A})$. But due to its Dynamic nature, this is also the least running time, hence it can not even be used for a particular set of instances. Until now there is no practical algorithm known that calculates a tree-decomposition of optimal width.

In [Ree92], Reed proposed a quadratic time algorithm for finding tree-decompositions of width $4 \cdot \mathrm{tw}(\mathfrak{A})$. Despite of bad constants in the worst case, there are reasons for a nice behavior in practice. In each step, the Divide and Conquer procedure determines a separator (out of a constant number of possible separators) of the graph, splits up the graph into its corresponding components and continues with these components. Since the first encountered separator suffices to continue, it is conceivable that we get reasonable running times in practice (this means for "natural" problem instances). Parts of this algorithm have already been implemented by the author, but there are still no experimental results. Above all there is the question of what are "natural" instances of graphs of tree-width, say, w.

For our purposes, it is convenient to work with tree-decompositions of a normalized form. For a vertex $r \in T$ (the *root*) the pair $(\mathcal{T}, r^{\mathcal{T}})$ is the tree $\mathcal{T}$ rooted in $r^{\mathcal{T}}$ (in the sequel we often omit the superscript). And once we have declared a root in a tree, we can speak about *parents*, *children* and *descendents*. This also induces the natural partial ordering $\leq^{\mathcal{T}}$ on T (the root is the least element). For $t \in T$, we denote the subtree of $\mathcal{T}$ rooted in t by $\mathcal{T}_t$. In relation with the underlying structure $\mathfrak{A}$, we denote by $\mathfrak{A}_t$ the substructure of $\mathfrak{A}$ induced by the set $\bigcup_{s \in T_t} B_s$.

Definition 7. *A* special tree-decomposition *(STD) of width w of a structure $\mathfrak{A}$ is a 3-tuple $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ where $\mathcal{T}$ is a binary tree, $r \in \mathcal{T}$ is the root and every $\bar{b}^t := (b^t_0, \ldots, b^t_w)$ is a $(w+1)$-tuple of elements of A, such that $(\mathcal{T}, \{b^t_0, \ldots, b^t_w\}_{t \in T})$ is a tree-decomposition of $\mathfrak{A}$ of width w.*

Obviously, having a tree-decomposition $(\mathcal{T}, (B_t)_{t \in T})$, a special tree-decomposition can be computed in linear time: Thereto, assume that w.l.o.g all B_t are non-empty and declare an arbitrary vertex to be the root. For each B_t we generate a $|B_t|$-tuple $\bar{b}$, which itself is extended to a $(w+1)$-tuple $\bar{b}^t$. For that, we fill up missing b^t_i with arbitrary "dummy" elements $b \in B_t$. To make $\mathcal{T}$ binary, assume that $t \in T$ has 3 children u_1, u_2, u_3. We create a new vertex t' being a child of t with the unique sibling u_1. u_2 and u_3 become the children of t', and we define $\bar{b}^{t'} := \bar{b}^t$. For more than 3 children we simply iterate this procedure. Altogether, we start with the root, proceed top-down to the leaves and end up with a special tree-decomposition.

Representation of tree-decompositions: Generally, for algorithms to run in linear time, it is important to structure the data adequately. We will use the concept of bounded tree-width to design model checking algorithms that work in time linear in the size of the input structure (plus the size of the output for evaluation problems). Hence structuring the data in a time saving manner becomes a central issue of our work. In chapter A we give a complete account on the machine model we use throughout this thesis and how we store and structure data in this model. The distinction between *small* and *big* objects turns out to be useful in the parameterized setting (cf. the introduction). Roughly, objects whose size depend on the input structure (e.g. subsets of the universe) are called big and require a careful implementation (by lists, arrays or whatsoever depending on the usage). For further details the reader is referred to section A.3.

Let $\mathfrak{A}$ be a τ-structure and $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ a tree-decomposition of $\mathfrak{A}$ of width w. In the bulk of applications, the algorithms work as follows: the tree is traversed in a bottom-up or top-down manner, while in each step the procedure maintains a certain set of partial solutions (usually w.r.t $\mathfrak{A}_t$). In each step, in order to establish the new partial solutions, the algorithm requires constant time access to $\langle B_t \rangle$ for the current t. Unfortunately, this information cannot be retrieved efficiently from the representation of the underlying structure (since the relations are stored as "unstructured" lists). To provide this tree-based access to $\langle B_t \rangle$, we associate with each tree node $t \in T$ and $R \in \tau$ the list of tuples contained in $R^{\langle B_t \rangle}$. By the next lemma, this can be done in linear time.

Lemma 8. *Given a special tree-decomposition $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ of a τ-structure $\mathfrak{A}$, the relations $R^{\langle B_t \rangle}$ for $R \in \tau$ and $t \in T$ can be calculated in time*

$$O(f(\|\tau\|, w) \cdot |A|)$$

for a suitable function $f : \mathbb{N}^2 \to \mathbb{N}$ and $w := tw(\mathfrak{A})$.

The actual problem is the space bound. If we had polynomial space, we would create a k-dimensional array $\mathtt{E_R}[\cdot]$ for each k-ary $R \in \tau$ as follows: for all $t \in T$ add t to the list $\mathtt{E_R}[\bar{c}]$ for all $\bar{c} \in B_t$. In a second loop over all $\bar{a} \in R^A$, we add $\bar{a}$ to $\mathtt{R}[t]$, for all $t \in \mathtt{E_R}[\bar{a}]$. After that, $\mathtt{R}[t]$ contains the set $R^{\langle B_t \rangle}$.

Proof: Let $R \in \tau$, k-ary. We proceed in two steps to compute an array $\mathtt{R}[\cdot]$ such that $\mathtt{R}[t]$ contains the list $R^{\langle B_t \rangle}$ for all $t \in T$. In the first step we generate a list $\mathtt{d}$ consisting of the $(k+1)$-tuples $(\bar{c}, t)$ for all $\bar{c} \in B_t$ and $t \in T$. Furthermore, we order this list ascendingly with respect to the first component. In a second pass, we compare $\mathtt{d}$ with the (also ascendingly ordered) list `rel` that contains the elements of $R^{\mathfrak{A}}$, and add those $\bar{a} \in \mathtt{rel}$ to $\mathtt{R}[t]$ for which $(\bar{a}, t) \in \mathtt{d}$. Since both list are ordered ascendingly, this can be done in time linear in the size of both lists.

The first step is displayed as algorithm 1. Observe that for each t at most $(w+1)^k$ tuples are added to $\mathtt{d}$. Together with the Radix-sort in the last line we thus need $O((k+1)((w+1)^k|T| + |A|))$ steps.

The subroutine $\mathtt{block\_relation}(\mathcal{T}, (B_t)_{t \in T}, R)$, finally, calculates the desired array $\mathtt{R}[\cdot]$ for $R \in \tau$. The only non-trivial part is comparing the two sorted lists. Essentially, we cycle over all $\bar{a} \in R^A$ and gather all $(\bar{c}, t) \in \mathtt{d}$ such that $\bar{a} = \bar{c}$. For these elements we add $\bar{c}$ to the list $\mathtt{R}[t]$. Note that this procedure requires the lists being sorted. Once explained, it is easily verified that the algorithm works correctly. □

Storing special tree-decompositions: Let $\mathfrak{A}$ be a τ structure with special tree-decomposition $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ of width w. By the above lemmas, we can

```
proc gen_ttuples(𝒯, (B_t)_{t∈T})
  for t ∈ T do
      for c̄ ∈ B_t do append(d, (c̄, t)) od
  od
  radix_sort(d);
  return(d);
.
```

Algorithm 1. Generate the list d

compute additional arrays $(\mathtt{R}[\cdot])_{R\in\tau}$ of size $|T|$ such that for all $R \in \tau$ and $t \in T$ we have:

$$\mathtt{R}[t] = R^{\langle B_t\rangle} \text{ as a linked list.}$$

Observe that since B_t has size $(w+1)$, each list $\mathtt{R}[t]$ for k-ary R has size $\leq (w+1)^k$, hence we have the desired constant time decision for queries like $\bar{a} \in R^{\langle B_t\rangle}$.

2.2 Courcelle's result and a simple extension

The following theorem is due to Courcelle [Cou90] and states that MSO-properties are linear time decidable over classes of bounded tree-width. Actually, he proved this theorem using context free hyperedge-grammars and classes definable by equational expressions. We take another way using games that characterize certain fragments of monadic second-order logic (MSO).

Theorem 9. *Let $w \geq 1$ and $\varphi \in$ MSO of quantifier rank q. There is function $f : \mathbb{N}^2 \to \mathbb{N}$ and an algorithm that answers $\mathfrak{A} \models \varphi$, for a structure $\mathfrak{A}$ in time*

$$f(w,q) \cdot ||\mathfrak{A}||$$

for $w := tw(\mathfrak{A})$.

For two structures $\mathfrak{A}, \mathfrak{B}$ and $q \geq 0$ we write $\mathfrak{A} \equiv_q^{\mathrm{MSO}} \mathfrak{B}$, if $\mathfrak{A}$ and $\mathfrak{B}$ satisfy the same sentences $\varphi \in$ MSO of rank $\leq q$. $\equiv_q^{\mathrm{MSO}}$ can be characterized by an *Ehrenfeucht-Fraisse* game that is played by the *Spoiler* and the *Duplicator*. The Spoiler starts choosing one of the two structures, say, $\mathfrak{A}$ and then decides whether he does a *point move* or *set move*. In a point (set) move, the Spoiler chooses some $a \in A$ ($P \subseteq A$), and the Duplicator answers $b \in B$ ($Q \subseteq B$). After q moves, elements $a_1, \ldots, a_r$ and subsets $P_1, \ldots, P_s$ ($q = r+s$) in A and corresponding elements $b_1, \ldots, b_r$ and subsets $Q_1, \ldots, Q_s$ of B have been chosen. Now, the Duplicator has won, if

```
proc block_relation(𝒯, (B_t)_{t∈T}, R)
  rel := R^A;
  d := gen_ttuples(𝒯, (B_t)_{t∈T});
  for ā ∈ rel do
      while () do
          (c̄, t) := current(d);
          if ā = c̄
            then
                append(R[t], ā);
            else
                if predecessor(rel) = c̄
                  then
                      break;
                fi
          fi
          next_element(d);
      od
  od
.
```

Algorithm 2. Calculate $R^{\langle B_t \rangle}$ for all $t \in T$

$\bar{a} \mapsto \bar{b} \in \mathrm{Part}((\mathfrak{A}, P_1, \ldots, P_s), (\mathfrak{B}, Q_1, \ldots, Q_s))$. The next theorem is folklore and can be found in, for instance, [EF95].

Theorem 10. $\mathfrak{A} \equiv_q^{\mathrm{MSO}} \mathfrak{B}$ *iff the Duplicator has a winning strategy for the q-move game on $\mathfrak{A}$ and $\mathfrak{B}$.*

For $q \geq 0$ the *q-type* $\mathrm{tp}_q(\mathfrak{A}, \bar{a})$ of a k-tuple $\bar{a} \in A$ in a τ-structure $\mathfrak{A}$ is the set of all monadic second-order formulas $\varphi(\bar{x})$ of quantifier-rank at most q such that $\mathfrak{A} \models \varphi(\bar{a})$. Using standard techniques, it is easy to see that, up to logical equivalence, there are only finitely many monadic second order formulas with free variables in $\bar{x}$ of quantifier rank at most q. Hence $\mathrm{tp}_q(\mathfrak{A}, \bar{a})$ has a finite description. Observe that $(\mathfrak{A}, \bar{a}) \equiv_q^{\mathrm{MSO}} (\mathfrak{B}, \bar{b})$ is equivalent to $\mathrm{tp}_q(\mathfrak{A}, \bar{a}) = \mathrm{tp}_q(\mathfrak{B}, \bar{b})$.

Since we consider q-types of the substructures $\mathfrak{A}_t$ of $\mathfrak{A}$, it is convenient to introduce the shortcuts $\mathit{stype}(t) := \mathrm{tp}_q(\mathfrak{A}_t, \bar{b}^t)$ and $\mathit{block}(t) := (\langle B_t \rangle, \bar{b}^t)$. We say that two blocks corresponding to tree vertices s, t (of maybe different tree-decompositions) are isomorphic ($\mathit{block}(s) \cong \mathit{block}(t)$), if there is a bijective function $f : B_s \to B_t$ such that for k-ary $R \in \tau$ and all $(a_1, \ldots, a_k) \in B_s$: $\bar{a} \in R^{\langle B_s \rangle} \Leftrightarrow (f(a_1), \ldots, f(a_k)) \in R^{\langle B_t \rangle}$ and $f(b_i^s) = b_i^t$ for all $i = 0, \ldots, w$.

Remark: When we treat special tree-decompositions, we always use $\bar{b}^t$ to denote the tuple associated with the tree node $t \in \mathcal{T}$. Even if we handle various tree-decompositions, this does not introduce ambiguities, if we assume that different tree-decompositions have disjoint underlying trees T. For convenience, we still use B_t to denote $\{b_0^t, \ldots, b_w^t\}$; of course only in situations that do not depend on the specific ordering of the tuple $\bar{b}^t$.

For a special tree-decomposition $(\mathcal{T}, r, (\bar{b}^t)_{t\in T})$, we define the *equality-type* at $s, t \in T$ as

$$id(s,t) := \{(i,j) \mid 0 \leq i < j \leq w, b_i^s = b_j^t\}.$$

The next lemma is essential for Courcelle's theorem.

Lemma 11. *Let $\mathfrak{A}$ and $\mathfrak{A}'$ be τ-structures, and $(\mathcal{T}, r, (\bar{b}^t)_{t\in T})$ and $(\mathcal{T}', r', (\bar{b}^{t'})_{t'\in T'})$ be special tree-decompositions of $\mathfrak{A}$ and $\mathfrak{A}'$ of width $w \geq 1$, respectively. For $t \in T$ and $t' \in T'$ with children u_1, u_2 and v_1, v_2, respectively, the following holds (cf. figure 2.1):*
If $stype(u_1) = stype(v_1)$, $id(t, u_1) = id(t', v_1)$
and $stype(u_2) = stype(v_2)$, $id(t, u_2) = id(t', v_2)$
and $block(t) \cong block(t')$,
then we have $stype(t) = stype(t')$.

Proof: We prove this lemma using Ehrenfeucht-Fraisse games. Because $\mathrm{tp}_q(\bar{b}^{u_1}, \mathfrak{A}_{u_1}) = \mathrm{tp}_q(\bar{b}^{v_1}, \mathfrak{A}'_{v_1})$, the Duplicator has a *local* winning strategy for the corresponding game on $(\mathfrak{A}_{u_1}, \bar{b}^{u_1})$ and $(\mathfrak{A}'_{v_1}, \bar{b}^{v_1})$. Likewise he has a local strategy on $(\mathfrak{A}_{u_2}, \bar{b}^{u_2})$ and $(\mathfrak{B}_{v_2}, \bar{b}^{v_2})$. We have to show that the Duplicator wins the game on $(\mathfrak{A}_t, \bar{b}^t)$ and $(\mathfrak{A}'_{t'}, \bar{b}^{t'})$. We show that the Duplicator can maintain the winning situation. Assume we have a partial isomorphism $\bar{c} \mapsto \bar{d}$ from $(\mathfrak{A}_t, \bar{P}, \bar{b}^t)$ to $(\mathfrak{A}'_{t'}, \bar{Q}, \bar{b}^{t'}$ (this means that $\bar{P}, \bar{Q}$ and $\bar{c}, \bar{d}$ have already been pebbled in the game) and that local strategies exist. We show how the Duplicator act in the set moves.

Assume that the Spoiler pebbles a set $X \subseteq A_t$. This set splits up into the parts $X_1 := X \cap A_{u_1}$ and $X_2 := X \cap A_{u_2}$ and $X^+ := X \setminus (X_1 \cup X_2)$. According to the local strategies, the Duplicator chooses sets $Y_1 \subseteq B_{u_1}, Y_2 \subseteq B_{u_2}$ corresponding to X_1, X_2. Take Y^+ to be the image of X^+ under the claimed isomorphism (recall that $block(t)$ and $block(t')$ are isomorphic). The Duplicator responds $Y := Y_1 \cup Y_2 \cup Y^+$.

We have to show that $\bar{c} \mapsto \bar{d}$ is still a partial isomorphism (now of the structures expanded by X and Y, respectively). Take some $a \in \{c_1, \ldots, c_r\}$. Essentially, there are 2 cases: (i) $a \in \bar{b}^t$: then the isomorphism between $\bar{b}^t$ and $\bar{b}^{t'}$ assures that a is mapped to an element a' with $a \in X \iff a' \in Y$.

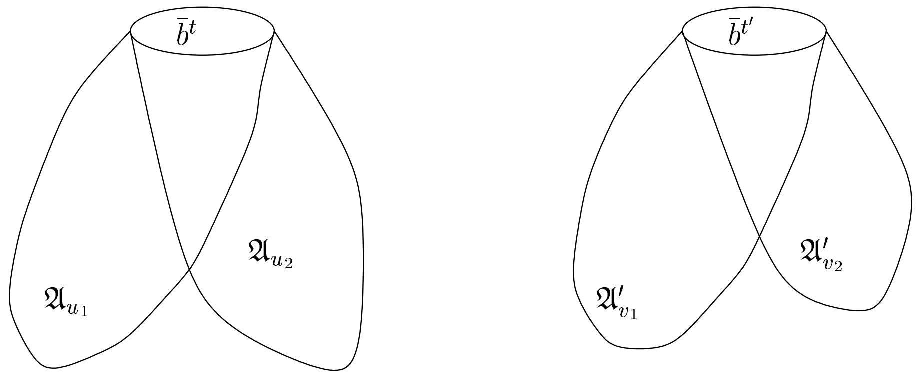

Figure 2.1: Dependency of types

If (ii) $a \notin \bar{b}^t$, then a is contained in, say, A_{u_1}. By the local strategy, according to which the Duplicator chose Y_1, we also have $a \in X_1 \iff a' \in Y_1$. Observe that we do not have to check for the relation of different $a, b \in \{c_1, \ldots, c_r\}$, since there conformity with the relations holds by assumption. Since the point moves work similarly, the proof is completed. □

Let $t \in T$ for a special tree-decomposition $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$. For each $R \in \tau$ we define $itype(R, t) := \{(i_1, \ldots, i_k) \mid (b^t_{i_1}, \ldots, b^t_{i_k}) \in R^{\mathfrak{A}}\}$. The *isomorphism-type* at $t \in T$ is the pair

$$itype(t) := (id(t,t), \{(R, itype(R,t)) \mid R \in \tau\}).$$

Observe that *itype* does not contain any reference to the underlying structure $\mathfrak{A}$. It is easy to see that there is an isomorphism between $(\langle B_t \rangle, \bar{b}^t)$ and $(\langle B_{t'} \rangle, \bar{b}^{t'})$ if, and only if, $itype(t) = itype(t')$. Then it is an easy corollary of the last lemma that for $t \in T$ with children u_1 and u_2, $stype(t)$ only depends on $stype(u_1), stype(u_2)$, $itype(t)$ and how $\bar{b}^t$ and $\bar{b}^{u_1}$, and $\bar{b}^t$ and $\bar{b}^{u_2}$ intersect, respectively. For leaves t, $stype(t)$ only depends on the isomorphism type of $itype(t)$. This means that there are functions Γ and Δ such that for all structures $\mathfrak{A}$ with an STD $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ of width w we have

$$\begin{aligned} stype(t) &= \Gamma(itype(t), stype(u_1), id(u_1,t), stype(u_2), id(u_2,t)) \\ &\qquad \text{for parent nodes } t \text{ with children } u_1, u_2 \\ stype(t) &= \Delta(itype(t)) \qquad \text{for leaves } t. \end{aligned}$$

As already mentioned, there is a finite number of q-types for a fixed $q \geq 0$. Furthermore, the number of different isomorphism types of blocks of the decomposition depends only on w. The same holds for the number of ways in which adjacent blocks may intersect. Hence, the functions Γ and Δ only depend on w and q, but not on the structure $\mathfrak{A}$ and can be assumed to be accessible in constant size lookup tables. This thoughts are directly translated to the program displayed as algorithm 3. The functions `itype(t)` and `id(s, t)` calculate the respective types in constant time by access to the arrays $\mathtt{R}[\cdot]$ for $R \in \tau$ (recall the remark about how tree-decompositions are stored).

```
proc model_check_{φ,w}(𝔄)
   Compute STD (𝒯, r, (b̄^t)_{t∈T}) of width w of 𝔄
   tp := stype(r);
   if φ ∈ tp then accept else reject fi
.
proc stype(t)
   comment: Compute and return stype[t]
   if t is a leaf
      then stype[t] := (Δ(itype(t)))
       else
             u_1 := first child of t;
             u_2 := second child of t
             stype[t] := Γ(itype(t), stype(u_1), id(u_1, t),
                              stype(u_2), id(u_2, t));
   fi
   return(stype[t]);
.
```

Algorithm 3. Checking monadic second-order property φ

The correctness of $\mathtt{model\_check}_{\varphi,w}$ follows directly from the properties of Γ, Δ and the fact that $\mathfrak{A} \models \varphi$ is equivalent to $\varphi \in \mathrm{tp}_q(\mathfrak{A}, \bar{b}^r) = \mathit{stype}(r)$. To estimate the time bound, recall that a STD $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ of $\mathfrak{A}$ can be computed in time $g(w) \cdot \|\mathfrak{A}\|$ for some function g.

Because Δ and Γ are stored in constant size look-up tables and the functions `itype` as well as `id` need constant time, each recursive call in `stype` costs time $g'(w, q)$ for a suitable g'. Summing this up, we obtain a time bound $g(w) \cdot \|\mathfrak{A}\| + g'(w, q)|T| = f(w, q) \cdot |A|$ for some suitable f. $\square$

2.2.1 Evaluation with one free point variable

Let $\mathfrak{A}$ be a τ structure. In this section we describe an extension of the result proved in the last section. In contrast to Courcelle's algorithm, which only computes $\mathrm{tp}_q(\mathfrak{A}, \bar{b}^r)$, we now compute (within essentially the same time bound) $\mathrm{tp}_q(\mathfrak{A}, \bar{b}^t)$ for all $t \in T$ at once (if $\mathcal{T}$ is the underlying tree of a suitable tree-decomposition). This enables us to associate with each vertex $a \in A$ its q-type $\mathrm{tp}_q(\mathfrak{A}, a) := \{\varphi(x) \mid \mathfrak{A} \models \varphi(a) \text{ and } \mathrm{rk}(\varphi) \leq q\}$.

Lemma 12. *Let $w, q \geq 0$. There is an algorithm that, on input $\mathfrak{A}$ with tree-width $\leq w$, calculates the array* $\texttt{tp}[a] := \mathrm{tp}_q(\mathfrak{A}, a)$*, for all $a \in A$ in time*

$$f(w, q) \cdot \|\mathfrak{A}\|$$

for an appropriate $f : \mathbb{N}^2 \to \mathbb{N}$.

Proof: Let $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ be an STD of width w of a τ-structure $\mathfrak{A}$. We adopt the notation from the above discussion. Let $t \in T$. Analogously to $\mathfrak{A}_t$, we define $\mathfrak{A}^t$ as the substructure of $\mathfrak{A}$ "above" t, i.e. $\mathfrak{A}^t := \langle (A \setminus A_t) \cup B_t \rangle$. The top-type is $\mathit{ttype}(t) := \mathrm{tp}_q(\mathfrak{A}^t, \bar{b}^t)$ and we define $\mathit{type}(t) := \mathrm{tp}_q(\mathfrak{A}, \bar{b}^t)$.

Our goal is to compute $\mathit{type}(t)$ for all $t \in T$, since from that we can easily extract the desired information. For $a = b_i^t$, we get $\mathrm{tp}_q(\mathfrak{A}, a) = \{\varphi(x_i) \mid \varphi(x_i) \in \mathit{type}(t)\}$, hence we can concentrate on $\mathit{type}(t)$ for all $t \in T$. Observe that for all $t \in T$ we have (cf. figure 2.2)

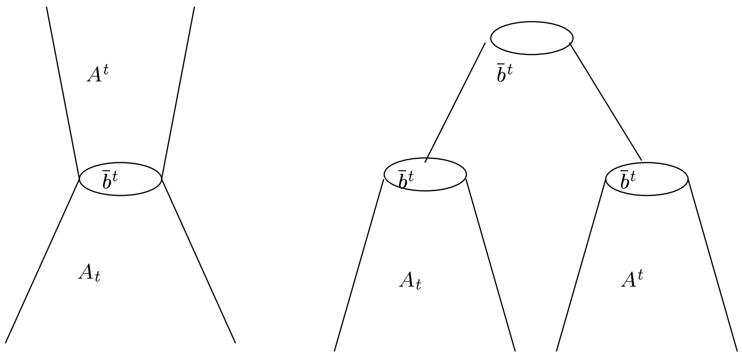

Figure 2.2: Computing $\mathit{type}(t)$.

$$\mathit{type}(t) = \Gamma(\mathit{itype}(t), \mathit{stype}(t), \mathit{id}(t,t), \mathit{ttype}(t), \mathit{id}(t,t)).$$

Hence we are done, if we know $\mathit{stype}(t)$ and $\mathit{ttype}(t)$ for all $t \in T$. The array $\texttt{stype}[\cdot]$ is computed (bottom-up) by the subroutine $\texttt{stype}(r)$ from the proof of Courcelle's theorem. For the computation of $\texttt{ttype}[t]$ for $t \in T$ observe the following:
Let t be a node, $t \neq r$ with parent s and sibling t'. Observe that

$$\mathrm{tp}_q(\langle A_{t'} \cup A^s \rangle, \bar{b}^s) = \Gamma(\mathit{itype}(s), \mathit{ttype}(s), \mathit{id}(s,s), \mathit{stype}(t'), \mathit{id}(t', s)).$$

If we denote this type with τ we get

$$ttype(t) = \Gamma(itype(t), \tau, id(s,t), \tau, id(s,t)).$$

The straightforward implementation is displayed as algorithms 4 and 5. After computing the STD of $\mathfrak{A}$ we compute the two arrays `stype` and `ttype` as explained above. Then an easy loop over all vertices $t \in T$ associates with t the corresponding type $type(t)$. This proves the correctness. For the time bound notice that each of the first two calls take time linear in $|A|$. The same holds for last loop. □

```
proc model_check_{φ,w}(𝔄)
  Compute STD (𝒯, r, (b̄^t)_{t∈T}) of width w of 𝔄
  stype(r); ttype(r);
  for t ∈ T do
      type[t] := Γ(itype(t), stype[t], id(t,t),
                   ttype[t], id(t,t));
      for i = 0 to w do
          if tp[b_i^t] ≠ NULL
            then
                 tp[b_i^t] := {φ(x_i) | φ(x_i) ∈ type[t]}
          fi
      od
  od
.
```

Algorithm 4. Compute `type`$[a]$ for all $a \in A$

The following lemma will be needed in the next chapter.

Corollary 13. *Let $w \geq 1$ and $\varphi(x) \in$ MSO of quantifier rank q. There is a function $f : \mathbb{N}^2 \to \mathbb{N}$ and an algorithm that on input $\mathfrak{A}$ calculates the set $\{a \in A \mid \mathfrak{A} \models \varphi(a)\}$ in time*

$$f(w,q) \cdot \|\mathfrak{A}\|.$$

Proof: In a first step we associate $\mathrm{tp}_q(\mathfrak{A}, a)$ with each vertex $a \in A$ (through the array `tp` in the last lemma). Then we perform a loop over all $a \in A$ and add those a to the set, for which $\varphi(x) \in$ `tp`$[a]$. □

Remark: Note that these model-checking algorithms require the functions Γ and Δ. Since we did not explain how their tables can be computed, our

```
proc ttype(t)
  if t = r then ttype[t] := Δ(itype(t)) fi
  if t is not a leave
    then
        u_1 := first child of t
        htp := Γ(itype(t), ttype[t], id(t, t),
                 stype[u_1], id(u_1, t));
        ttype[u_1] := Γ(itype(u_1), htp, id(u_1, t), htp, id(u_1, t));
        ttype(u_1);
        u_2 := second child of t
        htp := Γ(itype(t), ttype[t], id(t, t),
                 stype[u_2], id(u_2, t));
        ttype[u_2] := Γ(itype(u_2), htp, id(u_2, t), htp, id(u_2, t));
        ttype(u_2);
  fi
.
```

Algorithm 5. Compute the array `ttype`$[t]$

algorithms are non-uniform. In this context, this means that for each MSO-formula φ there is a linear time algorithm $\mathcal{A}_\varphi$ that decides whether φ holds. In the next section we will see that, using automata theoretic ideas, all these algorithms can be made uniform.

2.3 Evaluation on tree-like structures

The last corollary in the last section already had the flavour of an evaluation algorithm for queries. It allows the evaluation of MSO-queries restricted to one free first-order variable in linear time. In this section we develop an algorithm that allows to evaluate arbitrary MSO-queries in time linear in the size of the input plus the size of the output.

In the first section, we present an algorithm evaluating queries on trees. The algorithm is based on the correspondence between tree-automata and monadic second-order logic. Then this algorithm is extended to structures of bounded treewidth.

2.3.1 Evaluating MSO-queries on colored trees

Before we start, we need to introduce the basic concepts concerning tree automata. Let Γ be a finite alphabet. We define τ_Γ to be the vocabulary consisting of a binary relation symbol E, a constant r and an unary relation symbol P_γ for each $\gamma \in \Gamma$. A Γ-*tree* is a τ_Γ-structure whose underlying graph (the reduction to $\{E\}$) is a binary tree rooted in r and the P_γ for $\gamma \in \Gamma$ form a partition of the universe. A *colored tree* is a Γ-tree for some Γ. We say that a vertex t of a colored tree $\mathcal{T}$ has color γ, if $t \in P_\gamma^{\mathcal{T}}$. In this case we write $\gamma(t) = \gamma$.

A (bottom-up) Γ-tree automaton is a 4-tuple $\mathcal{A} = (Q, \delta, \Delta, F)$, where Q is the finite set of *states* and $\Delta : \Gamma \to Q$ the *starting function*. $F \subseteq Q$ is the set of *accepting states* and $\delta : \mathrm{Pow}^{\leq 2}(Q) \times \Gamma \to Q$ is the *transition function*.

The *run* (observe that δ is a function) $\rho : T \to Q$ of $\mathcal{A}$ on the Γ-tree $\mathcal{T}$ is defined in a bottom-up manner. If t is a leaf, then $\rho(t) := \Delta(\gamma(t))$. If t has children s_1, s_2 then $\rho(t) := \delta(\{\rho(s_1), \rho(s_2)\}, \gamma(t))$. The automaton $\mathcal{A}$ is said to *accept* $\mathcal{T}$ if $\rho(r^{\mathcal{T}}) \in F$. Otherwise we say that $\mathcal{A}$ *rejects* $\mathcal{T}$.

A class of colored trees is *recognizable*, if it is the class of trees accepted by some tree-automaton.

Theorem 14 (Thatcher and Wright [TW68, Tho97]). *Let Γ be a finite alphabet. A class of Γ-trees is recognizable, if and only if, it is definable by an* $\mathrm{MSO}[\tau_\Gamma]$ *sentence.*

We proceed as follows: since the above theorem only applies to sentences we show how formulas with free variables have to be treated. First, we replace free first-order variables by free second-order variables. Then we show how free second-order variables correspond to an extension of the vocabulary on the automaton side. So assume that we have given an MSO-formula $\varphi(X_1, \ldots, X_l, y_1, \ldots, y_m)$. We translate φ to a formula $\varphi'(X_1, \ldots, X_l, Y_1, \ldots, Y_m)$ such that for every structure $\mathfrak{A}$ and $\bar{B}, \bar{C} \subseteq A$

$$\mathfrak{A} \models \varphi'(\bar{B}, \bar{C}) \ \Leftrightarrow \quad \text{there are } c_1, \ldots, c_m \in A \text{ with } C_1 = \{c_1\}, \ldots, C_m = \{c_m\} \text{ and } \mathfrak{A} \models \varphi(\bar{B}, \bar{c})$$

Now let be given $\varphi(X_1, \ldots, X_k) \in \mathrm{MSO}[\tau_\Gamma]$ for an alphabet Γ and a Γ-tree $\mathcal{T}$. To code assignments of the free variables X_i into the tree, we extend the alphabet to $\Gamma' := \Gamma \times \{0,1\}^k$. Then, given an assignment $B_1, \ldots, B_k \subseteq T$ for our free variables, we obtain the corresponding Γ'-tree $\mathcal{T}' := (\mathcal{T};\ B_1, \ldots, B_k)$ by the obvious expansion of the vocabulary. In particular, if $\gamma(t)$ denotes the color of t in $\mathcal{T}$, we define

$$\gamma'(t) = (\gamma, \bar{\epsilon}) \quad \Leftrightarrow \quad \gamma(t) = \gamma \text{ and } (t \in B_i \Leftrightarrow \epsilon_i = 1) \text{ for } i = 1, \ldots, k.$$

The tuple $\bar{\epsilon}$ is called the *additional color* of t. This establishes the correspondence between assignments on the one side and colors on the other. Observe that $(\mathcal{T};\ B_1, \ldots, B_k)$ does not denote the expansion of $\mathcal{T}$ with k new unary relations (this is why we use a semicolon instead of a comma). In fact, $(\mathcal{T};\ B_1, \ldots, B_k)$ denotes the $\tau_{\Gamma'}$-structure $(T, E^T, (P^T_{(\gamma,\bar{\epsilon})})_{\gamma\in\Gamma,\bar{\epsilon}\in\{0,1\}^k})$, where $P_{(\gamma,\bar{\epsilon})}$ is defined above (cf. the definition of γ').

It is easy to see that the class

$$\{(\mathcal{T};\ \bar{B}) \mid \mathcal{T} \text{ a colored } \Gamma\text{-tree}, \bar{B} \in \varphi(\mathcal{T})\}$$

is axiomatized by the $\tau_{\Gamma'}$-sentence

$$\begin{aligned}\varphi' := \exists X_1 \ldots X_k \exists (Y_\gamma)_{\gamma\in\Gamma}\ \big(\ &\tilde{\varphi}(\bar{X}) \wedge \\ &\forall t \big(\bigwedge_{\gamma\in\Gamma} \bigwedge_{\bar{\epsilon}\in\{0,1\}^k} (P_{(\gamma,\bar{\epsilon}_i)} t \leftrightarrow (Y_\gamma t \wedge \bigwedge_{i:\epsilon_i=1} X_i t \wedge \bigwedge_{i:\epsilon_i=0} \neg X_i t)))\big)\big).\end{aligned}$$

where $\tilde{\varphi}(\bar{X})$ is obtained from $\varphi(\bar{X})$ by replacing all P_γ with the unary second-order variable Y_γ, and using $\exists(Y_\gamma)_{\gamma\in\Gamma}$ as a shortcut for the existential quantification of all Y_γ, $\gamma \in \Gamma$. As it will turn out, this characterization suffices to apply theorem 14 to our problem. Let us state our main theorem for the case of Γ-trees.

Theorem 15. *There exists a function f and an algorithm that solves the evaluation problem for* MSO*-formulas on colored trees in time*

$$f(\|\varphi\|) \cdot (\|\mathcal{T}\| + \|\varphi(\mathcal{T})\|)$$

on input $\varphi(X_1, \ldots, X_l, x_1, \ldots, x_m)$ and $\mathcal{T}$.

Proof: By the discussion above, we assume the input consisting of a formula of the form $\varphi(X_1, \ldots, X_k) \in \mathrm{MSO}[\tau_\Gamma]$ and a Γ-tree $\mathcal{T}$. Remember that the class $\mathcal{C}$ is defined by the above $\mathrm{MSO}[\tau_{\Gamma'}]$-sentence φ'. Hence, there is a Γ'-automaton $\mathcal{A} = (Q, \Delta, \delta, F)$ recognizing $\mathcal{C}$.

Now we get an easy and constructive characterization of the sought set. For the input Γ-tree $\mathcal{T}$ we have

$$\varphi(\mathcal{T}) = \{\bar{B} \mid \mathcal{A} \text{ accepts } (\mathcal{T};\ \bar{B})\}.$$

This gives us the idea for the algorithm. We simply collect the additional colors such that $\mathcal{A}$ accepts $\mathcal{T}$ expanded by those colors. The presentation of the algorithm (doing this in an optimal way) involves two parts. In the first part, we describe how the algorithm is going to work, including the correctness of the steps performed by the algorithm. In a second part, we give a detailed implementation of the algorithm together with an analysis of the running time and the data structures involved.

I.) The algorithm: We pass through the tree $\mathcal{T}$ three times. In a first bottom-up pass we collect the set of states that could be reached by the automaton. Then we go down, sorting out those states that do not lead to accepting states, and finally we go bottom-up again gathering the satisfying assignments for $\varphi(\bar{X})$ (even considering only those).

1. Bottom-up: By induction from the leaves to the root, we first compute, for every $t \in T$, a set P_t of "potential states" at t: If t is a leaf, then $P_t := \{\Delta((\gamma(t), \bar{\epsilon})) \mid \bar{\epsilon} \in \{0,1\}^k\}$. For an inner vertex t with children s_1 and s_2, we set

$$P_t := \{\delta(\{q_1, q_2\}, (\gamma(t), \bar{\epsilon})) \mid q_1 \in P_{s_1},\ q_2 \in P_{s_2},\ \bar{\epsilon} \in \{0,1\}^k\}.$$

Then for all $t \in T$ and $q \in Q$ we have $q \in P_t$ if, and only if, there are sets $B_1, \ldots, B_k \subseteq T$ such that for the run ρ of $\mathcal{A}$ on $(\mathcal{T};\ \bar{B})$ we have $\rho(t) = q$. This is easily proved by induction.

Note that if $P_r \cap F = \emptyset$ we have $\varphi(\mathcal{T}) = \emptyset$, and no further action is required.

2. Top-down: Starting at the root $r := r^{\mathcal{T}}$ we compute, for every $t \in T$, the subset S_t of P_t of "success states" at t: We let $S_r := F \cap P_r$. If t has parent s and sibling t', then

$$S_t := \{q \in P_t \mid \text{there are } q' \in P_{t'},\ \bar{\epsilon} \in \{0,1\}^k \text{ such that } \delta(\{q, q'\}, (\gamma(s), \bar{\epsilon})) \in S_s\}.$$

Then for all $t \in T$ and $q \in Q$ we have $q \in S_t$ if, and only if, there are sets $B_1, \ldots, B_k \subseteq T$ such that $\mathcal{A}$ accepts $(\mathcal{T}, \bar{B})$ and for the run ρ of $\mathcal{A}$ on $(\mathcal{T}, \bar{B})$ we have $\rho(t) = q$ (again easy by induction).

3. Bottom-up again: Recall that for $t \in T$, by $\mathcal{T}_t$ we denote the subtree of $\mathcal{T}$ rooted in t. For $t \in T$ and $q \in S_t$ we let

$$\mathrm{Sat}_{t,q} := \Big\{\bar{B} \subseteq T_t \ \Big|\ \begin{array}{l} \text{There are sets } B'_1, \ldots, B'_k \subseteq T \text{ such that} \\ B'_i \cap T_t = B_i \text{ for } 1 \leq i \leq k, \\ \mathcal{A} \text{ accepts } (\mathcal{T}, \bar{B}'), \\ \text{and for the run } \rho \text{ of } \mathcal{A} \text{ on } (\mathcal{T}, \bar{B}') \text{ we have} \\ \rho(t) = q \Big\}. \end{array}$$

We compute the sets $\mathrm{Sat}_{t,q}$ inductively from the leaves to the root. Let $t \in T$ and $q \in S_t$. Set $B_1^t := \{t\}$ and $B_0^t = \emptyset$. If t is a leaf, then

$$\mathrm{Sat}_{t,q} = \{(B^t_{\epsilon_1}, \ldots, B^t_{\epsilon_k}) \mid \Delta((\gamma(t), \bar{\epsilon})) = q\}.$$

If t has children u_1 and u_2, then

$$\begin{aligned}\mathrm{Sat}_{t,q} = \Big\{ (B_1 \cup B'_1 \cup B^t_{\epsilon_1}, \ldots, B_k \cup B'_k \cup B^t_{\epsilon_k}) \;\Big|\; & \\ \bar{\epsilon} \in \{0,1\}^k, \text{there exist } q_1 \in S_{u_1}, q_2 \in S_{u_2} \text{such that} & \\ \bar{B} \in \mathrm{Sat}_{u_1,q_1}, \bar{B}' \in \mathrm{Sat}_{u_2,q_2}, \delta(\{q_1, q_2\}, (\gamma(t), \bar{\epsilon})) = q \Big\}. &\end{aligned} \tag{2.1}$$

Note that $\varphi(\mathcal{T}) = \bigcup_{q \in S_r} \mathrm{Sat}_{r,q}$. Hence our algorithm performs the two passes through the tree to detect the necessary and sufficient set of states that have to be considered. Then in a last pass, it incrementally computes the sets $\mathrm{Sat}_{t,q}$ for all $q \in S_t$ in a bottom-up manner. An informal version of that procedure is displayed as algorithm 6.

Algorithm 6: Computing $\varphi(\mathcal{T})$

Input: A colored tree $\mathcal{T}$, an MSO-formula $\varphi(\bar{X})$
Output: $\varphi(\mathcal{T})$

1. Check if there is an alphabet Γ such that $\mathcal{T}$ is a colored Γ-tree and $\varphi(\bar{X})$ is an $\mathrm{MSO}[\tau_\Gamma]$-formula; if this is not the case, then return $\emptyset$
2. Compute the Γ'-tree automaton $\mathcal{A}$ corresponding to φ'
3. Compute P_t for all $t \in T$
4. Compute S_t for all $t \in T$
5. For all $t \in T$ and $q \in S_t$, compute $\mathrm{Sat}_{t,q}$
6. Return $\bigcup_{q \in S_r} \mathrm{Sat}_{r,q}$.

By the above discussion it is obvious that the algorithm works correctly. Next we examine the important steps.

II.) Implementation details: The computation of the automaton $\mathcal{A}$ follows the inductive construction given in the proof of theorem 14. Observe that the size of the vocabulary Γ' (including the arity k of the sought assignments) as well as of the automaton $\mathcal{A}$ only depend on $||\varphi||$. Hence, for the complexity analysis, it does not care how we store the description of $\mathcal{A}$; all queries posed to $\mathcal{A}$ can be answered in time only depending on $||\varphi||$, in particular, retrieving the values of the transition functions δ and Δ.

Having the automaton $\mathcal{A}$ in a convenient way, the sets P_t for all $t \in T$ are computed by the subroutine `calc_potentials` displayed as algorithm 7.

The call of `calc_potentials`(r) provides the set P_t as array element $\mathtt{p}[t]$ for all $t \in T$. For that, we recursively descent the tree and compute P_t by a direct evaluation of the definition. Note that the colors of tree vertices $t \in T$ are stored in an array over T (cf. the appendix on page 110). This admits access to $\gamma(t)$ in time $h(\|\Gamma'\|)$ for a suitable h. Since we also have constant time access to δ and Δ, we can evaluate the definition of P_t (in each level $t \in T$) in time $f(\|\varphi\|)$ for some function f.

proc calc_potentials(t)
  if t is a leaf
    then
        $\mathsf{p}[t] := \{\Delta((\gamma(t), \bar{\epsilon}) \mid \bar{\epsilon} \in \{0,1\}^k\}$
    else
        $u_1 :=$ first child of t
        $u_2 :=$ second child of t
        calc_potentials(u_1); calc_potentials(u_2);
        $\mathsf{p}[t] := \{\delta(\{q_1, q_2\}, (\gamma(t), \bar{\epsilon})) \mid q_1 \in \mathtt{p}[\mathtt{s}_1],$
                        $q_2 \in \mathtt{p}[\mathtt{s}_2], \bar{\epsilon} \in \{0,1\}^k\}$
  fi
.

Algorithm 7. Compute the sets P_t for $t \in T$

Together, the time for the entire procedure call sums up to $f(\|\varphi\|) \cdot |T|$. The subroutine `calc_successfuls` works like `calc_potentials`, implementing the above recursive procedure for the sets S_t (there we fill up the array $\mathtt{s}[\cdot]$). Thus, steps 3 and 4 need time $f(\|\varphi\|) \cdot |T|$ each.

Step 5 is the crucial one, and to compute the sets $\mathrm{Sat}_{t,q}$ we have to proceed particularly carefully to obtain the optimal time bound claimed in the theorem. We shall use the following two facts about the sets $\mathrm{Sat}_{t,q}$.

(1) $\mathrm{Sat}_{t,q}$ is non-empty for all $t \in T$ and $q \in S_t$

(2) $\mathrm{Sat}_{t,q} \cap \mathrm{Sat}_{t,q'} = \emptyset$ for all $t \in T$, $q, q' \in S_t$ such that $q \neq q'$

The first one claims that all sets $\mathrm{Sat}_{t,q}$ that are sufficient for the computation of $\varphi(\mathcal{T})$ are actually necessary, while the second essentially says that no tuple is handled twice.

Proof: (of the two claims) For the first claim assume $q \in S_t$. By definition, this is equivalent to there exists a $\bar{B}$ such that for the accepting run ρ of $\mathcal{A}$

on $(\mathcal{T}, \bar{B})$ we have $\rho(t) = q$; hence $\bar{B} \cap T_t \in \mathrm{Sat}_{t,q}$. The second claim follows from the fact that $\mathcal{A}$ is deterministic, i.e. different states in $t \in T$ can only be reached in runs over different colorings. □

By definition, $\mathrm{Sat}_{t,q}$ contains sets of k-tuples of subsets of the universe T (the assignments for $X_1, \dots, X_k$). To represent these sets we use a data structure of the following form: subsets of the universe T are stored as linked lists (cf. section A.3). Then a k-tuple $\bar{A}$ is represented as a k-size array such that the ith array element is a pointer to the list A_i. The arrays $\bar{A}$ themselves, are again arranged as lists (cf. the distinction between *big* and *small* data structures in the appendix).

Let T', T'' be disjoint sets and S', S'' two non-empty sets of k-tuples of subsets of T' and T'', respectively. The operation required to compute $\mathrm{Sat}_{t,q}$ (cf. the definition) corresponds to

$$\mathtt{merge}(S', S'') := \{(B'_1 \cup B''_1, \dots, B'_k \cup B''_k) \mid \bar{B}' \in S', \bar{B}'' \in S''\},$$

whose implementation is displayed as algorithm 8. Note that this is a so called *destructive procedure*, since after the call the arguments are part of the result.

Let us estimate the time needed by algorithm 8 to merge the sets $\mathtt{s}', \mathtt{s}''$. Apart from the if-statements at the beginning of the code, we perform a loop over all tuples $\mathtt{assign}' \in \mathtt{s}', \mathtt{assign}'' \in \mathtt{s}''$ and concatenate each of them componentwise. This concatenation is done in constant time (cf. the appendix). As mentioned above, we incorporate the arguments into the result. This is guaranteed by the two if-clauses, e.g. the first one assures that each but the last entry of $\mathtt{s}'$ is concatenated with a **copy** of the respective $\mathtt{assign}'' \in \mathtt{s}''$. On the other hand, the elements $\mathtt{assign}'' \in \mathtt{s}''$ (themselves) are concatenated with the last entry of $\mathtt{s}'$. The second if-clause does the same with reversed roles of $\mathtt{assign}'$ and $\mathtt{assign}''$. Hence, all elements of $\mathtt{s}'$ and $\mathtt{s}''$ are reused.

Making a copy of an object $\mathtt{x}$ takes time $\|\mathtt{x}\|$. Hence, altogether we need time

$$O(1 + |\mathtt{merge}(\mathtt{s}', \mathtt{s}'')| + \|\mathtt{merge}(\mathtt{s}', \mathtt{s}'')\| - \|\mathtt{s}'\| - \|\mathtt{s}''\|)$$

to compute the merge of two sets $\mathtt{s}', \mathtt{s}''$. Note here that $|\mathtt{s}|$ stands for the number of items contained in the list $\mathtt{s}$, while $\|\mathtt{s}\| := \sum_{(B_1,\dots,B_k) \in \mathtt{s}} \sum_{i=1}^{k} |B_i|$. Two get a more convenient representation of this upper bound, we use further conditions that are assured by the three if-clauses (i.e. $\mathtt{s}', \mathtt{s}'' \neq \emptyset$ and $\mathtt{s}', \mathtt{s}'' \neq \{(\emptyset, \dots, \emptyset)\}$) at the beginning of the procedure. Under these conditions we

```
proc merge(s', s")
  ms := ∅;  /* the merged set */
  if empty(s') or empty(s") then return(ms) fi;
  if s' = {(∅, ..., ∅)} then return(s") fi;
  if s" = {(∅, ..., ∅)} then return(s') fi;
  for assign' ∈ s' do
      for assign" ∈ s" do
          if last(s', assign')
             then a2 := assign"
              else a2 := copy(assign")
          fi
          if last(s", assign")
             then a1 := assign'
              else a1 := copy(assign')
          fi
          for i = 1 to k do  /* concatenate the sets */
              append(a1[i], a2[i]);
          od
          append(ms, a1);  /* add it to the result */
      od
  od
  return(ms);
.
```

Algorithm 8. Destructively merging two sets of assignments

can show that

$$O(1 + |\mathtt{merge}(\mathtt{s}', \mathtt{s}'')| + \|\mathtt{merge}(\mathtt{s}', \mathtt{s}'')\| - \|\mathtt{s}'\| - \|\mathtt{s}''\|) \\ = O(1 + \|\mathtt{merge}(\mathtt{s}', \mathtt{s}'')\| - \|\mathtt{s}'\| - \|\mathtt{s}''\|), \quad (2.2)$$

which simplifies the subsequent analysis of the main algorithm. To prove this, assume w.l.o.g. that $|\mathtt{s}'| \geq |\mathtt{s}''|$ and that $|\mathtt{s}'|$ is sufficiently big (otherwise (2.2) holds trivially). If $|\mathtt{s}''| = 1$ then $\|\mathtt{merge}(\mathtt{s}', \mathtt{s}'')\| = |\mathtt{s}'| \cdot \|\mathtt{s}''\| + \|\mathtt{s}'\|$, hence $\|\mathtt{merge}(\mathtt{s}', \mathtt{s}'')\| - \|\mathtt{s}'\| - \|\mathtt{s}''\| \geq |\mathtt{s}'| - 1$ (since $\|\mathtt{s}''\| \geq 1$). Together with $|\mathtt{merge}(\mathtt{s}', \mathtt{s}'')| = |\mathtt{s}'|$, this implies equation (2.2) for this case.

If $|\mathtt{s}''| > 1$, then $\|\mathtt{merge}(\mathtt{s}', \mathtt{s}'')\| - \|\mathtt{s}'\| - \|\mathtt{s}''\| \geq |\mathtt{merge}(\mathtt{s}', \mathtt{s}'')|$. Hence, as before, we obtain (2.2).

Having proved an upper bounded for algorithm 8, let us move to the main algorithm. Given a $t \in T$, algorithm 9 computes the sets $\mathrm{Sat}_{t,q}$ for all $q \in S_t$.

The case where t is a leaf is handled in the loop around line 5. Its correctness is obvious. For inner nodes we call the procedure `calc_sat`(t, u_1, u_2) displayed as algorithm 10, where u_1, u_2 are the children of t.

```
1  proc calc_sat(t)
2    if t is a leaf
3      then
4          for q ∈ S_t do
5              sat[t, q] := {(B^t_{ε_1}, ..., B^t_{ε_k}) | Δ((γ(t), ε̄)) = q}
6          od
7      else
8          u_1 := first child of t;
9          u_2 := second child of t;
10         calc_sat(t, u_1, u_2);
11   fi
12 .
```

Algorithm 9. Calculate $\mathrm{Sat}_{t,q}$ for all $q \in S_t$

Now consider algorithm 10. In the second line we compute the values Sat_{u_i,q_i} for $q_i \in S_{u_i}$, $i = 1, 2$. Based on these values we want to compute $\mathrm{Sat}_{t,q}$ for $q \in S_t$ in an efficient way (reusing all already obtained partial assignments). We explain the functionality in two phases: first, we describe why we get the right answer and second, we give a detailed account on the running time of the procedure.

Essentially, the procedure consists of two loops. In the first one (lines 4 to 15), we prepare for the second one (lines 16 to 21), which actually computes the desired sets by means of merges. To understand the first loop, take a pair $(q_1, q_2) \in S_{u_1} \times S_{u_2}$, a $q \in S_t$ and define

$$B(q, q_1, q_2) := \{(B^t_{\epsilon_1}, \ldots, B^t_{\epsilon_k}) \mid \delta(\{q_1, q_2\}, (\gamma(t), \bar{\epsilon})) = q\}.$$

The first inner loop computes this set `B`$[q, q_1, q_2]$, which intuitively contains the tuples by which assignments from Sat_{u_1,q_1} and Sat_{u_2,q_2} extend to assignments in $\mathrm{Sat}_{t,q}$. For the second inner loop (around the if-clause in line 10) we need to have a closer look at the problem of reusing intermediate results.

Our objective is to compute the sets $\mathrm{Sat}_{t,q}$, which in terms of the `merge`-function can be characterized as the following disjoint union:

$$\mathrm{Sat}_{t,q} = \bigcup_{\substack{(q_1,q_2) \in S_{u_1} \times S_{u_2} \\ B(q,q_1,q_2) \neq \emptyset}} \mathtt{merge}(\mathtt{merge}(\mathrm{Sat}_{u_1,q_1}, B(q, q_1, q_2)), \mathrm{Sat}_{u_2,q_2}). \qquad (*)$$

In a straightforward implementation, we would simply merge all sets Sat_{u_1,q_1}, $B(q, q_1, q_2)$, and Sat_{u_2,q_2} with $B(q, q_1, q_2) \neq \emptyset$. Recall here that `merge` is destructive, i.e. after the call of `merge(s', s'')` the parameters `s'`, `s''` are no longer available (they have become part of the result). On the other hand, we cannot copy each argument before passing it to the merge-procedure; this would be to wasteful.

Our solution to this problem is to make copies of all Sat_{u_1,q_1} and Sat_{u_2,q_2} that are to be merged with some $B(q, q_1, q_2)$ (one copy for each subsequent application of (*)). When doing the copies, we guarantee at the same time that we only make a copy, if it is really necessary. This allows direct calls of the merge-procedure in the subsequent loop and, as we will see, assures the claimed time bound.

To avoid unhandy pre-estimations of the number of copies that have to be made, we enhance the data structure for the sets $\mathrm{Sat}_{t,q}$ by a *"usage"*-flag, which indicates, if the set has already been copied. Initially, after the object is created, this flag is set "off". In this context we introduce the function `fcopy(x)` which first looks if the flag of `x` is "on", and if so, returns a copy of `x`. If the flag is "off", the flag is flipped "on" and `x` itself is returned.

So we come back to the second inner loop: for each non-empty $B(q, q_1, q_2)$ we store the triple $(\mathrm{Sat}_{u_1,q_1}, \mathrm{Sat}_{u_2,q_2}, B(q, q_1, q_2))$ in the list $\mathtt{l}[q]$. And these are exactly the arguments of the merges of the form (*) that have to be done to obtain $\mathrm{Sat}_{t,q}$ for all $q \in S_t$.

In the second outer loop (lines 16 to 21), we finally generate the sets $\mathrm{Sat}_{t,q}$ from the elements of the list $\mathtt{l}[q]$. This proves the correctness of the algorithm.

The running time: We will show by induction over $t \in T$ that all $\mathrm{Sat}_{t,q}$, $q \in S_t$ are computed in time

$$T(t) \leq c \cdot \sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\| + f(\|\mathcal{A}\|) \cdot |T_t|$$

for a constant c and a suitable function f (these will be defined later). Then the theorem follows with property (2) of $\mathrm{Sat}_{t,q}$. To prove this bound, we first give a recursive characterization of the time algorithm 9 spends at a node $t \in T$. In a second step, we show inductively that this recursive formula yields the upper bound claimed above.

```
proc calc_sat(t, u_1, u_2)
  calc_sat(u_1); calc_sat(u_2);
  initialize array B
  for (q_1, q_2) ∈ S_{u_1} × S_{u_2} do
      for ε̄ ∈ {0,1}^k do
          q := δ({q_1, q_2}, (γ(t), ε̄))
          B[q, q_1, q_2] := B[q, q_1, q_2] ∪ {(B^t_{ε_1}, ..., B^t_{ε_k})}
      od
      for q ∈ S_t do
          if B[q, q_1, q_2] ≠ ∅
            then
                  append(I[q], (B[q, q_1, q_2], fcopy(sat[u_1, q_1]),
                                                fcopy(sat[u_2, q_2])));
          fi
      od
  od
  for q ∈ S_t do
      for (B, cp_1, cp_2) ∈ I[q] do
          med := merge(B, cp_1);
          sat[t, q] := sat[t, q] ∪ merge(med, cp_2);
      od
  od
.
```

Algorithm 10. Compute $\mathrm{Sat}_{t,q}$ for t with children u_1, u_2

I.) For a leaf t and $q \in S_t$ the set $\mathrm{Sat}_{t,q}$ is computed in time $f_1(\|\mathcal{A}\|)$ for some function f_1 by a direct evaluation of the definition (in line 5, algorithm 9).

Let t be an inner node with children u_1, u_2. In this case, we call algorithm 10. In the first line, we descend recursively computing the sets Sat_{u_i,q_i} for $q_i \in S_{u_i}$ and $i = 1, 2$. So let us assume that these sets are already known.

To estimate the complexity of the first loop fix a $(q_1, q_2) \in S_{u_1} \times S_{u_2}$ and observe that the first inner loop, which computes $B(q, q_1, q_2)$, needs time $2^k \cdot (c_{\mathcal{A}} + k)$ ($c_{\mathcal{A}}$ charges the time to get the result of the transition function δ, and k steps are charged to produce one k-tuple $(B^t_{\epsilon_1}, \dots, B^t_{\epsilon_k})$).

It is easy to see that for the second inner loop, in each pass, we need c_Q steps to check the if-statement (by c_Q we denote the number of states of $\mathcal{A}$, i.e. $c_Q := |Q|$). Furthermore, for all (q_1, q_2) together, we need

$$c_1 \cdot \Big(\sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} \big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\|\big) - \sum_{q_1 \in S_{u_1}} \|\mathrm{Sat}_{u_1,q_1}\| - \sum_{q_2 \in S_{u_2}} \|\mathrm{Sat}_{u_2,q_2}\| \Big)$$

steps to do the copies. In particular, the first double sum accounts for the effort to make the copies (in line 12). From this cost we have to subtract the work we saved by reusing the sets Sat_{u_i,q_i}, $q_i \in S_{u_i}$. The constant c_1 takes care for the overhead produced by `fcopy` and the generation of $\mathtt{l}[q]$ (this constant is independent from $\mathcal{T}$ and $\mathcal{A}$). Hence, we need

$$c_Q^2 \cdot 2^k \cdot (c_{\mathcal{A}} + k) + c_Q^3 + c_1 \cdot \sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} \big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\|\big)$$
$$- c_1 \cdot \big(\sum_{q_1 \in S_{u_1}} \|\mathrm{Sat}_{u_1,q_1}\| + \sum_{q_2 \in S_{u_2}} \|\mathrm{Sat}_{u_2,q_2}\|\big)$$

steps for the entire first loop (in the sequel, we denote the term $c_Q^2 \cdot 2^k \cdot (c_{\mathcal{A}} + k) + c_Q^3$ by $f(\|\mathcal{A}\|)$).

The second loop simply merges all of the recently computed sets. The needed time sums up to

$$c_2 \cdot \sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} \big(\|\mathtt{merge}(\mathtt{merge}(\mathrm{Sat}_{u_1,q_1}, B(q,q_1,q_2)), \mathrm{Sat}_{u_2,q_2})\|$$
$$- \|\mathrm{Sat}_{u_1,q_1}\| - \|B(q,q_1,q_2)\| - \|\mathrm{Sat}_{u_2,q_2}\|\big),$$

where c_2 is the constant from the merge-function. Observe that we do not need to account for the loops, since there are no void loops (property (1) above). In characterization (*) of $\mathrm{Sat}_{t,q}$ the union is disjoint, hence, this value is dominated by

$$c_2 \cdot \big(\sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\| - \sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} \big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\|\big)\big).$$

Altogether, it is obvious that for the time $T(t)$ spent at an inner node $t \in T$

with children u_1, u_2 we have

$$\begin{aligned}T(t) \leq T(u_1) + T(u_2) + f(\|\mathcal{A}\|)&\\ + (c_1 - c_2) \cdot \sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} &\Big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\|\Big)\\ - c_1 \cdot \Big(\sum_{q_1 \in S_{u_1}} \|\mathrm{Sat}_{u_1,q_1}\| + \sum_{q_2 \in S_{u_2}} \|\mathrm{Sat}_{u_2,q_2}\|\Big) &+ c_2 \cdot \sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\|.\end{aligned}$$

II.) It remains to prove that this recursion formula yields the claimed upper bound. Define $c := c_1 + c_2$. By induction hypothesis, $T(u_i) \leq f(\|\mathcal{A}\|) \cdot |T_{u_i}| + c \cdot \sum_{q_i \in S_{u_i}} \|\mathrm{Sat}_{u_i,q_i}\|$ for $i = 1, 2$. If we insert this into the recursive characterization of $T(t)$, we get $T(t) \leq f(\|\mathcal{A}\|) \cdot (|T_{u_1}| + |T_{u_2}| + 1)$

$$\begin{aligned}+ (c_1 + c_2) \cdot \Big(\sum_{q_1 \in S_{u_1}} \|\mathrm{Sat}_{u_1,q_1}\| + \sum_{q_2 \in S_{u_2}} \|\mathrm{Sat}_{u_2,q_2}\|\Big)&\\ + (c_1 - c_2) \cdot \sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} &\Big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\|\Big)\\ - c_1 \cdot \Big(\sum_{q_1 \in S_{u_1}} \|\mathrm{Sat}_{u_1,q_1}\| + \sum_{q_2 \in S_{u_2}} \|\mathrm{Sat}_{u_2,q_2}\|\Big) &+ c_2 \cdot \sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\|,\end{aligned}$$

which is equal to

$$\begin{aligned}f(\|\mathcal{A}\|) \cdot |T_t| + c_2 \cdot \sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\|&\\ + (c_1 - c_2) \cdot \sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} &\Big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\|\Big)\\ &+ c_2 \cdot \Big(\sum_{q_1 \in S_{u_1}} \|\mathrm{Sat}_{u_1,q_1}\| + \sum_{q_2 \in S_{u_2}} \|\mathrm{Sat}_{u_2,q_2}\|\Big).\end{aligned}$$

It is easy to see that

$$\sum_{q \in S_t} \sum_{\substack{(q_1,q_2),\\ B(q,q_1,q_2)\neq\emptyset}} \Big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\|\Big) \leq \sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\|,$$

hence, we have

$$
\begin{aligned}
T(t) \leq f(\|\mathcal{A}\|) \cdot |T_t| + (c_1 + c_2) \cdot \sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\| \\
+ c_2 \cdot \Big(\big(\sum_{q_1 \in S_{u_1}} \|\mathrm{Sat}_{u_1,q_1}\| + \sum_{q_2 \in S_{u_2}} \|\mathrm{Sat}_{u_2,q_2}\| \big) \\
- \sum_{q \in S_t} \sum_{\substack{(q_1,q_2), \\ B(q,q_1,q_2) \neq \emptyset}} \big(\|\mathrm{Sat}_{u_1,q_1}\| + \|\mathrm{Sat}_{u_2,q_2}\| \big) \Big),
\end{aligned}
$$

which is smaller than $f(\|\mathcal{A}\|) \cdot |T_t| + c \cdot \sum_{q \in S_t} \|\mathrm{Sat}_{t,q}\|$ (since the last addend is ≤ 0). As mentioned in the beginning of the complexity analysis, this bound on $T(t)$ completes the proof of theorem 15. □

We immediately get the following corollary.

Corollary 16. *There exists a function f and an algorithm that solves the evaluation problem for* MSO*-formulas with no free set variables on colored trees in time*

$$f(\|\varphi\|) \cdot (\|\mathcal{T}\| + |\varphi(\mathcal{T})|)$$

on input $\varphi(x_1, \dots, x_k)$ and $\mathcal{T}$.

Proof: For a formula without free set variables we have $\|\varphi(\mathcal{T})\| = k \cdot |\varphi(\mathcal{T})| \leq \|\varphi\| \cdot |\varphi(\mathcal{T})|$ (since all variables x_i must occur freely in $\varphi(\bar{x})$). □

The witness case for MSO on colored trees is now an easy corollary of theorem 15.

Corollary 17. *There exists a function f and an algorithm that solves the witness problem for* MSO*-formulas on colored trees in time*

$$f(\|\varphi\|) \cdot \|\mathcal{T}\|$$

on input $\varphi(X_1, \dots, X_l, x_1, \dots, x_m)$ and $\mathcal{T}$.

Proof: Proceed as before, until the calculation of $\mathrm{Sat}_{t,q}$. Now we select an accepting run $\rho : T \to Q$ on $\mathcal{T}$ and then compute a particular coloring of $\mathcal{T}$ inducing that run. We start with an arbitrary $q \in S_r$ and proceed top-down. Assume that in node t we are in state $q \in S_t$ and t has children u_1, u_2. Now choose an additional color $\vec{\epsilon}^t \in \{0,1\}^k$ and states $q_1 \in S_{u_1}$ and $q_2 \in S_{u_2}$ such that $\delta(\{q_1, q_2\}, (\gamma(t), \vec{\epsilon}^t)) = q$. If t is a leaf and $\rho(t) = q$, choose an $\vec{\epsilon}^t \in \{0,1\}^k$ such that $\Delta((\gamma(t), \vec{\epsilon}^t)) = q$. The tuple $\bar{A} = (A_1, \dots, A_k)$ with $A_i := \{t \mid \epsilon_i^t = 1\}$ is an satisfying assignment. The analysis is straightforward and omitted. □

2.3.2 The extension to tree-like structures

In this section we extend the result presented in the previous section from colored trees to structures of bounded tree-width. For that, we reduce the question for tree-like structures to the question for colored trees and then apply the results proved in the last section. This reduction goes back to Arnborg, Lagergren and Seese [ALS91].

Some main techniques could already be seen in the proof of Courcelle's theorem. Essentially, we code all information, given by a tree-decomposition into the colors of the underlying tree. This is done by the following definitions. Let $\mathfrak{A}$ be a τ-structure and $(\mathcal{T}, r, (\bar{b}^t)_{t\in T})$ a special tree-decomposition of $\mathfrak{A}$ of width $w \geq 1$. With $\mathcal{T}$ we associate the colored tree $\mathcal{T}^*$ with underlying tree $\mathcal{T}$ and coloring $\gamma(t) = (\gamma_1(t), \gamma_2(t), \gamma_3(t))$ for $t \in T$, where

- $\gamma_1(t) := id(t,t)$
- $\gamma_2(t) := \begin{cases} id(t,s) & \text{for the parent } s \text{ of } t \text{ if } t \neq r^{\mathcal{T}}, \\ \emptyset & \text{if } t = r^{\mathcal{T}}, \end{cases}$
- $\gamma_3(t) := \{(R, itype(R,t)) \mid R \in \tau\}$

For the colors recall the definition of $id(t,s)$ and $itype(R,t)$ on page 23 in this chapter. Examining the definitions, we observe that $\mathcal{T}^*$ is a $\Gamma(w,\tau)$-tree for

$$\Gamma(w,\tau) := \mathrm{Pow}(\{0,\dots,w\}^2) \times \mathrm{Pow}(\{0,\dots,w\}^2) \times \\ \mathrm{Pow}\big(\tau \times \bigcup_{R\in\tau, r\text{-ary}} \mathrm{Pow}(\{0,\dots,w\}^r)\big).$$

The next lemma is an easy corollary of lemma 8.

Lemma 18. *Given a τ-structure $\mathfrak{A}$ and a special tree-decomposition $(\mathcal{T}, r, (\bar{b}^t)_{t\in T})$ of width w of $\mathfrak{A}$, the corresponding $\Gamma(w,\tau)$-tree $\mathcal{T}^*$ can be computed in time*

$$f(\tau, w) \cdot \|\mathcal{T}\|$$

for some function f.

Proof: By lemma 8, we dispose of the substructures $\langle B_t \rangle$ induced by the block of the tree-decomposition. From those substructures, it is easy to compute the sets $itype(R,t)$ for all $R \in \tau$ and $t \in T$. The way we compute the sets $id(t,t)$ for $t \in T$ and $id(t,s)$ for $t \in T$ with parent s is obvious. $\square$

$\mathcal{T}^*$ does not contain any reference to the structure $\mathfrak{A}$, hence the problem arises of how to "speak" about subsets and elements of A in the colored tree $\mathcal{T}^*$. Thereto, we code each element $a \in A$ by a canonical tuple of subsets of T. This tuple defines a unique (t, i) such that $a = b_i^t$.

For an element $a \in A$ we let $node(a)$ be the minimal $t \in T$ with respect to $\leq^{\mathcal{T}}$ such that $a \in \{b_0^t, \dots, b_w^t\}$. We define $\bar{U}(a) := (U_1(a), \dots, U_w(a))$ by

$$U_i(a) := \begin{cases} \{node(a)\} & \text{if } b_i^{node(a)} = a \text{ and } b_j^{node(a)} \neq a \text{ for } 1 \leq j < i, \\ \emptyset & \text{otherwise,} \end{cases}$$

for $0 \leq i \leq w$. This tuple is a unique representative for the element a: it encodes in which $\bar{b}^t$ the element a occurs first (w.r.t $\leq^{\mathcal{T}}$), and its minimal place (w.r.t. the ordered tuple) in the tuple $\bar{b}^t$. For a subset $B \subseteq A$ we let $U_i(B) := \bigcup_{a \in B} U_i(a)$ and $\bar{U}(B) := (U_0(B), \dots, U_w(B))$. Note that for subsets $U_0, \dots, U_w \subseteq T$ there exists an $a \in A$ such that $\bar{U} = \bar{U}(a)$ if, and only if,

(1) $\bigcup_{i=0}^{w} U_i$ is a singleton.

(2) For all $t \in T$ and $0 \leq i < j \leq w$: If $t \in U_j$ then $(i, j) \notin \gamma_1(t)$.

(3) For all $t \in T$, $t \neq r^{\mathcal{T}}$, with parent s and $0 \leq i, j \leq w$: If $t \in U_i$ then $(i, j) \notin \gamma_2(t)$.

Moreover, there is a subset $B \subseteq A$ with $\bar{U} = \bar{U}(B)$ just in case the conditions (2) and (3) are fulfilled.

To be able to talk about $\mathfrak{A}$ in $\mathcal{T}^*$ we define MSO-formulas $\mathrm{Elem}(X_0, \dots, X_w)$ and $\mathrm{Set}(X_0, \dots, X_w)$ such that for arbitrary $U_0, \dots, U_w \subseteq T$ we have

$$\begin{aligned} \mathcal{T}^* \models \mathrm{Elem}(\bar{U}) &\iff \text{there is an } a \in A \text{ with } \bar{U} = \bar{U}(a); \\ \mathcal{T}^* \models \mathrm{Set}(\bar{U}) &\iff \text{there is a subset } B \subseteq A \text{ with } \bar{U} = \bar{U}(B). \end{aligned}$$

The formula

$$\mathrm{Set}(\bar{U}) := \forall t \Big(\bigwedge_{0 \leq i < j \leq w} U_j t \to \bigwedge_{\substack{\gamma \in \Gamma(w,\tau) \\ (i,j) \in \gamma_1}} \neg P_\gamma t \wedge \bigwedge_{0 \leq i,j \leq w} U_i t \to \bigwedge_{\substack{\gamma \in \Gamma(w,\tau) \\ (i,j) \in \gamma_2}} \neg P_\gamma t \Big)$$

realizes the recognition of sets. For $\mathrm{Elem}(\bar{U})$ we just add the condition that $U_0 \cup \dots \cup U_w$ is a singleton. The next lemma completes the reduction.

Lemma 19. *Every* MSO*-formula* $\varphi(X_1, \ldots, X_k, y_1, \ldots, y_l)$ *can be effectively translated to a formula* $\varphi^*(\bar{X}_1, \ldots, \bar{X}_k, \bar{Y}_1, \ldots, \bar{Y}_l)$ *such that:*

(1) For all $B_1, \ldots, B_k \subseteq A, a_1, \ldots, a_l \in A$ *we have*

$$\mathfrak{A} \models \varphi(B_1, \ldots, B_k, a_1, \ldots, a_l) \\ \iff \mathcal{T}^* \models \varphi^*(\bar{U}(B_1), \ldots, \bar{U}(B_k), \bar{U}(a_1), \ldots, \bar{U}(a_l)).$$

(2) For all $\bar{U}_1, \ldots, \bar{U}_k, \bar{V}_1, \ldots, \bar{V}_l \subseteq T$ *such that* $\mathcal{T}^* \models \varphi^*(\bar{U}_1, \ldots, \bar{U}_k, \bar{V}_1, \ldots, \bar{V}_l)$ *there exist* $B_1, \ldots, B_k \subseteq A$, $a_1, \ldots, a_l \in A$ *such that* $\bar{U}_i = \bar{U}(B_i)$ *for* $1 \leq i \leq k$ *and* $\bar{V}_i = \bar{U}(a_i)$ *for* $1 \leq i \leq l$.

Proof: The proof of the first statement proceeds by induction over φ, hence is effective. The cases of Boolean connectives are trivial, and for an existential quantifier $\exists Y \varphi(Y)$ we take the formula $\exists \bar{Y}(\mathrm{Set}(\bar{Y}) \rightarrow \varphi^*(\bar{Y}))$. Likewise, we proceed for first-order quantifications. The only non-trivial part is the case of atomic φ.

Let $R \in \tau$ be an r-ary relation symbol and $\varphi = Ry_1 \ldots, y_r$. The variables $y_1, \ldots, y_r$ are assumed to be represented by tuples $\bar{Y}_1, \ldots, \bar{Y}_r \subseteq T$, which canonically encode vertices of A. In general, the components a_i of a tuple $\bar{a} \in A$ are not coded by the same minimal node in T, hence using $\bar{Y}_1, \ldots, \bar{Y}_r$ we cannot decide whether the tuple is contained in R^A or not (since we only dispose of the relation on elements of a common block).

To bypass this problem, we extend tuples $\bar{V} \subseteq T$ to maximally "consistent" ones. We say that $\bar{V} = (V_1, \ldots, V_w) \subseteq T$ is *closed*, if the following hold:

- If $t \in V_i$ and $(i, j) \in \gamma_1(t)$ then $t \in V_j$.
- If t has parent s, $s \in V_j$ and $(i, j) \in \gamma_2(t)$ then $t \in V_i$.

Clearly, in a similar way as for $\mathrm{Elem}(\bar{X})$ we can define an MSO-formula $\mathrm{Closed}(X_0, \ldots, X_w)$ such that

$$\mathcal{T}^* \models \mathrm{Closed}(\bar{V}) \iff \bar{V} \text{ is closed.}$$

Note that, for $a \in A$ and closed $\bar{V}$, if $\bar{U}(a) \subseteq \bar{V}$, $t \in T$, $0 \leq i \leq w$ and

$a_i^t = a$ then $t \in V_i$. Now φ is translated to the formula

$$\begin{aligned}\varphi^*(\bar{Y}_1, \ldots, \bar{Y}_r) := \mathrm{Elem}(\bar{Y}_1) \wedge \ldots \wedge \mathrm{Elem}(\bar{Y}_r) \wedge \\ \forall \bar{Z}_1 \ldots \bar{Z}_r \Big(\bigwedge_{i=1}^{r} \big(\bar{Y}_i \subseteq \bar{Z}_i \wedge \mathrm{Closed}(\bar{Z}_i)\big) \rightarrow \\ \exists x \big(\bigvee_{i_1, \ldots, i_r = 0}^{w} (Z_{1,i_1} x \wedge \ldots \wedge Z_{r,i_r} x \wedge \bigvee_{\substack{\gamma \in \Gamma(\tau, w) \\ (i_1, \ldots, i_r) \in \gamma_3(R)}} P_\gamma x)\big)\Big).\end{aligned}$$

By definition, it is obvious that for this case we have

$$\mathfrak{A} \models \varphi(a_1, \ldots, a_r) \Leftrightarrow \mathcal{T}^* \models \varphi^*(\bar{U}(a_1), \ldots, \bar{U}(a_r)).$$

Recall that, since for every occurrence of a tuple of variables $\bar{X}, \bar{Y}$ we additionally impose $\mathrm{Set}(\bar{Y})$ and $\mathrm{Elem}(\bar{X})$, respectively, the second claim trivially holds. This completes the proof of the lemma. ☐

Now the sought theorem is immediate.

Theorem 20. *There exists a function $f : \mathbb{N}^2 \rightarrow \mathbb{N}$ and an algorithm that solves the evaluation problem for* MSO*-formulas in time*

$$f(\|\varphi\|, tw(\mathfrak{A})) \cdot (\|\mathfrak{A}\| + \|\varphi(\mathfrak{A})\|)$$

with inputs $\varphi \in$ MSO *and the structure $\mathfrak{A}$.*

Proof: The main procedure is displayed as algorithm 11.

Algorithm 11: Computing $\varphi(\mathfrak{A})$

Input: A structure $\mathfrak{A}$, an MSO-formula $\varphi(\bar{X}, \bar{y})$
Output: $\varphi(\mathfrak{A})$

1. Check if $\mathfrak{A}$ and φ have the same vocabulary, say τ; if this is not the case then return $\emptyset$.

2. Compute a special tree-decomposition $(\mathcal{T}, r, (\bar{b}^t)_{t \in T})$ of tree-width $w :=$ $\mathrm{tw}(\mathfrak{A})$ of $\mathfrak{A}$.

3. Compute the corresponding $\Gamma(\tau, w)$-tree $\mathcal{T}^*$.

4. Compute the formula φ^*.

5. Compute $\varphi^*(\mathcal{T}^*)$.

6. Compute $\varphi(\mathcal{A})$.

The first line needs time linear in the size of the input. By theorem 6, line 2 requires time $f_1(\|\tau\|, w)|A|$ for some suitable f_1. The same holds for line 3 (by lemma 18).

The time needed by the forth line only depends on the size of φ (by the construction in the proof of lemma 19) and line 5 can be done in $f_3(\|\varphi^*\|) \cdot (|T| + \|\varphi^*(\mathcal{T}^*)\|)$ steps by theorem 15. Finally, given the set $\varphi^*(\mathcal{T}^*)$, the output $\varphi(\mathfrak{A})$ can be computed by applying the map $\bar{U}(a) \mapsto a$ in time $\|\varphi^*(\mathcal{T})\| = f(w)\|\varphi(\mathfrak{A})\|$. For that, we first create arrays $\mathtt{u_i}$ such that $\mathtt{u_i}[\mathtt{t}] = a$ iff $t \in U_i(a)$. This is done by a breadth-first search through $\mathcal{T}$ in which we gradually fill up the arrays $\mathtt{u_i}$. We have an Boolean array $\mathtt{a}$ initialized to false, and at each tree node t we perform a loop from $i = 0, \dots, w$ and if $\mathtt{a}[i] =$ false we set $\mathtt{u_i}[t] = b_i^t$, otherwise we do nothing (because then b_i^t has already its code).

Using this, for each $\bar{U} \subseteq T$ we can do look-ups in constant time and find the corresponding $U \subseteq A$ in time $\|\bar{U}\| = f(w) \cdot |U|$. Thus $\varphi(\mathcal{T})$ is translated to $\varphi(\mathfrak{A})$ within the claimed time bound.

Altogether, we get the desired time bound. □

Like in the case of trees, we easily get

Theorem 21. *There exists a function $f : \mathbb{N}^2 \to \mathbb{N}$ and an algorithm that solves the evaluation problem for* MSO*-formulas without free second-order variables in time*

$$f(\|\varphi\|, tw(\mathfrak{A})) \cdot (\|\mathfrak{A}\| + |\varphi(\mathfrak{A})|)$$

with inputs $\varphi \in$ MSO and a structure $\mathfrak{A}$.

as well as the witness case.

Theorem 22. *There exists a function $f : \mathbb{N}^2 \to \mathbb{N}$ and an algorithm that solves the witness problem for* MSO*-formulas in time*

$$f(\|\varphi\|, tw(\mathfrak{A})) \cdot \|\mathfrak{A}\|$$

with the formula $\varphi(X_1, \dots, X_l, x_1, \dots, x_m)$ and the structure $\mathfrak{A}$ as inputs.

Finally, for completeness, we state a theorem proven by Arnborg, Lagergren and Seese [ALS91], saying that the counting problem for structures of bounded tree-width is solvable in linear time. For this theorem we have to take the uniform cost model, which allows arithmetical operations $(*, +)$ being done in constant time (independently from their size).

Theorem 23 ([ALS91]). *There exists a function $f : \mathbb{N}^2 \to \mathbb{N}$ and an algorithm that solves the counting problem for* MSO*-formulas in time*

$$O(f(\|\varphi\|, tw(\mathfrak{A})) \cdot \|\mathfrak{A}\|)$$

with the formula $\varphi(X_1, \ldots, X_l, x_1, \ldots, x_m)$ and the structure $\mathfrak{A}$ as inputs.

Note that this theorem can easily be proved using the machinery developed here. The only difference is that in algorithm 6, instead of $\mathrm{Sat}_{t,q}$, we calculate its cardinality (and this can be done by a simple multiplication instead of a merge). Since the reduction of tree-like structures to binary trees conserves the cardinality of the solutions, this part is trivial.

Chapter 3

First-order Properties over Tree-decomposable Structures

In this chapter we present an extension of the notion of bounded tree-width motivated by the observation that common structures have the property that the tree-width only depends on the diameter of the structure. To capture this behaviour, Eppstein [Epp99] introduced the *diameter-treewidth* property of graph classes.

Another way to look at this is that for these structures the tree-width of a neighborhood of radius $r \geq 1$ of an arbitrary vertex is bounded by $f(r)$ for some suitable function $f : \mathbb{N} \to \mathbb{N}$. We say that classes of structures satisfying this property have *bounded local tree-width.* It turns out that over such classes the model checking problem is fixed parameter tractable. In particular, for each $k \geq 1$ there is an $f : \mathbb{N} \to \mathbb{N}$ and an algorithm that decides $\mathfrak{A} \models \varphi$ in time $f(\|\varphi\|) \cdot |A|^{1+1/k}$ for $\varphi \in \mathrm{FO}$.

Later on, we further restrict this notion to obtain a linear time algorithm for the decision problem of FO on so called *locally tree-decomposable* classes. At the end of the chapter we examine the complexity of special versions of the presented algorithm and exhibit the crucial bottlenecks.

3.1 Local tree-likeness

Let $\mathfrak{A}$ be a τ-structure for some vocabulary τ. Recall that by $\mathcal{G}(\mathfrak{A})$ we denote its Gaifman graph. The *distance* $d^{\mathfrak{A}}(a, b)$ between two elements $a, b \in A$ in $\mathfrak{A}$ is the length $d^{\mathfrak{A}}(a, b)$ of the shortest path in $\mathcal{G}(\mathfrak{A})$ connecting a and b. For $r \geq 1$ and $a \in A$ we define the *r-neighborhood* of a in $\mathfrak{A}$ to be $N_r^{\mathfrak{A}}(a) := \{b \in A \mid d^{\mathfrak{A}}(a, b) \leq r\}$.

Definition 24. *(1) The* local tree-width *of a structure* $\mathfrak{A}$ *is the function* $\mathrm{ltw}^{\mathfrak{A}} : \mathbb{N} \to \mathbb{N}$ *defined by*

$$\mathrm{ltw}^{\mathfrak{A}}(r) := \max\left\{ tw(\langle N_r^{\mathfrak{A}}(a)\rangle) \;\middle|\; a \in A\right\}.$$

(2) A class $\mathcal{C}$ *of structures has* bounded local tree-width*, if there is a function* $f : \mathbb{N} \to \mathbb{N}$ *such that* $\mathrm{ltw}^{\mathfrak{A}}(r) \leq f(r)$ *for all* $\mathfrak{A} \in \mathcal{C}$*,* $r \in \mathbb{N}$*.*

There are several important classes of structures (graphs) that have bounded local tree-width.

Example 25. *Structures of bounded tree-width.* Let $\mathfrak{A}$ be a structure with $\mathrm{tw}(\mathfrak{A}) \leq w$. Then $\mathrm{ltw}^{\mathfrak{A}}(r) \leq w$ for all $r \geq 1$.

The *degree* of an element $a \in A$ of a structure $\mathfrak{A}$ is the size of its 1-neighborhood, $|N_1^{\mathfrak{A}}(a)|$. The *degree* of a structure $\mathfrak{A}$ is the maximal degree realized by an element of A.

Example 26. *Structures of bounded degree.* Let $\mathfrak{A}$ be a structure of degree at most d, for a $d \geq 1$. Then $\mathrm{ltw}^{\mathfrak{A}}(r) \leq d(d-1)^{r-1}$ for all $r \geq 1$.

With reference to the Gaifman graph we can extend the notion of planarity to arbitrary structures. We say that a structure $\mathfrak{A}$ is *planar* if the corresponding Gaifman graph $\mathcal{G}(\mathfrak{A})$ is planar. Observe that a structure always has the same local tree-width as its Gaifman graph. Hence, all bounds on graph classes presented in the sequel extend to the corresponding classes of structures.

Example 27 (Robertson and Seymour [RS84]). *Planar graphs.* The class of planar graphs has bounded local tree-width. More precisely, for every planar graph $\mathcal{G}$ and $r \geq 1$ we have $\mathrm{ltw}^{\mathcal{G}}(r) \leq 3r$.

Example 28 (Eppstein [Epp99]). *Graphs of bounded genus.* Let S be a surface. Then the class of all graphs embeddable in S has bounded local tree-width. More precisely, there is a constant $c \geq 1$ such that for all graphs $\mathcal{G}$ embeddable in S and for all $r \geq 1$ we have $\mathrm{ltw}^{\mathcal{G}}(r) \leq c \cdot g(S) \cdot r$ (here $g(S)$ denotes the genus of the surface S).

A *minor* of a graph $\mathcal{G}$ is a graph $\mathcal{H}$ that is obtained from a subgraph of $\mathcal{G}$ by contracting edges. Classes of embeddable graphs (these include planar graphs) are examples of minor-closed graph classes. Eppstein gave the following characterization of all classes of bounded local tree-width that are closed under taking minors. An *apex* graph is a graph $\mathcal{G}$ that has a vertex $v \in G$ such that $\mathcal{G} \setminus \{v\}$ is planar.

Theorem 29 (Eppstein [Epp95, Epp99]). *Let $\mathcal{C}$ be a minor-closed class of graphs. Then $\mathcal{C}$ has bounded local tree-width if, and only if, $\mathcal{C}$ does not contain all apex graphs.*

Let $K_{m,n}$ be the *complete bipartite* graph with parts of size n and m, respectively.

Example 30. *$K_{3,m}$ free graphs.* Let $\mathcal{C}$ be a class of graphs that do not contain all $K_{3,m}$ as minors. Then $\mathcal{C}$ has bounded local tree-width.

This follows immediately from the observation that $K_{3,m}$, for $m \geq 3$, is an apex graph.

Observe that the class of structures of bounded degree is the only example that is not closed under taking minors.

Neighborhood and tree covers: A successful way to examine local properties of structures is to cover them by small neighborhoods. This has been done systematically in, for instance, [ABCP93, AP90, Pel93].

Definition 31. *Let $r, s \geq 1$. An (r, s)-*neighborhood cover *of a structure $\mathfrak{A}$ is a family $\mathcal{N}$ of subsets of A with the following properties:*

(1) For every $a \in A$ there exists an $N \in \mathcal{N}$ such that $N_r^{\mathfrak{A}}(a) \subseteq N$.

(2) For every $N \in \mathcal{N}$ there exists an $a \in A$ such that $N \subseteq N_s^{\mathfrak{A}}(a)$.

For a subset N of the universe of a structure $\mathfrak{A}$, the set of vertices $a \in N$ such that $N_r^{\mathfrak{A}}(a) \subseteq N$ is denoted by $K^r(N)$. Covers are coded as lists of lists (recall the definition of *big* objects in section A.3). Hence the *size* of a family $\mathcal{N}$ of sets is $\|\mathcal{N}\| := \sum_{N \in \mathcal{N}} |N|$. The algorithm in the following lemma is an adaptation of an algorithm due to Peleg [Pel93] to our situation.

Lemma 32 (Peleg [Pel93]). *Let $k \geq 1$. Then there is an algorithm that, given a graph $\mathcal{G}$ and an $r \geq 1$, computes an $(r, 2kr)$-neighborhood cover $\mathcal{N}$ of $\mathcal{G}$ of size $\|\mathcal{N}\| = O(|G|^{1+1/k})$ in time $O(\sum_{N \in \mathcal{N}} \|\langle N \rangle^{\mathcal{G}}\|)$.*

Proof: We compute the cover as it is displayed in algorithm 12. We iteratively compute a neighborhood cover $\mathcal{N}$, maintaining a set H of vertices whose r-neighborhood has not yet been covered by a set in $\mathcal{N}$. In each iteration step of the main loop (lines 4-14), the algorithm picks an arbitrary vertex a from H (actually the first in the list) and starts to compute increasing neighborhoods of a (in lines 7-11) until a certain threshold is reached (line 11). Then it adds the computed set N to the cover $\mathcal{N}$ and removes

1 **proc** comp_cover($\mathcal{G}, r$)
2   $H :=$ copy(G); $n := |G|$
3   $\mathcal{N} := \emptyset$;
4   **while** $H \neq \emptyset$ **do**
5       $a :=$ first(H);
6       $N := \{a\}$;
7   **do**
8           $M :=$ copy(N);
9           $L := N_r^{\mathcal{G}}(M) \cap H$;
10           $N := N_r^{\mathcal{G}}(L)$;
11   **od while** $|N| > n^{1/k}|M|$
12       $\mathcal{N} := \mathcal{N} \cup \{N\}$;
13       $H := H \setminus L$;
14   **od**
15 .

Algorithm 12. Computing a neighborhood-cover of $\mathcal{G}$

all points whose neighborhood has now been covered from H, before it goes to the next iteration of the main loop. This process is repeated until H is empty.

Correctness: Let $\mathcal{G}$ be a graph, $r \geq 1$ and $\mathcal{N}$ the cover computed by the algorithm. We prove a series of claims which entail the statement of the lemma.

Claim 1: For every $a \in G$ there exists an $N \in \mathcal{N}$ such that $N_r(a) \subseteq N$.

Proof: An element a is removed from the set H of uncovered elements of A in line 13 only if it belongs to a set L such that $N := N_r^{\mathcal{G}}(L)$ has been added to $\mathcal{N}$. Hence this N contains $N_r^{\mathcal{G}}(a)$. This proves Claim 1.

Claim 2: For every $N \in \mathcal{N}$ there exists an $a \in G$ such that $N \subseteq N_{2kr}^{\mathcal{G}}(a)$.

Proof: We consider the iteration of the main loop that leads to N. Let a be the element chosen in line 5, and let $N_0 := \{a\}$. Let $l \geq 1$ be the number of times the loop in lines 7-11 is repeated. For $1 \leq i \leq l$, let N_i be the value of N after the ith iteration. Then for $1 \leq i \leq l-1$ we have $|N_i| > n^{1/k}|N_{i-1}|$, and therefore $|N_i| > n^{i/k}$. Thus $l \leq k$.

Furthermore, it is easy to see that for $1 \leq i \leq l$ we have $N_i \subseteq N_{2ir}^{\mathcal{G}}(a)$. This implies Claim 2.

Claims 1 and 2 show that $\mathcal{N}$ is indeed an $(r, 2kr)$-neighborhood cover of $\mathcal{G}$. The following Claim 3 shows that the cover is not too large.

Claim 3. $\|\mathcal{N}\| \leq n^{1+(1/k)}$.

Proof: For $N \in \mathcal{N}$, let M be the corresponding set that is computed in the last iteration of the loop in lines 7-11 that let to N (i.e. M is the value of N after the second but last iteration of the loop).

We first show that for distinct $N_1, N_2 \in \mathcal{N}$ we have $M_1 \cap M_2 = \emptyset$. To see this, suppose that N_1 is computed first. Let H_1 be the value of H after the iteration of the main loop in which N_1 has been computed. Note that for every $a \in M_1$ and $b \in H_1$ we have $d^{\mathcal{G}}(a, b) > r$. Moreover, $M_2 \subseteq N_2 \subseteq N_r^{\mathcal{G}}(H_1)$. Thus $M_1 \cap M_2 = \emptyset$.

By the condition in line 11, we have $|N| \leq n^{1/k}|M|$ for all $N \in \mathcal{N}$, hence obtain

$$\|\mathcal{N}\| = \sum_{N \in \mathcal{N}} |N| \leq n^{1/k} \sum_{N \in \mathcal{N}} |M| \leq n^{1/k} \cdot n.$$

The last inequality holds because the M are disjoint subsets of G. This proves Claim 3.

Running time: To estimate the running time of the algorithm, we claim that each iteration of the main loop requires time $O(\|\langle N \rangle\|)$, for the vertex set N added to $\mathcal{N}$ in the corresponding iteration pass. Then the entire algorithm runs in the claimed time bound.

All sets except for H are implemented as lists. For H we must be able to compute set differences $H \setminus L$ for some list L (line 13) and to check whether $H \neq \emptyset$ (and if so, provide an example). We fit these requirements by using a *referenced list* as described in appendix A on page 108.

We prove the claimed time bound for each single iteration. To calculate L in line 9 we perform a multi-source *breadth-first search* starting in M. To assure that we stay within H, we additionally check for new vertices, if they belong to H (this takes constant time). Observe that we possibly consider edges that are not contained in $\langle L \rangle^{\mathcal{G}}$, but these are all contained in N, which is built in the subsequent line. Furthermore, each of these edges is considered at most r many times (the depth of the search tree). Thus, together we need at most $O(\|\langle N \rangle\|)$ steps.

Note the impact of the usage of referenced lists on the running time: we remove a set L (given as a list) from H calling the subroutine `remove()` displayed as algorithm 29 (in the appendix). Removing m vertices takes time $O(m)$. The emptiness check in line 4 and providing an example $a \in H$ in line 5 needs constant time.

So the overall running time to compute N is $O(\|\langle N\rangle\|)$.

It may seem that to check the condition of line 11 we need multiplication, which is not available as basic operation in our machine model (cf. chapter A). However, before we start the main computation, we can produce tables that store the values m^l and $m^l \cdot n$ for $1 \leq l \leq k$, $1 \leq m \leq n$ in linear time on a standard RAM. We use the fact that

$$(m+1)^l = \sum_{(\epsilon_1,\dots,\epsilon_l)\in\{0,1\}^l} m^{\sum_{i=1}^{l}\epsilon_i}$$

to inductively compute the tables. Remember that we treat k as a constant, hence the sum can be evaluated in constant time. We use these tables to check the condition of line 11 in constant time. $\square$

Let us apply this lemma to arbitrary structures. Let τ be a vocabulary and $\mathcal{C}$ be a class of τ structures of bounded local tree-width (witnessed by $f : \mathbb{N} \to \mathbb{N}$). Assume furthermore that $\mathcal{N}$ is an $(r, 2kr)$ neighborhood cover of the Gaifman graph $\mathcal{G}(\mathfrak{A})$ computed by algorithm 12. Pick some $N \in \mathcal{N}$: we know that $\mathrm{tw}(\langle N\rangle^{\mathfrak{A}}) \leq f(2kr)$, hence by lemma 3 there is an l such that $\|\langle N\rangle^{\mathfrak{A}}\| \leq l|N|$. Hence $\|\mathfrak{A}\| = O(\|\mathcal{N}\|)$ and we get the time bound for the subsequent corollary (observe that the constant behind the O-notation depends on k, r and τ).

Corollary 33. *Let $k, r \geq 1$, τ a vocabulary, and $\mathcal{C}$ a class of τ-structures of bounded local tree-width. Then there is an algorithm that, given a structure $\mathfrak{A} \in \mathcal{C}$, computes an $(r, 2kr)$-neighborhood cover $\mathcal{N}$ of $\mathfrak{A}$ of size $\|\mathcal{N}\| = O(|A|^{1+(1/k)})$ in time $O(|A|^{1+(1/k)})$.*

It is worth to mention the following immediate consequence of the preceding discussion.

Corollary 34. *Let τ be a vocabulary and $\mathcal{C}$ be a class of τ-structures of bounded local tree-width. Then for every $k \geq 1$ there is a constant c such that for all structures $\mathfrak{A} \in \mathcal{C}$ we have $\|\mathfrak{A}\| \leq c|A|^{1+(1/k)}$.*

As already mentioned, the constant c depends on k, τ and the function f bounding the local tree-width for $\mathcal{C}$. For most algorithmical purposes, a neighborhood cover is more than we need. To do local computations quickly, we can cover a structure by subsets of bounded tree-width (which is an immediate consequence, if we have bounded local tree-width and cover sets of bounded radius). This is captured by the next definition.

Definition 35. *Let $r, w \geq 1$. An (r, w)-*tree cover *of a structure $\mathfrak{A}$ is a family $\mathcal{T}$ of subsets of A with the following properties:*

(1) For every $a \in A$ there exists a $T \in \mathcal{T}$ such that $N_r^{\mathfrak{A}}(a) \subseteq T$.

(2) For every $T \in \mathcal{T}$ we have $tw(\langle T \rangle^{\mathfrak{A}}) \leq w$.

The next lemma concerns computing the substructures induced by the sets of a tree cover. This computation is necessary for most algorithms that work on the cover sets.

Lemma 36. *Let $\mathcal{T}$ be an $(r+1, w)$-tree cover for a structure $\mathfrak{A}$, $r, w \geq 1$. Then an (r, w)-tree cover $\mathcal{T}'$ and the set $\{\langle T' \rangle^{\mathfrak{A}} \mid T' \in \mathcal{T}'\}$ of induced structures can be computed in time*

$$O\big(\sum_{T \in \mathcal{T}} \|\langle T \rangle^{\mathfrak{A}}\|\big).$$

Proof: Let $\mathcal{T}$ be an $(r+1, w)$-tree cover of a structure $\mathfrak{A}$ for some $r, w \geq 1$. Take an arbitrary $T \in \mathcal{T}$. We call a vertex $v \in T$ *outermost*, if there is an adjacent vertex $u \in A \setminus T$. In a first step, we copy all but the outermost vertices in T, to a separate list T'. This is done by a breadth-first search, which terminates a branch, when a vertex outside T is to be put into the queue (cf. the standard implementation of a breadth-first search tree using queues). Then for each $v \in T'$ we copy its adjacency list, leaving out the edges to vertices not in T'. Obviously, we obtain $\langle T' \rangle$ and furthermore all the T' form an (r, w)-neighborhood cover of $\mathfrak{A}$.

Concerning the running time, observe that the set T' can be found in time $O(\|\langle T' \rangle\|)$ (since for the outermost vertices we only consider one edge per vertex that is not contained in $E^{\langle T \rangle}$). The same time bound holds for the second step. □

The construction of tree covers defined in the next lemma is implicit in [Epp99].

Lemma 37 (Eppstein [Epp99]). *Let $r \geq 1$ and $\mathcal{C}$ be a class of graphs that is closed under taking minors and has bounded local tree-width. Let $f : \mathbb{N} \to \mathbb{N}$ be a function bounding the local tree-width of the graphs in $\mathcal{C}$.*

Then there is an algorithm that, given a graph $\mathcal{G} \in \mathcal{C}$, computes an $(r, f(2r+1))$-tree cover $\mathcal{T}$ of $\mathcal{G}$ of size $\|\mathcal{T}\| = O(|G|)$ in time $O(|G|)$.

Proof: Let $\mathcal{G} \in \mathcal{C}$ and choose an arbitrary vertex $a_0 \in G$. For $0 \leq i \leq j$, let $G[i,j] := \{a \in G \mid i \leq d^{\mathcal{G}}(a_0, a) \leq j\}$.

We claim that $\mathrm{tw}(\langle G[i,j] \rangle) \leq f(j-i+1)$. This is immediate if $i = 0$ or $i = 1$, because then $G[i,j] \subseteq N_j^{\mathcal{G}}(a_0)$. If $i > 1$, we simply contract the connected subgraph $\langle G[0, i-1] \rangle^{\mathcal{G}}$ to a single vertex b_0. We obtain a minor

$\mathcal{G}'$ of $\mathcal{G}$, which is also an element of $\mathcal{C}$ by our assumption that $\mathcal{C}$ is closed under taking minors. $\mathcal{G}'$ still contains the set $G[i, j]$ as it is, but now this set is contained in $N^{\mathcal{G}'}_{j-i+1}(b_0)$. This proves the claim.

The claim implies that for all $r \geq 1$, the family $\mathcal{T} := \{G[i, i+2r] \mid i \geq 0\}$ is an $(r, f(2r+1))$-tree cover of $\mathcal{G}$ of size at most $(2r+1)|G|$. On input $\mathcal{G}$, we can choose an arbitrary a_0 and then compute this tree cover in linear time by a breadth-first search. □

It is not clear at all, whether we can find an (r, s)-neighborhood cover $\mathcal{N}$ for planar graphs $\mathcal{G}$ in $O(|G|)$ steps, for fixed $r, s \geq 1$, while the above construction offers an easy and effective way to compute tree covers. Tree covers of size linear in the size of the structure and a linear time algorithm computing such a cover is exactly what we need in our algorithms. The following definition makes this precise.

Definition 38. *A class $\mathcal{C}$ of structures is* locally tree-decomposable*, if there is an algorithm that, given an $r \geq 1$ and a structure $\mathfrak{A} \in \mathcal{C}$, computes an (r, w)-tree cover $\mathcal{T}$ of $\mathfrak{A}$ (for some suitable $w \geq 1$ depending only on r) such that $\|\mathcal{T}\| = O(|A|)$ in time $O(|A|)$.*

Immediately, we get a lemma corresponding to lemma 3 saying that locally tree-decomposable classes are *sparse*.

Corollary 39. *Let τ be a vocabulary and $\mathcal{C}$ be a locally tree-decomposable class of τ-structures. Then there is a $c \geq 1$ such that for all structures $\mathfrak{A} \in \mathcal{C}$ we have $\|\mathfrak{A}\| = c \cdot |A|$.*

Observe that the constant c depends on τ and the value w bounding the tree-width of the parts of the cover.

Example 40. *Classes closed under taking minors.* Classes of planar graphs, of graphs of bounded genus, and $K_{3,m}$-free classes that are closed under taking minors are locally tree-decomposable.

This follows immediately from lemma 37 and the fact that all these classes have bounded local tree-width. Although not closed under minors, classes of graphs of bounded degree are locally tree-decomposable. Let $\mathcal{C}$ be a class of graphs with degree bounded by, say, d and $r \geq 1$. Then the family $\{N^{\mathcal{G}}_r(v) \mid v \in G\}$ is an $(r, d(d-1)^{r-1})$-tree cover of $\mathcal{G}$ of size $|G| \cdot d(d-1)^{r-1}$.

Example 41. *Graphs of bounded degree.* The class of graphs of bounded degree is locally tree-decomposable.

Remark: If we carefully analyse the covers constructed in all examples, we find that either a cover consists of subsets with constant radius, or of rings of constant breadth. For the rings, to deduce bounded tree-width, we contract their interior to a single vertex and apply the bounded local tree-width property. For subsets with constant radius, the only example is the class of graphs of bounded degree. This appears to be a "natural" dichotomy, since only artificial classes of locally bounded tree-width, not closed under minors, with unbounded valence are known (e.g. bounded degree plus all trees). Recently Eppstein [Epp99] asked, if at all there is any other "natural" family of graphs, not minor-closed, but with bounded local tree-width.

Naturally the question arises, if classes of bounded local tree-width are sparse (recall that corollary 34 gave an upper bound of the size of such structures). The following counter example was provided by Grohe [FG99] and shows that local tree-decomposability is a proper restriction of bounded local tree-width.

Example 42. We construct a class $\mathcal{C}$ of graphs of bounded local tree-width such that for every constant c there is a graph $\mathcal{G} \in \mathcal{C}$ with $\|\mathcal{G}\| \geq c|G|$.

We use the following theorem due to Erdös [Erd59]: *For all $g, k \geq 1$ there exists a graph of girth greater than g and chromatic number greater than k.* Remember that the *girth* $g(\mathcal{G})$ of a graph $\mathcal{G}$ is the length of the shortest cycle in $\mathcal{G}$ and the *chromatic number* $\chi(\mathcal{G})$ of $\mathcal{G}$ is the least number of colors needed to color the vertices of $\mathcal{G}$ such that no two adjacent vertices have the same color. It is easy to see that every graph $\mathcal{G}$ with $\chi(\mathcal{G}) \geq k$ has a connected subgraph $\mathcal{H}$ with average degree

$$\frac{2|E^{\mathcal{H}}|}{|H|} \geq k - 1$$

(cf. [Die97], p. 98). The *diameter* of a connected graph $\mathcal{G}$ is the number $\mathrm{diam}(\mathcal{G}) := \max\{d^{\mathcal{G}}(a, b) \mid a, b \in G\}$.

We inductively construct a family $(\mathcal{G}_i)_{i \geq 1}$ of graphs as follows: $\mathcal{G}_1$ is the graph consisting of two vertices and an edge between them. Suppose now that $\mathcal{G}_i$ is already defined. Let $\mathcal{G}'_{i+1}$ be a graph with $g(\mathcal{G}'_{i+1}) \geq 2\mathrm{diam}(\mathcal{G}_i) + 1$ and $\chi(\mathcal{G}'_{i+1}) \geq 2i + 3$. Let $\mathcal{G}_{i+1}$ be a connected subgraph of $\mathcal{G}'_{i+1}$ with

$$\frac{2|E^{\mathcal{G}_{i+1}}|}{|G_{i+1}|} \geq 2i + 2.$$

Clearly, $g(\mathcal{G}_{i+1}) \geq g(\mathcal{G}'_{i+1}) \geq 2\mathrm{diam}(\mathcal{G}_i) + 1$.

Observe that for every $r \geq 1$ and every graph $\mathcal{G}$, if $2r + 1 < g(\mathcal{G})$ then $\mathrm{ltw}^{\mathcal{G}}(r) \leq 1$. Moreover, if $\mathcal{G}$ is connected then $\mathrm{ltw}^{\mathcal{G}}(r) = \mathrm{tw}(\mathcal{G})$ for all

$r \geq \mathrm{diam}(\mathcal{G})$. For every $i \geq 1$ and $\mathrm{diam}(\mathcal{G}_i) \leq r < \mathrm{diam}(\mathcal{G}_{i+1})$, we let $f(r) := \max\{\mathrm{tw}(\mathcal{G}_i), \mathrm{ltw}^{\mathcal{G}_{i+1}}(r)\}$. We claim that $\mathrm{ltw}^{\mathcal{G}_i}(r) \leq f(r)$ for all $i, r \geq 1$. This is obvious for $i = 1$. For $i \geq 2$, we have to distinguish between three cases: If $r < \mathrm{diam}(\mathcal{G}_{i-1}) \leq \frac{1}{2}(g(\mathcal{G}_i) - 1)$, then $\mathrm{ltw}^{\mathcal{G}_i}(r) \leq 1 \leq f(r)$. If $\mathrm{diam}(\mathcal{G}_{i-1}) \leq r < \mathrm{diam}(\mathcal{G}_i)$, then $\mathrm{ltw}^{\mathcal{G}_i}(r) \leq f(r)$ immediately by the definition of f. If $r \geq \mathrm{diam}(\mathcal{G}_i)$, then $\mathrm{ltw}^{\mathcal{G}_i}(r) = \mathrm{tw}(\mathcal{G}_i) \leq f(r)$.

Thus the class $\mathcal{C} := \{\mathcal{G}_i \mid i \geq 1\}$ has bounded local tree-width. On the other hand, for every $i \geq 2$ we have $\|\mathcal{G}_i\| \geq |E^{\mathcal{G}_i}| \geq i|G_i|$.

3.2 Gaifman's theorem

Recall the definition of the distance between two elements of a relational structure. Let τ be a vocabulary. Then for every $r \geq 1$ there is a formula $\delta_r(x, y) \in \mathrm{FO}[\tau]$ such that for all τ-structures $\mathfrak{A}$ and $a, b \in A$ we have $\mathfrak{A} \models \delta_r(a, b) \;\Leftrightarrow\; d^{\mathfrak{A}}(a, b) \leq r$. Within formulas we write $d(x, y) \leq r$ instead of $\delta_r(x, y)$ and $d(x, y) > r$ instead of $\neg\delta_r(x, y)$.

If $\varphi(x)$ is a first-order formula, then $\varphi^{N_r(x)}(x)$ is the formula obtained from $\varphi(x)$ by relativizing all quantifiers to $N_r(x)$, that is, by replacing every subformula of the form $\exists y \psi(x, y, \bar{z})$ by $\exists y(d(x, y) \leq r \wedge \psi(x, y, \bar{z}))$ and every subformula of the form $\forall y \psi(x, y, \bar{z})$ by $\forall y(d(x, y) \leq r \rightarrow \psi(x, y, \bar{z}))$. A formula $\psi(x)$ of the form $\varphi^{N_r(x)}(x)$, for some $\varphi(x)$, is called *r-local* . The basic property of r-local formulas $\psi(x)$ is that it only depends on the r-neighborhood of x whether they hold at x or not, that is, for all structures $\mathfrak{A}$ and $a \in A$ we have $\mathfrak{A} \models \psi(a) \iff \langle N_r^{\mathfrak{A}}(a) \rangle \models \psi(a)$.

The next theorem is due to Gaifman and states that properties expressible in first-order logic are local.

Theorem 43 (Gaifman [Gai82]). *Every first-order sentence is equivalent to a Boolean combination χ of sentences of the form*

$$\exists x_1 \ldots \exists x_m \big(\bigwedge_{1 \leq i < j \leq m} d(x_i, x_j) > 2r \wedge \bigwedge_{1 \leq i \leq m} \psi(x_i) \big), \tag{3.1}$$

for suitable $r, m \geq 1$ and an r-local $\psi(x)$. The minimal r satisfying the above properties is called the locality rank *of φ.*

In other words, for every sentence $\varphi \in \mathrm{FO}$ there is a Boolean formula F and sentences $\chi_1, \ldots, \chi_l$ of the form (3.1) such that $F(\chi_1, \ldots, \chi_l)$ is equivalent to φ.

Recall that $\mathrm{rk}(\varphi)$ denotes the maximal number of quantifier nestings in φ. The original proof of Gaifman yielded $r \leq 7^{\mathrm{rk}(\varphi)-1}$ and $m \leq \mathrm{rk}(\varphi)$.

Gaifman presented an explicit inductive construction of the formula χ. This enables the effective computation of this formula by an algorithm in time, say, $f(\|\varphi\|)$, for some suitable $f : \mathbb{N} \to \mathbb{N}$ (this function is assumed to be non-elementary [Woe00a]).

3.3 The main algorithm

In this final section we will present the main result of this chapter, namely, an algorithm that decides $\mathfrak{A} \models \varphi$ for structures $\mathfrak{A}$ of bounded local tree-width and first-order sentences φ.

Roughly, we first compute the formula χ (from Gaifman's theorem) that w.l.o.g. has the form (3.1). Then, using a tree cover $\mathcal{T}$ of $\mathfrak{A}$, we mark those elements $a \in A$ for which we have $\mathfrak{A} \models \psi(a)$. In a last step we try to find elements $a_1, \dots, a_m \in A$ such that $d^{\mathfrak{A}}(a_i, a_j) > 2r$ for all $i \neq j$. If there are such elements, we accept and reject, otherwise.

We start with two lemmas comprising the main difficulties of the algorithm. Assume that we have a family $\mathcal{T}$ of subsets of A. The first lemma is needed to recognize those elements a of a $T \in \mathcal{T}$ for which $\mathfrak{A} \models \psi(a)$ is equivalent to $\langle T \rangle^{\mathfrak{A}} \models \psi(a)$ ($\psi(x)$ is assumed to be r-local), i.e. the sets of elements $a \in T$ such that $N_r^{\mathfrak{A}}(a) \subseteq T$.

Lemma 44. *Let $\mathfrak{A}$ be a τ-structure and $\mathcal{T}$ a family of subsets of A. Then there exists an algorithm that computes the sets $K^r(T)$ for all $T \in \mathcal{T}$ in time*

$$O(\sum_{t \in \mathcal{T}} \|\langle T \rangle^{\mathfrak{A}}\|).$$

Proof: Recall that $K^r(T) := \{a \in T \mid N_r^{\mathfrak{A}}(a) \subseteq T\}$. Let $\mathcal{G} := \mathcal{G}(\mathfrak{A})$ be the Gaifman graph of $\mathfrak{A}$. We present an algorithm that computes $K^r(T)$ in time $O(\|\langle T \rangle^{\mathfrak{A}}\|)$ for each $T \in \mathcal{T}$.

Sets T of a cover are given as a list $a_1, \dots, a_l$. The pseudo-code for our algorithm is displayed as algorithm 13. We compute $K^r(T)$ iteratively, maintaining a set K that, in the ith pass, contains the set $\{a \in T \mid N_i^{\mathfrak{A}}(a) \subseteq T\}$. We code K by a referenced list (see page 108) what allows containment checks $a \in K$ and removing elements in constant time. For simplicity, we denote the entire data structure by `kernel` and remove elements by a call of `remove`.

Observe that, in contrast to the last usage of a referenced list, we do not require to read out single elements from K. Hence, it would suffice to code K by the two arrays `refc`, `contr` (cf. the definition of referenced lists).

```
1  proc comp_kernel(T, r)
2    kernel := copy(T);  /* working copy of T */
3    ref_init(kernel);  /* initialize referenced array */
4    for i = 1 to r do
5        aux := ∅;
6        for a ∈ kernel do
7            if  a has a neighbor b ∉ kernel
8              then
9                    append(aux, a);
10           fi
11       od
12       for b ∈ aux do remove(kernel, b) od
13   od
14   return(kernel);
15 .
```

Algorithm 13. Compute $K^r(T)$ for some T.

It is quite obvious that function `comp_kernel` works correctly. Observe that the condition in line 7 characterizes the elements of `kernel` that lie on the outermost ring. To check this condition we have to go through the list of vertices adjacent to a, hence this requires constant time for each such edge (the `append` in the subsequent line also takes constant time).

Running time: So let us analyse the running time of the algorithm. Each pass of the outer loop over i needs time $O(|T|)$: in particular we need $O(|E^{\mathfrak{A}} \cap T^2|)$ steps for line 7 plus $O(|T|)$ for line 12. Together this is $O(|T| + |E^{\mathfrak{A}} \cap T^2|) = O(\|\langle T\rangle\|) = O(|T|)$.

Since r is treated as a constant, the overall time need for the subroutine is $O(|T|)$ as claimed above. $\square$

The next lemma allow to determine, whether there is a certain number of elements whose pairwise distance is bigger than a given r.

Lemma 45. *Let $\mathcal{C}$ be a class of τ-structures of bounded local tree-width and $r, m \geq 1$. Then there is an algorithm that, given a structure $\mathfrak{A} \in \mathcal{C}$ and a set $P \subseteq A$, decides if there are $a_1, \ldots, a_m \in P$ such that*

$$d^{\mathfrak{A}}(a_i, a_j) > r \textit{ for } 1 \leq i < j \leq m. \tag{*}$$

Furthermore, this algorithm needs time $O(|A|)$.

```
1  proc find_remotes(𝔄, P, m, r)
2     valid := copy(P); p := 0;
3     ref_init(valid);
4     while valid ≠ ∅ and p < m do
5            p := p + 1; a_p := first(valid);
6            valid := valid \ N_r^𝔄(a_p);
7     od
8     if p = m then
9                     accept
10               else if l = 0 then reject fi
11    fi
12    𝓗 := (N_{2r}^𝔄({a_1, ..., a_p}), P);
13    if  there are pairwise remote a_1, ..., a_m ∈ H
14      then
15             accept;
16       else
17             reject;
18    fi
19 .
```

Algorithm 14. Decide if the are m remote vertices in P.

Proof: Let $f : \mathbb{N} \to \mathbb{N}$ be a function that guarantees that $\mathcal{C}$ is of bounded local tree-width. The decision procedure is displayed as algorithm 14

Correctness: The algorithm essentially involves two phases. In the first phase (lines 2-7) we compute elements $a_1, \ldots, a_p$ such that $d^{\mathfrak{A}}(a_i, a_j) > r$ for some $p \leq m$. If $p = m$ then we are done and the algorithm accepts, if $p = 0$ (this means $P = \emptyset$) we reject. Otherwise ($p < m$), we pass over to the second phase.

When we enter the second phase, we have $P \subseteq N_r^{\mathfrak{A}}(\{a_1, \ldots, a_p\})$, since otherwise we would find another a_{p+1} remote (with distance $> r$) to all preceding a_i. Let $\mathcal{H} := (\langle N_{2r}^{\mathfrak{A}}(\{a_1, \ldots, a_p\}), P)\rangle$. Then for all $b, b' \in P$ we have $d^{\mathfrak{A}}(b, b') \leq r \iff d^{\mathcal{H}}(b, b') \leq r$ (since we included the r-neighborhood of P). Thus, there are $a_1, \ldots, a_m \in P$ satisfying (*) in $\mathfrak{A}$ if and only if they satisfy (*) in $\mathcal{H}$. This proves the correctness of the algorithm.

Running time: The first phase requires $O(|A|)$ steps, since the loop is passed through at most m times and each pass requires at most $O(|A|)$ steps (set minus is done using a referenced list). For the second phase, note that for

B of constant size and $s \geq 1$ we have $\|\langle N_s^{\mathfrak{A}}(B)\rangle\| = O(|N_s^{\mathfrak{A}}(B)|)$ (by bounded local tree-width). Thus $\mathcal{H}$ can be calulated in time $O(\|\langle H\rangle\|) = O(|H|)$.

Furthermore we have $\mathrm{tw}(\mathcal{H}) \leq \mathrm{ltw}^{\mathfrak{A}}(2pr) \leq f(2pr)$, hence the condition in line 13 is equivalent to the satisfaction of the first-order formula

$$\exists x_1, \ldots, x_m \bigwedge_{1 \leq i < j \leq m} d(x_i, x_j) > r \wedge \bigwedge_{i=1}^{m} P x_i$$

in $\mathcal{H}$, and can thus be checked in time $O(|H|)$ by Courcelle's theorem (theorem 9). $\square$

Now we are ready to state and prove the main theorems.

Theorem 46. *Let $\mathcal{C}$ be a class of structures of bounded local tree-width. Then for every k there is an algorithm that, on input $\mathfrak{A} \in \mathcal{C}$ and $\varphi \in \mathrm{FO}$, decides $\mathfrak{A} \models \varphi$ in time*

$$f(\|\varphi\|) \cdot |A|^{1+(1/k)}$$

for some suitable function $f : \mathbb{N} \to \mathbb{N}$.

Proof: Algorithm 15 is claimed to do the job.

Algorithm 15: Deciding $\mathfrak{A} \models \varphi$

Input: Structure $\mathfrak{A} \in \mathcal{C}$, FO-sentence φ
Output: $\mathfrak{A} \models \varphi$?

1. Compute $F(\chi_1, \ldots, \chi_l)$ according to the recipe given by Gaifman's theorem; let r and m denote the values from Gaifman's theorem
2. Compute an $(r+1, 2k(r+1))$ neighborhood-cover $\mathcal{N}'$ of $\mathfrak{A}$
3. Compute an (r, w)-tree cover $\mathcal{N}$ from $\mathcal{N}'$ (for some suitable $w \geq 1$) plus the corresponding induced structures
4. Compute $K^r(N)$ for all $N \in \mathcal{N}$
5. for all $i = 1, \ldots, l$: (ψ_i denotes the local formula in χ_i)
   i. Compute $P_N := \{a \in N \mid N_r^{\mathfrak{A}}(a) \subseteq N, \langle N\rangle \models \psi_i(a)\}$ for all $N \in \mathcal{N}$
   ii. Compute $P := \bigcup_{N \in \mathcal{N}} P_N$
   iii. if there are $a_1, \ldots, a_m \in P$ such that $d^{\mathfrak{A}}(a_i, a_j) > 2r$ for $i \neq j$ then $p_i :=$ true, else $p_i :=$ false
6. if $F(p_1, \ldots, p_l) =$ true then accept, else reject

Correctness: Since $\mathcal{C}$ has bounded local tree-width, the $(r+1, 2k(r+1))$-neighborhood cover $\mathcal{N}$ computed in line 2 is a $(r+1, w)$-tree cover for some suitable $w \geq 1$ (only depending on $\mathcal{C}$ and r). Hence, we can apply lemma 36 to get a (r, w)-tree cover $\mathcal{N}$ with the corresponding induced substructures. The remainder is obviously correct.

Running time: By Gaifman's theorem and the subsequent remark, the first line needs $O(f(\|\varphi\|))$ steps. Line 2 requires time $O(|A|^{1+(1/k)})$ (by lemma 32). Recall that the obtained neighborhood cover has size $\|\mathcal{N}\| = O(|A|^{1+(1/k)})$. By lemma 44, step 4 needs time $O(\|\mathcal{N}\|)$.

l is bounded by a function on $\|\varphi\|$, hence we restrict attention to one cycle of the loop. Step 5i is performed by a call of the algorithm from lemma 12 for all $N \in \mathcal{N}$, each of which requires $O(g(\|\varphi\|) \cdot |N|)$ time. Together, this takes $O(\|\varphi\| \cdot \|\mathcal{N}\|)$ time. Line 5ii takes time $O(|\mathcal{N}|)$ and finally, using lemma 45 we can decide if the sought elements exist in time $O(|A|)$. Altogether, we get the desired time bound. $\square$

If we have a locally tree-decomposable class $\mathcal{C}$, we can do even better.

Theorem 47. *Let $\mathcal{C}$ be a locally tree-decomposable class of structures. Then there exists an algorithm that, on input $\mathfrak{A} \in \mathcal{C}$ and $\varphi \in \mathrm{FO}$, decides $\mathfrak{A} \models \varphi$ in time*

$$f(\|\varphi\|) \cdot \|\mathfrak{A}\|$$

for some suitable function $f : \mathbb{N} \to \mathbb{N}$.

Proof: The algorithm works exactly like in the previous theorem with the only difference that now, instead of computing a neighborhood-cover, we directly compute a tree cover $\mathcal{T}$ and the corresponding induced substructures. By the assumption on $\mathcal{C}$ being locally tree-decomposable this can be done in time $O(|A|)$ (cf. the definition). Since $\mathcal{T}$ has size $O(|A|)$ and the rest of the algorithm works in time $O(\|\mathcal{T}\|)$, we are done. $\square$

Remark: The constants involved in the running time depend on the class $\mathcal{C}$, that is, on a function g such that for $r \geq 1, \mathfrak{A} \in \mathcal{C}$ there is a $(r, g(r))$-tree cover of $\mathfrak{A}$.

3.4 Remarks on special cases

The results of the previous section cover a big variety of different problems. Due to their meta character, there are important drawbacks on the efficiency of the algorithms. These are caused by their design, which followed the guideline of being as universal as possible.

For particular cases, these general algorithms can be adjusted to get better constants. We are going to discuss the restriction of both, the class of admitted structures as well as the admitted formulas.

Recall the algorithm from theorem 47. Essentially, given a structure $\mathfrak{A}$ and an FO-sentence φ, the algorithm proceeds in four steps. First, we compute the Gaifman normal form for φ, which is w.l.o.g. of the form

$$\exists x_1 \ldots \exists x_m \big(\bigwedge_{1 \leq i < j \leq m} d(x_i, x_j) > 2r \wedge \bigwedge_{1 \leq i \leq m} \psi(x_i) \big),$$

where $\psi(x)$ is an r-local formula for some suitable $r \geq 1$ (the locality rank). Then, in the second step, a suitable cover for $\mathfrak{A}$ (this part depends on the admitted class; cf. the definition of locally tree-decomposable) is computed.

Using theorem 21, we compute the set $P := \{a \in A \mid \mathfrak{A} \models \psi(a)\}$ in the third step, and finally, we try to find a constant number of pairwise "remote" elements from P. A closer look shows that each step bears its problems with respect to the running time.

(i) For the computation of the Gaifman normal form, there is no exact bound known, but already easy formulas tend to yield big and complicated local formulas. For more details the reader is referred to the original work of Gaifman in which an inductive construction of the sought formula is presented.

(ii) Well known is the problem of computing a tree decomposition for a structure. As already discussed in the preceding chapter (cf. page 18) no practical algorithm is known for the general case.

(iii) Essentially the same holds for the Courcelle-like evaluation algorithm employed to compute P. But although the worst case analysis yields a lower bound consisting of a tower of height q, if q is the number of quantifier alternations of ψ, it turned out that for most cases the algorithm behaves well. Hence, apart from the problem of getting a tree-decomposition, this part can be considered as practical (or at least more practical than the other parts).

(iv) Since for the last step we again need Courcelle's method, there are the same objections against practicability as in (ii) and (iii).

In the remainder of this section, we exemplarily show how different restrictions affect the algorithms and their running time. In particular, we consider the restricitions to structures of bounded degree, planar structures and existential formulas.

Structures of bounded degree: For this case, the Gaifman part remains the same since we treat arbitrary first-order formula φ. Assume that $\mathfrak{A}$ denotes the input structure. In part (ii) and (iii), we first compute the sets $N_r^{\mathfrak{A}}(a)$ for all $v \in A$ and suitable $r \geq 1$, and then apply Courcelle's algorithm to determine whether or not $\psi(a)$ holds in $\mathfrak{A}$. The application of Courcelle's theorem is possible due to the fact that each $N_r^{\mathfrak{A}}(a)$ has size at most $f(r)$ for some function f, hence has tree-width only depending on r. An alternative to Courcelle's algorithm on tree-width $f(r)$ is the ad-hoc algorithm for first-order logic with l variables (see theorem 1) requiring time $O(f(r)^l)$ with very low constants hidden behind the O-notation (cf. the description of the algorithm on page 13).

Note that this does not necessarily provide a better running time. But it looks more feasible than the general approach. The same can be said about the last step of the algorithm, where we tried to find pairwise remote vertices. Here we similarly adapt the general algorithm to the special case and decide line 13 in algorithm 14 by an exhaustive search in the substructure $\mathcal{H}$ (since the size of $\mathcal{H}$ only depends on r). Another possiblity would be to adjust Dijkstra's algorithm to our setting.

The same ideas led Grohe [Gro00] to the definition of structure classes of *low degree*. We say that a class of structures $\mathcal{C}$ is of low degree, if for every $k \geq 1$ there is an $n_k \geq 1$ such that for all $\mathfrak{A} \in \mathcal{C}$ with $|A| \geq n_k$ we have $\deg(\mathfrak{A}) \leq |A|^{1/k}$. It is easy to see that the above modification of the algorithm also works for this class (observe that tree decompositions do not occur any more in the description of the algorithm). Given a $k \geq 1$, the complexity analysis yields the upper time bound $f(\|\varphi\|, k) \cdot |A|^{1+1/k}$ to decide $\mathfrak{A} \models \varphi$ for some suitable function f.

Planar graphs: For planar graphs there are essentially two places where we can (apparently) speed-up the general algorithm. The first place is where we compute the tree-decomposition of a planar graph of bounded diameter. In [Epp95] an algorithm was described that computes a tree-decomposition of width $O(\mathrm{diam}(\mathcal{G}))$ of a planar graph $\mathcal{G}$ in time $O(|G|)$. This algorithm essentially consists of a depth-first search through $\mathcal{G}$ and is thus quite efficient (in contrast to the algorithm from [Bod96] the constants hidden behind the O-notation are low). Unfortunately, this algorithm does not provide a decomposition of optimal width. Hence it is not clear whether the savings for the computation of the decomposition outweight the additional costs in the third part of the algorithm (caused by the bigger width of the decomposition).

Existential formulas: The restrictions described until now both lacked of an efficient treatment of the first step, the computation of the Gaifman normal form. There is a class of formulas that admit a fast recognition of the locality rank as well as the computation of a local formula. In the rest of the section we restrict our attention to existential first-order formulas, i.e. formulas of the form

$$\varphi := \exists x_1 \dots \exists x_k \psi(x_1, \dots, x_k),$$

such that $\psi(\bar{x})$ is quantifier free. The most restrictive version is to allow only conjunctive queries, that is, positive existential formulas without disjunctions. It is well known that satisfaction of conjunctive queries is essentially the same as finding homomorphic copies of a pattern structure. This problem has first been adressed by Eppstein [Epp95].

Let $\varphi := \exists \bar{x} \psi(\bar{x})$ be a conjunctive query, for convenience, over the vocabulary of graphs (the extension to arbitrary vocabularies is straightforward). Then φ induces a graph $\mathcal{G}_\varphi$ with universe $\{x_1, \dots, x_k\}$ and an edge between x_i and x_j iff Ex_ix_j occurs in $\psi(\bar{x})$. Assuming that $\mathcal{G}_\varphi$ is connected, it is easy to see that $\chi(x_1) := \exists x_2 \dots \exists x_k \psi(\bar{x})$ is k-local around x_1. Hence, we can omit the first step in the above algorithm (finding a local formula). For the second step, all objections made above remain valid. For the third step observe that there exists an efficient algorithm that finds homomorpic copies of a pattern graph in a graph of bounded treewidth [Epp95, Bod98]. This algorithm can easily be adjusted to our setting and runs in time $O(2^k \cdot |A|)$. By definition, the last step boils down to finding a single element of the universe that satisfies $\chi(x_1)$.

Now assume that $\mathcal{G}_\varphi$ is not connected and splits up into the components $C_1, \dots, C_l$. With each component we associate the induced conjunctive query φ_i. We define

$$\varphi_i := \exists C_i \bigwedge_{\text{free}(\lambda) \subseteq C_i} \lambda,$$

where $\exists C_i$ stands for the existiential quantification of all x_i that are in C_i and the λ range over all atoms occuring in φ. Then φ holds in a graph iff all φ_i hold. Observe that this approach can be extended to positive existential formulas (e.g. given in disjunctive normal form), but it does not work in the presence of negative conjuncts.

In his Master Thesis [Woe00b], Wöhrle exhibited a bigger fragment of existential first-order logic that admits an easy treatment with respect to local behaviour. Recall the definition of $\mathcal{G}_\varphi$ for a positive existential formula φ. For existential formulas φ, we define $\mathcal{G}_\varphi$ to be equal to $\mathcal{G}_{\varphi'}$, where φ' results from φ by deleting all negative literals. Then φ is called *guarded*, if for each $\neg Ex_ix_j$ occuring in φ, the variables x_i, x_j are both contained in the same component of $\mathcal{G}_\varphi$. In the thesis he was able to show that the algorithm for positive existential formulas also works for guarded ones.

Conclusion: Are these specialized algorithms really practical? The computation of the Gaifman normal form in the first step not only takes a long time, but also returns a huge formula, which afterwards has to be translated to a finite automaton (including a further blow up). So the only practical way seems to combine the above proposals, e.g. checking guarded existential formulas over planar structures or structures of bounded degree.

Chapter 4

First-order Formulas with free Variables

In the last section we haved showed that we can decide first-order properties over locally tree-decomposable structures in linear time. Naturally, the question turns up, if the techniques applied successfully to the decision problem, can be extended to get good algorithms for the evaluation and counting problems.

Recall the problem: We are given a first-order formula $\varphi(\bar{x})$ with free variables $\bar{x}$ and a structure $\mathfrak{A}$. Our task is to design efficient algorithms that solve the evaluation, witness and counting problem, respectively, i.e. calculate the set $\varphi(\mathfrak{A})$, find a tuple $\bar{a} \in A$ such that $\mathfrak{A} \models \varphi(\bar{a})$, or calculate the number $|\varphi(\mathfrak{A})|$.

We will proceed in three steps. In the first step, we develop a normal form for local formulas $\varphi(\bar{x})$. This normal form partitions the set $\varphi(\bar{a})$ with respect to "distance-patterns" realized by the free variables $\bar{x}$. Second, we include a special form of a tree cover $\mathcal{T}$ for $\mathfrak{A}$ to get an alternative description of $\varphi(\mathfrak{A})$. This characterization describes how local solutions (computed in the cover sets) are assembled to obtain the entire solution, which is accomplished by a combination of the locality properties of first-order logic and the respective properties of neighborhood covers. These ingredients will yield an alternative characterization of $\varphi(\mathfrak{A})$, which will be the starting point for the solution of the three mentioned problems.

The particular solutions will be provided in a third step. While the evaluation and witness case can be solved more or less straightforwardly, the counting case requires a bit more effort.

4.1 A normal form for local formulas

Let $k \geq 1$ and $t \geq 1$. The *t-distance-type* of a k-tuple $\bar{a} \in A$ in a structure $\mathfrak{A}$ is the undirected graph $\epsilon := ([k], E^\epsilon)$, where there is an edge between distinct vertices i and j if and only if $d^{\mathfrak{A}}(a_i, a_j) \leq 2t+1$. We denote this situation by $\bar{a} \models_{\mathfrak{A},t} \epsilon$ and, if the context is clear, we will frequently omit $\mathfrak{A}$ and t.

The satisfaction of t-distance-types can be expressed in first-order logic by the formula

$$\rho_{t,\epsilon}(\bar{x}) := \bigwedge_{(i,j)\in\epsilon, i<j} d(x_i, x_j) \leq 2t+1 \wedge \bigwedge_{(i,j)\notin\epsilon, i<j} d(x_i, x_j) > 2t+1,$$

i.e. we have $\bar{a} \models_{\mathfrak{A},t} \epsilon$ iff $\mathfrak{A} \models \rho_{t,\epsilon}(\bar{a})$. The sets $\{\bar{a} \mid \bar{a} \models \epsilon\}$ of tuples satisfying a certain distance-type ϵ form a partition of A^k.

A formula $\varphi(\bar{x})$ is called t-local if it is equivalent to a formula of the form $\psi^{N_t(\bar{x})}(\bar{x})$ for some $\psi(\bar{x})$. Observe that $\rho_{t,\epsilon}(\bar{x})$ is t-local around $\bar{x}$ (since $d^{\mathfrak{A}}(x,y) \leq 2t+1$ iff $\mathfrak{A} \models \exists z_1 \in N_t(x) \exists z_2 \in N_t(y)\ E z_1 z_2$).

Now, assume that ϵ splits into connected components $\epsilon_1, \dots, \epsilon_p$. We let $\epsilon_{i,\nu}$ stand for the ν-th coordinate of the i-th component (with respect to the natural ordering on the indices). With $\bar{a} \restriction \epsilon_i$ we denote the projection of $\bar{a}$ onto the coordinates contained in ϵ_i.

The restriction to tuples $\bar{a}$ satisfying some distance-type ϵ provides us with important a-priori information about how the t-ball around $\bar{a}$ looks like. The principal feature is that t-neighborhoods of elements corresponding to different components ϵ_i do not *touch*. We say that two subsets $U, V \subseteq A$ do not *touch* in $\mathfrak{A}$ if in the Gaifman graph of $\mathfrak{A}$ there are no edges between U and V, or equivalently, $\langle U \rangle^{\mathcal{G}(\mathfrak{A})} \cup \langle V \rangle^{\mathcal{G}(\mathfrak{A})} = \langle U \cup V \rangle^{\mathcal{G}(\mathfrak{A})}$. The next lemma provides the abovementioned normal form for t-local formulas, which will allow a separate treatment of the variables from different ϵ-components.

Lemma 48. *Given a t-local $\varphi(\bar{x}) \in$ FO for some $t \geq 1$. Then for every distance-type ϵ with connected components $\epsilon_1, \dots, \epsilon_p$ we can find a Boolean combination $F(\bar{\varphi}_1(\bar{x}), \dots, \bar{\varphi}_p(\bar{x}))$ of formulas $\varphi_{i,j}(\bar{x})$, $1 \leq i \leq p$ and $1 \leq j \leq m_i$ such that*

(1) the $\varphi_{i,j}(\bar{x})$ are t-local around their free variables,

(2) for all i, the free variables of $\varphi_{i,j}(\bar{x})$ are among $\bar{x} \restriction \epsilon_i$, and

(3) $\rho_{t,\epsilon}(\bar{x}) \models (\varphi(\bar{x}) \leftrightarrow F(\bar{\varphi}_1(\bar{x}), \dots, \bar{\varphi}_p(\bar{x})))$

To prove this recall the definition of MSO-Ehrenfeucht-Fraisse games defined in section 2.2. If we restrict the MSO game to point moves, we get a game that exactly characterizes FO [EF95]. For a structure $\mathfrak{A}$ and a k-tuple $\bar{a} \in A$ we let the *q-type* $\mathrm{tp}_q(\mathfrak{A}, \bar{a})$ be the set of all formulas $\psi(\bar{x}) \in$ FO of quantifier-rank $\leq q$ such that $\mathfrak{A} \models \psi(\bar{a})$. For two structures $\mathfrak{A}, \mathfrak{B}$ and k-tuples $\bar{a}$, $\bar{b}$ we write $(\mathfrak{A}, \bar{a}) \equiv_q^{\mathrm{FO}} (\mathfrak{B}, \bar{b})$, if $\mathrm{tp}_q(\mathfrak{A}, \bar{a}) = \mathrm{tp}_q(\mathfrak{B}, \bar{b})$. It is well known that the Duplicator wins the q-round FO-EF-game on $(\mathfrak{A}, \bar{a})$ and $(\mathfrak{B}, \bar{b})$ iff $(\mathfrak{A}, \bar{a}) \equiv_q (\mathfrak{B}, \bar{b})$, and that for all q-types $\mathrm{tp}_q(\mathfrak{A}, \bar{a})$ there is a formula $\chi^q_{\mathfrak{A},\bar{a}}(\bar{x})$ of quantifier-rank q such that $\mathfrak{B} \models \chi^q_{\mathfrak{A},\bar{a}}(\bar{b})$ iff $(\mathfrak{A}, \bar{a}) \equiv_q (\mathfrak{B}, \bar{b})$.

Proof: Let $\varphi(\bar{x})$ be a t-local first-order formula of quantifier-rank q. We claim that for a structure $\mathfrak{A}$ and a k-tuple $\bar{a} \in A$ realizing ϵ, the type of $\langle N_t(\bar{a})\rangle$ is already determined by the types of $\langle N_t(\bar{a} \restriction \epsilon_i)\rangle$ for $i = 1, \ldots, p$. Since the latter types are determined by t-local formulas around the variables $\bar{x} \restriction \epsilon_i$, this yields the lemma.

Observe that, if we restrict the EF-game on $(\mathfrak{A}, \bar{a})$ and $(\mathfrak{B}, \bar{b})$ such that both players are only allowed to put pebbles into $N_t^{\mathfrak{A}}(\bar{a})$ and $N_t^{\mathfrak{B}}(\bar{b})$ respectively, then this game characterizes exactly the formulas $\psi(\bar{x})$ which are t-local around $\bar{x}$. Now suppose $(\langle N_t(\bar{a} \restriction \epsilon_i)\rangle, \bar{a} \restriction \epsilon_i) \equiv_q (\langle N_t(\bar{b} \restriction \epsilon_i)\rangle, \bar{b} \restriction \epsilon_i)$ for two tuples $\bar{a}$, $\bar{b}$ and $i = 1, \ldots, p$. We have to show that this implies $(\langle N_t(\bar{a})\rangle, \bar{a}) \equiv_q (\langle N_t(\bar{b})\rangle, \bar{b})$, or equivalently, that the Duplicator has a winning strategy for the q-round EF-game. But this is obvious, since the t-balls around $\bar{a} \restriction \epsilon_i$ do not touch mutually, and therefore the assumed local strategies on the disjoint t-balls extend to a global strategy. Suppose, for instance, that in round $j \leq q$ the Spoiler puts a pebble onto vertex a, with $a \in N_t(\bar{a} \restriction \epsilon_i)$ for a unique $i \leq p$. Then the Duplicator responds with a pebble in $N_t(\bar{b} \restriction \epsilon_i)$ according to his local winning strategy (which exists by assumption). $\square$

For a Boolean formula F we denote with $\mathrm{SAT}(F)$ the set of assignments satisfying F. For a t-local formula $\varphi(\bar{x})$ and a distance-type $\epsilon = \epsilon_1 \cup \ldots \cup \epsilon_p$ denote the formula obtained in lemma 48 by $\varphi^{t,\epsilon}(\bar{x})$ (being of the form $F^{t,\epsilon}(\bar{\varphi}_1(\bar{x}), \ldots, \bar{\varphi}_p(\bar{x}))$ for some Boolean formula $F^{t,\epsilon}$).

Let $\varphi(\bar{x})$ be t-local around $\bar{x}$ and ϵ a t-distance-type. For a Boolean assignment $\alpha_{i,j}$ ($1 \leq i \leq p$ and $1 \leq j \leq m_i$) for $F^{t,\epsilon}$ define

$$\varphi^{t,\epsilon_i}_{\bar{\alpha}_i}(\bar{x}) := \rho_{t,\epsilon_i}(\bar{x}) \wedge \bigwedge_{\nu:\alpha_{i,\nu}=true} \varphi_{i,\nu}(\bar{x}) \wedge \bigwedge_{\nu:\alpha_{i,\nu}=false} \neg\varphi_{i,\nu}(\bar{x}).$$

Furthermore, we define $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\bar{x}) :=$

$$\varphi^{t,\epsilon_1}_{\bar{\alpha}_1}(\bar{x} \restriction \epsilon_1) \wedge \cdots \wedge \varphi^{t,\epsilon_p}_{\bar{\alpha}_p}(\bar{x} \restriction \epsilon_p) \wedge \bigwedge_{1 \leq i < j \leq p} d(\bar{x} \restriction \epsilon_i, \bar{x} \restriction \epsilon_j) > 2t + 1). \tag{4.2}$$

With this notation the next corollary follows directly from the last lemma and easy Boolean transformations:

Corollary 49. *Under the above assumptions, $\varphi(\bar{x})$ is equivalent to*

$$\bigvee_{\epsilon} \bigvee_{\substack{(\bar{\alpha}_1,\ldots,\bar{\alpha}_p) \\ \in \mathrm{SAT}(F^{t,\epsilon})}} \varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\bar{x}).$$

The next lemma is the key to exploit locality of formulas in the presence of a suitable neighborhood-cover. It will be used in the next section in slightly different contexts.

Lemma 50. *Let $\mathfrak{A}$ be a structure, $\varphi(\bar{x})$ be a t-local FO-formula and ϵ a distance-type over the free variables of $\varphi(\bar{x})$. Assume we have a family $U_1, \ldots, U_p$ of subsets of A with the property that $N_t(U_i)$ and $N_t(U_j)$ do not touch for all $1 \le i < j \le p$. Then the set $B := \{\bar{a} \mid \bar{a} \restriction \epsilon_i \in U_i$ and $\mathfrak{A} \models (\varphi \wedge \rho_{t,\epsilon})(\bar{a})\}$ can be characterized as follows:*

$$B = \bigcup_{(\bar{\alpha}_1,\ldots,\bar{\alpha}_p) \in \mathrm{SAT}(F^{t,\epsilon})} \prod_{i=1}^{p} \{\bar{c} \mid \bar{c} \in U_i \text{ and } \langle N_t(U_i)\rangle \models \varphi^{t,\epsilon_i}_{\bar{\alpha}_i}(\bar{c})\}, \tag{4.4}$$

The main feature provided by this lemma is that B is the union of a constant number of disjoint Cartesian products of locally computable sets. This representation will prove very useful later on where we are given a cover of a structure $\mathfrak{A}$ by subsets U_i such that $\langle U_i \rangle$ has bounded treewidth. In particular, this admits local computations to be done in optimal time (cf. for instance algorithm 21).

Proof: Observe that, since $N_t(U_i)$ and $N_t(U_j)$ do not touch, we know that vertices $x \in U_i$ and $y \in U_j$ have distance more than $2t + 1$, hence elements corresponding to different ϵ-components satisfy their claimed remoteness.

Take an $\bar{a} \in B$, hence $\mathfrak{A} \models (\varphi \wedge \rho_{t,\epsilon})(\bar{a})$, which implies $\mathfrak{A} \models F^{t,\epsilon}(\bar{\varphi}_1(\bar{a}), \ldots, \bar{\varphi}_p(\bar{a}))$. We denote the truth-value of $\varphi_{i,j}(\bar{a} \restriction \epsilon_i)$ in $\mathfrak{A}$ by $\alpha_{i,j}$, yielding a satisfying assignment $\bar{\alpha}_1, \ldots, \bar{\alpha}_p$ for $F^{t,\epsilon}$. Knowing that all $\varphi_{i,j}(\bar{x})$ are t-local around $\bar{x} \restriction \epsilon_i$, we get

$$\langle N_t(U_i)\rangle \models \varphi_{i,j}(\bar{a} \restriction \epsilon_i) \text{ iff } \mathfrak{A} \models \varphi_{i,j}(\bar{a} \restriction \epsilon_i).$$

Together with $\mathfrak{A} \models \rho_{t,\epsilon_i}(\bar{a} \restriction \epsilon_i)$, we get $\langle N_t(U_i)\rangle \models \varphi^{t,\epsilon_i}_{\bar{\alpha}_i}(\bar{a} \restriction \epsilon_i)$, hence, by definition, $\bar{a}$ is in the right side of equation (4.4).

Assume that $\bar{a}$ is in the right-side term, witnessed by $\bar{\alpha}_1, \ldots, \bar{\alpha}_p \in \mathrm{SAT}(F^{t,\epsilon})$. Then, by the satisfaction of $\rho_{t,\epsilon_i}(\bar{a} \restriction \epsilon_i)$, all components $\bar{a} \restriction \epsilon_i$ satisfy

ϵ_i, and, by the remark above, also remoteness is fulfilled, hence $\mathfrak{A} \models \rho_{t,\epsilon}(\bar{a})$. By corollary 49 we know that $\varphi(\bar{x}) \wedge \rho_{t,\epsilon}(\bar{x})$ is equivalent to

$$\bigvee_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p \in \mathrm{SAT}(F^{t,\epsilon})} \left(\varphi^{t,\epsilon_1}_{\bar{\alpha}_1}(\bar{x}) \wedge \cdots \wedge \varphi^{t,\epsilon_p}_{\bar{\alpha}_p}(\bar{x})\right) \wedge \bigwedge_{1 \le i < j \le p} d(\bar{x} \restriction \epsilon_i, \bar{x} \restriction \epsilon_j) > 2t+1, \tag{4.5}$$

and this gives that $\mathfrak{A} \models \varphi(\bar{a})$. $\square$

We end this section by gathering what we have achieved so far to characterize the target set $\varphi(\mathfrak{A}) := \{\bar{a} \mid \mathfrak{A} \models \varphi(\bar{x})\}$ for a t-local formula $\varphi(\bar{x})$ and a structure $\mathfrak{A}$. The formula equivalent to $\varphi(\bar{x})$ in the corollary is a disjoint disjunction, hence

$$\varphi(\mathfrak{A}) = \bigcup_{\epsilon} \bigcup_{(\bar{\alpha}_1,\ldots,\bar{\alpha}_p)} \varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\mathfrak{A}), \tag{4.6}$$

where both unions are over disjoint sets. In addition, a formula of the form $\varphi^{t,\epsilon}_{\bar{\alpha}}(\bar{x})$ is a conjunction of formulas with the property that no variable occurs freely in two different conjuncts. Hence, under certain additional conditions (like in the previous lemma), the set of satisfying assignments is a Cartesian product.

Computing the normal form: We close this section showing how to compute the formula $F^{t,\epsilon}$ from lemma 48. Recall that after this lemma, we continued with a formula $\varphi^{t,\epsilon}(\bar{x})$ where $\epsilon = \epsilon_1 \cup \cdots \cup \epsilon_p$ is a distance-type over the variables and $t \ge 1$. Hence, in order to get uniform algorithms, we need a procedure that effectively computes this formula. Assume that we have given a t-local first-order formula $\varphi(\bar{x})$ and a distance-type ϵ. We have to compute a Boolean formula F and a family of formulas $\varphi_{i,j}(\bar{x} \restriction \epsilon_i)$, ($1 \le i \le p$ and $1 \le j \le m_i$) such that:

$$\rho_{t,\epsilon}(\bar{x}) \models \varphi(\bar{x}) \leftrightarrow F(\bar{\varphi}_1(\bar{x} \restriction \epsilon_1), \ldots, \bar{\varphi}_p(\bar{x} \restriction \epsilon_p))$$

where the $\varphi_{i,j}(\bar{x} \restriction \epsilon_i)$ have quantifier rank $\le \mathrm{rk}(\varphi)$. The set of all possible candidates can be computed as follows. Define

$$\Gamma_i := \{\psi(\bar{x} \restriction \epsilon_i) \in \mathrm{FO} \mid \mathrm{rk}(\psi) \le \mathrm{rk}(\varphi)\}.$$

Modulo equivalence, Γ_i is finite, and a *complete* set of such formulas can be computed by an algorithm closing the set of atoms under Boolean combinations and existential quantification (the obtained set may contain equivalent formulas). Then the possible candidates for $\varphi^{t,\epsilon}(\bar{x})$ are exactly the Boolean combinations over $\bigcup_{i=1}^{p} \Gamma_i$. We call this Boolean closure Γ.

Now our problem is to find a $\psi(\bar{x}) \in \Gamma$ such that

$$\rho_{t,\epsilon}(\bar{x}) \models \varphi(\bar{x}) \leftrightarrow \psi(\bar{x}).$$

Remember Gödels completeness theorem for the *sequence calculus*[1]. A formula ψ is said to be *derivable* from φ ($\varphi \vdash \psi$), if there exists a sequence Δ of sequence rules such that $\varphi\Delta;\ \psi$ is a *valid* sequence (validness is a purely syntactic notion).

Theorem 51 (Gödel 1928 [EFT95]). *Let be φ and ψ first-order formulas. Then the following holds:*

$$\varphi \models \psi \quad \Longleftrightarrow \quad \varphi \vdash \psi.$$

This folklore theorem entails that there is an algorithm that enumerates all logical consequences of a given first-order formula φ. This algorithm enumerates all possible sequences Δ and "applies" them to φ, printing out the result. Note that we cannot decide, if a given formula ψ is a consequence of φ. Hence, the ad-hoc algorithm, checking the equivalence above for all $\psi(\bar{x}) \in \Gamma$ does not work.

Nonetheless, because of lemma 48, we know that such a formula $\varphi(\bar{x})$ exists, hence, by the completeness theorem, there must be a Δ such that

$$\Delta;\ \big((\rho_{t,\epsilon} \wedge \varphi)(\bar{x}) \leftrightarrow (\rho_{t,\epsilon} \wedge \psi)(\bar{x})\big)$$

is a valid sequence. Since validness can be decided by a simple syntax check, this offers the following solution to the problem:

> For all sequences Δ, check if $\Delta;\ \big((\rho_{t,\epsilon} \wedge \varphi)(\bar{x}) \leftrightarrow (\rho_{t,\epsilon} \wedge \psi)(\bar{x})\big)$ is a valid sequence for some formula $\psi(\bar{x}) \in \Gamma$. If so, return this $\psi(\bar{x})$.

Obviously, this algorithm is correct. And since we know that such a formula $\psi(\bar{x})$ exists and Γ is finite, we always terminate. This proves the following lemma.

Lemma 52. *There is an algorithm that, given $t \geq 1$, a t-local first-order formula $\varphi(\bar{x})$ and a fitting distance-type ϵ, computes the formula $\varphi^{t,\epsilon}(\bar{x})$.*

[1] Details about the sequence calculus can be found in every textbook about logic, for instance [EFT95].

4.2 Including the tree cover

What we have done so far only concerned the t-local formula $\varphi(\bar{x})$. In this section we bring together the normal form developed in the last section, and the property of the input structures of being locally tree-decomposable. These two ingredients will yield a characterization of $\varphi(\mathfrak{A})$ that only involves local computations.

Unfortunately, we have to strengthen our requirements on tree covers to be able to include the cover adequately. For instance, we need the property that the union of a constant number of cover sets induces a structure of bounded treewidth. This property holds for all example classes given in the previous chapter.

Definition 53. *Let $r, l \geq 1$ and $g : \mathbb{N} \to \mathbb{N}$. A* nice *$(r, l, g)$-tree cover $\mathcal{T}$ of a structure $\mathfrak{A}$ is a $(r, g(1))$-tree cover $\mathcal{T}$ such that*

(1) for each $U \in \mathcal{T}$ there are at most l many $V \in \mathcal{T}$ with $U \cap V \neq \emptyset$

(2) for all $U_1, \ldots, U_q \in \mathcal{T}$, $q \geq 1$ we have

$$tw(\langle U_1 \cup \cdots \cup U_q \rangle^{\mathfrak{A}}) \leq g(q)$$

A class $\mathcal{C}$ of structures is nicely locally tree-decomposable*, if there exists a linear time algorithm that, given an $r \geq 1$ and a structure $\mathfrak{A} \in \mathcal{C}$, computes a nice (r, l, g)-tree cover $\mathcal{T}$ of $\mathfrak{A}$ for some suitable $l \geq 1$ and $g : \mathbb{N} \to \mathbb{N}$.*

It is easy to see that all example classes $\mathcal{C}$, introduced in the last section admit nice tree covers, e.g. for the construction in lemma 37 we can choose $l = 3$ and $g(q) := f((2r+1) \cdot q)$, where f is the function bounding the local treewidth for $\mathcal{C}$. If $\mathcal{C}$ is a class of structures of degree bounded by $d \geq 1$, then choose $g(q) := q \cdot d(d-1)^{r-1}$ (the maximum size of q many r-neighborhoods), and $l = d(d-1)^{r-1}$.

Now adopt the assumptions from the end of the last section and let $\varphi(\bar{x})$ be of the form (4.2), i.e. $\varphi(\bar{x}) := \varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\bar{x})$ for a distance type ϵ and a matching assignment $(\bar{\alpha}_1, \ldots, \bar{\alpha}_p)$. This means that we have a formula $\varphi(\bar{x})$ of the following form:

$$\varphi(\bar{x}) := \bigwedge_{1 \leq i < j \leq p} d(\bar{x} \restriction \epsilon_i, \bar{x} \restriction \epsilon_j) > 2t + 1 \wedge \varphi_1(\bar{x} \restriction \epsilon_1) \wedge \cdots \wedge \varphi_p(\bar{x} \restriction \epsilon_p). \quad (4.7)$$

Furthermore, a nice (r, l, g) tree cover $\mathcal{T} = \{U_1, \ldots, U_m\}$ of the input structure $\mathfrak{A}$ is given, for $r := k(2t+1)$ and appropriate $l \geq 1, g : \mathbb{N} \to \mathbb{N}$.

In the sequel, we state and prove a couple of alternative descriptions of the target set $\varphi(\mathfrak{A})$. We use capital letters for the indices $I \in [m]$ over the tree cover. These indices often occur as tuples, which will be indexed by lower-case letters (e.g. $I_j \in [m]$).

(i) Including the cover

$$\varphi(\mathfrak{A}) = \bigcup_{(J_1,\dots,J_p)\in[m]^p} A_{J_1,\dots,J_p}, \tag{4.8}$$

where

$$A_{J_1,\dots,J_p} := \left\{\bar{a} \;\middle|\; \bar{a}\restriction\epsilon_i \cap K^r(U_{J_i}) \neq \emptyset \text{ and } \langle U_{J_1} \cup \dots \cup U_{J_p}\rangle \models \varphi(\bar{a})\right\}. \tag{4.9}$$

Proof: This follows directly from the definitions. In fact, if $\mathfrak{A} \models \varphi(\bar{a})$, then, by property (1) of tree covers, there are indices $J_1, \dots, J_p$ such that $\bar{a}\restriction\epsilon_i \cap K^r(U_{J_i}) \neq \emptyset$.

For the tree cover $U_1, \dots, U_m$, we have chosen r $(= k(2t+1))$ such that for all $\bar{a}$ with $\bar{a}\restriction\epsilon_i \cap K^r(U_I) \neq \emptyset$ we have $N_t(\bar{a}\restriction\epsilon_i) \subseteq U_I$, hence $N_t(\bar{a}) \subseteq U_{J_1} \cup \dots \cup U_{J_p}$. Therefore, by t-locality

$$\mathfrak{A} \models \varphi(\bar{a}) \text{ iff } \langle U_{J_1} \cup \dots \cup U_{J_p}\rangle \models \varphi(\bar{a}).$$

This gives us the left to right inclusion of equation (4.8).
The other direction is trivial, by the definition of $A_{J_1,\dots,J_p}$. □

(ii) Structuring the cover

The graph $\mathrm{c}(\mathcal{T})$ serves to summarize the relevant topological information of a tree cover $\mathcal{T}$, and is the graph $([m], E^{\mathrm{c}(\mathcal{T})})$ with $E^{\mathrm{c}(\mathcal{T})} := \{(I,J) \mid U_I \cap U_J \neq \emptyset\}$. To partition the set of indices, we introduce the *p-contiguousness-type* of a tuple $(J_1, \dots, J_p)$ w.r.t. $\mathcal{T}$ as the graph $\kappa = ([p], E^\kappa)$ such that $(i,j) \in E^\kappa$ if, and only if $(J_i, J_j) \in E^{\mathrm{c}(\mathcal{T})}$ (we write $(J_1, \dots, J_p) \models \kappa$). By $\mathrm{Mod}^{\mathcal{T}}(\kappa)$ we denote the set of tuples of indices that *realize* κ: $\mathrm{Mod}^{\mathcal{T}}(\kappa) := \{(J_1, \dots, J_p) \mid (J_1, \dots, J_p) \models \kappa\}$.

Convention: We arrange the coordinates of a tuple $(J_1, \dots, J_p)$ realizing κ according to the κ-components they belong to. Tacitly assuming that the ordering remains unchanged, we will write $(\bar{J}_1, \dots, \bar{J}_q)$ for the rearranged tuple; e.g. $A_{\bar{J}_1,\dots,\bar{J}_q}$ refers to the set defined in equation (4.9), although the indices are permuted (actually, this definition depends on κ).

Furthermore, we declare that if we write J_j, this refers to the j'th coordinate of $\bar{J}$, while $J_{i,\nu}$ refers to ν'th coordinate contained in κ_i.

Now assume that we have given a tuple of indices $(J_1, \ldots, J_p) \in [m]^p$ and that this tuple satisfies the p-contiguousness-type κ. The objective of introducing κ is to deliver a priori information about the distances between elements contained in the sets $U_{J_1}, \ldots, U_{J_p}$: take two indices J_i, J_j not adjacent in κ, i.e. $(i,j) \notin E^\kappa$. It is easy to see that

$$d^{\mathfrak{A}}(K^r(U_{J_i}), y) > 2r \text{ for all } y \in K^r(U_{J_j})$$

follows directly from $U_{J_i} \cap U_{J_j} = \emptyset$. Let κ be a p-contiguousness-type with connected components $\kappa_1, \ldots, \kappa_q$. Then $\epsilon(\kappa_i) := \bigcup_{j \in \kappa_i} \epsilon_j$ is the union of the ϵ-components contained in κ_i. By the remark just made, we know that if $\bar{J} \models \kappa$ and J_i, J_j belong to different κ-components, then the distance between elements of their respective kernels is $> 2r$. This fact will be used to get the next characterization of $\varphi(\mathfrak{A})$.

Restricting the definition of $A_{J_1,\ldots,J_p}$ to the indices corresponding to the components $\epsilon(\kappa_i)$ we obtain the following definition:

$$\begin{aligned} A^{\kappa_i}_{\bar{J}_i} := \{\bar{a} \restriction \epsilon(\kappa_i) \mid & \text{ for all } j, \mu \text{ such that } j \text{ is the } \mu\text{'th element from } \kappa_i : \\ & \bar{a} \restriction \epsilon_j \cap K^r(U_{J_{i,\mu}}) \neq \emptyset \text{ and } \mathfrak{A} \models \rho_{t,\epsilon(\kappa_i)}(\bar{a} \restriction \epsilon(\kappa_i)) \wedge \bigwedge_{j \in \kappa_i} \varphi_j(\bar{a} \restriction \epsilon_j)\}. \end{aligned} \quad (4.10)$$

This set can be computed in the substructure of $\mathfrak{A}$ induced by the set $U_{\bar{J}_i} := \bigcup_{\nu=1}^{l} U_{J_{i,\nu}}$, where l denotes the size of the tuple $\bar{J}_i$. As a matter of fact, $A^{\kappa_i}_{\bar{J}_i}$ is the set of tuples satisfying the formula

$$\rho_{t,\epsilon(\kappa_i)}(\bar{x} \restriction \epsilon(\kappa_i)) \wedge \bigwedge_{j \in \kappa_i} \big(\varphi_j(\bar{x} \restriction \epsilon_j) \wedge \bigvee_{\nu \in \epsilon_j} P_\nu x_\nu\big),$$

where the P_ν are new unary relation variables, interpreted by the kernels of the cover sets U_{J_ν}. Observe that, by t-locality, this formula can be evaluated correctly[2] in $\langle U_{\bar{J}_i} \rangle$. The next lemma combines the above remark about remoteness of indices and the definition of $A^{\kappa_i}_{\bar{J}_i}$ with lemma 50. It claims that the set $A_{\bar{J}_1,\ldots,\bar{J}_q}$ is a Cartesian product of the locally computable sets $A^{\kappa_1}_{\bar{J}_1}, \ldots, A^{\kappa_q}_{\bar{J}_q}$.

Lemma 54. *Let be κ a contiguousness-type as above and $(\bar{J}_1, \ldots, \bar{J}_q)$ a tuple of indices realising κ. Then*

$$A_{\bar{J}_1,\ldots,\bar{J}_q} = A^{\kappa_1}_{\bar{J}_1} \times \cdots \times A^{\kappa_q}_{\bar{J}_q}$$

holds, using the above definitions.

[2]Note that, even if some $x_\nu \in \bar{x} \restriction \epsilon_i$ does not occur freely in φ_i, it must occur in the set. This is against the convention made in the preliminaries, but is necessary here.

Here we run into the problem already mentioned above. Since the grouping with respect to the components of κ permutes the coordinates of a tuple $\bar{J}$ this also happens to the Cartesian product in the lemma. Therefore, we state that the Cartesian product $\times$ is just a notation for the operation that takes care of this changes.

Now we rewrite equation (4.8) using the preceding lemma and obtain:

$$\begin{aligned}\varphi(\mathfrak{A}) &= \bigcup_{\kappa} \bigcup_{(\bar{J}_1,\ldots,\bar{J}_q)\models\kappa} A_{\bar{J}_1,\ldots,\bar{J}_q} \\ &= \bigcup_{\kappa} \bigcup_{(\bar{J}_1,\ldots,\bar{J}_q)\models\kappa} A^{\kappa_1}_{\bar{J}_1} \times \cdots \times A^{\kappa_q}_{\bar{J}_q}. \qquad (4.11)\end{aligned}$$

What did we gain with this reformulation? First of all, we grouped the indices into parts that are mutually independent. Thereby, we opened the way up to a classification of what parts are to be computed together and how these have to be assembled afterwards. The crucial point is that dependent parts are nearby neighborhoods, and thus the corresponding indices form small connected subgraphs of $\mathrm{c}(\mathcal{T})$ (and as it will turn out in the next section, there are only linearly many of them).

Proof: (of lemma 54) Let κ be given with components $\kappa_1, \ldots, \kappa_q$ and $(\bar{J}_1, \ldots, \bar{J}_q)$ be a tuple of indices realising κ. First, note that the condition $\bar{a}\restriction\epsilon_j \cap K^r(U_{J_j}) \neq \emptyset$ for all $j \in [p]$ must be satisfied on both sides of the equation, hence we can ignore it.

Thus, we have the following: $\bar{a} \in A_{\bar{J}_1,\ldots,\bar{J}_q} \Leftrightarrow \mathfrak{A} \models \rho_{t,\epsilon}(\bar{a}) \wedge \varphi_1(\bar{a}\restriction\epsilon_1) \wedge \cdots \wedge \varphi_p(\bar{a}\restriction\epsilon_p)$. This directly implies $\bar{a}\restriction\epsilon(\kappa_i) \in A^{\kappa_i}_{\bar{J}_i}$ (equation (4.10)).

To prove the other direction, assume a tuple $\bar{a}$ with $\bar{a}\restriction\epsilon(\kappa_i) \in A^{\kappa_i}_{\bar{J}_i}$, for all $i = 1, \ldots, q$. We show that for j_1, j_2 from different κ-components, remoteness is satisfied, i.e. $d(\bar{a}\restriction\epsilon_{j_1}, \bar{a}\restriction\epsilon_{j_2}) > 2t+1$. This gives $\bar{a} \models \epsilon$, since $\bar{a}\restriction\epsilon(\kappa_i) \models \epsilon(\kappa_i)$, for all i, is explicitly claimed in the definition of $A^{\kappa_i}_{\bar{J}_i}$.

For $j_1 \neq j_2$ there are $\nu_1 \in \epsilon_{j_1}, \nu_2 \in \epsilon_{j_2}$ such that $a_{\nu_1} \in K^r(U_{J_{j_1}})$ and $a_{\nu_2} \in K^r(U_{J_{j_2}})$. Using the definition of kernels (cf. the above remark) this gives us the inequality $d(a_{\nu_1}, a_{\nu_2}) > 2r = 2k(2t+1)$. On the other hand, the connectedness of the ϵ-components yields $\bar{a}\restriction\epsilon_{j_1} \subseteq N_{k(2t+1)}(a_{\nu_1})$, $\bar{a}\restriction\epsilon_{j_2} \subseteq N_{k(2t+1)}(a_{\nu_2})$, which together give us the desired inequality $d(\bar{a}\restriction\epsilon_{j_1}, \bar{a}\restriction\epsilon_{j_2}) > 2t+1$ (since either of the ϵ-components has length less than k). The definition of $A^{\kappa_i}_{\bar{J}_i}$ claims

$$\langle U_{\bar{J}_i}\rangle \models \rho_{t,\epsilon(\kappa_i)}(\bar{a}\restriction\epsilon(\kappa_i)) \wedge \bigwedge_{j\in\kappa_i} \varphi_j(\bar{a}\restriction\epsilon_j),$$

for all $i = 1, \ldots, q$, hence $\mathfrak{A} \models \rho_{t,\epsilon}(\bar{x}) \wedge \varphi_1(\bar{a} \restriction \epsilon_1) \wedge \cdots \wedge \varphi_p(\bar{a} \restriction \epsilon_p)$. □

4.2.1 The contiguousness-graph

Recall the last characterization of our target set $\varphi(\mathfrak{A})$

$$\varphi(\mathfrak{A}) = \bigcup_{\kappa} \bigcup_{(\bar{J}_1,\ldots,\bar{J}_q) \models \kappa} A^{\kappa_1}_{\bar{J}_1} \times \cdots \times A^{\kappa_q}_{\bar{J}_q}. \tag{4.12}$$

The goal of this section is to provide a characterization of the set of admissible indices $(\bar{J}_1, \ldots, \bar{J}_q)$ in the above equation. The graph introduced in the next definition captures the combinatorial properties of this index set.

Definition 55. *Let κ be a p-contiguousness-type with components $\kappa_1, \ldots, \kappa_q$ and $\mathcal{T}$ be a neighborhood cover. The* contiguousness graph $\mathrm{c}(\mathcal{T}, \kappa)$ *of $\mathcal{T}$ with respect to κ has vertex set* $\mathrm{Mod}(\kappa_1) \dot\cup \cdots \dot\cup \mathrm{Mod}(\kappa_q)$ *and edge relation* $E^{\mathrm{c}(\mathcal{T},\kappa)}$ *defined as follows:*

$$(\bar{J}_i, \bar{J}_j) \in E^{\mathrm{c}(\mathcal{T},\kappa)} \textit{ iff there are } \nu, \mu : (J_{i,\nu}, J_{j,\mu}) \in E^{\mathrm{c}(\mathcal{T})}$$

Additionally, we give the vertices corresponding to $\mathrm{Mod}(\kappa_i)$ *the color i, or more formally, we add unary relations $C_1, \ldots, C_q$ with* $C_i^{\mathrm{c}(\mathcal{T},\kappa)} := \mathrm{Mod}(\kappa_i)$.

Remark: It is easy to see that a tuple $(\bar{J}_1, \ldots, \bar{J}_q)$ is admitted in the above union (4.12) for κ if, and only if, $\bar{J}_i \in C_i$ for all i, and $\{\bar{J}_1, \ldots, \bar{J}_q\}$ is independent in $\mathrm{c}(\mathcal{T}, \kappa)$. We call tuples satisfying these two conditions *independent tuples* . Thus our problems translates to the problem of finding independent tuples in a graph. This is, apart from the constraints $\bar{J}_i \in C_i$, a classical NP-complete problem [GJ79] in the general case.

The solution, we will provide, takes advantage of the specific combinatorial structure of $\mathrm{c}(\mathcal{T}, \kappa)$ induced by the restrictions on $\mathcal{T}$. (r, l, g)-tree covers have the crucial property that each $U \in \mathcal{T}$ has at most l contiguous $V \in \mathcal{T}$ (or equivalently: $\mathrm{c}(\mathcal{T})$ has degree bounded by l). The next lemma summarizes the properties of $\mathrm{c}(\mathcal{T}, \kappa)$.

Lemma 56. *Let $\mathcal{T}$ be a nice (r, l, g)-tree cover of $\mathfrak{A}$. Then for all κ:*

(1) $\mathrm{c}(\mathcal{T}, \kappa)$ *can be calculated in time $O(|\mathcal{T}|)$.*

(2) $\mathrm{c}(\mathcal{T}, \kappa)$ *has maximal degree bounded by some function on l, k.*

(3) for all $i = 1, \ldots, q$: $\{U_{\bar{J}_i} \mid \bar{J}_i \in C_i^{\mathrm{c}(\mathcal{T},\kappa)}\}$ is a $(r, k^l, g \circ h_i)$-tree cover of $\mathfrak{A}$, where $h_i(n) := |\kappa_i| \cdot n$.

Proof: Let $\mathcal{T} = \{U_1, \dots, U_m\}$ be an (r, l, g)-tree cover of $\mathfrak{A}$ and κ a contiguousness-type with components $\kappa_1, \dots, \kappa_q$. Denote the size of κ_i by p_i, and recall that $\mathrm{c}(\mathcal{T})$ has maximal degree $\leq l$.

Let $J \in [m]$ and $i \leq q$. We define $\kappa_i(J) := \{\bar{J} \mid \bar{J} \models \kappa_i \text{ and } J_j = J \text{ for some } j\}$, the set of all vertices in $C_i^{\mathrm{c}(\mathcal{T},\kappa)}$ that contain J as a part.

For the proof of (1), we give an algorithm at a quite low level. A non-primitive operation will be the loop over all $\bar{J} \in \kappa_i(J)$ for $J \in [m]$. This loop can be implemented in the following straightforward manner: calculate $\kappa_i(J)$, sort it, and identify the set elements with integers from an initial segment of the naturals. Since $\bar{J} \in \kappa_i(J)$ can be checked in constant time, we do not lose any time (see also the complexity analysis at the end of the proof, where we show that $|\kappa_i(J)| \leq l^{p_i(p_i-1)}$).

```
proc calc_cgraph(T, κ)
  comment: calculate the vertex set
  for i = 1 to q do  /* the colors */
      n := 1;
      for J = 1 to m do
          for J̄ ∈ κ_i(J) do
              vertex[i, n] := J̄;
              comment: calculate the adjacency list
              adj[i, n] := NULL;
              for μ = 1 to p_i do
                  for I ∈ N^{c(T)}(J_μ) do
                      for j = 1 to q do
                          for Ī ∈ κ_j(I) do
                              append(adj[i, n], Ī);
                          od
                      od
                  od
              od
              n := n + 1;
          od
      od
  od
.
```

Algorithm 16. Calculate $\mathrm{c}(\mathcal{T}, \kappa)$

At the end of the algorithm $\mathrm{c}(\mathcal{T}, \kappa)$ is contained in the arrays `vertex` and `adj`. Their semantics is as follows: $\texttt{vertex}[i,j] = \bar{J}$ means that the j'th vertex of color i is $\bar{J}$. $\texttt{adj}[i,j]$ represents a list of the vertices adjacent to the $\bar{J}$.

It is important for the algorithm to work that the arrays `vertex` and `adj` have only one unbounded dimension (the one that corresponds to the index j). The others are bounded, hence the arrays are essentially one dimensional. Observe here that vertices can occur multiple times and the vertices of the graph are not an initial segment of the natural numbers. But with the algorithm described at the end of section A.3 we get a normalized version. The correctness of the algorithm is immediate.

The running time: For the linear time bound, note that $|\kappa_i(J)| \leq (l^{p_i})^{p_i-1} = l^{p_i(p_i-1)}$, for all $J \in [m]$; we simply choose p_i many indices from the $p_1 - 1$-neigborhood (in $\mathrm{c}(\mathcal{T})$) of J. Thus every loop, apart from the loop over J, is constant. Hence the algorithm needs time $O(m)$. Implicitly, this forces $\mathrm{c}(\mathcal{T}, \kappa)$ to have linear size, in fact, $|C_i^{\mathrm{c}(\mathcal{T})}| \leq \sum_{J \in \mathrm{c}(\mathcal{T})} |\kappa_i(J)| \leq m \cdot l^{p_i(p_i-1)} = O(m)$ (this is the maximum value of the variable n).

Finally for the normalization of the graph we need time $O(p \cdot m + \|\mathrm{c}(\mathcal{T}, \kappa)\|)$ which is, by the previous considerations, equal to $O(m)$.

To prove (2) consider the loop from line 10 to line 18. Recall that $N(J)$ is the set of all vertices adjacent to J. In this loop, we generate the adjacency list for each vertex $\bar{J}$, which has size $\leq p_i \cdot l \cdot l^{p_i(p_i-1)}$.

At last, for (3), we have to confirm the conditions of definition 53. Condition (1) of definition 35 transfers immediately to covers with the property that each set in the old cover is contained in a set of the new cover.

To examine the treewidth, fix some i and $\bar{J} \models \kappa_i$. Because $\mathcal{T}$ is a nice (r, l, g)-tree cover, the set $\langle U_{\bar{J}} \rangle$ has treewidth bounded by $g(p_i)$. Building the union over s such sets we get $g(p_i \cdot s)$ as an upper bound on the treewidth.

Furthermore, there is an edge between $\bar{J}$ and $\bar{J}'$ iff $U_{\bar{J}} \cap U_{\bar{J}'} \neq \emptyset$, hence property (1) of definition 53 is satisfied (a bound on the non-empty intersections). □

4.3 The evaluation- and witness problem

In this section we present an algorithm that solves the FO-evaluation problem over nicely tree-decomposable classes $\mathcal{C}$. The running time of the algorithm is linear in the size of the input structure plus the size of the output, hence optimal.

We use two steps to describe the algorithm. First, we show that the reduction described in the first two sections of this chapter is feasible in linear time. In fact, we provide an algorithm in the style of a Turing reduction, where the oracle is assumed to be an algorithm computing $\varphi^{\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A})$ for the input structure $\mathfrak{A}$.

Second, we present a subroutine that fills the gap in the above program (i.e. calculates the sets $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}$ and their union). This completes the algorithm.

4.3.1 The evaluation problem

We can evaluate an FO-formula on nicely tree-decomposable structures in optimal time.

Theorem 57. *Let $\mathcal{C}$ be a class of structures being nicely tree-decomposable. Then there exists a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm that solves the evaluation problem for* FO*-formulas $\varphi(\bar{x})$ on structures $\mathfrak{A} \in \mathcal{C}$ in time*

$$f(\|\varphi\|) \cdot (\|\mathfrak{A}\| + |\varphi(\mathfrak{A})|).$$

The algorithm evaluating FO-queries looks as follows:

Algorithm 17: Computing $\varphi(\mathfrak{A})$

Input: Structure $\mathfrak{A} \in \mathcal{C}$, FO-formula $\varphi(\bar{x})$
Output: $\varphi(\mathfrak{A})$

1. Compute the Boolean combination $\varphi'(\bar{x}) = F(\varphi_1(\bar{x}),\dots,\varphi_{l_1}(\bar{x}),\psi_1,\dots,\psi_{l_2})$ equivalent to $\varphi(\bar{x})$, where all $\varphi_i(\bar{x})$ are t-local around $\bar{x}$ (lemma 48).

2. For all $i = 1,\dots,l_2$ decide, if $\mathfrak{A} \models \psi_i$ and replace ψ_i in $\varphi'(\bar{x})$ with its truth value. Denote the resulting formula with $\varphi''(\bar{x})$.

3. Compute a nice $(k(2t+1), l, g)$-tree cover $\mathcal{T}$ for appropriate $l \geq 1, g : \mathbb{N} \to \mathbb{N}$.

4. For all distance-types $\epsilon = \epsilon_1 \cup \dots \cup \epsilon_p$

    i. calculate $\varphi^{t,\epsilon}(\bar{x}) = F(\bar{\varphi}_1(\bar{x}),\dots,\bar{\varphi}_p(\bar{x}))$ from lemma 48

    ii. For all $(\bar{\alpha}_1,\dots,\bar{\alpha}_p) \in \mathrm{SAT}(F)$, calculate $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A})$

5. Return $\bigcup_\epsilon \bigcup_{\bar{\alpha}} \varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A})$

The first line needs time linear in $\|\varphi'(\bar{x})\|$, which itself only depends on $\|\varphi(\bar{x})\|$ (cf. the end of section 4.1, where we explained how to calculate the normal form from lemma 48).

By definition, a nice $(k(2t+1), l, g)$-tree cover can be computed in time $O(|A|)$, the same holds for the decision of $\mathfrak{A} \models \chi_i$ (by theorem 47). Thus line 2 and line 3 need time linear in the structure size.

The loop in line 4 is the crucial one. First, the number of different ϵ is constant (depends only on k). Calculating $\varphi^{t,\epsilon}(\bar{x})$ needs time only depending on $\|\varphi\|$, and the number of different Boolean assignments $(\bar{\alpha}_1, \ldots, \bar{\alpha}_p)$ which have to be considered is also bounded by a constant. Hence, the subroutine calculating $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\mathfrak{A})$ is called constantly often.

By a result, being proved later on, $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\mathfrak{A})$ can be calculated in time $O(|\varphi(\mathfrak{A})|)$ (actually in time $O(|\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\mathfrak{A})|)$), hence the entire loop is performed in time $O(\sum_\epsilon \sum_{\bar{\alpha}} |\varphi(\mathfrak{A})|) = O(|\varphi(\mathfrak{A})|)$. The last step, finally, outputs the just computed sets.

Enumerating independent tuples: Consider the assumptions in algorithm 17 before the call of the subroutine in line 4ii, which does the main computation. That is, we have given a formula $\varphi(\bar{x})$ of the form $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\bar{x})$ (recall formula 4.7), and a tree cover $\mathcal{T}$ of $\mathfrak{A}$. The procedure `enum_assign` that enumerates all tuples $\bar{a}$ such that $\mathfrak{A} \models \varphi(\bar{a})$ is displayed as algorithm 18.

Correctness: `enum_assign` first calculates $\mathcal{G} := \mathrm{c}(\mathcal{T}, \kappa)$ and then eliminates all vertices $\bar{J} \in C_i^{\mathrm{c}(\mathcal{T},\kappa)}$ with $A^{\kappa_i}_{\bar{J}} = \emptyset$. Note that a Cartesian product is empty if, and only if, one of its parts is empty. This means that we throw away those indices that, in any case, do not contribute to the result. Calling `indep_tuples`$(\mathcal{G})$ (displayed as algorithm 19) then computes the set of independent tuples $\mathcal{I}$ (line 9), which directly provides the set $\bigcup_{(\bar{J}_1,\ldots,\bar{J}_q)\in\mathcal{I}} A^{\kappa_1}_{\bar{J}_1} \times \cdots \times A^{\kappa_q}_{\bar{J}_q}$. This proves the correctness of the algorithm. And as it will turn out, this, performed in a clever way, is enough to obtain an optimal time bound.

Running time: Concerning the running time, we have to go more into details: Note that the outermost loop over κ is passed through a constant number of times. Consider the first part between line 4 and line 8: by definition, $A^{\kappa_i}_{\bar{J}_i} = \emptyset$ can be decided in the structure $\langle U_{\bar{J}_i} \rangle$, which, by the additional property of nice tree covers, has bounded treewidth (by $g(|\kappa_i|)$, cf. lemma 56). Hence, we can apply the algorithm from theorem 23 and decide emptiness in time $f'(\|\varphi\|) \cdot |U_{\bar{J}_i}|$, for some $f' : \mathbb{N} \to \mathbb{N}$. Furthermore, each $a \in A$ is contained in at most l sets U_J, hence $\sum_{\bar{J}_i \in C_i^{\mathcal{G}}} |U_{\bar{J}_i}| \leq l^{|\epsilon(\kappa_i)|} \cdot |A|$

```
1  proc enum_assign(𝒯, φ(x̄))
2    for κ a p-contiguousness-type do
3      𝒢 := c(𝒯, κ);
4      for i = 1 to q do
5        for J̄ ∈ C_i^𝒢 do
6          if A_J̄^{κ_i} = ∅ then G := G \ {J̄} fi
7        od
8      od
9      ℐ := indep_tuples(𝒢);
10     for i = 1 to q do
11       ℐ_i := ℐ ↾ i;
12       for J̄ ∈ ℐ_i do
13         calculate A_J̄^{κ_i};
14       od
15     od
16     B := ∅;
17     for (J̄_1, ..., J̄_q) ∈ ℐ do
18       B := B ∪ A_{J̄_1}^{κ_1} × ··· × A_{J̄_q}^{κ_q};
19     od
20   od
21   return(B);
22 .
```

Algorithm 18. Evaluating a t-local formula

for all $i \in [q]$. Therefore, the running time for this loop is linear in the size of A. From now on, $\mathcal{G}$ denotes the graph after removing these "superflous" vertices.
In line 9, we call the function `indep_tuples`$(\mathcal{G})$ (algorithm 19) to compute the set of independent tuples of $\mathcal{G}$.

Claim: `indep_tuples`$(\mathcal{G})$ returns all independent tuples of $\mathcal{G}$ and needs time $O(|$`indep_tuples`$(\mathcal{G})|)$.

Proof of the Claim: Consider algorithm 19 and assume that $\mathcal{G}$ has maximal degree d (it is an induced subgraph of a contiguousness-graph). In a first loop, we partition the set of indices into "big" and "small" ones (lines 4 to 8). The actual computation proceeds in two steps. In the first step, we look exhaustively for all independent tuples over small indices (note that there are at most $(2dq)^q$ many of them).

```
proc indep_tuples(𝒢)
  ℐ := ∅;
  d := max-degree(𝒢); big = NULL; small = NULL;
  for i = 1 to q do
      if |C_i^𝒢| > 2dq
         then append(big, i) else append(small, i)
      fi
  od
  if small = NULL
    then
          ℐ_s := {()} else
          ℐ_s := all independent v̄ ∈ ∏_{i∈small} C_i^𝒢
  fi
  for v̄ ∈ ℐ_s do
      for ū ∈ ∏_{i∈big} C_i^𝒢
          if (v̄, ū) independent
             then ℐ := ℐ ∪ {(v̄, ū)}
          fi
      od
  od
  return(ℐ);
.
```

Algorithm 19. Calculate all independent tuples

Then, in the second step, we extend these "partial" tuples to tuples over $[q]$, which gives the sought result. The code should be clear.

Let us estimate the time necessary for the second part of the search (the first part trivially needs constant time): for simplicity, we assume that all indices are big; the set of all possible tuples of indices $(\bar{J}_1, \ldots, \bar{J}_q)$ has size $\prod_{i=1}^{q} |C_i^{\mathcal{G}}|$. A lower bound for the number of independent tuples is easily seen to be $|C_1^{\mathcal{G}}| \cdot (|C_2^{\mathcal{G}}| - d) \cdots (|C_q^{\mathcal{G}}| - (q-1)d)$. Since our loop goes through all tuples in $\prod_{i=1}^{q} C_i^{\mathcal{G}}$, it is enough to show that there is a $c > 0$ (actually, we choose $c := 2^q$) such that

$$\prod_{i=1}^{q} |C_i| \leq c \cdot \prod_{i=1}^{q} (|C_i| - dq).$$

To prove this, observe that by $|C_i| > 2dq$ we have $|C_i| \leq 2 \cdot (|C_i| - dq)$.

Applied to the above situation we get

$$\prod_{i=1}^{q} |C_i| \quad \leq \quad 2(|C_1| \; - \; dq) \; \cdot \; \prod_{i=2}^{q} |C_i| \quad \leq \quad \cdots \quad \leq \quad 2^q \prod_{i=1}^{q} (|C_i| \; - \; dq)$$

as an upper bound on the number of times the loop (between lines 14 and 20) is passed. Checking, if a tuple $(\bar{J}_1, \dots, \bar{J}_q)$ is independent can be done in constant time (a simple lookup in q adjacency lists, all of constant size). Together, we need time $\prod_{i=1}^{q} |C_i^{\mathcal{G}}| = O(\prod_{i=1}^{q}(|C_i| - dq))$. $\square$

So let us revert to the analysis of the running time of `enum_assign`. We just computed the set $\mathcal{I}$ of independent tuples of $\mathcal{G} = \mathrm{c}(\mathcal{T}, \kappa)$ in time $O(|\mathcal{I}|)$ (line 9). Because $A_{\bar{J}}^{\kappa_i} \neq \emptyset$, for all $\bar{J} \in C_i^{\mathcal{G}}$, the set $\mathcal{I}$ has at most $\sum_{(\bar{J}_1, \dots, \bar{J}_q) \in \mathcal{I}} |A_{\bar{J}_1}^{\kappa_1} \times \cdots \times A_{\bar{J}_q}^{\kappa_q}| = O(|\varphi(\mathfrak{A})|)$ elements. Hence the loop needs $O(|\varphi(\mathfrak{A})|)$ time.

Now consider the subsequent loop for each i (lines 10 to 15): first, we calculate the projection of $\mathcal{I}$ onto coordinate i. This can be done in $O(|\mathcal{I}|)$ steps. A bit more involved, and actually the reason for the first simplifications is the loop over $\mathcal{I}_i$. Observe that $A_{\bar{J}_i}^{\kappa_i}$ can be calculated in $\langle U_{\bar{J}_i} \rangle$. This substructure has bounded treewidth (recall the first loop), thus we can calculate $A_{\bar{J}_i}^{\kappa_i}$ in time $O(|U_{\bar{J}_i}| + |A_{\bar{J}_i}^{\kappa_i}|)$ (recall theorem 21). Hence, the entire loop requires

$$\sum_{\bar{J}_i \in \mathcal{I}_i} \Big(|U_{\bar{J}_i}| + |A_{\bar{J}_i}^{\kappa_i}| \Big) = O(|A| + \sum_{\bar{J}_i \in \mathcal{I}_i} |A_{\bar{J}_i}^{\kappa_i}|)$$

steps (for the equality, cf. the first loop). It remains to show that $\sum_{\bar{J}_i \in \mathcal{I}_i} |A_{\bar{J}_i}^{\kappa_i}| = O(|\varphi(\mathfrak{A})|)$. For a tuple $\bar{b}$ define $I(\bar{b}) := \{\bar{J}_i \mid \bar{b} \in U_{\bar{J}_i}\}$ the number of times a $\bar{b}$ is computed. Assume $\bar{J}_i \in I(\bar{b})$. This entails for all $\nu : \bar{J}_i \cap \{J \mid b_\nu \in U_J\} \neq \emptyset$, hence

$$|I(\bar{b})| \leq \left(|\kappa_i| \cdot l^{|\epsilon(\kappa_i)|}\right)^{|\kappa_i|}.$$

Thus, each tuple occurring in $\bigcup_{\bar{J}_i \in \mathcal{I}_i} A_{\bar{J}_i}^{\kappa_i}$ is computed at most a constant number times, i.e.

$$\sum_{\bar{J}_i \in \mathcal{I}_i} |A_{\bar{J}_i}^{\kappa_i}| \leq \left(|\kappa_i| \cdot l^{|\epsilon(\kappa_i)|}\right)^{|\kappa_i|} \cdot |\bigcup_{\bar{J}_i \in \mathcal{I}_i} A_{\bar{J}_i}^{\kappa_i}|.$$

The same argument gives us: $\mathcal{I} = O(|\varphi(\mathfrak{A})|)$. Finally, we injectively map each $\bar{b} \in \bigcup_{\bar{J}_i \in \mathcal{I}_i} A_{\bar{J}_i}^{\kappa_i}$ to the lexicographically minimal $\bar{a} \in \varphi(\mathfrak{A})$ such that $\bar{b} = \bar{a} \restriction \epsilon(\kappa_i)$, hence the last term is $O(|\varphi(\mathfrak{A})|)$. Such a minimal $\bar{a}$ exists, since (i) we removed all $\bar{J}_i$ with empty $A^{\kappa_i}_{\bar{J}_i}$ and (ii) in $\mathcal{I}_i$ there are only indices, which are part of an independent tuple (actually, this was the reason for calculating the projections $\mathcal{I}_i$). For else, we would possibly compute "big" sets $A^{\kappa_i}_{\bar{J}_i}$ that do not contribute to $\varphi(\mathfrak{A})$ (because $\bar{J}_i$ is not part of a independent tuple).

To finish, we sum up the times we just estimated. We have $c_1 \cdot |A|$ for the first loop, and $O(|\varphi(\mathfrak{A})|)$ for the second one. As we have seen, the last two loops need time $O(|\varphi(\mathfrak{A})|)$. This amounts to $O(\|\mathfrak{A}\| + |\varphi(\mathfrak{A})|)$, as claimed in the theorem. □

4.3.2 Finding a witnessing tuple

The task of finding a witness for a formula reduces to the task of finding an independent tuple $(\bar{J}_1, \ldots, \bar{J}_q)$ in the contiguousness graph $\mathrm{c}(\mathcal{T}, \kappa)$ for some κ. We describe a pigeonhole-like trick, which immediately provides the algorithm.

Theorem 58. *Let $\mathcal{C}$ be a class of nicely tree-decomposable structures. Then there is a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm that solves the witness problem for* FO*-formulas $\varphi(\bar{x})$ on structures $\mathfrak{A} \in \mathcal{C}$ in time*

$$f(\|\varphi\|) \cdot \|\mathfrak{A}\|.$$

Proof: The algorithm resembles very much the one of the evaluation case. It is displayed as algorithm 20. The analysis of the evaluation case shows that step 1 to step 3 can be done in linear time. For the last step, assume that, given a tree cover $\mathcal{T}$, and a formula of the form $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_q}(\bar{x})$, we find a witness in linear time. Then the entire algorithm works in linear time (again by the already known fact that there are only constantly many different ϵ and corresponding Boolean assignments $\bar{\alpha}$).

Algorithm 20: Computing an $\bar{a} \in \varphi(\mathfrak{A})$

Input: Structure $\mathfrak{A} \in \mathcal{C}$, FO-formula $\varphi(\bar{x})$
Output: an $\bar{a}$ with $\mathfrak{A} \models \varphi(\bar{a})$

1. Compute the Boolean combination $\varphi'(\bar{x}) = F(\varphi_1(\bar{x}), \dots, \varphi_{l_1}(\bar{x}), \psi_1, \dots, \psi_{l_2})$ equivalent to $\varphi(\bar{x})$, where all $\varphi_i(\bar{x})$ are t-local around $\bar{x}$.

2. For all $i = 1, \dots, l_2$ decide, if $\mathfrak{A} \models \psi_i$ and replace ψ_i in $\varphi'(\bar{x})$ with its truth value. Denote the resulting formula with $\varphi''(\bar{x})$.

3. Compute a nice $(k(2t+1), l, g)$-tree cover $\mathcal{T}$ for appropriate $l \geq 1, g : \mathbb{N} \to \mathbb{N}$.

4. For all distance-types $\epsilon = \epsilon_1 \cup \dots \cup \epsilon_p$

    i. calculate $\varphi^{t,\epsilon}(\bar{x}) = F(\bar{\varphi}_1(\bar{x}), \dots, \bar{\varphi}_p(\bar{x}))$ from lemma 48

    ii. For all $(\bar{\alpha}_1, \dots, \bar{\alpha}_p) \in \mathrm{SAT}(F)$, calculate a witness for $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\bar{x})$ in $\mathfrak{A}$

5. return one of the found witnesses, if it exists

So it remains to show that finding a witness for formulas of the special form $\varphi^{t,\epsilon}_{\bar{\alpha}}(\bar{x})$ can be done in linear time. We already saw in the evaluation case that satisfying tuples roughly correspond to independent tuples in the contiguousness graph. Fortunately, in graphs of bounded valence, independent tuples are easy to find.

Lemma 59. *Let $(\mathcal{G}, C_1, \dots, C_q)$ be a q-colored graph with maximal degree d. Then there is an algorithm that finds an independent tuple $(v_1, \dots, v_q) \in \prod_{i=1}^{q} C_i^{\mathcal{G}}$ in linear time.*

Proof: At the beginning, the subroutine partitions the colors into "big" and "small" ones. The point is that small colors have constant size, hence we can try all possibilities to find a partial independent tuple. On the other hand, big colors are sufficiently big that, by any means, there exists an extension of the formerly found partial independent tuple.

The subroutine `indep_tuple` is displayed as algorithm 21. Lines 2 to 8 do the partitioning. Then the routine exhaustively searches for an independent tuple in the subgraph induced by $\bigcup_{i \in \texttt{small}} C_i^{\mathcal{G}}$. Obviously, this requires $\leq O((d \cdot q)^q)$ steps. The following claim is the key to the correctness of the algorithm:

Claim: If $\bar{u}$ is an independent tuple in $\langle \bigcup_{i \in \texttt{small}} C_i^{\mathcal{G}} \rangle$, then line 15 to 23 expand $\bar{u}$ to an independent tuple of $\mathcal{G}$.

Proof of the claim: Observe that $i \leq q$ vertices in $\mathcal{G}$ have together at most $d \cdot i$ neighbors. Of each "big" color there are more than $d \cdot q$ vertices, thus

```
proc indep_tuple(𝒢) 𝒢 is a q-colored graph
  d := max-degree(𝒢); big = NULL; small = NULL;
  for i = 1 to q do
      if |C_i^𝒢| > d · q
        then append(big, i) else append(small, i)
      fi
      if small = NULL then go 16 fi
  od
  found := false;
  for v̄ ∈ ∏_{i∈small} C_i^𝒢 do
      if v̄ is independent in 𝒢 then found := true; break;
      fi
  od
  if found = false then return(NULL) fi
  marked := N^𝒢(v̄);
  for i ∈ big do
      for u_i ∈ C_i^𝒢 do
          if u_i ∉ marked
            then
                  append(marked, N^𝒢(u_i)); break;
          fi
      od
  od
  return((v̄, ū));
.
```

Algorithm 21. Finding an independent tuple

we always find a new vertex not adjacent to all the vertices already in the tuple. In that way we definitely find an independent q-tuple. $\square$

Having proved this claim, assume that there is an independent tuple $(u_1, \ldots, u_q)$ in $\mathcal{G}$. The projection onto small colors is independent in $\bigcup_{i \in \mathtt{small}} C_i^{\mathcal{G}}$, hence the exhaustive search in the algorithm finds an independent tuple $\bar{w}$ (maybe this is another one). By the claim, this tuple is extended to a q-tuple for $\mathcal{G}$, and we are done.

The running time is easily estimated. We partition the colors in $c \cdot q$ steps, the search for the "small" independent tuple needs $c \cdot (d \cdot q)^q$ steps (for some $c \geq 0$), and finally, the expansion proceeds in at most $c \cdot q \cdot (q \cdot d)$ steps. Determining the size of $C_i^{\mathcal{G}}$ depends on the way the relation is given but, in any case, it can be done in linear time. $\square$

We now turn to the routine that finds a witnessing tuple for $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\bar{x})$. This subroutine is displayed as algorithm 22. The correctness of `witness_assign` is obvious.

```
proc witness_assign(𝒯, φ(x̄))
  for κ a p-contiguousness-type do
      𝒢 := c(𝒯, κ);
      for i = 1 to q do
          for J̄ ∈ C_i^𝒢 do
              if A_J̄^{κ_i} = ∅ then G := G \ {J̄} fi
          od
      od
      (J̄_1, ..., J̄_q) := indep_tuple(𝒢);
      if (J̄_1, ..., J̄_q) ≠ NULL
        then
              calculate an ā ∈ ∏_{i=1}^{q} A_{J̄_i}^{κ_i}
              return(ā)
      fi
  od
  return(NULL);
.
```

Algorithm 22. Finding a witness

Concerning the running time, recall that the first loop needs linear time (cf. evaluation case). Then, by a subroutine call, we determine an independent tuple $(\bar{J}_1, \ldots, \bar{J}_q)$ (if one exists!). The last step calculates a representative $\bar{a}_i \in A^{\kappa_i}_{\bar{J}_i}$, for each $i \leq q$. By theorem 22, this requires linear time. Altogether, we stay within linear time. $\Box$

4.4 The counting problem

The last problem remaining open for first-order formulas over nicely tree-decomposable structures is counting the number of satisfying assignments. We proceed in two steps. In the first one, we reduce the counting problem to the calculation of a sum over integers associated with independent sets of a graph. Then, in the last section, we show how this sum can be evaluated quickly.

Remember that, in the context of counting algorithms, we assume that arithmetical operations can be done in constant time. Like before we reduce the question for an FO-formula $\varphi(\bar{x})$ to the case where $\varphi(\bar{x})$ is of the form $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\bar{x})$ for a k-distance-type $\epsilon = \epsilon_1 \cup \dots \cup \epsilon_p$, the locality rank t of $\varphi(\bar{x})$, and a fitting Boolean assignment $(\bar{\alpha}_1, \dots, \bar{\alpha}_p)$. Recall formula (4.6) on page 71:

$$\varphi(\mathfrak{A}) = \bigcup_{\epsilon} \bigcup_{(\bar{\alpha}_1,\dots,\bar{\alpha}_p)} \varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A}). \tag{4.13}$$

This union goes over disjoint sets, hence

$$|\varphi(\mathfrak{A})| = \sum_{\epsilon} \sum_{(\bar{\alpha}_1,\dots,\bar{\alpha}_p)} |\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A})|.$$

This immediately leads to algorithm 4.4 for the counting problem, leaving open how to calculate the values $|\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A})|$ of the double sum. The lines until line 3 are already familiar. After that, we simply cycle over all possible ϵ and fitting $\bar{\alpha}$, and sum up the respective $|\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A})|$.

Algorithm 23: Computing $|\varphi(\mathfrak{A})|$

Input: Structure $\mathfrak{A} \in \mathcal{C}$, FO-formula $\varphi(\bar{x})$
Output: $|\varphi(\mathfrak{A})|$

1. Compute the Boolean combination $\varphi'(\bar{x}) = F(\varphi_1(\bar{x}), \dots, \varphi_{l_1}(\bar{x}), \psi_1, \dots, \psi_{l_2})$ equivalent to $\varphi(\bar{x})$, where all $\varphi_i(\bar{x})$ are t-local around $\bar{x}$.

2. For all $i = 1, \dots, l_2$ decide, if $\mathfrak{A} \models \psi_i$ and replace ψ_i in $\varphi'(\bar{x})$ with its truth value. Denote the resulting formula with $\varphi''(\bar{x})$.

3. Compute a nice $(k(2t+1), l, g)$-tree cover $\mathcal{T}$ for appropriate $l \geq 1, g : \mathbb{N} \to \mathbb{N}$.

4. For all distance-types $\epsilon = \epsilon_1 \cup \dots \cup \epsilon_p$

   i. calculate $\varphi^{t,\epsilon(\bar{x})} = F(\bar{\varphi}_1(\bar{x}), \dots, \bar{\varphi}_p(\bar{x}))$ (cf. 48)

   ii. for all $(\bar{\alpha}_1, \dots, \bar{\alpha}_p) \in \mathrm{SAT}(F)$, calculate $\gamma^{\epsilon}_{\bar{\alpha}} := |\varphi^{t,\epsilon}_{\bar{\alpha}_1,\dots,\bar{\alpha}_p}(\mathfrak{A})|$

5. Return $\sum_{\epsilon} \sum_{\bar{\alpha}} \gamma^{\epsilon}_{\bar{\alpha}}$

Like in the algorithm for the evaluation problem the lines until line 3 need linear time. The subsequent loop goes over a constant number of distance types ϵ and respective Boolean assignments $\bar{\alpha}$. Hence, assuming that $\gamma^{\epsilon}_{\bar{\alpha}}$ can be computed in linear time, we are done. $\square$

This algorithm, together with theorem 67 (in the next subsection), which allows us to evaluate the double sum, yield the intended counting result:

Theorem 60. *Let $\mathcal{C}$ be a class of nicely tree-decomposable structures. Then there is a function $f : \mathbb{N} \to \mathbb{N}$ and an algorithm that solves the counting problem for* FO*-formulas $\varphi(\bar{x})$ on structures $\mathfrak{A} \in \mathcal{C}$ in time*

$$f(\|\varphi\|) \cdot \|\mathfrak{A}\|.$$

4.4.1 Reducing the starting problem

In the sequel, let $\varphi(\bar{x})$ be of the form $\varphi^{t,\epsilon}_{\bar{\alpha}_1,\ldots,\bar{\alpha}_p}(\bar{x})$ for t being the locality rank of φ, a distance type ϵ with components $\epsilon_1, \ldots, \epsilon_p$, together with a suitable Boolean assignment $(\bar{\alpha}_1, \ldots, \bar{\alpha}_p)$. k denotes the number of free variables of $\varphi(\bar{x})$ and $\mathcal{T} = \{U_1, \ldots, U_m\}$ is a nice (r, l, g)-tree cover for $\mathfrak{A}$.

We will develop a procedure that involves two steps: First, we reduce the problem of computing $|\varphi(\mathfrak{A})|$ to computing a sum over the *independent* tuples of a q-colored graph $\mathcal{G}$. Essentially, this sum will look as follows:

$$\gamma = \sum_{\substack{\bar{v} \text{ independent} \\ v_i \in C_i}} \gamma(v_1) \cdots \gamma(v_q),$$

where $\gamma(v)$ is an integer attached to vertex v in the graph $\mathcal{G}$. This reduction is described in the next section, which resembles the evaluation case, but is a bit more complicated. In particular, we give an algorithm that calculates the graph $\mathcal{G}$ (an extension of the contiguousness graph $\mathrm{c}(\mathcal{T}, \kappa)$ from section 4.2) and the labels $\gamma(v)$. The section ends with a procedure that calculates the above sum in linear time, completing the proof of theorem 60.

Let us start with the first part. The following equation characterizes the cardinality of $\varphi(\mathfrak{A})$ and is a consequence of equation (4.8) on page 74 and the principle of inclusion and exclusion (PIE). We define $A_{\mathcal{J}} := \bigcap_{J \in \mathcal{J}} A_J$ for an arbitrary index set $\mathcal{J}$.

$$|\varphi(\mathfrak{A})| \;=\; |\bigcup_{(J_1,\ldots,J_p)\in[m]^p} A_{J_1,\ldots,J_p}| \overset{\text{PIE}}{=} \sum_{\emptyset \neq \mathcal{J} \subseteq [m]^p} (-1)^{|\mathcal{J}|+1} |A_{\mathcal{J}}| \qquad (4.14)$$

It is worthwhile to compare the situation with the one we had in the evaluation case. After including the tree cover into equation (4.8) we grouped the indices $(J_1, \ldots, J_p)$ with respect to their mutual dependence in the computation of $A_{J_1,\ldots,J_p}$ (cf. lemma 50). In the present case this looks more difficult, because now we have sets of tuples, which apparently lost the relevant topological information.

To circumvent this problem, we pass from sets of indices to their projections. More formally, for $\emptyset \neq \mathcal{J} \subseteq [m]^p$ we define $\mathcal{J} \restriction j := \{J_j \mid \text{there is a } (J_1, \ldots, J_j, \ldots, J_p) \in \mathcal{J}\}$, the projection of $\mathcal{J}$ onto coordinate j.

The natural extension of the definition of $A_{J_1,\ldots,J_p}$ (4.9) to sets of indices relates these projections to the sets $A_{\mathcal{J}}$. We set

$$A_{\mathcal{J}_1,\ldots,\mathcal{J}_p} := A_{\mathcal{J}_1 \times \cdots \times \mathcal{J}_p},$$

which receives further justification by the next lemma, stating that Cartesian product and projection are inverse to each other.

Lemma 61. *With the above notation, the following holds for all $\mathcal{J} \subseteq [m]^p$:*

$$A_{\mathcal{J}} = A_{\mathcal{J}\restriction 1,\ldots,\mathcal{J}\restriction p}.$$

Proof: The sets on both sides only contain tuples $\bar{a}$ with $\mathfrak{A} \models \varphi(\bar{a})$, hence it is not necessary to state that explicitly, and we can restrict the argumentation on the requirements imposed by the index sets.

Now, let be $\mathcal{J} \subseteq [m]^p$ and $\bar{a} \in A_{\mathcal{J}}$. By definition, this is equivalent to $\bar{a} \restriction \epsilon_i \cap K^r(U_{J_i}) \neq \emptyset$ for all $i = 1, \ldots, p$ and all $(J_1, \ldots, J_p) \in \mathcal{J}$. The same holds for all $(J_1, \ldots, J_p) \in \mathcal{J} \restriction 1 \times \cdots \times \mathcal{J} \restriction p$. The backward direction is trivial because $\mathcal{J} \subseteq \mathcal{J} \restriction 1 \times \cdots \times \mathcal{J} \restriction p$ implies $A_{\mathcal{J}} \supseteq A_{\mathcal{J}\restriction 1,\ldots,\mathcal{J}\restriction p}$. $\square$

We rewrite equation (4.14) as follows:

$$\begin{aligned}
|\varphi(\mathfrak{A})| &= \sum_{j=1}^{m^p} (-1)^{j+1} \sum_{\mathcal{J} \subseteq [m]^p, |\mathcal{J}| = j} |A_{\mathcal{J}}| \\
&= \sum_{j=1}^{m^p} (-1)^{j+1} \sum_{\mathcal{J}_1,\ldots,\mathcal{J}_p \subseteq [m]} \sum_{\substack{\mathcal{J}, |\mathcal{J}|=j \\ \mathcal{J}\restriction i = \mathcal{J}_i}} |A_{\mathcal{J}}|. \qquad (4.15)
\end{aligned}$$

To examine the last equation fix some $1 \leq j \leq m^p$ and index sets $\mathcal{J}_1, \ldots, \mathcal{J}_p \subseteq [m]$. We claim that the number of addends of the innermost sum (the sum over the sets $\mathcal{J} \subseteq [m]^p$) only depends on the sizes $m_1 := |\mathcal{J}_1|, \ldots, m_p := |\mathcal{J}_p|$ and j. Thereto, take new index sets $\mathcal{J}_i' \subseteq [m]$ with $m_i = |\mathcal{J}_i'|$ for $i = 1, \ldots, p$. By assumption, there are bijections $\beta_i : \mathcal{J}_i \to \mathcal{J}_i'$. These bijections lead to a bijection β from the admitted sets for $\mathcal{J}_1, \ldots, \mathcal{J}_p$ to the admitted sets for $\mathcal{J}_1', \ldots, \mathcal{J}_p'$ by the following definition:

$$\beta : \mathcal{J} \mapsto \{(\beta_1(J_1), \ldots, \beta_p(J_p)) \mid (J_1, \ldots, J_p) \in \mathcal{J}\}.$$

Observe that this mapping preserves the size, i.e. $|\mathcal{J}| = |\beta(\mathcal{J})|$. If we denote the number of sets $\mathcal{J}$ with $|\mathcal{J}| = j$ and $|\mathcal{J} \restriction i| = m_i$ by $\mu_j(m_1, \ldots, m_p)$, i.e.

$$\mu_j(m_1, \ldots, m_p) := |\{\mathcal{J} \subseteq [m]^p \mid |\mathcal{J} \restriction i| = m_i, i = 1, \ldots, p \text{ and } |\mathcal{J}| = j\}|,$$

then we get $|\varphi(\mathfrak{A})| =$

$$\sum_{\mathcal{J}_1,\ldots,\mathcal{J}_p\subseteq[m]} \sum_{j=1}^{|\mathcal{J}_1|\cdots|\mathcal{J}_p|} (-1)^{j+1}\cdot\mu_j(|\mathcal{J}_1|,\ldots,|\mathcal{J}_p|)\cdot|A_{\mathcal{J}_1,\ldots,\mathcal{J}_p}|. \tag{4.16}$$

The next lemma gathers the combinatorial properties important for the evaluation of the formula. Recall that l is the maximal degree of $\mathrm{c}(\mathcal{T})$.

Lemma 62. *Let $\mathcal{J}_1,\ldots,\mathcal{J}_p$ be subsets of $[m]$. If $A_{\mathcal{J}_1,\ldots,\mathcal{J}_p}\neq\emptyset$ then $|\mathcal{J}_i|\leq k\cdot l$ for all $i=1,\ldots,p$. For $j>(k\cdot l)^p$ the function μ_j vanishes, and for $j\leq(k\cdot l)^p$, μ_j is a function from $[k\cdot l]^p$ to $[2^{(k\cdot l)^p}]$.*

Proof: Let $\mathcal{J}_1,\ldots,\mathcal{J}_p$ be given and assume that $|\mathcal{J}_i|>k\cdot l$ for an i. By contradiction, assume further that there is an $\bar{a}\in A_{\mathcal{J}_1,\ldots,\mathcal{J}_p}$, particularly, $\bar{a}\restriction\epsilon_i\cap U_{J_i}\neq\emptyset$ for all $J_i\in\mathcal{J}_i$ and $\mathfrak{A}\models\varphi(\bar{a})$. But each coordinate of $\bar{a}\restriction\epsilon_i$ may be contained in at most l neighborhoods, summing up to at most $|\epsilon_i|\cdot l\leq k\cdot l$ neighborhoods (under the assumption that all coordinates lie in different neighborhoods). This contradicts the assumption $|\mathcal{J}_i|>k\cdot l$.

Assume $|\mathcal{J}|>(k\cdot l)^p$. Then there must exist an i such that $|\mathcal{J}\restriction i|>k\cdot l$, hence $A_{\mathcal{J}}=\emptyset$ (by the first claim).

For the last claim $\mathcal{J}_1,\ldots,\mathcal{J}_p\subseteq[m]$ are given. By the first claim, all $\mathcal{J}_i$ have $\leq k\cdot l$ elements. So fix the cardinalities $m_i:=|\mathcal{J}_i|$. We are looking for the number of sets $\mathcal{J}\subseteq[m]^p$ such that $\mathcal{J}\restriction i=\mathcal{J}_i$, for all $i=1,\ldots,p$. If $(J_1,\ldots,J_p)\in\mathcal{J}$ then $J_i\in\mathcal{J}_i$, hence there are at most $m_1\cdots m_p\leq(k\cdot l)^p$ many such tuples, hence less than $2^{(k\cdot l)^p}$ sets of such tuples. This proves the lemma. $\square$

This lemma implies that (i) the outer sum of (4.16) goes over index sets $\mathcal{J}_i$ of size $\leq k\cdot l$, and (ii) the inner sum only has a constant number of addends. Furthermore, the functions μ_j can be calculated in time bounded by $g(k,l,p)$, for some function g.

In order to continue like in the evaluation case, we partition the set of admissible indices (now sets instead of simple indices) into disjoint parts, according to their mutual dependence (i.e. which ones are pairwise far away, and which ones are not).

Let us introduce the contiguousness-type of a tuple $(\mathcal{J}_1,\ldots,\mathcal{J}_p)\in\mathrm{Pow}([m])^p$ w.r.t $\mathcal{T}$. Recall the definition of $\mathrm{c}(\mathcal{T})$ on page 74. We define $\tilde{\mathrm{c}}(\mathcal{T})$ to be the graph $(\mathrm{Pow}^{\leq kl}([m])\setminus\{\emptyset\},E^{\tilde{\mathrm{c}}(\mathcal{T})})$ where

$$(\mathcal{I},\mathcal{J})\in E^{\tilde{\mathrm{c}}(\mathcal{T})} \text{ iff for all } I\in\mathcal{I} \text{ and } J\in\mathcal{J} : (I,J)\in E^{\mathrm{c}(\mathcal{T})}.$$

The *p-contiguousness-type* of a tuple $(\mathcal{J}_1, \dots, \mathcal{J}_p) \in \mathrm{Pow}([m])^p$ w.r.t. $\mathcal{T}$ is the graph $\kappa := ([p], E^\kappa)$, such that $(i,j) \in E^\kappa$ iff $(\mathcal{J}_i, \mathcal{J}_j) \in E^{\tilde{\mathfrak{c}}(\mathcal{T})}$ (again, we write $(\mathcal{J}_1, \dots, \mathcal{J}_p) \models \kappa$ in this situation).

Let κ be a contiguousness-type with components $\kappa_1, \dots, \kappa_q$. Like before, we group tuples $(\mathcal{J}_1, \dots, \mathcal{J}_p)$ according to the κ-components the coordinates belong to (cf. the evaluation case). With the easy fact that p-contiguousness-types κ partition the set of index tuples we get

$$|\varphi(\mathfrak{A})| = \sum_{\kappa} \sum_{(\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q) \models \kappa} \sum_{j=1}^{(kl)^p} (-1)^{j+1} \cdot \mu_j(|\mathcal{J}_1|, \dots, |\mathcal{J}_p|) \cdot |A_{\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q}|.$$

We fix the arguments of the μ_j, pull the sum over the tuples $(\mathcal{J}_1, \dots, \mathcal{J}_p)$ inwards and obtain

$$= \sum_{j=1}^{(kl)^p} (-1)^{j+1} \cdot \sum_{\bar{m} \in [kl]^p} \mu_j(\bar{m}) \cdot \sum_{\kappa} \sum_{(\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q) \models \kappa, |\mathcal{J}_\nu| = m_\nu} |A_{\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q}|. \tag{4.17}$$

Our intention is to replace global computations (of $A_{\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q}$) by local ones. For that, we restrict the definition of $A_{\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q}$ to the indices that correspond to a connected component κ_i of κ. In particular, we define

$$\begin{aligned} A^{\kappa_i}_{\bar{\mathcal{J}}_i} := \{\bar{a} \restriction \epsilon(\kappa_i) \mid & \text{ for all } j, \mu \text{ s.t. } j \text{ is the } \mu\text{'th element from } \kappa_i \\ & \text{we have } \bar{a} \restriction \epsilon_j \cap K^r(U_J) \neq \emptyset \text{ for all } J \in \mathcal{J}_{i,\mu} \text{ and} \\ & \mathfrak{A} \models \rho_{t, \epsilon(\kappa_i)}(\bar{a} \restriction \epsilon(\kappa_i)) \wedge \bigwedge_{j \in \kappa_i} \varphi_j(\bar{a} \restriction \epsilon_j)\}. \end{aligned} \tag{4.18}$$

Observe that the definition of these sets conform with the sets $A^{\kappa_i}_{\bar{J}_i}$ (cf. equation (4.10)) and their extension to sets of indices, i.e. we have

$$A^{\kappa_i}_{\bar{\mathcal{J}}_i} = \bigcap_{\bar{J}_i \in \mathcal{J}_{i,1} \times \dots \times \mathcal{J}_{i,p_i}} A^{\kappa_i}_{\bar{J}_i} = A^{\kappa_i}_{\mathcal{J}_{i,1} \times \dots \times \mathcal{J}_{i,p_i}}.$$

This property justifies the shortcut $\bar{J}_i \in \bar{\mathcal{J}}_i$ which stands for $\bar{J}_i \in \mathcal{J}_{i,1} \times \dots \times \mathcal{J}_{i,p_i}$, where p_i is the size of the component κ_i. With this terminology, we can state the next lemma, which essentially is an extension of lemma 54 and allows us to substitute $A_{\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q}$ by a Cartesian product.

Lemma 63. *Let κ be a contiguousness-type with components $\kappa_1, \dots, \kappa_q$ and $(\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q) \in \mathrm{Pow}([m])^p$ be a tuple realising κ. Then*

$$A_{\bar{\mathcal{J}}_1, \dots, \bar{\mathcal{J}}_q} = A^{\kappa_1}_{\bar{\mathcal{J}}_1} \times \dots \times A^{\kappa_q}_{\bar{\mathcal{J}}_q}$$

holds.

Proof: To prove the lemma we reduce the problem to the case where we have indices instead of sets of indices (as in lemma 54). Thereto recall that, by definition, we have

$$A_{\bar{\mathcal{J}}_1,\ldots,\bar{\mathcal{J}}_q} := A_{\bar{\mathcal{J}}_1\times\cdots\times\bar{\mathcal{J}}_q} = \bigcap_{\bar{J}_1,\ldots,\bar{J}_q\in\bar{\mathcal{J}}_1\times\cdots\times\bar{\mathcal{J}}_q} A_{\bar{J}_1,\ldots,\bar{J}_q}.$$

For arbitrary $(\bar{J}_1,\ldots,\bar{J}_q) \in \bar{\mathcal{J}}_1 \times\cdots\times \bar{\mathcal{J}}_q$ we have $(\bar{J}_1,\ldots,\bar{J}_q) \models \kappa$ (by definition). For such tuples, lemma 54 yields $A_{\bar{J}_1,\ldots,\bar{J}_q} = A^{\kappa_1}_{\bar{J}_1}\times\cdots\times A^{\kappa_q}_{\bar{J}_q}$, hence

$$A_{\bar{\mathcal{J}}_1,\ldots,\bar{\mathcal{J}}_q} = \bigcap_{(\bar{J}_1,\ldots,\bar{J}_q)\in\bar{\mathcal{J}}_1\times\cdots\times\bar{\mathcal{J}}_q} A^{\kappa_1}_{\bar{J}_1}\times\cdots\times A^{\kappa_q}_{\bar{J}_q} = \bigcap_{\bar{J}_1\in\bar{\mathcal{J}}_1} A^{\kappa_1}_{\bar{J}_1}\times\cdots\times\bigcap_{\bar{J}_q\in\bar{\mathcal{J}}_q} A^{\kappa_q}_{\bar{J}_q},$$

which is equal to $A^{\kappa_1}_{\bar{\mathcal{J}}_1}\times\cdots\times A^{\kappa_q}_{\bar{\mathcal{J}}_q}$. □

In the sequel, we treat sets of the form $A^{\kappa_i}_{\bar{\mathcal{J}}_i}$. These sets are empty, if at least for one of the indices $\mathcal{J}_{i,\nu}$ we have: $U_{\mathcal{J}_{i,\nu}} = \emptyset$. Since these sets do not contribute to the result, we omit them, i.e. if $(\bar{\mathcal{J}}_1,\ldots,\bar{\mathcal{J}}_q) \models \kappa$, we implicitly assume that $U_{\mathcal{J}_i} \neq \emptyset$ for $i = 1,\ldots,p$. Later, we will see that this restriction has crucial impact on the tractability of $\tilde{\mathrm{c}}(\mathcal{T})$.

With the above lemma, equation (4.17) can be transformed to

$$= \sum_{j}^{(kl)^p} (-1)^{j+1} \sum_{\bar{m}\in[kl]^p} \mu_j(\bar{m}) \cdot \sum_{\kappa} \sum_{(\bar{\mathcal{J}}_1,\ldots,\bar{\mathcal{J}}_q)\models\kappa,|\mathcal{J}_\nu|=m_\nu} |A^{\kappa_1}_{\bar{\mathcal{J}}_1}|\cdots|A^{\kappa_q}_{\bar{\mathcal{J}}_q}|. \tag{4.19}$$

Since the outer sums have a constant number of addends and μ_j can be calculated easily, this characterization of $|\varphi(\mathfrak{A})|$ gives us immediatly a reduction of the starting problem to the calculation of

$$\gamma(\bar{m},\kappa) := \sum_{(\bar{\mathcal{J}}_1,\ldots,\bar{\mathcal{J}}_q)\models\kappa,|\mathcal{J}_\nu|=m_\nu} |A^{\kappa_1}_{\bar{\mathcal{J}}_1}|\cdots|A^{\kappa_q}_{\bar{\mathcal{J}}_q}|. \tag{4.20}$$

4.4.2 Counting

In this last section, we show how to calculate the value

$$\gamma(\bar{m},\kappa) := \sum_{(\bar{\mathcal{J}}_1,\ldots,\bar{\mathcal{J}}_q)\models\kappa,|\mathcal{J}_\nu|=m_\nu} |A^{\kappa_1}_{\bar{\mathcal{J}}_1}|\cdots|A^{\kappa_q}_{\bar{\mathcal{J}}_q}| \tag{4.21}$$

in linear time. Therefore, like in the evaluation case, we translate this question to a graph theoretic one. What follows is a slight variation of section

4.2.1. Let $\mathcal{T} = \{U_1, \ldots, U_m\}$ be a nice (r, l, g)-tree cover. For a contiguousness graph κ and $\bar{m} \in [k \cdot l]^p$ define $\mathrm{PMod}^{\mathcal{T}}(\kappa, \bar{m}) := \{\bar{\mathcal{J}} \mid \bar{\mathcal{J}} \models \kappa, U_{\mathcal{J}_i} \neq \emptyset$ and $|\mathcal{J}_i| = m_i\}$. This is the obvious extension of $\mathrm{Mod}^{\mathcal{T}}(\kappa)$ to the present case.

Definition 64. *Let κ be a p-contiguousness-type with components $\kappa_1, \ldots, \kappa_q$ and $(\bar{m}_1, \ldots \bar{m}_q) \in [k \cdot l]^p$. The* contiguousness graph *$\tilde{\mathrm{c}}(\mathcal{T}, \kappa, \bar{m})$ of $\mathcal{T}$ with respect to κ and $\bar{m}$ has vertex set $\mathrm{PMod}^{\mathcal{T}}(\kappa_1, \bar{m}_1) \dot\cup \cdots \dot\cup \mathrm{PMod}^{\mathcal{T}}(\kappa_q, \bar{m}_q)$ and edge relation $E^{\tilde{\mathrm{c}}(\mathcal{T},\kappa,\bar{m})}$ defined as follows:*

$$(\bar{\mathcal{J}}_i, \bar{\mathcal{J}}_j) \in E^{\tilde{\mathrm{c}}(\mathcal{T},\kappa,\bar{m})} \quad \textit{iff} \quad \textit{there are } \nu, \mu : (\mathcal{J}_{i,\nu}, \mathcal{J}_{j,\mu}) \in E^{\tilde{\mathrm{c}}(\mathcal{T})}.$$

Additionally, we give the vertices corresponding to $\mathrm{PMod}^{\mathcal{T}}(\kappa_i, \bar{m}_i)$ the color i or, more formally, we add unary relations $C_1, \ldots, C_q$ with $C_i^{\tilde{\mathrm{c}}(\mathcal{T},\kappa,\bar{m})} := \mathrm{PMod}^{\mathcal{T}}(\kappa_i, \bar{m}_i)$.

The next lemma corresponds to lemma 56 and gathers the relevant properties of contiguousness graphs $\tilde{\mathrm{c}}(\mathcal{T}, \kappa, \bar{m})$ for a tree cover $\mathcal{T}$.

Lemma 65. *Let $\mathcal{T}$ be an (r, l, g)-tree cover of a structure $\mathfrak{A}$. Then for all κ and $\bar{m} \in [k \cdot l]^p$:*

(1) $\tilde{\mathrm{c}}(\mathcal{T}, \kappa, \bar{m})$ can be calculated in time $O(|\mathcal{T}|)$.

(2) $\tilde{\mathrm{c}}(\mathcal{T}, \kappa, \bar{m})$ has maximal degree bounded by some function on l.

Proof: To prove (1) we present an algorithm computing $\tilde{\mathrm{c}}(\mathcal{T}, \kappa, \bar{m})$. In the first step we compute the graph $\tilde{\mathrm{c}}(\mathcal{T})$ without the unnecessary vertices $\mathcal{J}$ (those for which $U_{\mathcal{J}} = \emptyset$). This procedure is displayed as algorithm 24. For the correctness, note that if for $\mathcal{J} \subseteq [m]$ we have $U_{\mathcal{J}} \neq \emptyset$, then there is a $J \in \mathcal{J}$ with $\mathcal{J} \subseteq N^{\mathrm{c}(\mathcal{T})}(J)$. Hence, we really get all relevant vertices of $\tilde{\mathrm{c}}(\mathcal{T})$. For the loop which generates the adjacency lists, note further that $\mathcal{I} \times \mathcal{J} \subseteq E^{\mathrm{c}(\mathcal{T})}$ coincides with the definition of $E^{\tilde{\mathrm{c}}(\mathcal{T})}$. Furthermore, any two adjacent $\mathcal{I}, \mathcal{J}$ there must be $I \in \mathcal{I}$ and $J \in \mathcal{J}$ such that I and J are adjacent in $\mathrm{c}(\mathcal{T})$.

It is easy to see that the running time is bounded by $m \cdot 2^l \cdot l \cdot 2^l \cdot c$ where c is the time necessary to check $\mathcal{I} \times \mathcal{J} \subseteq E^{\mathrm{c}(\mathcal{T})}$ in a graph of bounded degree, given in adjacency list representation (this constant is independent from m).

To continue, recall algorithm 16, which calculates the contiguousness graph $\mathrm{c}(\mathcal{T}, \kappa)$, and the definition of $\kappa_i(J)$ for $J \in [m]$ and $1 \leq i \leq q$. This definition is naturally extended to

$$\kappa_i(\mathcal{J}, \bar{m}) := \{\bar{\mathcal{J}} \in \mathrm{PMod}^{\mathcal{T}}(\kappa_i, \bar{m}) \mid \mathcal{J} = \mathcal{J}_j \text{ for some } j\}.$$

```
proc calc_pgraph(\mathcal{T})
  for J = 1 to m do
      i := 1;
      for \mathcal{I} \subseteq N^{c(\mathcal{T})}(J) do
          vertex[i] := \mathcal{I}; adj[i] := NULL;
          /* add adjacencies */
          for I \in N^{c(\mathcal{T})}(J) do
              for \mathcal{J} \subseteq N^{c(\mathcal{T})}(I) do
                  if \mathcal{I} \times \mathcal{J} \subseteq E^{c(\mathcal{T})}
                    then
                          append(adj[i], \mathcal{J});
                  fi
              od
          od
          i := i + 1;
      od
  od
  normalize the graph (vertex[\cdot], adj[\cdot]);
.
```

Algorithm 24. Calculate $\tilde{c}(\mathcal{T})$

Algorithm 25 essentially works like algorithm 16, with roles of $\tilde{c}(\mathcal{T})$ and $c(\mathcal{T})$ interchanged.

Correctness is immediate by the definition of the $\tilde{c}(\mathcal{T}, \kappa, \bar{m})$. For example, there is an edge between $\bar{\mathcal{I}}$ and $\bar{\mathcal{J}}$ iff there are μ,ν such that $(\mathcal{I}_\nu, \mathcal{J}_\mu) \in E^{\tilde{c}(\mathcal{T})}$. The μ is delivered by the corresponding for-loop, and ν is implicit in the expression $\bar{\mathcal{I}} \in \kappa_l(\mathcal{I}, \bar{m}_l)$ (the ν such that $\mathcal{I} = \mathcal{I}_\nu$). Finally $\mathcal{I} \in N^{\tilde{c}(\mathcal{T})}(\mathcal{J}_\mu)$ assures that the tuples are adjacent.

For the running time, recall that $\bar{\mathcal{J}} \in \kappa_i(\mathcal{J}, \bar{m}_i)$ can be checked in constant time (cf. the case of primitive vertices). Thus, all loops except from the one cycling over $\mathcal{J} \in \tilde{c}(\mathcal{T})$ are bounded by a constant of the form $f(q, l)$ for a function $f : \mathbb{N}^2 \to \mathbb{N}$. Together, we get the running time dominated by $f(q, l) \cdot 2^l \cdot m$.

To prove (2), note that $\tilde{c}(\mathcal{T})$ has maximum degree 2^l. Thus, $|\kappa_i(\mathcal{J}, \bar{m}_i)| \leq ((2^l)^{p_i})^{p_i - 1}$ for a vertex $\mathcal{J} \in \tilde{c}(\mathcal{T})$. Thus, summing up over all loops that compute the adjacency list, we get $p_i \cdot q \cdot 2^l \cdot ((2^l)^p)^{p-1}$ as an upper bound on the degree (the maximal number of different $\bar{\mathcal{I}}$ the algorithm can choose in the loops between line 10 and line 18). $\square$

Finally, by the following lemma, we can assume the values $|A^{\kappa_i}_{\bar{\mathcal{J}}_i}|$ being

```
proc calc_cgraph($\mathcal{T}, \kappa$)
  comment: calculate the vertex set
  for $i = 1$ to $q$ do  /* the colors */
      $j := 1$;
      for $\mathcal{J} \in \tilde{c}(\mathcal{T})$ do
          for $\bar{\mathcal{J}} \in \kappa_i(\mathcal{J}, \bar{m}_i)$ do
              vertex$[i, j] := \bar{\mathcal{J}}$;
              comment: calculate the adjacency list
              adj$[i, j]$ := NULL;
              for $\mu = 1$ to $p_i$ do
                  for $l = 1$ to $q$ do
                      for $\mathcal{I} \in N^{\tilde{c}(\mathcal{T})}(\mathcal{J}_\mu)$ do
                          for $\bar{\mathcal{I}} \in \kappa_l(\mathcal{I}, \bar{m}_l)$ do
                              append(adj$[i, j], \bar{\mathcal{I}}$);
                          od
                      od
                  od
              od
              $j := j + 1$;
          od
      od
  od
  normalize the graph (vertex$[\cdot, \cdot]$, adj$[\cdot, \cdot]$);
.
```

Algorithm 25. Calculate $\tilde{c}(\mathcal{T}, \kappa, \bar{m})$

known. Observe that only here we really need that our tree covers are nice.

Lemma 66. *The values $|A^{\kappa_i}_{\bar{\mathcal{J}}_i}|$ for $i = 1, \ldots, q$ and $\bar{\mathcal{J}}_i \in C^{\tilde{c}(\mathcal{T},\kappa,\bar{m})}$ can be computed in linear time.*

Proof: We compute the $|A^{\kappa_i}_{\bar{\mathcal{J}}_i}|$ for all $\bar{\mathcal{J}}_i \in C_i$ and fixed $1 \le i \le q$ straightforwardly by a simple loop over $\bar{\mathcal{J}}_i$ and calls of the procedure from theorem 23. For that, observe that $|A^{\kappa_i}_{\bar{\mathcal{J}}_i}|$ can be computed in the structure $\langle U_{\bar{J}} \rangle^{\mathfrak{A}}$ for an index $\bar{J}$ with $J_\nu \in \mathcal{J}_{i,\nu}$, for all ν. Furthermore, for an index $\bar{J}$ there are at most $(2^l)^{|\kappa_i|}$ many $\bar{\mathcal{J}}_i$ satisfying this condition. Like before, since each index $\bar{J}$ is used constantly often, also each $a \in A$ is contained constantly often in a set $\langle U_{\bar{J}} \rangle$, hence the entire computation needs linear time. $\square$

Now it is easy to see that we are done, if we are able to evaluate the formula (4.21) in linear time. So fix an $\bar{m} \in [k \cdot l]^q$ and a contiguousness-type

κ with components $\kappa_1, \ldots, \kappa_q$. Define $\mathcal{G} := \tilde{\mathrm{c}}(\mathcal{T}, \kappa, \bar{m})$ and set $\gamma(\bar{\mathcal{J}}_i) := |A^{\kappa_i}_{\bar{\mathcal{J}}_i}|$ for $\bar{\mathcal{J}}_i \in C_i^{\mathcal{G}} := \mathrm{PMod}^{\mathcal{T}}(\kappa_i, \bar{m}_i)$. Then the tuples $(\bar{\mathcal{J}}_1, \ldots, \bar{\mathcal{J}}_q) \in C_1^{\mathcal{G}} \times \cdots \times C_q^{\mathcal{G}}$ admitted in the sum are exactly the ones that are independent in $\mathcal{G}$.

By lemma 65 the q-colored graph $(\mathcal{G}, C_1, \ldots, C_q)$ has degree bounded by, say, $d \geq 1$. The next theorem fills the gap still missing in the proof of theorem 60

Theorem 67. *There is an algorithm that, given $\mathcal{G}$ as above, calculates the value*

$$\sigma(\mathcal{G}, \gamma) := \sum_{(v_1,\ldots,v_q) \text{ independent}} \gamma(v_1) \cdots \gamma(v_q) \tag{4.22}$$

in time $O(\|\mathcal{G}\|)$.

If we plug in this algorithm into the gap left open in algorithm 4.4, then the counting problem for FO over classes that admit nice tree covers is solved. We assume the uniform cost measure for arithmetic operations on integers. To attack theorem 67 we need some new definitions.

Let $\mathcal{G}$ and a labelling γ be given as above. We group the tuples $\bar{v}$ according to the subgraph they induce in $\mathcal{G}$. Fix a graph π over the set $[q]$ and define the subsets

$$\mathrm{Hom}(\pi) := \{\bar{v} \mid (i,j) \in E^{\pi} \;\Rightarrow (v_i, v_j) \in E^{\mathcal{G}}\}$$

and

$$\mathrm{Iso}(\pi) := \{\bar{v} \mid (i,j) \in E^{\pi} \;\Leftrightarrow (v_i, v_j) \in E^{\mathcal{G}}\}$$

of $\prod_{i=1}^{q} C_i^{\mathcal{G}}$. It is easy to see that $\bar{v} \in \mathrm{Iso}(\pi)$ iff $\bar{v} \in \mathrm{Hom}(\pi)$ and for all $\pi' \supsetneq \pi$: $\bar{v} \notin \mathrm{Hom}(\pi')$. Using these index sets, we define $c_\pi := \sum_{\bar{v} \in \mathrm{Hom}(\pi)} \prod_{i \in [q]} \gamma(v_i)$ and $d_\pi := \sum_{\bar{v} \in \mathrm{Iso}(\pi)} \prod_{i \in [q]} \gamma(v_i)$. It is easy to see that $\sigma(\mathcal{G}, \gamma)$ coincides with d_π for π being the graph consisting of q isolated vertices.

We start with a lemma that allows us to calculate the values c_π.

Lemma 68. *Let π be a arbitrary graph over $[q]$. Then c_π can be calculated in time $O(qd^q \cdot \|\mathcal{G}\|)$, where d is the maximal degree of $\mathcal{G}$.*

Proof: Let π be a graph over $[q]$ splitting up into components $\pi_1, \ldots, \pi_r$. By definition, we have $c_\pi = c_{\pi_1} \cdots c_{\pi_r}$ (just apply the distributive law). Hence it is enough to show that the claim holds for connected π.

So assume that π is connected and take an arbitrary $j \in [q]$. If $\bar{v} \in \mathrm{Hom}(\pi)$ then, by connectedness, $v_i \in N_q^{\mathcal{G}}(v) \cap C_i$ for all $i \neq j$. Since $\mathcal{G}$ has degree at most d, $N_q^{\mathcal{G}}(v)$ contains $\leq d^q$ elements. Observe that the situation is similar to that in lemma 56 where we calculated $\mathrm{c}(\mathcal{T},\kappa)$ for a tree cover $\mathcal{T}$ and a contiguousness-type κ.

Now the algorithm proceeds as follows: In a first phase, using the just derived characterization, we compute a list containing all tuples $\bar{v}$ that are in $\mathrm{Hom}(\pi)$. This list may contain multiples. Then we remove multiple occurences from the list, and, in a last step, we sum up the $\gamma(\bar{v}) := \gamma(v_1)\cdots\gamma(v_q)$ for $\bar{v}$ from the list. This is displayed as algorithm 26.

proc calc_pi$(\mathcal{G},\gamma)$
    comment: calculate the vertex set
    h $:= \emptyset$; $j \in [q]$;
    for $v \in C_j$ **do**
        for $\bar{v}$ such that $v_j = v$ and all $v_i \in N_q(v) \cap C_i$ **do**
            if $\bar{v} \in \mathrm{Hom}(\pi)$ **then**
                append(h, $\bar{v}$);
            fi
        od
    od
    remove multiples from h;
    return$(\sum_{\bar{v}\in h}\gamma(v_1)\cdots\gamma(v_q))$
.

Algorithm 26. Calculate c_π

It is easy to see that the algorithm works correctly. The time bounds are easily verified. For instance, the loops require time $O(|G|\cdot d^q)$ (since $|N_q(v)| \leq d^q$). Multiple occurrences of tuples can be removed by a simple Radix-sort (cf. the appendix) and the sum in the last line is computed in time linear in the size of h. $\square$

With this lemma we are almost finished. By the above observation on the relation between homomorphisms and isomorphisms, we have

$$d_\pi = c_\pi - \sum_{\pi' \supsetneq \pi} d_{\pi'}, \tag{4.23}$$

which leads us to the following procedure:

Algorithm 27: Computing $\sigma(\mathcal{G},\gamma)$

Input: A graph $(\mathcal{G}, C_1,\dots,C_q)$ and $\gamma : G \to \mathbb{N}$
Output: $\sigma(\mathcal{G},\gamma)$

1. Compute c_π for all π over $[q]$
2. Compute d_π for all π using formula (4.23)
3. Return d_π for π the empty graph

The correctness is immediate. Since there are $\leq 2^{q^2}$ different π the first line needs time $O(\|\mathcal{G}\|)$ (by lemma 68). Line 2 needs some more explanations: by induction, we proceed downwards starting with the complete graph π (for which we have $c_\pi = d_\pi$). Having computed all d_π for π with $l+1$ edges, we can compute d_π for π with l edges in constant time (simply using formula (4.23)).

Altogether we stay within the time bounds claimed in theorem 67. $\square$

Appendix A

The Machine Model

This appendix is dedicated to a detailed definition of the machine model used. This is of particular interest, because linear time classes depend heavily on the model, and still there is no agreement on which one is the most appropriate. We will use Random Access Machines (RAM) that have an infinite supply of registers, each of which may contain a natural number.

The choice for RAMs is motivated by the fact that most algorithms, considered to run in linear time, depend on pointers and direct access to data, a feature that is not provided by Turing machines. Essentially, there are two ways to charge time for a single operation. The *uniform* time measure charges one time unit for each operation, whatever the size of the involved registers is. In this framework, the content of a register is considered to have constant size. To some extend this model is justified by the fact that most objects in algorithms are represented by pointers, which in a real computer have constant size.

In contrast to that, the *logarithmic* cost measure charges the logarithm of the values of the registers involved in the operation. This cost measure reflects the fact that logarithmic space is necessary to store numbers. For most algorithms (that do not involve too much arithmetic) this difference is neglected because of the following reason: the collection of primitive input objects (e.g. the vertices of a graph) is assumed to be an initial segment of the natural numbers. This problem of *normalized* inputs will be addressed later.

We slightly modify the model defined by Grandjean in a series of papers [Gra94a, Gra94b, Gra96]. This model is an extension of the basic RAM-model described in, for example, [CR73], based on the logarithmic cost measure. We present the definition of the model in the first section. Then we address the problem of relational structures as inputs of machines, and its impact on the model checking problem. There we encounter the problem of normalizing the input.

Finally we give some sample algorithms to illustrate the used pseudo-code and frequently used data structures.

A.1 The Definition

The following definition of a DRAM is essentially due to Grandjean and Schwentick [GS99]. We fix an ordered finite alphabet $\Sigma = \{1, \dots, d\}$. We identify a word $w \in \Sigma^n$ with the integer it represents in d-adic notation, that is

$$w = \sum_{i=1}^{n} w_i d^i.$$

The empty word is identified with zero.

A $\{+, -\}$-DRAM M is a *Random Access Machine* with two accumulators A, B, a special register N and registers $R^\nu(i)$, for every $i, \nu \geq 0$. Each of these registers contains a non-negative integer. Its program is a sequence $I(1), \dots, I(r)$ of instructions, each of which is of one of the following forms:

- $A := c$, for some constant c
- $A := A * B$, where $* \in \{+, -\}$
- $A := N$, $N := A$ and $B := A$
- $B := R^\nu(A)$ and $R^\nu(A) := B$, for $\nu \geq 0$
- IF $A = B$ THEN $I(i_0)$ ELSE $I(i_1)$
- HALT
- WRITE($R^\nu(A)$) and READ$_{R^\mu(B)}$($R^\nu(A)$)

Most of these instructions have the expected meaning. For instance, if A contains a number i then the execution of $R^\nu(A) := B$ copies the content of B to register $R^\nu(i)$. This is a kind of addressing mechanism, where A (or i respectively) is the referred *address*.

The input-output operations need some explanations: WRITE($R^\nu(A)$) writes the content of $R^\nu(A)$ to the output-tape. If the length of $R^\mu(B)$ is l, then READ$_{R^\mu(B)}$($R^\nu(A)$) stores in register $R^\nu(A)$ the next l digits of the input word and advances the input head accordingly. As input, a RAM expects a word $w \in \Sigma^*$. We define $\|w\|$ to be the length of the word. If register $R^\nu(i)$ contains the number n then $\|R^\nu(i)\| := \log n$. The space usage of a computation of a DRAM is the maximal space used during the computation. That is, the maximal $\sum_{i,\nu\geq 0} \|R^\nu(i)\|$. Now, we are ready to define time complexity classes.

Primarily, we are interested in linear time algorithms, which are a particularly delicate question. The models proposed in the literature, differ in how they charge time, what primitive objects are, and how much space is admitted. We propose a particularly restricted definition of linear time that only allows as much space as time.

Definition 69. *Let $T(n) \geq n$ be any time function. A function $f : \Sigma^* \to \Sigma^*$ (language $A \subseteq \Sigma^*$) belongs to class* $\mathrm{DTIME}(T(n))$, *if f is computable (respectively, A is recognizable) by a DRAM that executes $O(T(n))$ instructions (logarithmic time criterion), where n denotes the length $\|w\|$ of the input w. Furthermore, the available space is restricted to $c \cdot T(n)$ for some $c \geq 1$.*

We do not admit multiplication as a primitive operation. For addresses, multiplication and division by 2 can be done using precomputed tables [GS99].

For *counting problems* we also allow multiplication to be done in linear time, i.e. the operation $A \cdot B$ is charged with $\|A \cdot B\|$ steps.

Usually, we have structured input, that is input of the form $w = w_1, \ldots, w_n$, where w.l.o.g. $, \notin \Sigma$ is a separator. The *normalization*-problem is defined as follows:

Input: a string $w_1, \cdots, w_m \in (\Sigma \cup \{,\})^*$
Output: a string $v_1, \cdots, v_m \in (\Sigma \cup \{,\})^*$, such that

(1) for all $i, j : v_i = v_j$ if, and only if, $w_i = w_j$ and

(2) considered as integers, we have $\{v_1, \ldots, v_m\} = [l]$ for some $l \leq m$.

Grandjean proved the following nice result:

Theorem 70 (Grandjean [Gra96]). *There exists an algorithm that solves the normalization problem for an input w in time $O(\|w\|)$.*

The algorithm is a generalization of the well-known Radix-sort to tuples of arbitrary degree. Since it works in time linear in the size of the input, it justifies the assumption that inputs are already normalized. By that, all

w_i can be considered as addresses in the memory. This is an indispensable feature for linear time algorithms. In section A.4 we present an easy version of Grandjean's algorithm, which does not tightly obey the linear time bound. Nevertheless, there are reasons that this simplification is the right choice for practice.

A.2 RAMs on relational structures

All our algorithms work on relational structures. Let $\tau := \{R_1, \ldots, R_l\}$ be a vocabulary, where R_i is of arity $r_i \geq 0$. Let $\mathfrak{A} \in \mathrm{Str}(\tau)$ be a finite τ-structure. An encoding of $\mathfrak{A}$ is composed of the following: It starts with an encoding of the vocabulary, followed by the size of the universe A and the sizes of the relations $R_i^{\mathfrak{A}}$. The next $|A|$ numbers are the elements of the universe followed by enumerations of the tuples of the relations $R_1^{\mathfrak{A}}, \ldots, R_l^{\mathfrak{A}}$.

In particular, the encoding $\mathrm{enc}(\tau)$ of τ as above is a sequence of $|\tau| + 1$ natural numbers. The first number is $|\tau|$, followed by the numbers $r_1, \ldots, r_l$. The length of $\mathrm{enc}(\tau)$ is $\|\tau\| := |\tau| + 1$.

For τ as above, we say that a word w *encodes* a τ-structure $\mathfrak{A}$, if w has the form

$$w = \mathrm{enc}(\tau), n, m_1, \ldots, m_l, x_1, \ldots, x_n, \bar{a}_1^1, \ldots, \bar{a}_{m_1}^1, \ldots, \bar{a}_1^l, \ldots, \bar{a}_{m_l}^l$$

where $n = |A|$, $m_i = |R_i^{\mathfrak{A}}|$ and $\{\bar{a}_1^i, \ldots, \bar{a}_{m_i}^i\} = R_i^{\mathfrak{A}}$ for all $i = 1, \ldots, l$. The tuples $\bar{a}_j^i$ are represented as mere sequences of r_i natural numbers (recall that R_i is r_i-ary). Observe that an encoding of a structure $\mathfrak{A}$ is not unique, since it depends on the ordering in which the elements of the universe appear. The size $\|\mathfrak{A}\|$ of the encoding of the structure $\mathfrak{A}$ is the sum

$$\|\mathrm{enc}(\tau)\| + \|n\| + \sum_{j=1}^{n}(\|x_i\| + 1) + \sum_{i=1}^{l}((\|m_i\| + 1) + \sum_{j=1}^{m_i}(\|\bar{a}_j^i\| + 1)),$$

which coincides with the size of w (note the we have to count the ","s as part of the input). Often it is assumed that elements of the universe are primitive, what means that they need constant space. We think that this is not realistic as the following example will show.

Querying against databases:
In database theory, we are given a countable set $\mathrm{dom} = \{v_1, v_2, \ldots\}$ and a database scheme τ, which corresponds to a relational vocabulary. An

instance of τ is a mapping I over domain τ, such that $I(R)$ is a finite relation over dom for each $R \in \tau$. The *active domain* of I is the set of all elements of dom that occur in a relation $I(R)$, $R \in \tau$. Now queries can be evaluated over the active domain, which, in logical terms, is a finite τ-structure.

In practice, the domain elements correspond to maybe complex objects like strings or even bitmaps. Thus, given a database instance I, it is not realistic to assume that the active domain is an initial segment of the natural numbers. This leads to the failure of techniques (e.g. using domain elements as array indices) that are crucial for linear time algorithms.

Recall that, by theorem 70, we can normalize our input in linear time. In our setting, algorithms will look as follows: first we compute a normalized version of the input structure $\mathfrak{A}$, i.e. a structure $\mathfrak{B}$ isomorphic to $\mathfrak{A}$, such that B is an initial segment of the natural numbers. Then the actual algorithm is applied to $\mathfrak{B}$.

Maybe, this algorithm produces some output in terms of $\mathfrak{B}$, e.g. tuples, or sets of tuples from B. To provide an answer in terms of the input structure, we have to retranslate these objects from $\mathfrak{B}$ to $\mathfrak{A}$. This is done using a table `retrans`$[\cdot]$ such that for $b \in B$ `retrans`$[b]$ is the pre-image of b in A according to the isomorphism computed by the normalization. Note that Grandjean's algorithm provides this table as a by-product. With this table (which is of linear size), the output $O_{\mathfrak{B}}$ can be translated to $O_{\mathfrak{A}}$ in time $O(\|O_{\mathfrak{A}}\|)$, i.e. in time linear in the size of the output.

This justifies the next convention, which has been assumed in almost all papers concerning algorithms on graphs and structures, but by the theorem of Grandjean receives its validity without any further assumptions:

Convention:

> Without loss of generality, we assume that the set of elements of the universe of an input structure is an initial segment of the natural numbers.

A.3 Data-structures and algorithms

In this section we describe the pseudo-code language used in presentation of the algorithms and the data-structures we work with. At the end, we give the implementation of radix-sort that illustrates the introduced concepts (the data structures as well as the language elements). We always use `typewriter` face to refer to variables and procedures.

Notice that the algorithmic problems we consider mostly have an input of the form $(\mathfrak{A}, \varphi)$ where $\mathfrak{A}$ is a τ-structure for some relational vocabulary τ and φ is a query. We consequently adopt the viewpoint of paramerized complexity for algorithms and their running time (cf. the introduction). In a typical situation, we express the running time of an algorithm on input $(\mathfrak{A}, \varphi)$ by functions on $||\varphi||$, $\lambda(\mathfrak{A})$ for some numerical structure invariant[1] λ, and $||\mathfrak{A}||$. In this context we call a data structure *big* if its size may depend on $||\mathfrak{A}||$, and *small*, otherwise. This convention is motivated by the fact that the input structure is considered to be "big", whereas queries or invariants tend to be "small". We focus on linear time bounds with respect to $||\mathfrak{A}||$ and arbitrary dependence on the remaining parameters: hence, whatever representation of small objects we choose, the running time will yield results of essentially the same type.

Big objects comprise e.g. subsets of the universe A and data associated with elements of the universe. For the latter, recall that the input is an initial segment of the natural numbers. This permits arrays over the universe and therefore constant time access to the data that corresponds to an $a \in A$. Subsets of A or A^k for $k \geq 1$ whose size depend on $||\mathfrak{A}||$ are always represented by *doubly linked cursor lists.* A list consists of a collection of *list items*, each of which has a pointer to the preceding and the succeeding list item, and a *content* field. The list data structure has a pointer to the first and the last item of the list. Additionally, there is a pointer to some arbitrary item, the *cursor*. An example of such a list is displayed in figure A.1. Observe that, in this example, the contents are pointers to objects x_i.

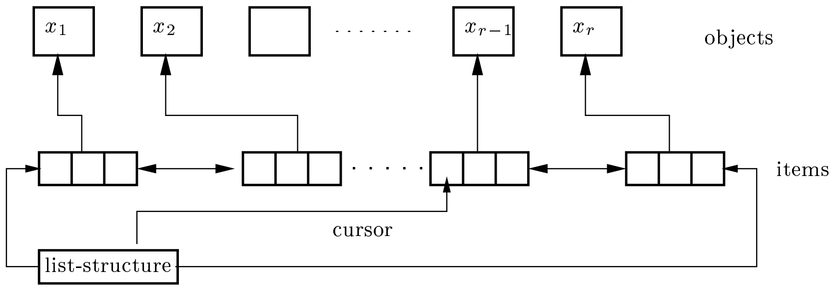

Figure A.1: The representation of lists.

The items: As displayed in figure A.1, a list consists of an ordered collection of items. Each of these items contains two pointers (establishing the list

[1] Formally a numerical invariant for τ-structures is a function $f : \mathrm{Str}(\tau) \to \mathbb{N}$ that maps isomorphic structures to the same value.

structure) and a content field. We give an example of the usage of pointers to items at the end of the section. Let `it` denote a item variable; with the following routines we can access items.

- `content(it)`
- `next(it)`, `previous(it)`
- `remove(it)`

Observe that items are *pointer-types*, i.e. a variable `it` contains a pointer to an item. Furthermore, we have a convention that items never occur isolatedly. This means that an item is always an item of a list, and thus removing an item always means removing an item from a list.

The following procedures provide access to lists. Their semantics should be clear. Let `list` denote a list, and `x` refer to some object.

- `first(list)` and `last(list)`
- `append(list, x)`, `push(list, x)`
- `append(list, it)`, `push(list, it)`
- `append(list, list′)`, `push(list, list′)`

All this subroutines need constant time. This is remarkable for the case `append(list, list′)`, where we simply let the end-pointer of `list` point to the last element of `list′` and add a pointer from the last element of `list` to the first element of `list′`. Observe that we do not copy the lists, instead we incorporate one list into the other. For removing elements quickly, it is crucial to have double links.

Observe that we do not support access via the content field. This would involve a linear search through the list and thus make it useless for linear time algorithms (of course only if the list codes a *big* object).

Convention:

> In the bulk of applications we blur the difference between items and elements of a list. This allows, for instance, statements like `remove(x)` or `next(x)` (but nevertheless, one should assure that the code can be rewritten just using items).

In the rest of the section we present examples that introduce our used pseudo-code. These examples also describe repeatedly used simple data structures and algorithms.

Example 1: *Radix-sort.* Assume that we are given a structure $\mathfrak{A}$ and a set $X \subseteq A^k$. As explained above, A is identified with an initial segment $[n]$ of the natural numbers, whereas X corresponds to a list `list` of k-size arrays `a` where $\mathtt{a}[i] \in [n]$ for $i = 1, \ldots, k$. The procedure $\mathtt{radix\_sort}(\mathtt{list}, k, n)$ displayed as algorithm 28 sorts `list` with respect to the lexicographical ordering in time

$$O(k \cdot (\|X\| + |A|)).$$

After the execution the sorted list is contained in the variable `list`. Notice that we strictly distinguish between subroutine calls $\mathtt{procedure}(\mathtt{params})$ and access to array elements $\mathtt{array}[\mathtt{index}]$.

```
proc radix_sort(list, k, n)
  newsort[n];  /* initialize array of size n */
  for ν := k to 1 do
      temp := NULL;
      for j := 1 to n do
          sort[j] := NULL;
      od
      for it ∈ list do
          append(sort[content(it)[ν]], it);
      od
      list := NULL;  the empty list
      for j := 1 to n do
          for it ∈ sort[j] do
              append(list, it);
          od
      od
  od
  return(sort)
.
```

Algorithm 28. radix-sort

The algorithm is well known, so we continue without further explanations on the algorithm. The code itself should be clear.

Example 2: *Referenced lists.* The next example introduces a data structure that allows to handle subsets of A (for a given structure $\mathfrak{A}$) in a very efficient manner. After the initialization with some linked list (representing the set), it supports (i) membership checks in constant time. On the other hand it also allows to (ii) find an arbitrary element of the set in constant time (this prevents us from simply taking an array). We call this data structure *referenced list*.

Let $X \subseteq A$ be given by a list `list` of size l and assume that A has size n. Remember that, by our assumptions, we have $A = [n]$ and $X \subseteq [n]$. Hence, elements of A can be used as array indices. We need three additional arrays: `ref` of size n associates with each $a \in X$ the corresponding item in `list` (whose item content is a). Formally, for each item `it` of `list` with $\texttt{content(it)} = a$ we have $\texttt{ref}[a] = \texttt{it}$. But what happens if $a \notin X$? In general, we do not have the time to initialize the array `ref` to assure that the ath entry does not accidently point to some valid position (maybe an item of another list). For that we introduce new "control" arrays `contr` of size $|X| = l$ and `refc` of size n, and initialize them such that $\texttt{contr}[i] = a \iff a$ is the ith element in `list` (at the time of initialization).

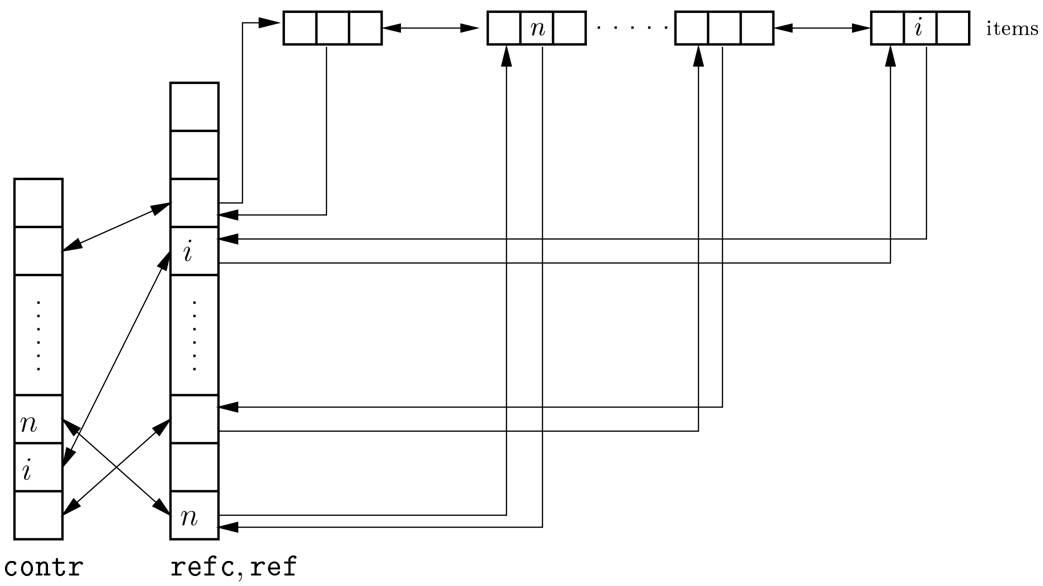

Figure A.2: Implementation of a referenced list.

An example of this construction is displayed as figure A.2. Observe that $a \in X$ if and only if $\texttt{contr}[\texttt{refc}[a]] = a$. We implement a couple of subroutines that perform basic queries to a referenced list (algorithm 29). `ref_init(list)` initializes the auxiliary arrays for `list`, while `remove(list)` removes the elements contained in `list` from the referenced list.

```
proc ref_init(list)
   i := 1;
   for it ∈ list do
        a := content(it);
        ref[a] := it; refc[a] := i; contr[i] := a;
        i := i + 1;
   od
.
proc remove(list)
   for it ∈ list do
        remove(ref[content(it)]);
        a := content(it);
        ref[a] := NULL; contr[refc[a]] := 0; refc[a] := 0;
   od
.
proc is_element(a)
   return[contr[refc[a]] = a];
.
```

Algorithm 29. Accessing a referenced list.

Answering $a \in$ `list` is done by the obvious check in an additional procedure `is_elem`(a). Observe that this can be done in constant time, while we can remove k elements in time $O(k)$.

A standard application of this data structure is a set that is made smaller gradually, and each step depends on some elements still contained in the set (cf. algorithm 12 on page 50).

We end this section with three easy, but useful procedures.

A structure prescan: We go through the whole input and assign to each tuple $\bar{a}$ occurring in a relation $R^{\mathfrak{A}}$ a unique index $\iota(\bar{a})$. For a τ-structure $\mathfrak{A}$ as above and $R \in \tau$ of arity r we define arrays E_R^ν for $\nu = 1, \ldots, r$ each of which has size $|R^{\mathfrak{A}}|$. $E_R^\nu[j] = k$ means that if $\bar{a}$ has index $\iota(\bar{a}) = j$, then $a_\nu = k$ (recall that the universe is represented by an initial segment of the natural numbers, hence each element can be seen as an array index). This prescan requires linear time in the size of $R^{\mathfrak{A}}$, and it enables us to associate in constant time labels with vertices and hyperedges.

Graphs: Graphs admit a more structured representation, referred to as *adjacency list* representation. This representation associates with each vertex the list of adjacent vertices. Having given an $\{E\}$-structure $\mathcal{G}$ we can calculate such a representation in linear time. Note that this representation is necessary for techniques like depth-first-search and breadth-first-search to be feasible in linear time.

Removing multiple edges: Let A be a set of size n. Assume that we just have computed an auxiliary graph $\mathcal{G}$ such that G consists of a list of vertices $\bar{a} \in A^k$ for some k. Assume further that vertices as well as edges may occur multiple times. To apply standard algorithms on $\mathcal{G}$ we have to compute a normalized version of $\mathcal{G}$ without multiple vertices/edges. Let $\mathcal{G}$ be structured as described in the previous section A.2 (the edges are an unstructured list of pairs).

In a first step we separately sort the universe and the set of edges. This is done using radix-sort explained above and requires $O(k \cdot n + |G|)$ steps for the vertices and $O(2k \cdot n + |E^{\mathcal{G}}|)$ steps for the edges. Then we normalize the result applying the algorithm from theorem 70. We will use this procedure in algorithm 16.

A.4 The linear time sort

In this last part of the appendix we present a simple version of the normalization algorithm of Grandjean (theorem 70). In contrast to the normalization explained before, this sorting procedure expects as input a list `w` of integer arrays of arbitrary size and sorts these arrays with respect to their size and lexicographical ordering. To see the connection between these procedures let $w = w_1, \dots, w_m \in (\{0,1\} \cup \{,\})^*$ be an input of the normalization algorithm, $n := |w|$ and $l := \lceil \log |w| \rceil$. We explain how to build a list of integer arrays `w` from the string w, which then can be sorted by the subsequent sorting routine.

We define w_i' to be the least word of size $k \cdot l$ (for some $k \geq 1$) that results from w_i by introducing leading zeros. This naturally induces a partition of w_i into k blocks of size l, which itself corresponds to an k-array of integers. Note that this array can be interpreted as the 2^l-adic representation of w_i (cf. page 102). For convenience, we assume that the first array entry contains the least significant bit. Note here that this transformation can easily be done in linear time.

After this transformation, sorting `w` is essentially the same as normalizing w. In particular, after sorting `w`, we assign numbers to all `it` $\in$ `w` such that, if the contents of two items are equal then they have the same associated number. For that, assume that `v` is a sorted version of `w` (i.e. it contains the same items, but in a different order). We loop over all `it` $\in$ `v` maintaining a counter c (intially set to 0), which is incremented in each step in which changes the current item. In the loop body we assign c to the current list item `it`, which gives us the desired isomorphism between the input and its normalized version.

Our sorting procedure is displayed as algorithm 30 (and essentially is a generalization of the Radix-sort). We claim that under the assumption that the integers are of size $O(\log \|\mathtt{w}\|)$, this algorithm works in time $O(\|\mathtt{w}\|)$. After an analysis of the procedure, we discuss why this algorithm doesn't fit the requirements of theorem 70, but nevertheless seems to be an adequate choice for our purposes.

```
1  proc sort(w)
2    ν := 1; sorted := NULL;
3    compute used_buckets[·]
4    while not-empty(w) do
5           ext_radix_sort(w, ν);
6           for x ∈ w do
7                if size(x) = ν
8                  then
9                        append(sorted, x); remove(w, x);
10               fi
11          od
12          ν := ν + 1;
13   od
14   return(sorted)
```

Algorithm 30. Sorting tuples of arbitrary length

The routine `normalize(w)` works in three steps. First we compute the array `used_buckets`, which is intended to contain in its ν-th field the indices i such that there is an $x \in$ `w` with $x[\nu] = i$. This array of lists can be obtained in linear time by a Radix-sort of the list $\{(\nu, i) \mid$ there is a $x \in$ `w` such that $x[\nu] = i\}$.

In the second step, `w` is sorted. To see this, consider the loop between lines 4 and 13. We have the following loop invariant at the top of the loop: `w` contains all x with `size`$(x) \geq \nu$ (`size` returns the number of array entries), ordered with respect to their ν least significant digits. All other arrays are already sorted in the list `sorted`. The call of `ext_radix_sort`$(\mathtt{w}, \nu)$ corresponds to the ν-th cycle in the known Radix-sort, and essentially sorts the list with respect to the ν-digit, conserving the ordering between items having identical ν-th digits.

Obviously, after leaving the loop, `sorted` is ordered as follows: x precedes y in `sorted`, iff $|x| < |y|$ or $x < y$ with respect to the lexicographical ordering.

```
proc ext_radix_sort(ilist, ν)
   for x ∈ ilist do
       append(sort[x[ν]], x);
   od
   ilist := NULL;   the empty list
   for i ∈ used_buckets[ν] do
       for x ∈ sort[i] do
           append(ilist, x);
       od
       delete(sort[i]);
   od
```

Algorithm 31. One cycle of the extended radix-sort

To finish the proof, consider subroutine `ext_radix_sort`$(\mathtt{ilist}, \nu)$ (displayed as algorithm 31), which behaves as follows: ν is expected to be a natural number, and `ilist` is a list of arrays of size $\geq \nu$. It puts each $x \in$ `ilist` in the bucket $x[\nu]$ (the ν'th least significant digit). Then it looks up in `used_buckets`$[\nu]$ which buckets have been used and copies the items back to (the previously deleted) `ilist`. Using this array we assure that in fact we only look in those buckets, which contain relevant data. All other buckets are ignored. It is important to note that for each $x \in$ `ilist` this procedure needs time $O(l)$ (two append operations and an array lookup), where l is the size of a pointer.

Analysis: Define $n := \|\mathtt{w}\| = \sum_{x \in \mathtt{w}} \|x\|$, where $\|x\| = \sum_{\nu=1}^{\mathtt{size}(x)} \log x[\nu]$, because of the logarithmic cost measure. As mentioned at the beginning, we assume that all $x[\nu]$ have the same size l, and take $m = n/l$ to be the number of integers in `w`.

It is easy to see that the array `used_buckets` needs the same amout of space as `w` and is computed in time $O(n)$ (cf. Radix sort in the previous section). Observe that we always refer with pointers to our objects (e.g. arrays or list items), which are all of size $O(\log n)$ (this is sufficient to access all objects). Consider the main loop of `sort`, let $x \in$ `w` and assume that `size`$(x) = k$. By the `if`-condition in line 7 it is obvious that x is treated

k-times in the procedure `ext_radix_sort` and the subsequent `append` operation, each of which costs time $O(\log n)$ (to move the pointers). Together, we get $O(m \cdot \log n)$ as an upper bound on the running time. With $l = O(\log n)$ this proves the claim.

Discussion: As mentioned before, this algorithm does not fully fit the requirements of theorem 70, although it looks very much alike. Consider the transformation of the input $w = w_1, \ldots, w_m$ to a list of integer arrays. To see that this is not sufficient assume that $w_i = 1$ for $i = 1, \ldots, m-1$ and $w_m = 1^m$. Then this transformation yields a data-structure of size $\log m \cdot (m-1) + m = \Omega(n \log n)$, which already breaks the time bound.

Grandjean overcomes this problem by dividing the words w_i into big and small ones. The small words are sorted in a certain space saving manner, while the big words are sorted as we just saw. Although Grandjean's version gives the desired theoretical result, the author believes that for practice (if it applies) the normalization algorithm presented here is the adequate one. First, the assumption that a natural number n has size $\log n$ is not realistic, in particular, there is a minimal de-facto integer size (usually 32 bit, soon becoming 64 bit). And this allows to adress the space for all sensible input sizes (it allows to adress $2^{32} \approx 10^{10}$ registers). On the other hand, it is not realistic to assume that inputs come as plain text (which is the reason that e.g. 1 needs space 1); in all applications, algorithms receive their data in a structured way.

Index

Bibliography

[ABCP93] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In *Proceedings of the 34th Annual IEEE Symposium on Fondations of Computer Science*, pages 638–647, 1993.

[ACP87] S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8:277–284, 1987.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[ALS91] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[AP90] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31st Annual IEEE Symposium on Fondations of Computer Science*, pages 503–513, 1990.

[BDG95] J. Balcazar, J. Diaz, and J. Gabarro. *Structural Complexity 1.* Springer, Berlin, 1995.

[Bod96] H.L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on Computing, 25:1305–1317, 1996.

[Bod97] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In Igor Privara and Peter Ruzicka, editors, *Proceedings 22nd International Symposium on Mathematical Foundations of Computer Science*, volume 1295 of *Lecture Notes in Computer Science*, pages 29–36. Springer-Verlag, Berlin, 1997.

[Bod98] H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, (209):1–45, 1998.

[CH82] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 1982.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 9th ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[CM93] B. Courcelle and M. Mosbah. Monadic second-order evaluations over on tree-decomposable graphs. *Theoretical Computer Science*, 103:49–82, 1993.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. *In J. van Leeuwen, editor,* Handbook of Theoretical Computer Science, Elsevier Science Publishers, 2:194–242, 1990.

[CR73] S.A. Cook and R.A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7:354–375, 1973.

[DF99] R.G. Downey and M.R. Fellows. *Parameterized Complexity.* Monographs in Computer Science. Springer-Verlag, 1999.

[DFT96] R.G. Downey, M.R. Fellows, and U. Taylor. On the parametric complexity fo relational databases queries and a sharper characterization of $W[1]$. In *Combinatorics, Complexity and Logic*, DMTCS, pages 194–213. Springer-Verlag, 1996.

[Die97] R. Diestel. *Graph theory.* Graduate texts in mathematics. Springer, 1997.

[EF95] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer Verlag, 1995. ISBN 3-540-60149-X.

[EFT95] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic.* Springer Verlag, 2 edition, 1995. ISBN 3-540-60149-X.

[Epp95] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms and Applications*, volume 3, pages 1–27, 1995.

[Epp99] D. Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27(3):275–291, 1999.

[Erd59] P. Erdös. Graph theory and probability. *Canadian Journal of Mathematics*, (11):34–38, 1959.

[FG99] M. Frick and M. Grohe. Checking first-order properties of tree-decomposable graphs. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Automata, Languages and Programming, 26th International Colloquium, ICALP'99, Prague, Czech Republic, July 11-15, 1999, Proceedings*, volume 1644 of *Lecture Notes in Computer Science*, pages 331–340. Springer, 1999.

[FV93] T. Feder and M.Y. Vardi. Monotone monadic SNP and constraint satisfaction. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 612–622, 1993.

[Gai82] H. Gaifman. On local and non-local properties. In J. Stern, editor, *Logic Colloquium '81*, pages 105–135. North Holland, 1982.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[Gra94a] E. Grandjean. Invariance properties of RAMs and linear time. *Computational Complexity*, 4:62–106, 1994.

[Gra94b] E. Grandjean. Linear time algortihms and NP-complete problems. *SIAM Journal of computing*, 23:573–597, 1994.

[Gra96] E. Grandjean. Sorting, linear time and the satisfyability problem. *Annals of Mathematics and Artificial Intelligence*, 16:183–236, 1996.

[Gro00] M. Grohe. Generalized model-checking problems for first-order logic. *to appear*, 2000.

[GS99] E. Grandjean and T. Schwentick. Machine-independent characterizations and complete problems for deterministic linear time. *to be published*, 1999.

[Pel93] D. Peleg. Distance-dependent distributed directories. *Information and Computation*, (103):270–298, 1993.

[PY97] C.H. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *Proceedings of the 16th ACM Symposium on Principles of Databases Systems*, pages 12–19, 1997.

[Ree92] B. Reed. Finding approximate separators and computing tree width quickly. *ACM Symposium on theory of computing*, 1992.

[RS84] N. Robertson and P.D. Seymour. Graph minors III. planar tree-width. *Journal of Combinatorial Theory, Series B*, (36):49–64, 1984.

[RS86] N. Robertson and P.D. Seymour. Graph minors II. algorithmic aspects of tree-width. *Journal of Algorithms*, (7):309–322, 1986.

[Tho97] W. Thomas. Languages, automata, and logic. *In A. Salomaa and G. Rozenberg, editors, Handbook of Formal Languages*, 3, 1997.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second order logic. *Math. Syst. Theory*, (2):57–82, 1968.

[Var82] M.Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

[Var95] M.Y. Vardi. On the complexity of bounded-variable queries. In *Proceedings of the 14th ACM Symposium on Principles of Data bases Systems*, pages 266–276, 1995.

[Woe00a] A. Woehrle. private communication, 2000.

[Woe00b] A. Woehrle. Lokalität in der Logik und ihre algorithmischen Anwendungen. Master's thesis, Albert-Ludwigs-Universität Freiburg, 2000.