ENSC-(n d'ordre)

THÈSE DE DOCTORAT DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Présentée par

Monsieur Wojciech Kazana

Pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Domaine :
Informatique

Sujet de la thèse :

Query Evaluation with Constant Delay

Thèse présentée et soutenue à Cachan le 16ème Septembre 2013 devant le jury composé de :

Nicole Bidoit	Professeur	Président
Arnaud Durand	Professeur	Rapporteur
Patrice Ossona de Mendez	Chargé de Recherche	Rapporteur
Victor Vianu	Professeur	Examinateur
Luc Segoufin	Directeur de Recherche	Directeur de thèse

Laboratoire Spécification et Vérification
ENS de Cachan, UMR 8643 du CNRS
61, avenue du Président Wilson
94235 CACHAN Cedex, France

Acknowledgments

I am grateful to Luc Segoufin who kindly accepted me to be his PhD student. He introduced me to the problem of query enumeration and encouraged me to look for the answers to all the questions that emerged during our collaboration. He was a truly great advisor, always supportive and available for discussions and advice.

I would like to thank Arnaud Durand and Patrice Ossona de Mendez for accepting to review this thesis and for their constructive comments on the earlier versions of this manuscript. Many thanks to Nicole Bidoit and Victor Vianu for accepting to be in my jury.

Special thanks go to my Parents, who directed me to follow the scientific path and where fully supportive at all times.

Thanks to all the members of LSV – it was a pleasure working with You.

I would like to thank the awesome people whom I have been working with in the same office on the 4th floor. Without all the nice riddles and the magnificent French lessons my time there wouldn't be as chouette as it was.

Big thanks go to my son Boris, who was kind enough to sleep through the whole nights, allowing his dad to actually write the thesis down.

Last but not least, the biggest thanks to my beloved wife Kinga. Without her continuous support the whole Paris adventure wouldn't be possible at all.

Contents

1

Introduction

"Knowledge is power."

– Francis Bacon

Knowledge is power. Somewhere between the end of sixteenth and the beginning of seventeenth century, when Francis Bacon said those memorable words, they certainly had different flavor than they do nowadays. Guilds were still at their best, not threatened by any global criticism and successively shielding the outer world from their secret knowledge. Books and even the most basic education were highly limited and accessible only to a small fraction of the society. Without a doubt, Bacon's words had far more practical tone back then.

Unarguably, the knowledge still IS power. We are far from claiming that it is not. But with the continuous spread of knowledge throughout the passed ages and its real worldwide explosion thanks to the popularization of the Internet, there seems to be something more to it nowadays. "When did Bacon actually lived? Has Napoléon Bonaparte ever seen Sphinx? In which part of Warsaw is Umińskiego street?" you might wonder? Well, just turn on your web browser, type the question into one of the search engines and voilà! It is all there. Just waiting for someone curious to ask.

But hold on right there! Whom is it that we actually ASK?

The above story, despite being rather naive, points out two important facts. First of all the randomness of the three mentioned questions and the fact that they do have easily accessible answers[1] is supposed to show us that the amount of knowledge that everyone has access to these days, is truly overwhelming. The second emphasize is almost too abruptly put on the ASK word. There of course is no human being on the other side. These are computer programs that do all the dirty work.

Querying the database

From now on let us move to the world of computers, in particular to its databases segment.

An integral part of every database is the way the data is being stored. It should definitely be efficient in order not to use too much memory. It should definitely be well protected so that an unauthorized third party would not have access to it. But the main idea behind storing the data is and will always be the same: at some point we want to have access to it. The reason can be arbitrary: we may want to check the balance of our bank account or we may simply be hungry for some new gossip about our favorite movie star. What we want is information that we know is out there.

We think that it is justified to say that query evaluation is the most important problem in databases. Given a query q and a database $\mathbf{D}$ it is to compute the set $q(\mathbf{D})$ of all tuples in the output of q on $\mathbf{D}$.

[1]It works – we've checked.

However, the set $q(\mathbf{D})$ may be larger than the database itself as it can have a size of the form n^l where n is the size of the database and l the arity of the query. It can therefore require too many of the available resources to compute it entirely.

Example 1.0.1 *Imagine a social network with* 100 *million of users. Assume that we want to output all the pairs of people whose "handshake distance" is less than* 10*. Although not formally proved, there are many reasons to predict that the output will contain every possible pair of users!* 100 *million squared already is a scary number, while the query was only binary. The more of so called* free variables*, the more quickly the output of the query would tend to get out of hand.*

There are many solutions to overcome this problem. For instance one could imagine that a small subset of $q(\mathbf{D})$ can be quickly computed and that this subset will be enough for the user needs. Typically, one could imagine computing the top-ℓ most relevant answers relative to some ranking function or to provide a sampling of $q(\mathbf{D})$ relative to some distribution. One could also imagine computing only the number of solutions $|q(\mathbf{D})|$ or providing an efficient test for whether a given tuple belongs to $q(\mathbf{D})$ or not.

Enumeration of queries

Our main focus is the following scenario: given a query q and a database $\mathbf{D}$, we want to enumerate $q(\mathbf{D})$ with constant delay. Intuitively, this means that we are looking for a two-phases algorithm working as follows: it starts with a preprocessing phase that works in time linear in the size of the database, followed by an enumeration phase outputting one by one all the elements of $q(\mathbf{D})$ with a constant delay between any two consecutive outputs. In particular, the first answer is output after a time linear in the size of the database and once the enumeration starts, a new answer is being output regularly at a speed independent from the size of the database. Altogether, the set $q(\mathbf{D})$ is entirely computed in time $f(q)(\|\mathbf{D}\| + |q(\mathbf{D})|)$ for some function f depending only on q and not on $\mathbf{D}$.

The reason why we want to somehow separate the size of the query from the size of the database and that we insist on the linearity of the database part, while we do not restrict the query size at all, is illustrated by Example 1.0.1: while a database of 100 million people would be huge, it took us roughly one line of text to describe the query. It is in general believed to be the case that queries tend to be small, while the databases grow bigger and bigger in size.

Example 1.0.2 *Consider the scenario from Example 1.0.1. The mentioned query has a natural enumeration algorithm for the mentioned database:*

The preprocessing phase simply puts all the people onto a linked list and stores two pointers, both pointing at its first element.

The enumeration phase just moves second of the pointers along the list, outputting the pair currently pointed by first and second pointer at each step. As soon as it traverses all the elements, it moves the first pointer (just once) along the list, resets the second pointer to the current value of the first pointer and starts all over with the enumeration procedure. It continues this loop until the first pointer reaches the last element on the list.

There is also another way of viewing constant delay enumeration that we find truly appealing. One could also view the enumeration algorithm as follows: The preprocessing phase computes in linear time an index structure representing the set $q(\mathbf{D})$ in a compact way (of size linear in the size of the database). The enumeration algorithm is then a streaming decompression algorithm.

As an addition, one could also require that the enumeration phase outputs the answers in some given order. To some extent we also address in this thesis the problem of the output order and the impact it has on the complexity (or the existence) of the enumeration algorithm.

Around enumeration

There are many problems related to enumeration. The main one is the model checking problem. This is the case when the query is boolean, i.e. outputs only 0 or 1. In this case a constant delay enumeration algorithm is a Fixed Parameter Linear (FPL) algorithm for the model checking problem of q, i.e. it works in time $f(q)\|\mathbf{D}\|$. This is a rather strong constraint as even the model checking problem for conjunctive queries is not FPL (modulo some hypothesis in parametrized complexity) [57]. Hence, in order to obtain constant delay enumeration algorithms, we need to make restrictions on the queries and/or on the databases.

Another problem related to enumeration is the already described counting problem, where we are only interested in computing the number of solutions to the query.

The also mentioned before testing problem is of providing an efficient test for whether a given tuple belongs to the output of the query or not.

Since the output of the query may be huge, even its compact representation and its efficient decompression might not be satisfactory enough. We might be interested in directly computing the j-th solution (with respect to some chosen order) of the output set. We then speak of the j-th solution problem.

Although our main focus remains with the enumeration problem, we always try to address the above four problems too.

Structure of the thesis

This thesis is organized as follows.

- In Preliminaries Chapter 2 we introduce all the necessary terminology and show some examples.
- In the following Chapter 3 we present simple results that are being used later. It might as well have been part of the Preliminaries Chapter 2, but we decided to extract it away for the sake of readability.
- In State of the Art Chapter 4 we try to bring to light all the results concerning enumeration that we are aware of. This is divided into two sections. In the first we consider the database scenario as described above. In the other we mention a variety of enumeration algorithms understood in a broader sense: instead of enumerating solutions to a query we might for example be interested in enumerating all the spanning trees of a given graph or any other objects satisfying an arbitrary property.
- The next three chapters contain the heart of this thesis and its main contribution. We in turn consider:

  • first-order queries over classes of structures with bounded degree (cf. Chapter 5). We follow the lines of [46] in the presentation of the enumeration algorithm. We then show new proofs of the existence of efficient solutions to the corresponding counting, testing and j-th solution problems by extending the approach of [46];

  • monadic second-order queries over classes of structures with bounded treewidth (cf. Chapter 6). We follow the lines of [48] in the presentation of the enumeration algorithm. We also this time show new proofs of the existence of efficient solutions to the corresponding counting, testing and j-th solution problems by extending the approach of [48];

  • first-order queries over classes of structures with bounded expansion (cf. Chapter 7). We follow the lines of [47] in the presentation of the model-checking, enumeration, testing and counting algorithms. We later on show why [47] gives us an efficient solution to the j-th solution problem "for free".

- In the Discussion Chapter 8 we first sum up the content of the previous chapters. We then discuss some interesting open problems that naturally emerged from the considered problems and we try to outline our intuition on possible ways of solving them.

For the details concerning particular definitions, the reader is referred to Chapter 2. We decided not to present the outlines of the proofs from Chapters 5, 6 and 7 here, since the necessary terminology has not yet been defined and we wanted to keep this Introduction chapter light and on a rather informal level. A reader familiar with all the mentioned concepts might move directly to Chapter 4 to find the omitted sketches. Otherwise we feel that it might be beneficial to get used to all the definitions and notations that we use throughout this thesis by parsing through Chapters 2 and 3.

2

Preliminaries

Contents

As mentioned in Chapter 1, throughout this thesis we work with problems concerning queries over databases. As we are dealing with linear time complexities, it is mandatory to make the model of computation precise. In this chapter we present all the necessary definitions. The way we introduce all the involved notations and definitions follows the lines of a recent survey on this topic by Luc Segoufin [62].

2.1 Databases, relational structures and queries

We define databases as relational structures.

Definition 2.1.1 *A* relational signature *is a tuple $\sigma = (R_1, \ldots, R_l)$, each R_i being a relation symbol of arity r_i. A* relational structure *over σ is a tuple $\boldsymbol{D} = \big(D, R_1^{\boldsymbol{D}}, \ldots, R_l^{\boldsymbol{D}}\big)$, where D is a finite set (called* the domain *of $\boldsymbol{D}$) and each $R_i^{\boldsymbol{D}}$ is a subset of D^{r_i}. The number of elements in the domain of $\boldsymbol{D}$ is denoted with $|\boldsymbol{D}|$.*

In the above definition, if for some i we have $r_i = 1$, then we also call relation R_i a *color*. Moreover, we often use the term *schema* instead of the term relational signature.

A *query* is a computable function associating to a database **D** a relation over the domain of **D**. The arity of the associated relation depends only on the query (that is, given a query q, there exists an integer r_q such that for an input database **D** query q returns a subset of D^{r_q}) and is called the *arity of the query*. If this arity is 0, we say that q is a *sentence*, it is either true or false on **D** and it defines a property of **D**. A query is *unary* if its arity is 1 and it then defines a color on **D**. Given a query q, database **D** and a tuple $\bar{a} \subseteq D$ we write $\mathbf{D} \models q(\bar{a})$ to denote the fact that $\bar{a}$ is in the image of q on **D** and $\mathbf{D} \models \neg q(\bar{a})$ to denote the opposite. In the particular case of sentences we write $\mathbf{D} \models q$ if q is true on **D** (or, in other words, **D** has property q) and $\mathbf{D} \models \neg q$ otherwise. We set $q(\mathbf{D}) = \{\bar{a} : \mathbf{D} \models q(\bar{a})\}$ and we call this set *the set of solutions to q over $\boldsymbol{D}$* . If **D** is clear from the contexts, we just write *the set of solutions to q*. We say that two queries q and q' are *mutually exclusive over $\boldsymbol{D}$* if their sets of solutions over **D** are disjoint, that is $q(\mathbf{D}) \cap q'(\mathbf{D}) = \emptyset$. Again, if the above intersection happens to be empty and **D** is clear from context, we just say that q and q' are mutually exclusive. Note that if the arity r_q of the query is greater than one, then the size of $q(\mathbf{D})$ may be exponential in r_q.

A query language is a class of queries. Typically it is defined as a logical formalism such as FO (for *first-order* queries) or MSO (for monadic *second-order* queries). In fact these are the two formalisms that are considered in Chapters 5 , 6 and 7 and they are in the center of attention of this thesis.

Given a query language $\mathcal{L}$, the *model checking problem of $\mathcal{L}$* is the following computational problem: given a sentence $q \in \mathcal{L}$ and a database **D** decide whether $\mathbf{D} \models q$ or not. If the database **D** is restricted to a class $\mathcal{C}$ of structures (which will most of the time be the case throughout this thesis), we speak of the *model checking problem of $\mathcal{L}$ over $\mathcal{C}$* .

Given a query language $\mathcal{L}$, the *query evaluation problem of $\mathcal{L}$* is the following computational problem: given a query $q \in \mathcal{L}$ and a database **D** compute set $q(\mathbf{D})$. Similarly as it was the case for model checking problem, if the database **D** is restricted to a class $\mathcal{C}$ of structures, we speak of the *query evaluation problem of $\mathcal{L}$ over $\mathcal{C}$* .

The definitions of both the model checking and the query evaluation problems as given above correspond to their *combined complexity*, meaning that the input for each of the problems is both query q and database **D**. In the sequel we most of the time assume the query to be a parameter of the problem (the only input then is the database). We then speak of the *data complexity* of the problems.

2.2 Model of computation

In Section 2.5 we define a series of problems that are central for this thesis. As their solutions are all going to mention linear time, it is mandatory to make our model of computation precise. We use Random Access Machines (RAM or RAM machines) with addition and uniform cost measure as a model of computation. As it is not the focus of this thesis, we do not give detailed definition of this model (for more information on the model itself see [2] and for its use in logic see [26]). We only list its key features that are the most relevant for this thesis:

• On an input of *size* n, a RAM machine has a certain number or registers, each consisting of its *value* and a unique *address*, both containing $\log n$ bits. We often write *the memory* of a RAM machine to denote its set of registers and a *memory cell* to denote a single register. Given a register r, we also write *a value stored in* r to refer to the value of r.

• The machine can modify the content of the registers just as a Turing Machine would do.

• Additionally, it can use any register r for accessing directly an other register r' such that the address of r' is given by the value of r. In this scenario we say that r is a *pointer* to r'.

• Moreover, RAM machine can perform numerical operations (typically additions or multiplications) on values of registers.

• An access to memory using a register or an addition of two registers takes a unit time in RAM, i.e. counts as a constant time.

A good approximation of RAM from the real life is a low level imperative programing in assembler. The values stored in assembler's registers can be used in two ways: either as direct addressing of the memory or as numerical input for some basic arithmetical operations. This is exactly what is offered by RAM, only the arithmetical part is limited to just addition and the times of performing a single addition and of accessing a memory cell are considered to be equal and are treated as a unit time. Moreover, while in assembler it is usually necessary to first load the content of a memory cell into a register before manipulating its value, in RAM model we in a sense assume that the memory is already divided into registers.

Continuing the "real life" analogy to imperative programming, when describing RAM algorithms we would often like to talk about lists or arrays of elements. And, perhaps even as an initial step, we would like to have records of constant size (for instance records that store pairs of elements). A good way of thinking about those objects can be the following:

• A record containing k "units of information" can be see as k registers with consecutive addresses. For instance to represent pair of values n, m one can use two registers r and r' with values n and m and addresses a and $a+1$ respectively. Note that having access to just r or r' we can then easily recover the other value in constant time (as the addresses are consecutive and a register knows its address). Also note that we can enrich such records with additional values by nesting records inside records.

• If we would like a register r to store both some value v and a pointer to some other register r', we can implement it with the value of r pointing to a *record describing* r. This record can then store in the values of consecutive registers both the value v and the address of r'.

• A list can then be seen as a sequence of registers such that each of them, except from storing its value, stores a pointer to the next element of the list (this can be achieved using records describing registers as shown above).

• Finally, an array can be seen as a sequence of registers with consecutive addresses.

In our setting the input will always be a database. We do not give a precise definition of how relational structures are encoded as inputs of RAM machines. This is identical as for Turing Machines and is described in many textbooks, cf. [1]. It only matters for this thesis that we can enumerate the domain or a relation of a database in linear time. For the simplicity of the presentation (in particular for a clear description of examples from Section 2.6) we later on assume that the domain and each of the relations is given either as a list or as an array containing the consecutive tuples of the matching list. In particular, we do not allow the use of uninitialized memory. The advantage of array representation is only the possibility to directly access its j-th element in constant time. We fix from now on a reasonable encoding of structures by binary words. The *size* of $\mathbf{D}$, denoted $\|\mathbf{D}\|$, is the length of the encoding of $\mathbf{D}$.

Note that for a fixed schema σ and a database $\mathbf{D}$ over σ with a domain of size n and the total number of tuples in all relations in $\mathbf{D}$ equal to m, the encoding of $\mathbf{D}$ has length $O((n+m)\log(n+m))$ bits and can be stored in $O(n+m)$ registers.

An immediate observation is that with the RAM model it is possible to sort m elements of size $O(\log m)$ in time $O(m \log m)$. In particular, we can sort the domain of $\mathbf{D}$ in time $O(\|\mathbf{D}\|)$, i.e. in linear time in terms of the input of the RAM machine. An important observation is that we may also sort tuples of nodes in linear time [40].

When we say that the model checking problem of $\mathcal{L}$ over $\mathcal{C}$ can be solved, we mean that there exists a RAM machine that solves the problem. Of course this extends to other problems, like query evaluation, etc.

In the sequel, when we say that the model checking problem of $\mathcal{L}$ over $\mathcal{C}$ can be solved in linear time, we refer to the data complexity only, i.e. we mean that it can be solved in $O(\|\mathbf{D}\|)$, when the big O may depend on q.

2.3 Parametrized complexity

Already at the end of both Section 2.1 and Section 2.2 we mentioned data complexity of problems. The reason for this is that the database $\mathbf{D}$ and the query q play different roles as inputs of our algorithms: $\|\mathbf{D}\|$ is assumed to be large, while $|q|$ is assumed to be small. Hence it would be useful to distinguish these two inputs in a formal way. Parametrized complexity is a suitable framework for analyzing such situations. Similarly to the RAM model, as parametrized complexity itself is not a topic of this thesis, we only provide here its basics. An interested reader is referred to the book [33] for extensive cover of the topics concerning parametrized complexity.

In parametrized complexity, a problem consists of the following triple:
- an input,
- a parameter that is a number computable from the input,
- and a question.

A typical example is the parametrized model checking problem where the input is a database $\mathbf{D}$ and a sentence q, the parameter is $|q|$ and the question asks whether $\mathbf{D} \models q$.

A parametrized problem is *Fixed Parameter Tractable* if, on input of size n and parameter k, it can be solved in time $f(k)n^c$ for some suitable computable function f and a constant c. We write that *a problem is in* FPT if it is Fixed Parameter Tractable.

The idea behind this definition is that for many scenarios (in particular for query evaluation problem in the database setting) it is preferable to have an algorithm working in time $2^k n$ rather than n^k.

A parametrized complexity turns out to be a very robust notion. There is a suitable notion of reduction, called FPT-reductions. It is the case that the class FPT is closed under FPT-reductions. There are some hard classes of parametrized problems that are also closed under FPT-reductions, but they contain problems that have no known FPT algorithms and that are believed to be different from the class FPT. The notion of completeness inside parametrized complexity is always understood modulo FPT-reductions. The class FPT in the parametrized complexity setting plays the role of P in the classical complexity.

An important hard class is denoted W[1]. It plays in the parametrized complexity setting the role of NP in the classical complexity. A typical problem which is complete for W[1] is the parametrized model checking problem for Conjunctive Queries (CQ) (see [57] for details). It takes as input a database $\mathbf{D}$ and a sentence $q \in$ CQ, as a parameter $|q|$ and asks whether $\mathbf{D} \models q$.

Another important hard class is denoted AW[*]. It plays in parametrized complexity setting the role of PSpace in the classical complexity. A typical problem which is complete for AW[*] is the parametrized model checking problem for First-Order logic (FO) (see [57] for details). It takes as input a database $\mathbf{D}$ and a sentence $q \in$ FO, as a parameter $|q|$ and asks whether $\mathbf{D} \models q$.

While we know the obvious inclusions of classes FPT $\subseteq$ W[1] $\subseteq$ AW[$*$], it is not know whether these inclusion are strict or not. Just as it is the case for P, NP and PSpace in the classical complexity, all these inclusion are strongly believed to be strict.

2.4 Logics

In Section 2.1 we already mentioned that typical query languages are logical formalisms. This is indeed going to be the case for Chapters 5 , 6 and 7, where we consider first-order (FO) and monadic second-order (MSO) logics. We assume that the reader is familiar with both these formalisms, so in the sequel we only give their brief overviews.

First-order logic (FO) is built from atomic formulas of the form $x = y$ or $R_i(x_1, \dots, x_{r_i})$ for some relation R_i and is closed under the usual Boolean connectives ($\neg, \vee, \wedge$) and existential and universal quantifications ($\exists, \forall$).

Monadic second-order logic (MSO) is an extension of FO with set variables X and set quantifications. Since we define queries as functions associating to a database a relation over its domain, when we talk about MSO queries, we disallow free set variables. To denote set variables we always use upper-case letters (X, Y) to distinguish them from first-order variables (x, y) and similarly we distinguish first-order quantification ($\exists x$) from second-order quantification ($\exists X$).

We use Greek letters like ϕ, φ, ψ, etc. to denote formulas.

When writing $\phi(\bar{x})$ we always mean that $\bar{x}$ are exactly the free first-order variables of ϕ. Given a database $\mathbf{D}$ and a tuple $\bar{a}$ of elements of $\mathbf{D}$, we write $\mathbf{D} \models \phi(\bar{a})$ if the formula ϕ is true in $\mathbf{D}$ after replacing its free variables with $\bar{a}$. As usual $|\phi|$ denotes the size of formula ϕ, which is the sum of the number of free variables, the number of quantifications, the number of Boolean connectives and the number of atomic formulas that appear in ϕ.

2.5 The core query problems

In this section we introduce a series of problems that are of central interest for this thesis. Some of them were already mentioned, but we repeat them here anyway to have all of them grouped in one place.

Fix a query language $\mathcal{L}$ and a class of databases $\mathcal{C}$. Each of the problems mentioned below is of the form: given a query $q \in \mathcal{L}$ (possibly with some additional constraints on q) and a database $\mathbf{D} \in \mathcal{C}$, compute the solution to the problem (which may depend on q and $\mathbf{D}$ only).

For the following sections assume $\mathcal{L}$ and $\mathcal{C}$ to be a fixed query language and a class of databases respectively.

2.5.1 Model checking problem

The *model checking problem of $\mathcal{L}$ over $\mathcal{C}$* is the computational problem of, given a **sentence** $q \in L$ and a database $\mathbf{D} \in \mathcal{C}$, decide whether $\mathbf{D} \models q$ or not.

2.5.2 Query evaluation problem

The *query evaluation problem of $\mathcal{L}$ over $\mathcal{C}$* is the computational problem of, given a query $q \in \mathcal{L}$ and a database $\mathbf{D} \in \mathcal{C}$, compute the set $q(\mathbf{D})$.

2.5.3 Query enumeration problem

The *enumeration of $\mathcal{L}$ over $\mathcal{C}$* is the computational problem of, given a query $q \in \mathcal{L}$ and a database $\mathbf{D} \in \mathcal{C}$, output the elements of $q(\mathbf{D})$ one by one with no repetitions.

In the enumeration scenario we denote the maximal time between any two consecutive outputs of elements from $q(\mathbf{D})$ as *the delay*. The goal of this thesis is to show pairs $\mathcal{L}, \mathcal{C}$ such that the enumeration of $\mathcal{L}$ over $\mathcal{C}$ allows this delay to be constant.

2.5.4 Query testing problem

The *testing of $\mathcal{L}$ over $\mathcal{C}$* is the computational problem of, given a query $q \in \mathcal{L}$ with k free variables and a database $\mathbf{D} \in \mathcal{C}$, allow to answer the following question: given a tuple $\bar{a}$ of k elements from $\mathbf{D}$, decide whether $\mathbf{D} \models q(\bar{a})$ or not.

We call $\bar{a}$ the *dynamical input* of the testing problem and an answer to the question whether $\mathbf{D} \models q(\bar{a})$ is true is called the *dynamical answer*.

2.5.5 The counting problem for a query

The *counting problem for $\mathcal{L}$ over $\mathcal{C}$* is the computational problem of, given a query $q \in \mathcal{L}$ and a database $\mathbf{D} \in \mathcal{C}$, compute the value $|q(\mathbf{D})|$.

2.5.6 The j-th solution problem for a query

The *j-th solution problem for $\mathcal{L}$ over $\mathcal{C}$* is the computational problem of, given a query $q \in \mathcal{L}$ with k free variables and a database $\mathbf{D} \in \mathcal{C}$, allow to answer the following problem: given $j \in \mathbb{N}$, compute the j-th element of the set $q(\mathbf{D})$ (with respect to some order $\leq$ on tuples of k nodes from $\mathbf{D}$) or say that such an element does not exist.

Similarly as it was the case for the testing problem, we call j the *dynamical input* of the testing problem and the returned j-th element (or the response that such an element does not exist) the *dynamical answer*.

Before we move to the examples, we would like to point out two things.

Notice that testing and j-th solution problems have a similar "dynamical" flavor: depending on q and database $\mathbf{D}$ they require the RAM algorithm to be able to answer certain questions in "real-time". The important common factor is that both the dynamical inputs have sizes independent from the size of the input database (size of $\bar{a}$ depends only on the arity of q and the value represented by j can be assumed to fit into a single register) and the same is true for the respective dynamical outputs. For this we write a *dynamical query problem* whenever we want to talk about one of these two problems but we do not want to specify about which one. Of course this could be generalized into a more broad and abstract notion, but as the only dynamical problems that we consider are testing and j-th solution, we stick to this very limited definition of dynamical query problems.

Moreover, the order in which we introduced these problems is not random.

• Model checking and query evaluation are the most classical query problems. Model checking, where the input query turns out to be a sentence, being the starting point (and a well established problem that plays a huge role in other fields of computer science, like for instance in verification) and query evaluation being its natural extension that is strongly database-oriented. In the following chapters we most often introduce them as known black-boxes with the exception of Chapter 7, where we address the model checking problem directly.

• We then introduce the enumeration problem as a refinement of the evaluation problem. As mentioned in Chapter 1, this is in fact the problem around which this thesis was concentrated from the very beginning and is in the center of interest of the author.

• The next to follow is the testing problem, which in all cases considered in this thesis can be solved using tools developed for the enumeration algorithms.

• We then move to the counting problem which is useful for two reasons: assuming that we have a bound on the delay, it allows to predict the time necessary for the enumeration of all the solutions to a query; and is usually a good start on the path to understanding and solving the j-th solution problem.

• Finally we have the j-th solution problem. While it has strong connections to the enumeration problem (where in a sense we may ask for a particular solution from the "middle" rather than wait for the enumeration process to reach it), it most of the time comes with a price: either the mentioned order $\leq$ turns out to be very unintuitive and the solution to the j-th solution problem can be used to generate random solutions (but without duplicates) rather than particular ones; or there is a loss in the complexity, where applying j-th solution consecutively to $1, 2, \ldots, |q(\mathbf{D})|$ enumerates $q(\mathbf{D})$ with delays between consecutive solutions strictly worse than the ones obtained when going directly via the enumeration procedure.

2.6 Examples

We devote this section to the analysis of a few simple examples that we find representative enough to point out the major difficulties that one challenges when dealing with problems described in Section 2.5.

Let σ be a relational schema containing relational symbol P and let $\mathbf{D}$ be a database over σ. Although formally we should write $P^{\mathbf{D}}$ to denote the actual relation P inside $\mathbf{D}$, for simplicity of the presentation we skip the superscript $\mathbf{D}$ and simply write P.

Before introducing the examples we would like to note that algorithms solving enumeration, testing and j-th solution problems are going to have two distinguished phases:

• the so called *preprocessing* that is performed before enumerating the first solution or reading the first dynamical input;

• followed by and enumeration/dynamical phase.

The reason for that is that without enhancing the input database with some additional navigational powers (the preprocessing phase is exactly for that), it would be very difficult to solve even the simplest cases. We typically aim at the preprocessing phase that is linear in $\|\mathbf{D}\|$ and constant/logarithmic delays/dynamical outputs.

Example 2.6.1 *Consider a database schema containing a single unary relational symbol P and the query $q(x) := P(x)$. Let $\mathbf{D}$ be the input of each of the algorithms.*

The enumeration of q is trivial: the algorithm goes through the list of nodes that belong to P, outputting current node at each step.

The counting problem is also immediate: the algorithm goes through the above list and counts its length.

The algorithm for the j-th solution problem first performs the counting procedure. Then, whenever it is given j, it first tests if there are in fact at least j solutions and only if this is the case, it performs some other steps. We do not mention this part in the following examples, but the reader should keep in mind that while being trivial, this step is always necessary. The "easiness" of the rest of the j-th solution problem depends on the way $\mathbf{D}$ is given. If the elements describing P are already in an array, whenever we are given j it is enough to output the j-th element of this array and that is all. If the description of P is a list, we first turn it into an array in linear time and then proceed as in the previous case.

What might come as a surprise, the testing problem seems to have the most demanding solution (although still simple). Given an element a of the domain it would be too time consuming to iterate *through the list describing P in the search of a. The solution is to enrich the record describing each node a of the domain (see Section 2.2 for details on records describing nodes) with an additional information saying whether $\boldsymbol{D} \models P(a)$ or not. This can easily be done with a linear pass over the list describing P and then using the direct addressing available in* RAM *machines. Having this done upfront, the testing problem is then trivial: given a we access record describing a in constant time and this record stores the desired answer to the question whether $\boldsymbol{D} \models P(a)$.*

Example 2.6.2 *Consider a database schema containing a single binary relational symbol R and the query $q(x, y) := \neg R(x, y)$. Let $\boldsymbol{D}$ be the input of each of the algorithms.*

The enumeration problem of q is this time not trivial as it happened to be the case for Example 2.6.1. A correct enumeration algorithm could be: iterate through all pairs $(a, b) \in D^2$ and for each pair test if $\boldsymbol{D} \models R(a, b)$. If this is the case, skip (a, b) and output it otherwise.

There are two problems with this approach. First of all, we do not yet know how to test whether $\boldsymbol{D} \models R(a, b)$ efficiently (and iteration through the list describing R is very costly). Moreover, the delay can be even quadratic in the size of D as R might be "dense" for big parts of D^2.

In order to enumerate this query efficiently, we need to do some additional preprocessing. *We first sort R lexicographically. This can be done on* RAM *machines in linear time as described in [40].*

For each pair (a, b) on the sorted list of R we compute the smallest (with respect to the lexicographical order) value $(c, d) = f(a, b)$ such that $(c, d) \notin R$ and $(c, d) > (a, b)$. For each value $f(a, b)$ obtained this way we compute the smallest element on R that is greater than $f(a, b)$ (we denote it with $g(f(a, b))$. This can easily be done in linear time by a single pass from the last element of R to its first one.

This concludes the preprocessing phase and note that it was performed in linear time. Having the above structure, the enumeration phase admits then only constant delay:

The algorithm starts with the smallest pair (e, e) and as long it is not the first element on R, it outputs the currently considered pair and moves to the next one in lexicographical order. The moment we see $(a, b) \in R$, we have in constant time access to $f(a, b)$ and $g(f(a, b))$. Note that: all the pairs inside interval $[(a, b), f(a, b))$ are in R and as such should not be output; all the pairs in the interval $[f(a, b), g(f(a, b)))$ are not in R and should be output; and $g(f(a, b)) \in R$. It is then enough to jump directly to $f(a, b)$ and keep outputting the current pair until we reach $g(f(a, b))$. We end up with a pair in R so we can then continue the same reasoning as we did for (a, b). It follows from the definitions of f and g pointers that we do not skip any solutions and the constant delay property is clear.

The counting problem turns out to be trivial also in this case: we count the number of elements in R (denoted $|R|$) in linear time and output $|D|^2 - |R|$.

To solve the testing problem we perform the same preprocessing as we did for the enumeration problem. For simplicity assume that the smallest element in the lexicographical order (a_1, a_1) is in R and set $f(a_1, a_1) = (a'_1, b'_1)$, $g(f(a_1, a_1)) = (a_2, b_2)$, $\dots$, $f(a_i, b_i) = (a'_i, b'_i)$, $g(f(a_i, b_i)) = (a_{i+1}, b_{i+1})$, $\dots$.

The other way of looking at the output of the preprocessing phase of the enumeration algorithm is that we are given a list L of pairs $((a'_1, b'_1), (a_2, b_2)), ((a'_2, b'_2), (a_3, b_3)), \dots, ((a'_i, b'_i), (a_{i+1}, b_{i+1})), \dots$ such that $q(\boldsymbol{D}) = [(a'_1, b'_1), (a_2, b_2)) \cup \dots \cup [(a'_i, b'_i), (a_{i+1}, b_{i+1})) \cup \dots$. Moreover, observe that this list is increasing (in the sense that $(a_i, b_i) < (a'_i, b'_i) < (a_{i+1}, b_{i+1})$ with respect to the lexicographical order).

This concludes the preprocessing phase and we now move to the testing phase. Given pair (a, b) we need to check whether $(a, b) \in [(a'_1, b'_1), (a_2, b_2)) \cup \dots \cup [(a'_i, b'_i), (a_{i+1}, b_{i+1})) \cup \dots$. Note that a single test if (a, b) is inside $[(a'_i, b'_i), (a_{i+1}, b_{i+1}))$ or is to the left or to the right of it (in terms of the lexicographical order) can be performed in constant time (by just comparing a to a'_i and a_{i+1} and similarly comparing b to b'_i and b_{i+1}). The testing phase is then easily doable in logarithmic time: we *apply a binary search on L that either finds an interval containing (a, b) or proves that such an interval does not exist.*

It should be mentioned that we do not know if a better testing time can be achieved and we strongly believe that it cannot.

To solve the j-th solution problem we perform the same preprocessing as we did for the testing problem. We also do the following:

With a single linear pass we can compute the sequence of values $s_1, \ldots, s_i, \ldots$ such that s_i is equal to the number of elements in $[(a'_i, b'_i), (a_{i+1}, b_{i+1}))$.

We set $S_0 = 0$ and with another linear pass we compute values $S_i = \sum_{1 \le j \le i} s_i$.

This concludes the preprocessing for the j-th solution algorithm and note that it was done in linear time.

We now switch to the dynamical phase: given j we find (again using a binary search) smallest i such that $S_{i-1} < j \le S_i$. It is then enough to output $(j - S_{i-1})$-th element of interval $[(a'_i, b'_i), (a_{i+1}, b_{i+1}))$. This being doable in constant time (by comparing a to a'_i and a_{i+1} and similarly comparing b to b'_i and b_{i+1} and knowing $|D|$), we altogether compute the dynamical answer to the j-th solution problem in logarithmic time.

Also this time it should be mentioned that we do not know if a better time for obtaining the dynamical answer can be achieved and we again strongly believe that it cannot.

What we wanted to show with the above examples is that all the presented solutions, in order for them to be effective, heavily relied on the *precomputation phases* that computed additional index structures that were later used. Only having these structures in hand, we were able to give the desired answers within satisfactory time constraints. Moreover, it is worth noticing that all the computed structure were linear in the size of the input database and were computable in linear time. Moreover, the enumeration phase and both dynamical phases (in both examples) did not modify these structures at all and were only traversing them in a specific manner.

The other point of these examples was to show that even very simple queries as we saw here (no quantification at all, each time just a single relation in the schema and its single appearance in the query) already required some nontrivial reasoning.

2.7 Complexity classes

In this section we introduce certain complexity classes over RAM machines that are of central interest for this thesis. Before defining them, we introduce notions that explain how we are going to relate these classes to problems mentioned in Section 2.5. While this is done in a purely abstract way in here, the reader is referred to the forthcoming sections for its application in practice.

Note that the use of symbols $\mathcal{A}$ and $\mathcal{S}_{\mathcal{A}}$ below is a rather ad-hoc choice. In the sequel $\mathcal{A}$ will always be defined as a class of RAM algorithms that have some particular properties (for instance as a class of algorithms that on input of size n run in time $O(n)$) and $\mathcal{S}_{\mathcal{A}}$ will always be defined as a class of problems, for which we will use a more self-explanatory names (for instance LINEAR-TIME).

Establishing this, we finally move to defining the common scenario present in the following sections.

Fix a query q and a computational problem P that associates to a database $\mathbf{D}$ an answer $P(q, \mathbf{D})$ (for particular problems see Section 2.5). Let $\mathcal{A}$ be a class of RAM machines.

We say that *the problem P for q is in the class $\mathcal{S}_{\mathcal{A}}$* if there exists a RAM algorithm $a_q \in \mathcal{A}$ that computes $P(q, \mathbf{D})$.

Let $\mathcal{C}$ be a class of databases. We say that *the problem P for q over $\mathcal{C}$ is in the class $\mathcal{S}_{\mathcal{A}}$* if there exists a RAM algorithm $a_q \in \mathcal{A}$ that, given database $\mathbf{D} \in \mathcal{C}$, computes $P(q, \mathbf{D})$.

Let $\mathcal{L}$ be a query language. We say that *the problem P of $\mathcal{L}$ is in the class $\mathcal{S}_\mathcal{A}$* if for every $q \in \mathcal{L}$ the problem P for q is in $\mathcal{S}_\mathcal{A}$.

Let $\mathcal{L}$ be a query language and $\mathcal{C}$ a class of databases. We say that *the problem P of $\mathcal{L}$ over $\mathcal{C}$ is in the class $\mathcal{S}_\mathcal{A}$* if for every $q \in \mathcal{L}$ the problem P for q over $\mathcal{C}$ is in $\mathcal{S}_\mathcal{A}$

Notice that in the last two definitions we do not say anything about whether the appropriate algorithm $a_q \in \mathcal{A}$ can be computed from the query $q \in \mathcal{L}$ or not. It only requires the existence of such an algorithm. If it happens to be the case that the algorithm a_q can be automatically obtained from q, we then say that the problem P on $\mathcal{L}$ is *generic*. This is the case for all the scenarios mentioned in this paper and all the scenarios that the author is aware of. The generic case is exactly the reason for introducing the notion of parametrized complexity (cf. Section 2.3): treating $|q|$ as a parameter we show (under some parametrized complexity assumptions) non membership of certain problems into some of the classes defined in the forthcoming sections.

In the sequel we define five classes of RAM algorithms (cf. $\mathcal{A}$ in the scenario above) and classes of query problems that they induce (cf. $\mathcal{S}_\mathcal{A}$). We do it only for a fixed query q and assume that the mentioned RAM algorithms always take a database $\mathbf{D}$ as input. These naturally extends (as explained in the scenario above) to situations where q is limited to a query language $\mathcal{L}$ and/or input databases $\mathbf{D}$ are limited to class $\mathcal{C}$.

2.7.1 The Linear-Time class

We say that a computational problem P is in Linear-Time if there exists a RAM algorithm that on input $\mathbf{D}$ computes $P(q, \mathbf{D})$ in time $O(\|\mathbf{D}\|)$.

The example problems that are in Linear-Time are:

• The counting problem for FO queries over the class of structures of bounded degree is in Linear-Time. (See Theorem 5.1.3)

• The counting problem for MSO queries over the class of structures of bounded treewidth is in Linear-Time. (See Theorem 6.1.4)

• The model checking problem for FO queries over the class of structures with bounded expansion is in Linear-Time. (See Theorem 7.1.1)

2.7.2 The evaluation class Linear-Eval

We say that the evaluation problem of q is in the class Linear-Eval if there exists a RAM algorithm that, given $\mathbf{D}$, computes $q(\mathbf{D})$ in time $O(\|\mathbf{D}\| + |q(\mathbf{D})|)$.

In the presence of Remark 2.7.1 we do not give any examples of problems from the Linear-Eval class, but we rather directly define the enumeration class and present our examples there.

2.7.3 The enumeration class Constant-Delay$_{lin}$

Recall the definition of the enumeration problem from Section 2.5.

We say that the enumeration problem of q is in the class Constant-Delay$_{lin}$ if it can be solved by a RAM algorithm which, on input $\mathbf{D}$, can be decomposed into two steps:

- a *precomputation* phase that is performed in time $O(\|\mathbf{D}\|)$,
- followed by an enumeration phases that outputs $q(\mathbf{D})$ with no repetitions and a constant delay between two consecutive outputs. The enumeration phase has a full read-access to the output of the precomputation phase, but it can use only a constant total amount of extra read-write memory.

The example problems that are in CONSTANT-DELAY$_{lin}$ are:

• The enumeration of FO queries over the class of structures of bounded degree is in CONSTANT-DELAY$_{lin}$. (See Theorem 5.1.1)

• The enumeration of MSO queries over the class of structures of bounded treewidth is in CONSTANT-DELAY$_{lin}$. (See Theorem 6.1.2)

• The enumeration of FO queries over the class of structures with bounded expansion is in CONSTANT-DELAY$_{lin}$. (See Theorem 7.1.2)

In the literature one can sometimes find a more liberal definition only requiring constant delay with no constraints on the memory. For more detailed comparison of the two models see Section 8.1.2.

Remark 2.7.1 *Notice that if the enumeration problem of q is in* CONSTANT-DELAY$_{lin}$, *then the evaluation problem of q can be solved in $O(\|\boldsymbol{D}\| + |q(\boldsymbol{D})|)$. In particular, if q is a sentence, then the model checking problem for q is in* LINEAR-TIME.

This also implies that CONSTANT-DELAY$_{lin}$ $\subseteq$ LINEAR-EVAL.

2.7.4 The answering classes CONSTANT-TIME$_{lin}$ and LOGARITHMIC-TIME$_{lin}$

Recall the definition of a dynamical query problem from Section 2.5 (which is either a testing or a j-th solution problem).

We say that a dynamical query problem of q is in the class CONSTANT-TIME$_{lin}$ (LOGARITHMIC-TIME$_{lin}$ respectively) if it can be solved by a RAM algorithm which, on input **D**, can be decomposed into two steps:

- a *precomputation* phase that is performed in time $O(\|\mathbf{D}\|)$,
- followed by an answering phases that for a dynamical input I computes the dynamical output $O(q, \mathbf{D}, I)$ in constant time (in time $O(\log \|\mathbf{D}\|)$ respectively). The answering phase has a full read-access to the output of the precomputation phase, but it can use only a constant total amount of extra read-write memory.

The example problems that are in in these classes are:

• The j-th solution problem for FO queries over the class of structures of bounded degree is in CONSTANT-TIME$_{lin}$. (See Theorem 5.1.4)

• The testing problem for FO queries over the class of structures with bounded expansion is in CONSTANT-TIME$_{lin}$. (See Corollary 7.3.1)

• The j-th solution problem for MSO queries over the class of structures of bounded treewidth is in LOGARITHMIC-TIME$_{lin}$. (See Theorem 6.1.5)

To avoid continuous repetitions, throughout this thesis we use word precomputation and *preprocessing* interchangeably.

2.8 Graphs

Graphs play an important role throughout this thesis. All the results from Chapters 5, 6 and 7 are of the form "For a class of databases with property X there exists an algorithm…". But each time the definition of the property X refers to a *graph of the structure representing the database* rather than the database directly. We now describe the way in which we view graphs. In Section 2.8.1 we introduce two notions of graphs of a relational structures. In Section 2.8.2 we define classes of graphs that are used in Chapters 5, 6 and 7. Later on, in Section 3.3, we compare the two notions of graphs of a structure with respect to the classes of graphs from Section 2.8.2.

Throughout this thesis, whenever we talk about graphs, we implicitly assume that their nodes may have different color and so a graph is always a particular example of a colored graph.

Definition 2.8.1 *A* (colored, oriented) graph $\boldsymbol{G}$ *is a relational structure* $\{V, E, P_1, \dots, P_n\}$ *where* E *is a binary relation and each* P_i *is a unary relation. We denote the elements of the structure as* nodes, *we call* E *the* edge *relation and each* P_i *a* color *relation. If* $(u, v) \in E$*, then we say that* there is an edge from u to v*. An* in-degree *of node* u *is* $|\{v : (v, u) \in E\}|$*. By* $\Delta^-(\boldsymbol{G})$ *we mean the maximal in-degree of a node of* $\boldsymbol{G}$*. We say that* u *and* v *are* adjacent *if there is an edge between them (no matter the orientation).*

If the relation E *is* symmetric *(i.e. for each* $(u, v) \in E$ *it is the case that also* $(v, u) \in E$*), we say that the graph is* unoriented. *Instead of the in-degree, we then speak of the* degree *of a node which is the number of nodes adjacent to it.*

A *path* in a graph is a sequence of adjacent nodes that uses each edge at most once. A path is *simple* if it contains each node no more than once. A graph is a *tree* if for every pair of distinct nodes of that graph there exists a unique path connecting those nodes.

Until Chapter 7 all the considered graphs are going to be unoriented. We elaborate more on oriented graphs in the Preliminaries Section 7.2 of Chapter 7.

2.8.1 Graphs of a structure

Definition 2.8.2 *The* Gaifman graph *of a relational structure* $\mathcal{A}$*, denoted by Gaifman*$(\mathcal{A})$*, is defined as follows: the set of vertices of Gaifman*$(\mathcal{A})$ *is* A *and there is an edge* (a, b) *in Gaifman*$(\mathcal{A})$ *iff there exists a relation* R_i *and a tuple* $t \in R_i$ *such that both* a *and* b *occur in* t*.*

Definition 2.8.3 *The* adjacency graph *of a relational structure* $\boldsymbol{D}$*, denoted by Adjacency*$(\boldsymbol{D})$*, is the following bipartite graph: the set of vertices of Adjacency*$(\boldsymbol{D})$ *is* $D \cup T$ *where* T *is the set of all tuples occurring in some relation of* $\boldsymbol{D}$*. For* $a \in D$ *and* $t \in T$ *there is an edge between* a *and* t *if node* a *belongs to tuple* t *in* $\boldsymbol{D}$*. We call nodes from* D real nodes *and nodes from* T tuple nodes.

The above definition of adjacency graph is a bit too weak. A node representing tuple (a, b) should be connected with a with an edge of *color* 1 and with b with an edge of color 2 to reflect the order of a and b inside this tuple. As we did not introduce colors on edges when defining graphs and we only care about the structure of the graph (that is its nodes and the way their are connected) for now, we stick to the above definition. We do not look into the details of adjacency graphs until Chapter 7, but there we look at these graphs in a slightly different (functional) way (see Section 7.2 for details).

2.8.2 Classes of graphs

In the forthcoming sections we define three classes of graphs that are going to be used in Chapters 5, 6 and 7. Recall from Section 2.3 that already a parametrized model checking problem for CQ is W[1] complete and so it is highly unlikely that it can be solved in FPT. As we already mentioned, the model checking problem being in LINEAR-TIME is just a starting point for all our investigations. Since in this thesis we do not want to limit the query languages below the power of first-order logic, it is a necessity to restrict the class of databases in order to obtain efficient solutions to the problems mentioned in Section 2.5. The following classes of graphs are going to be used to define these restrictions.

Bounded degree

Fix $d \in \mathbb{N}$. A graph **G** is *d-degree bounded* (or in other words *has d-bounded degree*) if every node of **G** has degree at most d. A class $\mathcal{C}$ of graphs is *d-degree bounded* (or equivalently *has d-bounded degree*) if every graph $\mathbf{G} \in \mathcal{C}$ is d-degree bounded.

Bounded treewidth

In order to define the next class of graphs, we need to recall the notion of a *tree decomposition* and *treewidth* first [43, 60]. It extends the definition of trees and is a well established and well studied notion in computer science. Interested reader is referred to the Chapter 11 of [33] for more detailed information on tree decompositions, treewidth and their properties. For the purpose of this thesis we just give a definition of tree decomposition and treewidth here. We use it only in Chapter 3 to compare the class of structures such that the underlying class of Gaifman graphs has bounded treewidth with the class of structures such that the underlying class of adjacency graphs has bounded treewidth. Although all the results from Chapter 6 talk about bounded treewidth, we do not work with this notion directly as Courcelle's theorem 6.3.1 allows us to immediately switch to trees.

This being said, we now move to the definitions of the tree decomposition and treewidth.

Definition 2.8.4 *Let* $\mathbf{G} = \{V, E\}$ *be a graph. A* tree decomposition *of* $\mathbf{G}$ *is a pair* (X, T)*, where* $X = \{X_1, \ldots, X_s\}$ *is a family of nonempty subsets of* V *and* T *is a tree whose set of nodes is exactly* X *and such that:*

- $\bigcup_{1 \leq i \leq s} X_i = V$*,*
- *for every edge* $(u, v) \in E$ *of* $\mathbf{G}$ *there exists* $1 \leq i \leq s$ *such that* $u, v \in X_i$*,*
- *if* $v \in X_i \cap X_j$ *for some* $1 \leq i, j \leq s$*, then for every* k *such that* X_k *is on the unique path from* X_i *to* X_j *in* T*, we have that* $v \in X_k$*.*

We refer to sets X_i *as* bags.

Intuitively, the tree decomposition of a graph tests how "close" a given graph is to actually being a tree. As we said before, for more details on the tree decompositions see Chapter 11 of [33].

A *width* of a tree decomposition (X, T), where $X = \{X_1, \ldots, X_s\}$, is the value $\max_{X_i \in X} |X_i| - 1$. A *treewidth* of a graph is the minimal width among all its tree decompositions.

Finally we are ready to define:

Fix $d \in \mathbb{N}$. A graph $\mathbf{G}$ *has* d*-bounded treewidth* if the treewidth of $\mathbf{G}$ is bounded by d. A class $\mathcal{C}$ of graphs *has* d*-bounded treewidth* if every graph $\mathbf{G} \in \mathcal{C}$ has d-bounded treewidth.

Bounded Expansion

The notion of a class of graphs with bounded expansion was introduced by Nešetřil and Ossona de Mendez in [53]. This is a rather broad class of graphs that generalizes not only both the class of graphs of bounded degree and the class of graphs of bounded treewidth, but for example also the class of planar graphs and any class of graphs excluding at least one minor. Despite being very general, it still admits strong algorithmic properties. Moreover, the notion of bounded expansion is very robust as it can be rephrased using quite a few equivalent characterizations. Interested reader is referred to [53] for a broad overview of all the equivalent characterizations of a class of graphs with bounded expansion and to [54] for a detailed description of the algorithmic properties that this classes of graphs admit.

This section is dedicated to defining the class of graphs with bounded expansion. We actually give its two characterizations here: the original definition using r-minors and the one using augmentations that will be heavily exploited in Chapter 7. For other properties of these classes that are going to be useful for this thesis, see Section 3.2.

We now turn to the definition of a class of graphs with bounded expansion.

To avoid confusion with the notion of size of a structure, we use the following notion in the case of graphs: we write $|\mathbf{G}|_{\text{VERT}}$ to denote the number of nodes of $\mathbf{G}$ (i.e. the size of V from the Definition 2.8.1), while we write $|\mathbf{G}|_{\text{EDGE}}$ to denote the number of edges of $\mathbf{G}$ (i.e. the size of E from the Definition 2.8.1).

Let $\mathbf{G} = (V, E)$ be an uncolored (we remove the colors as they do not affect the forthcoming definitions) and unoriented graph. For any node $v \in V$ and any $r \in \mathbb{N}$ we denote by $B_r(v)$ the r-ball around v, i.e. the set of nodes of $\mathbf{G}$ that are reachable from v by paths of lengths up to r. We say that a graph $\mathbf{H}$ is a r-minor of $\mathbf{G}$ if: all the nodes $v_1, \ldots, v_k$ of $\mathbf{H}$ are also nodes of $\mathbf{G}$; for $1 \leq i \leq k$ there exists a subset V_i of nodes of $\mathbf{G}$ that contains v_i, that is connected inside $\mathbf{G}$ and such that $V_i \subseteq B_r(v_i)$; the mentioned sets V_i (which we denote as *ball nodes*) are pairwise non-overlapping; and there is an edge between v_i and v_j in $\mathbf{H}$ iff there is an edge in $\mathbf{G}$ from a node of V_i to a node of V_j. The set of all r-minors of $\mathbf{G}$ is denoted by $\mathbf{G}\nabla r$. For a graph $\mathbf{G}$ the *greatest reduced average density (grad) of* $\mathbf{G}$ *with rank* r is:

$$\nabla_r(\mathbf{G}) = \max_{\mathbf{H} \in \mathbf{G}\nabla r} \frac{|\mathbf{H}|_{\text{EDGE}}}{|\mathbf{H}|_{\text{VERT}}}.$$

Having all this terminology we are ready to define:

Definition 2.8.5 *Let $\mathcal{C}$ be a class of graphs. We say that $\mathcal{C}$ has* bounded expansion *if there exists a function $f : \mathbb{N} \to \mathbb{R}$ such that for all graphs $\mathbf{G} \in \mathcal{C}$ and for all $r \in \mathbb{N}$ we have:*

$$\nabla_r(\mathbf{G}) \leq f(r).$$

In [53] several equivalent characterizations of bounded expansion were proven. We will almost never use in this thesis the initial definition as presented above, but the characterization exploiting the notion of "augmentations".

Let $\mathbf{G}$ be an oriented graph. A 1*-transitive fraternal augmentation of* $\mathbf{G}$ is any graph $\mathbf{H}$ with the same vertex set as $\mathbf{G}$ and the same colors of vertices, including all edges of $\mathbf{G}$ (with their orientation) and such that for any three vertices x, y, z of $\mathbf{G}$ we have the following:

(transitivity) if (x, y) and (y, z) are edges in $\mathbf{G}$, then (x, z) is an edge in $\mathbf{H}$,
(fraternity) if (x, z) and (y, z) are edges in $\mathbf{G}$, then at least one of the edges: (x, y), (y, x) is in $\mathbf{H}$,
(strictness) moreover, if $\mathbf{H}$ contains an edge that was not present in $\mathbf{G}$, then it must have been added by one of the previous two rules.

Note that the notion of 1-transitive fraternal augmentation is not a deterministic operation. Although transitivity induces precise edges, fraternity is underspecified and thus there can possibly be many different 1-transitive fraternal augmentations. We care here about choosing the orientations of the edges resulting from the fraternity rule in order to minimize the maximal in-degree.

For the sake of the presentation, later on in Chapter 7, we actually fix a deterministic algorithm computing a "good" choice of orientations of the edges induced by the fraternity property. This algorithm is described in details in [54], but its main property is contained in Lemma 7.2.1: for a class of graphs with bounded expansion this algorithm works in time linear in the size of the graph that is to be augmented. With this algorithm fixed, we can later on speak of **the** 1-transitive fraternal augmentation of $\mathbf{G}$.

But for now the nondeterministic process is going to suffice.

Let $\mathbf{G}$ be an unoriented graph. A *transitive fraternal augmentation* of $\mathbf{G}$ is any sequence $\mathbf{G} = \mathbf{G}_0 \subseteq \mathbf{G}_1 \subseteq \mathbf{G}_2 \subseteq \ldots$ such that for each $i \geq 1$ the graph $\mathbf{G}_{i+1}$ is a 1-transitive fraternal augmentation of $\mathbf{G}_i$.

Theorem 2.8.1 ([53]) *Let $\mathcal{C}$ be a class of graphs. The following conditions are equivalent:*

1. *$\mathcal{C}$ has bounded expansion,*

2. *there exists a function $\Gamma_{\mathcal{C}} : \mathbb{N} \to \mathbb{R}$ such that for each graph $\boldsymbol{G} \in \mathcal{C}$ there exists a transitive fraternal augmentation $\boldsymbol{G} = \boldsymbol{G}_0 \subseteq \boldsymbol{G}_1 \subseteq \boldsymbol{G}_2 \subseteq \ldots$ of $\mathbf{G}$ such that for each $i \geq 0$ we have $\Delta^-(\boldsymbol{G}_i) \leq \Gamma_{\mathcal{C}}(i)$.*

The above definition of the transitive fraternal augmentation is slightly imprecise. We start with an unoriented graph $\mathbf{G}$, while we need its oriented version to perform augmentations and so there is no real equality between $\mathbf{G}$ and $\vec{\mathbf{G}}_0$. What is in fact required by Theorem 2.8.1 is the existence of the orientation $\vec{\mathbf{G}}_0$ of $\mathbf{G}$ such that $\Gamma_{\mathcal{C}}(i)$ bounds the maximal in-degree of nodes from $\vec{\mathbf{G}}_i$. We allow this little confusion since in the sequel we will only talk about very specific classes of graphs. The scenario is going to be as follows: we first fix a relational schema. Then we consider classes of structures over this schema and look at their underlying classes of adjacency graphs. Since the schema is fixed, the maximal arity of tuple nodes is uniformly bounded and so we may assume that all the graphs are always oriented in such a way that there are edges from real nodes to tuple nodes.

Example 2.8.1 *Consider for instance a graph of degree d. Notice that a 1-transitive fraternal augmentation introduces an edge between nodes that were at distance at most 2 in the initial graph. Hence, when starting with a graph of degree d, we end up with a graph of degree at most d^2. This observation shows that the class of graphs of degree d has bounded expansion as witnessed by the function $\Gamma(i) = d^{2^i}$.*

Exhibiting the function Γ for the other examples of classes with bounded expansion mentioned in the introduction: bounded treewidth, planar graphs, graphs excluding at least one minor, requires more work [53].

3

Basic results

Contents

Just as we explained in the introduction, this chapter is a continuation of the Preliminaries Chapter 2. As Chapter 2 is already heavy with notation, we thought that it would be the best to leave it on a strict "definition" level. That is why all the simple observations and basic results that consider notions introduced in Chapter 2 are pushed to this chapter. Despite their simplicity, all these results find themselves being extensively used throughout the rest of this thesis, in particular in Chapters 5, 6 and 7. So without further ado, we move to these observations.

3.1 The query problems

Throughout this section the considered query problems are enumeration, testing, counting and j-th solution, just as they were described in Section 2.5. To present the results in a compact way we shall talk about a problem P meaning one of the above four problems. Similarly, when we talk about a solution from some class C, we mean one of the classes CONSTANT-DELAY$_{lin}$, LINEAR-TIME, CONSTANT-TIME$_{lin}$, LOGARITHMIC-TIME$_{lin}$ as they were introduced in Section 2.7. Although we do not precise it, the class C is assumed to be chosen from the appropriate subset that makes sense with respect to the chosen problem, that is if P is counting, then C is always LINEAR-TIME, if it is j-th solution, then C is CONSTANT-TIME$_{lin}$ or LOGARITHMIC-TIME$_{lin}$, etc.

We first present a series of simple facts that hold for any of the above problems. Since they are immediate consequences of the appropriate definitions, we just state the results here and either omit the proofs or present their brief sketches.

Fact 3.1.1 *Let P be a query problem. Let $\boldsymbol{D}$ and $\boldsymbol{D}'$ be two databases such that $\|\boldsymbol{D}'\| = O(\|\boldsymbol{D}\|)$ and let ϕ and ϕ' be two queries such that their sizes do not depend on $\|\boldsymbol{D}\|$ and $\|\boldsymbol{D}'\|$ respectively. If $\phi(\boldsymbol{D}) = \phi'(\boldsymbol{D}')$, then a solution from class C to the problem P for ϕ' and $\boldsymbol{D}'$ gives a solution from C to the problem P for ϕ and $\boldsymbol{D}$.*

Fact 3.1.2 *Let P be a query problem. Let **D** be a database and let $\phi(\bar{x})$ be a query. Let $\phi'(\bar{x}\bar{y})$ be a query such that ϕ' logically implies that $\bar{y} = f(\bar{x})$ ($\bar{y}$ functionally depends on $\bar{x}$ for some computable function f). If **D** $\models \phi(\bar{x}) \leftrightarrow \phi'(\bar{x}f(\bar{x}))$, then a solution from class C to the problem P for ϕ' and **D** gives a solution from C to the problem P for ϕ and **D**.*

A particular use for the Fact 3.1.2 can be when ϕ' logically implies that two of its free variables are equal, but of course there are more complicated situations too.

Fact 3.1.3 *Let P be a query problem. Let **D** be a database and let $\phi = \bigvee_{i\in I} \phi_i$ be a query such that all the ϕ_i are pairwise mutually exclusive and $|I|$ does not depend on $\|\mathbf{D}\|$. Then solutions from class C to the problem P for each ϕ_i and **D** give a solution from C to the problem P for ϕ and **D**. In the particular case when P is the j-th solution problem, we additionally assume that for each $i \in I$ we have a solution to the counting problem for ϕ_i and **D** that is in* LINEAR-TIME*.*

PROOF While the other cases are immediate, let us have a closer look at the special case when P is the j-th solution problem.

The preprocessing phase for ϕ is composed from the preprocessing phases of all ϕ_i together with computing numbers $s_i = \sum_{j\le i} |\phi_i(\mathbf{D})|$. This can altogether be achieved in linear time using the additional assumption about the counting problems. Additionally (just for the sake of readability) set $s_0 = 0$.

Then the j-th solution phase works as follows: given $j \in \mathbb{N}$, find in constant time i such that $s_{i-1} < j \le s_i$ (if such an i does not exists, it responds that there are less than j solutions in total). The j-th solution is then $(j - s_{i-1})$-th solution to ϕ_i and **D**.

■

It is worth mentioning that the use of Fact 3.1.3 has an impact on the order in which the solutions to ϕ are enumerated and with respect to which the j-th solution is returned (in the resulting order all the solutions to ϕ_i precede solutions to ϕ_j if $i < j$ and the order inside each $\phi_i(\mathbf{D})$ is inherited from the appropriate algorithm for ϕ_i and **D**).

3.1.1 The enumeration problem

We now present a fact that is a refinement of Fact 3.1.3 in the special case when P is the enumeration problem. It says that, in the setting of Fact 3.1.3, if the output to the sub-queries is enumerated with respect to some fixed order, then to output to the main query can also be enumerated with respect to this order (and there is no additional overhead). Moreover, the assumption that the sub-queries are mutually exclusive can be omitted in this case. Although we state it for just two queries here, it can easily be extended to an arbitrary number of fixed queries by a recursive application.

Fact 3.1.4 *Let **D** be a database. Let $\phi(\bar{x})$ and $\phi'(\bar{x})$ be two queries with k free variables and let $<$ be an order on k-tuples of elements from **D**. Then* CONSTANT-DELAY$_{lin}$ *enumeration procedures for ϕ over **D** and for ϕ' over **D** that output their answers in increasing order relative to $<$, give a* CONSTANT-DELAY$_{lin}$ *solution for $\phi \vee \phi'$ over **D** that enumerates the answers set in increasing order relative to $<$.*

PROOF The CONSTANT-DELAY$_{lin}$ enumeration procedure for $\phi \vee \phi'$ over **D** resembles a procedure of merging two sorted list. The preprocessing consists of the preprocessing phases for ϕ over **D** and for ϕ' over **D**.

The enumeration phase keeps two values - the smallest element from $\phi(\mathbf{D})$ that was not yet output and similarly the smallest element from $\phi'(\mathbf{D})$ that was not yet output. It then outputs the smaller of the two values and replaces it in constant time with the next element from the appropriate set (using the enumeration procedure for one of the two problems). In case the elements are equal, the value is output once and both stored values are replaced with their appropriate successors.

■

3.1.2 The j-th solution problem

We now present a fact that finds its use for the j-th solution problem. This is somewhat a continuation of Fact 3.1.3 in the spirit of Fact 3.1.4 for the enumeration problem. It says that, in the setting of Fact 3.1.3, if the j-th solution for the sub-queries is returned with respect to some fixed order, then the j-th solution for the main query can also be returned with respect to this order (but this comes with a cost of logarithmic overhead during the dynamical phase). Note that the assumption that the sub-queries are mutually exclusive is still necessary in this case. Although we state it for just two queries here, it can be extended to an arbitrary number of fixed queries. This time a recursive application would result in a poly-logarithmic overhead, but one can verify that in order to handle k queries it is enough to replace $\frac{j}{2}$ with $\frac{j}{k}$ in the proof of Fact 3.1.5 and the reasoning still holds.

Fact 3.1.5 *Let $\mathbf{D}$ be a database. Let $\phi(\bar{x})$ and $\phi'(\bar{x})$ be two mutually exclusive queries with k free variables and let $<$ be an order on k-tuples of elements from $\mathbf{D}$. Then j-th solution procedures from class C for ϕ over $\mathbf{D}$ and for ϕ' over $\mathbf{D}$ that both return their answers with respect to the order $<$, give a solution from C' for $\phi \vee \phi'$ over $\mathbf{D}$ that returns its answers with respect to the order $<$ and C' is such that its preprocessing phase has the same complexity as the preprocessing phase of C, but its dynamical phase is worse by a logarithmic factor.*

PROOF As expected, the precomputation phase for $\phi \vee \phi'$ consists of the precomputation phases of the j-th solution procedures for ϕ and for ϕ'.

The j-th solution phase is then a recursive procedure that at each step divides the input value j by 2, resulting in the logarithmic overhead as explained in the statement of this fact. The case when $|\phi \vee \phi'(\mathbf{D})| < j$ is not considered in the sequel, but it should be clear from the proof that this case does not introduce any additional difficulties.

We first make a simple observation. There are two ways of looking at the j-th solution to ϕ: either it is a solution such that there are exactly $j-1$ smaller solutions or this is a solution such that there are exactly $|\phi(\mathbf{D})| - j$ greater solutions. We will later on make use of the latter point of view.

We now introduce a bit of notation. A *s-shift of ϕ* is a procedure that given j outputs the $(j+s)$-th solution to ϕ. Note that this procedure is trivially obtained from the j-th solution procedure for ϕ and that it does not require any additional preprocessing except from the one performed by the j-th solution procedure for ϕ. We denote the s-shift of ϕ with s-ϕ and we call the first s solutions to ϕ the *shifted solutions*.

Assume j comes as an input for the dynamical phase of $\phi \vee \phi'$. Let $\bar{v}$ be the $\frac{j}{2}$-th solution to ϕ and $\bar{v}'$ be the $\frac{j}{2}$-th solution to ϕ'. Assume that $\bar{v} < \bar{v}'$ (the case when $\bar{v} > \bar{v}'$ is fully symmetric and is handled by analogous reasoning).

We now show that the j-th solution to $\phi \vee \phi'$ is in that case the $\frac{j}{2}$-th solution to $(\frac{j}{2}$-$\phi) \vee \phi'$ (we denote the latter solution with $\bar{v}''$).

First note that there are exactly $|\phi(\mathbf{D})| - \frac{j}{2} + |\phi'(\mathbf{D})| - \frac{j}{2} = |\phi(\mathbf{D})| + |\phi'(\mathbf{D})| - j$ solutions to $(\frac{j}{2}$-$\phi) \vee \phi'$ that are greater than $\bar{v}''$. Indeed: $(\frac{j}{2}$-$\phi)$ has in total $|\phi(\mathbf{D})| - \frac{j}{2}$ solutions, ϕ' has in total $|\phi'(\mathbf{D})|$ solutions, $(\frac{j}{2}$-$\phi)$ and ϕ' are mutually exclusive and $\bar{v}''$ is the $\frac{j}{2}$-th solution to $(\frac{j}{2}$-$\phi) \vee \phi'$.

In view of the above it is enough to prove that $\bar{v}''$ is greater than all the $\frac{j}{2}$ shifted solutions to ϕ. If this would in fact be the case, then all the solutions to $\phi \vee \phi'$ that are greater than $\bar{v}''$ would exactly be the solutions to $(\frac{j}{2}\text{-}\phi) \vee \phi'$ that are greater than $\bar{v}''$ and we already know that there are $|\phi(\mathbf{D})| + |\phi'(\mathbf{D})| - j$ of them. This would then prove that $\bar{v}''$ is the desired j-th solution to $\phi \vee \phi'$.

The proof that $\bar{v}''$ is greater than all the $\frac{j}{2}$ shifted solutions to ϕ is now a simple case analysis:

• if $\mathbf{D} \models \phi(\bar{v}'')$, then clearly $\bar{v}''$ is greater than all the shifted solutions as we always shift the smallest ones;

• if $\bar{v}' = \bar{v}''$, then it is also the case since $\bar{v} < \bar{v}'$ and $\bar{v}$ was the greatest among all the shifted solutions;

• if $\mathbf{D} \models \phi'(\bar{v}'')$ but $\bar{v}'' < \bar{v}'$, then there exists a solution to $\frac{j}{2}\text{-}\phi$ that is smaller than $\bar{v}''$ and so $\bar{v}''$ is also in this case greater than all the shifted solutions.

As we said earlier, the problem of finding j-th solution to $\phi \vee \phi'$ was reduced to the problem of finding $\frac{j}{2}$-th solution to $(\frac{j}{2}\text{-}\phi) \vee \phi'$. Since the preprocessing phase for $(\frac{j}{2}\text{-}\phi)$ is the same as the one for ϕ and its dynamical phase has the same complexity, with two calls to the j-solutions dynamical procedures for ϕ and ϕ' respectively, we managed to reduce the value of j by half. Recursive application of this procedure indeed results in a j-th solution phase to $\phi \vee \phi'$ that has a logarithmic overhead with respect to the j-th solution phases of ϕ and ϕ'.

■

3.2 Bounded expansion

We state here a simple fact about classes of graphs with bounded expansion.

It is an immediate consequence of the characterization of this class from Point 2 of Theorem 2.8.1.

Fact 3.2.1 *If class $\mathcal{C}$ of graphs has bounded expansion, then the class $\mathcal{C}$' of the 1-transitive fraternal augmentations of graphs from $\mathcal{C}$ also has bounded expansion.*

3.3 Gaifman vs adjacency

Fix class $\mathcal{C}$ of databases. In this section we compare the class $\mathcal{C}$' of Gaifman graphs of structures from $\mathcal{C}$ with the class $\mathcal{C}$" of adjacency graphs of structures from $\mathcal{C}$. We do it with respect to the classes of graphs that were defined in Section 2.8.2. It turns out that in all cases we actually need to fix a schema to have an "equivalence" of $\mathcal{C}$' and $\mathcal{C}$".

For the purpose of this section we introduce one new notion:

Let $\mathbf{D}$ be a relational structure over signature σ, let R be a relation from σ of arity r and let $t \in R$ be a tuple of R in $\mathbf{D}$. The *effective arity of* t is the number of different elements in t.

3.3.1 Bounded degree

This section is devoted to proving the following theorem:

Theorem 3.3.1 *Let σ be a fixed schema. Let $\mathcal{C}$ be a class of structures over σ. The class $\mathcal{C}$' of Gaifman graphs of structures from $\mathcal{C}$ has bounded degree iff the class $\mathcal{C}$" of adjacency graphs of structures from $\mathcal{C}$ has bounded degree.*

The reason why we need to fix the schema is shown in Example 3.3.1. It is worth to note that the right-to-left direction (the implication that the bounded degree of adjacency graphs implies the bounded degree of Gaifman graphs) does not use the fixed schema assumption.

PROOF [of Theorem 3.3.1] We start with the left-to-right direction, that is the implication that the bounded degree of graphs from $\mathcal{C}$' implies bounded degree of graphs from $\mathcal{C}$". Assume that graphs from $\mathcal{C}$' have degree bounded by k.

Fix structure $\mathbf{D}$ from $\mathcal{C}$. Let u be a node of Gaifman($\mathbf{D}$) and let $v_1, \ldots, v_s$ be all its neighbors (recall that $s < k$). From the definition of the Gaifman graph, if u belongs to a tuple t of some relation from $\mathbf{D}$, then all the nodes from t are in the set $\{u, v_1, \ldots, v_s\}$. Since the schema is fixed, there are only constantly many tuples t that contain u and this constant gives us the bound on the degree of Adjacency($\mathbf{D}$).

This concludes the first direction of the proof.

We now move to the right-to-left direction, that is the implication that the bounded degree of graphs from $\mathcal{C}$" implies bounded degree of graphs from $\mathcal{C}$'. Assume that graphs from $\mathcal{C}$" have degree bounded by k.

Fix structure $\mathbf{D}$ from $\mathcal{C}$. Note that each tuple node t has effective arity bounded by k. Moreover, each real node appears in up to k tuples and so the degree of Gaifman($\mathbf{D}$) is bounded by k^2 (each node is in up to k tuples, each time with up to $k-1$ different nodes).

This concludes the second direction and the proof of Theorem 3.3.1.

■

Example 3.3.1 *Consider a class $\mathcal{C}$ of the following structures:*

$\boldsymbol{D}_i$ has universe $\{u, v\}$ and i binary relations $P_1, \ldots, P_i$, each containing a single tuple (u, v). Now:

- *graph Gaifman($\boldsymbol{D}_i$) has degree 1 for each i;*
- *graph Adjacency($\boldsymbol{D}_i$) has degree i (this is the degree of both nodes u and v) and so the class $\{$Adjacency$(\boldsymbol{D}_i)\}_{i \in \mathbb{N}}$ has unbounded degree.*

3.3.2 Bounded treewidth

This section is devoted to proving the following theorem:

Theorem 3.3.2 *Let σ be a fixed schema. Let $\mathcal{C}$ be a class of structures over σ. The class $\mathcal{C}$' of Gaifman graphs of structures from $\mathcal{C}$ has bounded treewidth iff the class $\mathcal{C}$" of adjacency graphs of structures from $\mathcal{C}$ has bounded treewidth.*

The reason why we need to fix the schema is shown in Example 3.3.2. It is worth to note that the left-to-right direction (the implication that the bounded treewidth of Gaifman graphs implies the bounded treewidth of adjacency graphs) does not use the fixed schema assumption, which is in opposite to what we have seen in the bounded degree case.

PROOF [of Theorem 3.3.2] We start with the left-to-right direction, that is the implication that the bounded treewidth of graphs from $\mathcal{C}$' implies bounded treewidth of graphs from $\mathcal{C}$". Assume that graphs from $\mathcal{C}$' have treewidth bounded by k.

Fix structure $\mathbf{D}$ from $\mathcal{C}$. Let T be a tree decomposition of Gaifman($\mathbf{D}$) of width $k+1$. Let t be a tuple from a relation from $\mathbf{D}$. By the definition of the Gaifman graph, nodes that appear in t form a clique in Gaifman($\mathbf{D}$). It is a well known fact that this implies that there is at least one bag X_t in T that contains all the nodes from t.

This simple observation tells us how to construct a tree decomposition T' of Adjacency($\mathbf{D}$) that has a small (to be precise: bounded by $k+2$) width:

- T' contains T;

• moreover, for each tuple t we fix a bag X_t that contains all the nodes from t and add $X'_t := X_t \cup \{t\}$ as a child of X_t.

Clearly points 1 and 2 of the definition of tree decomposition are satisfied for Adjacency($\mathbf{D}$) and T'. But the last point is also true as each tuple node appears in exactly one bag and real nodes are covered by the fact that T was a proper tree decomposition.

This concludes the first direction of the proof.

We now move to the right-to-left direction, that is the implication that the bounded treewidth of graphs from $\mathcal{C}$" implies bounded treewidth of graphs from $\mathcal{C}$'. Assume that graphs from $\mathcal{C}$" have treewidth bounded by k.

Fix structure $\mathbf{D}$ from $\mathcal{C}$. Let T be a tree decomposition of Adjacency($\mathbf{D}$) of treewidth $k+1$. Let s be the maximal arity of a relational symbol from σ.

We now construct a tree decomposition T' of Gaifman($\mathbf{D}$) that has a small (to be precise: bounded by $k+s+1$) treewidth:

The construction replaces each occurrence of a tuple node t with the set of real nodes that appear in t. Clearly points 1 and 2 of the definition of tree decomposition are satisfied for Gaifman($\mathbf{D}$) and T'. The fact that the last point also holds is that from point 2 for each node v in t there has to be a bag in T that contains both t and v. As sub-graphs induced by v and by t separately were sub-trees of T and v and t share a bag, sub-graph induced by v is in fact a sub-tree of T'.

This concludes the second direction and the proof of Theorem 3.3.2.

■

Example 3.3.2 *Consider a class $\mathcal{C}$ of the following structures:*

$\boldsymbol{D}_i$ has universe $\{1, \dots, i\}$ and a single relation P_i with just one tuple $(1, \dots, i)$. Now:

• *graph Adjacency($\boldsymbol{D}_i$) is a tree, so it has treewidth 1;*

• *graph Gaifman($\boldsymbol{D}_i$) is an i-clique and so the class $\{\text{Gaifman}(\boldsymbol{D}_i)\}_{i \in \mathbb{N}}$ has unbounded treewidth.*

3.3.3 Bounded expansion

This section is devoted to proving the following theorem:

Theorem 3.3.3 *[56] Let σ be a fixed schema. Let $\mathcal{C}$ be a class of structures over σ. The class $\mathcal{C}$' of Gaifman graphs of structures from $\mathcal{C}$ has bounded expansion iff the class $\mathcal{C}$" of adjacency graphs of structures from $\mathcal{C}$ has bounded expansion.*

The reason why we need to fix the schema is shown in Example 3.3.2 from the previous section (the class of all cliques obviously does not have bounded expansion). It is worth to note that the left-to-right direction (the implication that the bounded expansion of the class of Gaifman graphs implies the bounded expansion of the class of adjacency graphs) does not use the fixed schema assumption, just as it was the case for bounded treewidth.

In this section we interchangeably use the characterizations of bounded expansion from Definition 2.8.5 and from Point 2 of Theorem 2.8.1, but we always explicitly say which one are we using at a given moment.

Recall the definition of Adjacency($\mathbf{D}$) from Section 2.8.1. In particular, nodes of Adjacency($\mathbf{D}$) are divided into two sets: D – containing real nodes and T – containing tuple nodes. Note that Adjacency($\mathbf{D}$) is a bipartite graph (neither any two nodes from D nor any two nodes from T are ever connected) and the maximal in-degree of a node from T is bounded by the maximal arity of a relation in $\mathbf{D}$.

Lemma 3.3.1 *Let $\mathcal{C}$ be class of relational structures and let $\mathcal{C}$' be the underlying class of Gaifman graphs of structures from $\mathcal{C}$. If $\mathcal{C}$' has bounded expansion, then there exists a constant k such that for any structure $\mathbf{D} \in \mathcal{C}$ and for any tuple $t \in \mathbf{D}$ the effective arity of t is less than k.*

PROOF Fix class $\mathcal{C}$ of structures and let $\mathcal{C}$' be the class of Gaifman graphs of structures from $\mathcal{C}$. Let f be the function from Definition 2.8.5 witnessing the fact that $\mathcal{C}$' has bounded expansion.

Set $k = 2f(0)$. Let $\mathbf{D} \in \mathcal{C}$ and t be an arbitrary tuple from $\mathbf{D}$ with effective arity s. Let $A = \{a_1, \dots, a_s\}$ be the set of different elements in t. By the definition of Gaifman($\mathbf{D}$), vertices from A are pairwise connected. Consider the 0-minor $\mathbf{H}$ of Gaifman($\mathbf{D}$) induced by A. We have that $\frac{|\mathbf{H}|_{\text{EDGE}}}{|\mathbf{H}|_{\text{VERT}}} = \frac{|A| \cdot (|A|-1)}{2|A|} = \frac{s-1}{2}$. By definition $\nabla_0(\text{Gaifman}(\mathbf{D})) \geq \frac{|\mathbf{H}|_{\text{EDGE}}}{|\mathbf{H}|_{\text{VERT}}} \geq \frac{s-1}{2}$. On the other hand the characterization of bounded expansion from Definition 2.8.5 gives $f(0) \geq \nabla_0(\text{Gaifman}(\mathbf{D}))$ and we have $k > s$ as desired.

■

Theorem 3.3.3 is a consequence of Proposition 3.3.1 and Proposition 3.3.2.

Proposition 3.3.1 *Let $\mathcal{C}$ be a class of structures such that the class $\mathcal{C}$' of Gaifman graphs of structures from $\mathcal{C}$ has bounded expansion. Then the class $\mathcal{C}$" of adjacency graphs of structures from $\mathcal{C}$ has bounded expansion.*

PROOF

We use the characterization of class of graphs with bounded expansion from Definition 2.8.5. Then Proposition 3.3.1 is the direct consequence of the following lemma:

Lemma 3.3.2 *Let $\mathcal{C}$ be a class of structures such that the class $\mathcal{C}$' of Gaifman graphs of structures from $\mathcal{C}$ has bounded expansion. There exists a constant k such that for any structure $\mathbf{D} \in \mathcal{C}$ and for any natural number r we have that $\nabla_r(\text{Adjacency}(\mathcal{A})) \leq \nabla_r(\text{Gaifman}(\mathcal{A})) + k$.*

PROOF Fix class $\mathcal{C}$ of structures such that the class $\mathcal{C}$' of Gaifman graphs of structures from $\mathcal{C}$ has bounded expansion.

Let k be the constant given by Lemma 3.3.1.

Let $\mathbf{D} \in \mathcal{C}$ and let r be a natural number and $\mathbf{H}$ be a r-minor of Adjacency($\mathbf{D}$). From $\mathbf{H}$ we construct a graph $\mathbf{H}'$ which is a r-minor of Gaifman($\mathbf{D}$) and such that:

$$\frac{|\mathbf{H}|_{\text{EDGE}}}{|\mathbf{H}|_{\text{VERT}}} \leq \frac{|\mathbf{H}'|_{\text{EDGE}}}{|\mathbf{H}'|_{\text{VERT}}} + k.$$

This immediately yields the result.

Recall from Section 7.2.3 that Adjacency($\mathbf{D}$) is a bipartite graph that contains *tuple* nodes and *real* nodes and such that neither any two tuple nodes nor any two real nodes are connected. By the definition of constant k from Lemma 3.3.1, each tuple node has up to k neighbors in Adjacency($\mathbf{D}$).

Consider a node v of $\mathbf{H}$. By construction, v is derived from a ball node $S_v \subseteq B_r(v)$ of Adjacency($\mathbf{D}$).

If S_v contains no real nodes, then it simply is a single tuple node. As each tuple node has up to k neighbors in Adjacency($\mathbf{D}$), then if S_v contains no real nodes, v has at most k neighbors in $\mathbf{H}$. Let X be the set of all such nodes v in $\mathbf{H}$.

Otherwise, let S'_v be the set of real nodes of S_v. By definition S'_v is not empty and it is easy to verify that it forms a connected component contained in $B_{\frac{r}{2}}(v)$ in Gaifman($\mathbf{D}$): for every $u \in S'_v$ the longest path from v to u in S_v is $v = u_1, t_{(1,2)}, u_2, t_{(2,3)}, \dots, t_{(\frac{r}{2}-1,\frac{r}{2})}, u_{\frac{r}{2}} = u$, where each $t_{(i,i+1)}$ is a tuple node. By the definition of Gaifman($\mathbf{D}$) we have that u_i is connected to u_{i+1} (which is witnessed by $t_{(i,i+1)}$), which yields that $v = u_1, u_2, \dots, u_{\frac{r}{2}} = u$ is a path in S'_v. Let $\mathbf{H}'$ be the r-minor of Gaifman($\mathbf{D}$) constructed from the elements S'_v, where $v \notin X$.

By construction we have : $|\mathbf{H}'|_{\text{VERT}} + |X| = |\mathbf{H}|_{\text{VERT}}$.

Consider now an edge (u, v) in $\mathbf{H}$ where both u and v are not in X. This means that there is an edge (a, b) in Adjacency($\mathcal{A}$) with $a \in S_u$ and $b \in S_v$. As Adjacency($\mathcal{A}$) is bipartite, this means that a is a real node and b a tuple node (or vice versa). Wlog assume that a is the real node. As v is not in X, S_v contains a real node b' adjacent to b. Hence b witnesses that (a, b') is an edge in Gaifman($\mathbf{D}$) and so (u, v) is an edge in $\mathbf{H}'$. As we have seen that there are at most $k|X|$ edges (u, v) in $\mathbf{H}$ where either u or v belongs to X, we get: $|\mathbf{H}|_{\text{EDGE}} \leq |\mathbf{H}'|_{\text{EDGE}} + k|X|$.

Summing up we get:

$$\frac{|\mathbf{H}|_{\text{EDGE}}}{|\mathbf{H}|_{\text{VERT}}} \leq \frac{|\mathbf{H}'|_{\text{EDGE}}+k|X|}{|\mathbf{H}'|_{\text{VERT}}+|X|} \leq \frac{|\mathbf{H}'|_{\text{EDGE}}}{|\mathbf{H}'|_{\text{VERT}}+|X|} + \frac{k|X|}{|\mathbf{H}'|_{\text{VERT}}+|X|} \leq \frac{|\mathbf{H}'|_{\text{EDGE}}}{|\mathbf{H}'|_{\text{VERT}}} + k.$$

as desired.

■

As we said before, Lemma 3.3.2 concludes the proof of Proposition 3.3.1.

■

Proposition 3.3.2 *Let σ be a fixed schema. Let $\mathcal{C}$ be a class of structures over σ such that the class $\mathcal{C}$' of adjacency graphs of structures from $\mathcal{C}$ has bounded expansion. Then the class $\mathcal{C}$" of Gaifman graphs of structures from $\mathcal{C}$ has bounded expansion.*

PROOF This time we use characterization of class of graphs with bounded expansion from Point 2 of Theorem 2.8.1. Let $\mathcal{C}$' be witnessed by the function $\Gamma_{\mathcal{C}'}$ as described in this theorem. We show that function $\Gamma_{\mathcal{C}''}$ defined as:

$$\Gamma_{\mathcal{C}''}(i) := \Gamma_{\mathcal{C}'}(i+1) \text{ for } 0 \leq i,$$

witnesses the fact that $\mathcal{C}$" has bounded expansion.

Fix $\mathbf{D} \in \mathcal{C}$. $\mathcal{C}'$ has bounded expansion witnessed by $\Gamma_{\mathcal{C}'}$, so the transitive fraternal augmentation Adjacency($\mathbf{D}$) $= \mathbf{G} = \mathbf{G}_0 \subseteq \mathbf{G}_1 \subseteq \mathbf{G}_2 \subseteq \ldots$ is such that for each $i \geq 0$ we have $\Delta^-(\mathbf{G}_i) \leq \Gamma_{\mathcal{C}'}(i)$.

Following the discussion from the end of Section 2.8.2 we assume that $\mathbf{G}_0$ is oriented in such a way that all the edges point from real nodes to tuple nodes.

Let a, b be any two nodes of Gaifman($\mathbf{D}$) such that there is an edge between a and b. By the definition of Gaifman graph there is a tuple t in $\mathbf{D}$ containing both a and b. This means that in Adjacency($\mathbf{D}$) there is an edge from a to t and from b to t. This shows that a and b form a fraternal pair of nodes in Adjacency($\mathbf{D}$) and that they are connected in the graph $\mathbf{G}_1$ that is the 1-transitive fraternal augmentation of Adjacency($\mathbf{D}$). As (a, b) was an arbitrary pair of nodes connected in Gaifman($\mathbf{D}$) we see that $\mathbf{G}_1$ contains as a subgraph an oriented copy of Gaifman($\mathbf{D}$). We denote this subgraph with $\mathbf{H}_0$. As $\Delta^-(\mathbf{G}_1) \leq \Gamma_{\mathcal{C}'}(1)$, clearly $\Delta^-(\mathbf{H}_0) \leq \Gamma_{\mathcal{C}'}(1)$.

Observe now that $\mathbf{G}_2$ contains as a subgraph some 1-transitive fraternal augmentation of $\mathbf{H}_0$ (as $\mathbf{H}_0 \subseteq \mathbf{G}_1$ and $\mathbf{G}_2$ is a 1-transitive fraternal augmentation of $\mathbf{G}_1$). We denote this 1-transitive fraternal augmentation of $\mathbf{H}_0$ with $\mathbf{H}_1$ and note that $\Delta^-(\mathbf{H}_1) \leq \Delta^-(\mathbf{G}_2) \leq \Gamma_{\mathcal{C}'}(2)$. Continuing this reasoning we get a transitive fraternal augmentation $\mathbf{H} = \mathbf{H}_0 \subseteq \mathbf{H}_1 \subseteq \mathbf{H}_2 \subseteq \ldots$ such that $\mathbf{H}_0$ is an orientation of Gaifman($\mathbf{D}$) and for $i \geq 0$ we have $\Delta^-(\mathbf{H}_i) \leq \Delta^-(\mathbf{G}_{i+1}) \leq \Gamma_{\mathcal{C}'}(i+1) = \Gamma_{\mathcal{C}''}(i)$. This concludes the proof.

■

4

State of the art

Contents

In this chapter we present a current state of the art concerning enumeration algorithms. This is divided into two parts.

Section 4.1 is an overview of the enumeration algorithms with respect to the database querying scenario which is in the center of interest of this thesis. The general goal is to say for which classes of databases and which query languages efficient enumeration is possible. In some of the considered scenarios we actually reach out from this framework:

• in case of MSO queries we also have a closer look at what happens when we allow free second-order predicates in a query (in which case a single solution might contain a set of elements of the input database of arbitrary size),

• in Section 4.1.4 we look at the class of data trees being queried using XPath formulas. We consider this case not to be contained in the database scenario for the following reason: XPath queries can compare data values. We could of course model all this comparators with binary relations, but each such relation would have size quadratic in the size of the data tree. This would then trivialize all the results as "linear in the size of the database" would in fact mean "quadratic in the size of the data tree". Since the best known algorithms are linear in the size of the input data tree, we allow XPath queries to use constructions like $x < y$, where order $<$ is not provided by the database, but is inferred from the "type" of the data values (in our case it is the natural order over $\mathbb{N}$).

Since both these cases seem to be very closely related to the database querying scenario, we decided to group them together with the "regular" enumeration algorithms.

Section 4.2 is a brief overview of other kinds of enumeration algorithms, where the goal is usually to enumerate all the structures of some sort (like all the perfect matchings of a bipartite graph) and in many of the considered cases there is no real querying language involved: they all aim at solving a fixed problem. Since this is rather far away from what we are focusing on in this thesis, we just present the statements of the results here. We annotate each result from that section with an appropriate reference, so that an interested reader could refer to a corresponding source for more details.

It should be mentioned that the content of Section 4.1 makes a great use of a recent survey on the constant delay enumeration by Luc Segoufin [62].

4.1 Query enumeration in database setting

In this section we consider the core scenario with respect to this thesis: fix a class $\mathcal{C}$ of databases and a query language $\mathcal{L}$. We are interested in solving the following problem efficiently: given a query $q \in \mathcal{L}$ and a database $\mathbf{D} \in \mathcal{C}$, enumerate the set of solutions $q(\mathbf{D})$. By "efficiently" we mean here an FPT algorithm that treats $|q|$ as a parameter and by "enumerate" we mean a constant delay enumeration.

For details on all the necessary definitions, see Chapter 2.

We are going to proceed in the following way:

- we first give no restrictions on the class of databases, which will force us to dramatically bound the query language.
- we then consider consecutive restrictions on the class of databases (namely $\underline{\text{X}}$-strucutre, bounded degree, bounded treewidth and bounded expansion) and for each of those classes we propose a suitable query language.

Each time, along with the enumeration algorithm, we also address the other problems mentioned in Section 2.5, namely: model checking, testing, counting and j-th solution.

4.1.1 Arbitrary relational structures

Conjunctive queries

A conjunctive query is a first-order query of the following form:

$$q(\bar{x}) = \exists \bar{y} \bigwedge_i R_i(\bar{z}_i)$$

where each $\bar{z}_i$ contains variables from $\bar{x}$ and $\bar{y}$. As usual, we denote conjunctive queries with CQ.

We already mentioned in Section 2.3 that the parametrized model checking problem for CQ is W[1] complete (see [57] for details). Since it is strongly believed that FPT $\neq$ W[1], this is highly unlikely that an FPT enumeration algorithm exists in this case, so there is no hope for a one belonging to CONSTANT-DELAY$_{lin}$.

However, for *acyclic* conjunctive queries (ACQ) the parametrized model checking problem is known to be in FPT with a linear dependency in the size of the database [73]. Therefore we can hope for a constant delay enumeration.

Before we turn to the main definitions we should also mention that, except from the references provided in the sequel, an interested reader is also referred to thesis [9] for a detailed cover of the enumeration of conjunctive queries.

Acyclic conjunctive queries

A *join tree* for a conjunctive query q is a tree T whose nodes are atomic formulas (*atoms*) of q and such that:

- each atom of q is the label of exactly on node of T,
- for each variable x of q, the set of nodes of T in which x occurs is connected.

A linear-time algorithm for computing a join tree was shown in [69].

A conjunctive query q is said to be *acyclic* if it has a join tree. In graph theoretical terms this is equivalent to saying that the hypergraph associated to q is α-acyclic [13].

Given as an input a database $\mathbf{D}$, the best known algorithm for evaluating a query $q \in$ ACQ runs in time $|q| \cdot \|\mathbf{D}\| \cdot |q(\mathbf{D})|$ [73]. This is not yet of the form $f(|q|) \cdot (\|\mathbf{D}\| + |q(\mathbf{D})|)$ implied by any constant delay enumeration algorithm. Actually, we will see that it is very unlikely that constant delay enumeration can be achieved for all queries in ACQ. Constant delay enumeration is only obtained for a subset of ACQ called *free-connex* that we are about to define. Before we do that, let us present the best know result concerning full ACQ.

Theorem 4.1.1 ([11]) *The enumeration of* ACQ *queries over the class of all structures can be done with constant time preprocessing and delay linear in the size of the database.*

Concerning the other problems related to enumeration, only the problem of counting the number of solutions to an acyclic conjunctive query was addressed. The combined complexity turns out to be $\#P$-complete [58] and is known to be linear only in the quantifier free case [9]. For this reason in [27] a new parameter named *quantified-star size* was introduced. Intuitively it measures "how far free variables are spread in the formula". It turns out that this parameter characterizes exactly the subclass of ACQ having a tractable counting problem:

Theorem 4.1.2 ([27]) *For each number s, the counting problem for* ACQ *with quantifier-star size bounded by s over the class of all structures can be solved in time polynomial both in the size of the query and the size of the structure.*

If a class of ACQ *does not have a bounded quantified-star size, then its associated counting problem is* $\#$W[1]*-hard. In particular, it is not in* FPT *unless* $\#$W[1] $=$ FPT.

Free-connex acyclic conjunctive queries

An acyclic conjunctive query $q(\bar{x})$ is said to be *free-connex* if the query $q'(\bar{x}) = q(\bar{x}) \wedge R(\bar{x})$, where R is a new symbol of the appropriate arity, is acyclic. It should be mentioned, that this definition is due to [19], but it is equivalent to the original definition of [11].

Remark 4.1.1 *Notice that for boolean and unary* ACQ*, free-connex queries are exactly acyclic ones. Indeed:*

• If boolean conjunctive query q is acyclic, then there exists a join tree T for q. Then an example join tree T' for $q' = q \wedge R$ is T with constant R added as a new leaf of any of the nodes from T.

• If unary conjunctive query $q(x)$ is acyclic, then there exists a join tree T for $q(x)$. Then an example join tree T' for $q' = q \wedge R(x)$ is T with $R(x)$ added as a new leaf of any of the nodes from T that contained x as one of the variables. If such a node in T does not exist, then $R(x)$ can be added as a new leaf of any of the nodes from T.

It is immediate to verify that the above constructions give proper join trees.

Remark 4.1.1 shows that the free-connexity does not introduce any restrictions for boolean or unary ACQ queries. We shall see in Example 4.1.2 that this is no longer the case for binary queries and beyond. But before going to Example 4.1.2 we first show a simple example of a binary acyclic conjunctive query that happens to be free-connex.

Example 4.1.1 *Consider the following query:*

$q(x, y) = \exists u, vA(x, u) \wedge B(v, y)$.

It is free-connex, because $q'(x, y) = \exists u, vA(x, u) \wedge B(v, y) \wedge R(x, y)$ is acyclic. The example join tree for q' has $R(x, y)$ as root and $A(x, u)$ as the left child of $R(x, y)$ and $B(v, y)$ as the right child of $R(x, y)$.

Example 4.1.2 *This time consider the query:*

$\Pi(x, y) = \exists zA(x, z) \wedge B(z, y)$.

Then $\Pi'(x, y) = \exists zA(x, z) \wedge B(z, y) \wedge R(x, y)$ is clearly NOT acyclic, so $\Pi(x, y)$ is not free-connex.

This example is important for one more reason: notice that if one would interpret relations A and B as boolean matrices (where $A(x, y)$ iff there is a 1 in cell $[x, y]$ of matrix represented by A and $B(x, y)$ behaves in an analogous way), then $\Pi(x, y)$ represents the product of matrices represented by $A(x, y)$ and $B(x, y)$.

It turns out that free-connex assumption for acyclic conjunctive queries is enough to obtain constant delay enumeration:

Theorem 4.1.3 ([11]) *The enumeration of free-connex* ACQ *over the class of all structures is in* CONSTANT-DELAY$_{lin}$.

The result of Theorem 4.1.3 also holds if the queries contain inequalities (we then speak of the class ACQ$^{\neq}$). In this case the atoms with inequalities are not involved when building the (generalized) join trees. The best known evaluation algorithm for full ACQ$^{\neq}$ (without the free-connex assumption) requires $f(|q|) \cdot \|\mathbf{D}\| \cdot |q(\mathbf{D})|$ steps, where f is exponential function (see [11]).

Beyond free-connex ACQ

It turns out that the free-connexity characterizes exactly those acyclic queries that can be enumerated with constant delay. This is under an assumption that boolean matrix multiplication cannot be done in quadratic time. The encoding of boolean matrix multiplication problem into evaluating an acyclic conjunctive query was already explained in Example 4.1.2. The best known algorithm so far for solving the matrix multiplication problem requires more than $n^{2.37}$ steps (the method is based on Coppersmith-Winograd algorithm [21]).

Theorem 4.1.4 ([11]) *If boolean matrix multiplication problem cannot be solved in quadratic time, then the following are equivalent for any $q \in$* ACQ*:*

1. *q is free-connex,*

2. *q can be enumerated in* CONSTANT-DELAY$_{lin}$*,*

3. *q can be evaluated in time $O(\|\boldsymbol{D}\| + |q(\boldsymbol{D})|)$.*

Remark 4.1.2 *We already mentioned that the results for free-connex* ACQ *extend to the case where atoms could also be inequality statements. In the case of* signed conjunctive queries *(*SCQ*), where atoms could be negated, under a suitable notion of acyclicity (somewhere between α-acyclicity and β-acyclicity) the model checking problem was shown to be $O(\|\boldsymbol{D}\| \log \|\boldsymbol{D}\|)$, where the constant involving the query are of polynomial size.* SCQ *can be enumerated with an algorithm having a $O(\|\boldsymbol{D}\| \log \|\boldsymbol{D}\|)$ preprocessing phase and $O(\log \|\boldsymbol{D}\|)$ delay [19].*

4.1.2 X-underbar structures

Let us stick to the conjunctive queries for a while. A class of structures, called $\underline{\text{X}}$-structures, has been exhibited such that CQ and ACQ can be enumerated more efficiently over those classes of structures.

Let **D** be a database with domain D. Assume **D** does not contain any relations of arities greater than 2. A binary relation R has the $\underline{\text{X}}$ *property* with respect to a total order $<$ on D iff the following holds:

$$\mathbf{D} \models \forall v_0, v_1, v_2, v_3\, (R(v_0, v_1) \wedge R(v_2, v_3)) \to R(\min{(v_0, v_2)}, \min{(v_1, v_3)}).$$

A set of binary relations over D has the $\underline{\text{X}}$ property if there is a total order $<$ on D such that all the relations from that set have the $\underline{\text{X}}$ property with respect to $<$. Similarly, a structure has $\underline{\text{X}}$ property if the set of its binary relations has the $\underline{\text{X}}$ property. We call such structure an $\underline{\text{X}}$ *structure*.

Example 4.1.3 *Over trees, subset of axes relations of* XPath *(we define* XPath *in details in Section 4.1.4) given by* {CHILD, NEXT-SIBLING} *has the $\underline{\text{X}}$ property for the order induced by a breadth-first left to right traversal of the tree. It turns out that there is no order $<$ on the set of nodes of a tree such that all* XPath *axes would have the $\underline{\text{X}}$ property with respect to that order. A complete list of subsets of* XPath *axes having the $\underline{\text{X}}$ property can be found in [39].*

Restricting the class of structures to the $\underline{\text{X}}$ structures yields the following results.

Theorem 4.1.5 ([10]) *The enumeration of* CQ *over $\underline{\text{X}}$ structures can be done with constant time preprocessing and delay linear in the size of the database.*

Theorem 4.1.6 ([10]) *The enumeration of* ACQ *over $\underline{\text{X}}$ structures can be done with linear time preprocessing and delay linear in the size of the domain of the input database.*

4.1.3 Sparse structures

In this section we restrict the class of structures in order to extend the class of queries. We are going to consider the following three restrictions:

- bounded degree,
- bounded treewidth,
- bounded expansion.

Rather than on the class of structures itself, they are all defined with respect the underlying class of graphs of these structures. There are two well established definitions of a graph of a structure, namely its Gaifman graph and its adjacency graph (see Section 2.8.1 for details). As we have seen in Section 3.3, namely in Examples 3.3.1 and 3.3.2, for some classes of structures these underlying classes of graphs do not necessarily admit the same properties of having bounded degree/treewidth/expansion. We have a full "equivalence" only if we fix a schema of the databases, which a priori is not a necessary requirement. Fortunately in our case we in a sense get the fixed schema property "for free". The reason for this is that we are interested in FPT algorithms, where we always assume the input query to be fixed. Having a fixed query, we may restrict all the databases to only those relations which actually appear in the query, which in fact fixes the schema.

In the sequel, when we say that a class $\mathcal{C}$ of databases has bounded degree/treewidth/expansion, we always refer to the underlying class $\mathcal{C}'$ of Gaifman graphs of structures from $\mathcal{C}$. In view of the above, the reader should keep in mind that it is equivalent to defining $\mathcal{C}$ with respect to the underlying class of adjacency graphs.

Bounded degree

The first restriction that we are going to consider is the bounded degree case. As we said in the beginning of this chapter, we are limiting the allowed databases in order to increase the power of the query language. In Section 4.1.1 we saw that without any restrictions we cannot get constant delay enumeration for conjunctive queries or even for its acyclic part. Bounding the degree, we not only handle CQ, but even the whole first-order logic (FO).

Recall the definitions of first order logic (see Section 2.4) and of the class of graphs with bounded degree (see Section 2.8.2).

As we said earlier, the starting point for the potential enumeration algorithm is always the model checking problem.

Theorem 4.1.7 ([61, 36]) *Fix $d \in \mathbb{N}$. The problem of whether a given d-degree-bounded structure $\boldsymbol{D}$ satisfies a given first-order sentence ϕ is decidable in time $2^{2^{2^{O(|\phi|)}}} \|\boldsymbol{D}\|$.*

This can be lifted to an enumeration algorithm:

Theorem 4.1.8 ([26, 46]) *The enumeration of* FO *queries over the class of structures of bounded degree is in* CONSTANT-DELAY$_{lin}$. *Moreover, the output is returned in a lexicographical order.*

In particular, the mentioned CONSTANT-DELAY$_{lin}$ *algorithm has precomputation phase taking time $2^{2^{2^{O(|\phi|)}}} \cdot \|\boldsymbol{D}\|$ and a delay during the enumeration phase that is triply exponential in $|\phi|$.*

There are two fundamental properties of structures of degree d that can be used to prove the above result.

One key property of structure of degree d is that for a given radius r there are only finitely many (up to isomorphism) possible r-neighborhoods of a single node (i.e. substructures involving all nodes at distance of up to r from the center node). Given query $q \in$ FO the Gaifman Locality Theorem (see Theorem 5.2.1 for details) tells us that only the r-neighborhood types are relevant, where r depends only on q. One can then show that it is possible to recolor in linear time, hence during the preprocessing phase, each node with its neighborhood type. Based on these colors and the Gaifman Locality Theorem, it is then possible to derive an enumeration algorithm. This approach was taken in [46] and is the one that we will explain in details in Chapter 5.

Another key property of structures of degree d is that they can be encoded using bijective unary functions (where exactly d unary functions suffice to represent all the neighbors of each node). Using this bijective encoding, it is then possible to obtain a quantifier elimination procedure for FO. This procedure is designed in such a way that it returns the quantifier free formula in a very special form: as a mutually exclusive disjunction of *inductive descriptions*. Working only with inductive description, it is then possible to derive an enumeration algorithm. This approach was taken in [26], but the interested reader is also referred to [12] and [9]. A similar idea can be also seen in a more general case of classes of structures with bounded expansion (which we describe in details later on).

Concerning the involved constants (the factor in the preprocessing phase and the constant delay from the enumeration phase), the triply exponential bound comes from the approach of [46]. A naive implementation of the approach of [26] yields constants that are towers of exponential depending on $|q|$ and we do not know whether a more careful reasoning could lead to a better approximation in this case.

It should also be noted that the triply exponential bound cannot be significantly improved: it was shown in [36] that already for degree 2 doubly exponential constants cannot be achieved unless AW$[*] =$ FPT.

Concerning the other problems related to constant delay enumeration, it turns out that the first-order logic over classes of structures with bounded degree admits the best possible properties. Namely, we have:

Theorem 4.1.9 ([26, 52]) *The solution testing for* FO *queries over the class of structures of bounded degree is in* CONSTANT-TIME$_{lin}$.

Theorem 4.1.10 ([12]) *The counting problem for* FO *queries over the class of structures of bounded degree is in* LINEAR-TIME.

Theorem 4.1.11 ([12]) *The j-th solution problem for* FO *queries over the class of structures of bounded degree is in* CONSTANT-TIME$_{lin}$.

The approach of [52] is similar to the one of [26]. It also relies on bijective representation and also performs a quantifier elimination procedure.

Concerning the constants involved, all the proofs of [26, 12, 52] do not give any bound other than the tower of exponential depending on $|q|$.

In this thesis, present new proofs of the above three results. Extending the approach of [46], we derive triply exponential constants in all the cases. The reader is referred to Chapter 5 for details.

For a closer look into the problem of whether we might obtain constant delay enumeration over structures of bounded degree, but for logics stronger than FO, the reader is referred to the Discussions Chapter 8. For now we only state that we most likely cannot go as far as the MSO logic, as a particular problem expressible in that logic, namely the tiling problem, is NP-complete already for grids. Thus, unless P $=$ NP, there is no hope for a constant delay enumeration of MSO queries over the class of structures of bounded degree.

Bounded treewidth

We now turn to the class of structures having bounded treewidth (for the definition of bounded treewidth see Section 2.8.2). Bounded treewidth is a well established notion that has a huge success in many fields of computer science when looking for FPT solutions to some otherwise intractable problems. The constant delay enumeration is not going to be an exception.

It turns out that this time we can even go beyond first-order logic and deal with monadic second-order logic (MSO) (see Section 2.4) instead.

Just as before, the starting point for the potential enumeration algorithm is always the model checking problem. Its linear time solution is a very well known result of Courcelle:

Theorem 4.1.12 ([22]) *Fix $d \in \mathbb{N}$. The problem of whether a given structure $\boldsymbol{D}$ of d-bounded treewidth satisfies a given second-order sentence ϕ is decidable in time $O(\|\boldsymbol{D}\|)$.*

We would like to lift it to an enumeration algorithm. Although with MSO we could potentially encounter free set variables, recall that our definition of a query is restricted to only first-order free variables (but set quantification inside the formula is of course allowed). We later on refer to the case when set free variables are allowed.

Theorem 4.1.13 ([8, 48]) *The enumeration of* MSO *queries over the class of structures of bounded treewidth is in* CONSTANT-DELAY$_{lin}$.

Let us first note that there is also a third solution to the enumeration problem, as described in [23]. The main difference is that the index structure built there has size $O(\|\mathbf{D}\| \log \|\mathbf{D}\|)$ rather than $O(\|\mathbf{D}\|)$, so it does not fully satisfy all the requirements of CONSTANT-DELAY$_{lin}$ class.

As for the proofs of [8] and [48], they do differ significantly. They both rely on a seminal result of Courcelle which explains how a tree decomposition can be efficiently encoded in MSO. Together with Bodlaender's algorithm for computing in linear time a tree decompositions of a bounded width (see [16]), Courcelle's result states:

Theorem 4.1.14 ([22]) *Fix $k \in \mathbb{N}$. There exists an algorithm which given an* MSO *formula ϕ and a structure $\boldsymbol{D}$ with treewidth bounded by k, computes in time $O(\|\boldsymbol{D}\|)$ an* MSO *formula ϕ' and a tree $\boldsymbol{T}$ such that $\phi(\boldsymbol{D}) = \phi'(\boldsymbol{T})$.*

This shows that the difficulties with obtaining the enumeration procedure lie entirely in the tree case. Application of the Courcelle's theorem is the first step of the enumeration procedures of both [8] and [48]. Starting from the tree case, the two algorithms strongly differ.

The proof of [8] exploits the well known fact that one can compute a finite tree automaton that is equivalent to a given MSO sentence (which accepts the input tree iff this tree is a model for the given MSO formula). Based on this automaton, an intricate index structure is described which allows constant delay enumeration. It should be noted that the algorithm of [8] works in a slightly weaker version of the constant delay enumeration, where we do not assume constant write memory limitations during the enumeration phase (while new solutions are being output with constant delay between two consecutive ones, the algorithm uses recursive procedures whose depth seems to be as big as the height of the input tree).

We now briefly outline the ideas behind the proof of [48]. We later on present it in full details in Chapter 6.

Let us first focus on a special case, which is also of independent interest on its own.

Let L be a regular word language over an alphabet $\mathbb{A}$. A typical binary MSO query over trees relating to L is $q_L(x, y)$, which returns pairs of nodes (u, v) of the input tree such that u is an ancestor of v and the word formed by the labels of the nodes on the path from u to v belongs to L.

Given a tree $\mathbf{T}$, there exists an index structure such that, given two nodes u and v of $\mathbf{T}$, one can test in constant time whether $(u, v) \in q_L(\mathbf{T})$. Moreover, this index structure can be computed in time linear in $\|\mathbf{T}\|$ (and therefore has size linear in $\|\mathbf{T}\|$). This is a nontrivial result of Colcombet:

Theorem 4.1.15 ([20]) *For any regular language L over an alphabet $\mathbb{A}$ and any $\mathbb{A}$-labeled tree $\boldsymbol{T}$, there exists an index structure such that:*

- *it can be constructed in time $O(\|\boldsymbol{T}\|)$,*

- *for all nodes $u, v \in \boldsymbol{T}$, one can decide whether $(u, v) \in q_L(\boldsymbol{T})$ in constant time (i.e. in time depending only on L).*

This is a deep result based on algebraic constructions. The constants involved during the construction of the index structure and during the constant time tests depend on the presentation of L. They are non elementary in L if the language is given as an MSO sentence, but are only exponential in L if it is given as an automaton (being the case both for deterministic and non-deterministic automatons). However, in some situations this constants may be only polynomial. This is for example the case for the *basic automata model* introduced in [18] in order to capture the navigational power of XPath and used in the proof of Theorem 4.1.26.

It turns out that the index structure built for proving Theorem 4.1.15 has many other important consequences. One of them is a normal form for MSO queries over trees.

Theorem 4.1.16 (Implicit in [20]) *Over binary trees, every binary* MSO *query $q(x, y)$ is equivalent to a disjunction of queries of the form $\exists \bar{y} \forall \bar{z}\ \theta$, where θ is a disjunction of conjunctions of atomic predicates or* MSO *queries with one free variable or atoms using $<$, where $<$ is the ancestor relation.*

The index structure of [48] for enumerating MSO queries highly relies on the above theorem. The so called "compositional method" or a natural Ehrenfeucht-Fraïssé game argument shows that any MSO query is equivalent to a boolean combination of binary queries and for binary queries the above theorem finds its use. The unary MSO subformulas can be precomputed in linear time by Courcelle's theorem (to be precise, by its extension as given by Theorem 6.2.1) and can therefore be considered as new colors. Hence it is enough to consider $\exists \bar{y} \forall \bar{z}$ first order queries. Those queries can be handled rather smoothly, as we explain in details in Chapter 6.

It is worth mentioning that the constants involved in the enumeration algorithm deviate from Theorem 4.1.15 only by a polynomial factor. Hence their size depends on the presentation of the MSO query as explained above.

For the detailed proof of the sketched approach, the reader is referred to Chapter 6.

Concerning the other problems related to constant delay enumeration, it turns out that the second-order logic over classes of structures with bounded treewidth admits rather good properties. There might possibly be some room for improvement concerning the j-th solution problem, but it is highly unlikely that such an improvement is actually achievable (cf. [9]). Going into details, we have:

Theorem 4.1.17 *The solution testing for* MSO *queries over the class of structures of bounded treewidth is in* CONSTANT-TIME$_{lin}$.

Theorem 4.1.18 ([5]) *The counting problem for* MSO *queries over the class of structures of bounded treewidth is in* LINEAR-TIME.

Theorem 4.1.19 ([8]) *The j-th solution problem for* MSO *queries over the class of structures of bounded treewidth is in* LOGARITHMIC-TIME$_{lin}$.

Both the approach of [5] and of [8] highly rely on the finite automaton representing the MSO query. For more details on the approach of [8], the interested reader is also referred to thesis [9].

In this thesis we present new proofs of the above three results. We follow the approach of [46] and show how it can be extended to the above three theorems. The reader is referred to Chapter 6 for details.

Before we move to further results, let us mention an extension of the enumeration algorithm first. Recall that our definition of a query allows only first-order free variables. If we skip this restriction we can no longer hope for CONSTANT-DELAY$_{lin}$ enumeration since even writing a single solution could take linear time. We might then follow two approaches that we now mention.

• One way is to consider the so called *output-linear delay*, which allows for the delay between two consecutive solutions to be linear in the size of the latter solution. This is clearly a generalization of the constant delay for query enumeration, since the size of a tuple is always constant. With set free-variables allowed this starts to play a role but we still get:

Theorem 4.1.20 ([8, 23]) *The enumeration of* MSO *formulas (allowing monadic second-order free predicates) over the class of structures of bounded treewidth can be done with linear time preprocessing and output-linear delay.*

It should be noted that the preprocessing phase of [23] works in fact in time $O(\|\mathbf{D}\| \log(\|\mathbf{D}\|))$.

• The other approach consists in having an output tape on which the current output is stored and during the enumeration phase the algorithm only modifies the content of this tape to transform the previous solution into a new one. In some special cases the delta between two consecutive solutions only affects a constant part of the output and can be performed in constant time resulting in a procedure with *delta-constant delay*. For details on this approach see [28].

Bounded expansion

We now turn to the class of structures with bounded expansion (see Section 2.8.2 for necessary definitions). In [53] a number of equivalent definitions of this class was shown giving evidence that this class is very robust. It turns out that many known families of structures have bounded expansion. Among others, this list contains for example:

- class of structures of bounded degree,
- class of structures of bounded treewidth,
- class of planar structures,
- class of structures excluding at least one minor.

The logic that we are going to consider in this case is again FO.

As usual, the starting point for the potential enumeration algorithm is the model checking problem.

Theorem 4.1.21 ([29, 42, 47]) *The model checking of* FO *sentences over the class of structures with bounded expansion is in* LINEAR-TIME.

The key ingredient that is common for all the three proofs of this theorem is that they all perform some kind of quantifier elimination procedure.

Proofs of both [29] and [47] use the functional representation of the input graph (which is up to a certain level similar to the approach of [26]) to obtain the quantifier elimination procedure for FO over the class of graphs with bounded expansion. Unfortunately, in the bounded expansion case this encoding is no longer bijective (which happened to be the case for bounded degree), which strongly affects the combinatorics.

In [42] there is no switch to the functional representation, but the quantifier elimination procedure does not remove the quantifiers in this case, but rather replaces the universal quantifications with existential ones, so that the resulting query is in the existential fragment of FO. Existential fragment being a lot simpler, the appropriate linear model checking solution can be shown.

The major difference between the proofs of [29, 42] and the one of [47] is that while the first two are based on the low tree depth coloring characterization (which is yet another characterization of bounded expansion, cf. [53]) the latter is based on transitive fraternal augmentations (see Theorem 2.8.1).

We argue that the use of transitive fraternal augmentations gives a simpler proof. The reason is that it gives a useful normal form on quantifier-free formulas that is the core of not only the quantifier elimination procedure algorithm, but also the algorithms for constant delay enumeration and for counting the number of solutions as presented in [47].

In Chapter 7 we present the proof of [47] in details.

The model checking solution can be lifted to an enumeration algorithm:

Theorem 4.1.22 ([47]) *The enumeration of* FO *queries over the class of structures with bounded expansion is in* CONSTANT-DELAY$_{lin}$. *Moreover, the output is returned in a lexicographical order.*

The above theorem generalizes the corresponding result for bounded degree case, but this comes with a cost. While previously being triply exponential, the hidden constants are in this case a tower of exponential in the quantifier alternation depth of the FO query. This nonelementary constant factor is unavoidable already on the class of unranked trees, assuming FPT $\neq$ AW$[*]$ (cf. [36]).

Concerning the other problems related to constant delay enumeration, it turns out that the first-order logic over classes of structures with bounded expansion still admits rather good properties. There might possibly be some room for improvement concerning the j-th solution problem (which was the case for the simpler case of bounded degree), but we do not know whether this is actually the case or not. Going into details, we have:

Theorem 4.1.23 ([47]) *The solution testing for* FO *queries over the class of structures with bounded expansion is in* CONSTANT-TIME$_{lin}$.

Theorem 4.1.24 ([56, 47]) *The counting problem for* FO *queries over the class of structures with bounded expansion is in* LINEAR-TIME.

Theorem 4.1.25 *The j-th solution problem for* FO *queries over the class of structures with bounded expansion is in* LOGARITHMIC-TIME$_{lin}$.

For details on the above theorems, see Chapter 7. In the proofs presented there we follow the lines of [47].

4.1.4 Data trees

In this section we no longer talk about relational databases, but rather about data trees. The query language is going to be XPath.

A *data tree* is a tree whose every node carries a label from a finite alphabet $\mathbb{A}$ (this alphabet can be viewed as the set of colors in the database context) and a datum from some infinite domain (for simplicity we restrict ourselves to $\mathbb{N}$). This structure has been considered in the realm of semistructured data, timed automata, program verification and generally in systems manipulating data values. In particular, data tress can model XML documents (see for instance [17]).

By XPath we refer to a fragment of XPath 1.0. It is a two-sorted language, with *path* expressions (that we write α, β) and *node* expressions (ϕ, ψ). These expressions are defined in a mutually recursive manner. Path expressions are binary relations resulting from composing the axis relations and node expressions. The axis relations, denotes AXIS, are the usual CHILD, PARENT, DESCENDANT, ANCESTOR, NEXT-SIBLING, RIGHT-SIBLING, PREVIOUS-SIBLING, LEFT-SIBLING relations. Node expressions are boolean formulas that test a property of a node, like for example that it has a certain label, or that it has a child labeled a with the same data value as an ancestor labeled b and so on. For comparing data values we allow any predicate in the set $\textsc{Relop} = \{=, \neq, <, >, \leq, \geq\}$.

The syntax of XPath is given below:

$$\begin{array}{lll} \alpha, \beta & :: & \textsc{Axis} \mid [\phi] \mid \alpha\beta \mid \alpha \cup \beta \\ \phi, \psi & :: & \mathbb{A} \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \langle\alpha\rangle \mid \alpha \ \textsc{Relop} \ \beta \end{array}$$

We also consider an extension of XPath allowing the Kleene star on any path expression and we denote it by regular XPath. Its semantic over data trees is classical and rather intuitive. It can be found in details for instance in [18]. Typically a path expression $[\phi]$ selects all the pairs (u, u) such that the node expression ϕ is true at u. A node expression $\langle\alpha\rangle$ selects all nodes u such that there is a node v with (u, v) selected by the path expression α. Finally, a node expression α $\textsc{Relop}$ β selects all nodes u such that there exist nodes v, w such that (u, v) is selected by the path expression α, (u, w) is selected by the path expression β and the data values of v and w are related accordingly to the predicate from $\textsc{Relop}$.

We view formulas of regular XPath as queries over data trees. This query is unary if the formula is a node expression and binary otherwise.

Theorem 4.1.26 ([18]) *The enumeration problem of regular* XPath *over the class of data trees is in* $\textsc{Constant-Delay}_{lin}$.

It turns out that the constants involved in the enumeration algorithm are reasonably low. They are polynomial in the size of the query if the query is in XPath and are exponential if the query is in regular XPath.

As we mentioned earlier in Section 4.1.3 when we considered the class of databases with bounded treewidth, the index structure constructed for the enumeration algorithm highly relies on Theorem 4.1.15.

To our knowledge the other problems related to constant delay enumeration, namely testing, counting and j-th solution, have not been yet addressed and we are not aware of any results with that respect.

4.2 Other enumeration problems

The enumeration outside of the database context turns out to be a rather fruitful field, though it also seems to still be a bit fragmented. Fairly recently there has been a very interesting attempt to classifying the complexity of different enumeration problems. The interested reader is referred to the thesis [65] for learning more details on this classification. In here we give only its brief outline. We use the notation from [65] to express the results on enumeration problems outside of the database context.

We do not look into the mentioned problems in full details, since the techniques developed for their solutions seem to be of different sort than the ones used in database context. Although this might still be the case, in general we do not see a way of transferring them for the needs of the database scenario. The are two reasons for this:

- All the mentioned algorithms enumerate structures of some sort (in particular, sizes of the output objects may be arbitrary), while in the database context we are focused on enumerating tuples of fixed sizes (the only somewhat "common" problem could then be the enumeration of MSO queries with second-order predicate variables).
- And what is probably more important, the enumeration problems mentioned in this section most often refer to a fixed property (like perfect matching), while in the database context the query language most of the time plays the central role.

In some cases (see for example Theorem 4.2.4) when there are more similarities to the database scenario, we try to elaborate a bit more on the proof techniques.

In general we restrict ourselves to just stating the key results. The interested reader should follow the corresponding references for both the definitions and the proofs.

4.2.1 Abstract enumeration problems

We now define the notion of an enumeration problem in an abstract way. For the sake of readability, throughout this section we skip the word "abstract", but the reader should keep in mind that the enumeration problems mentioned here are not understood as in the database context.

An *(abstract) enumeration problem* is a binary relation. Given an enumeration problem R and an input x, a *solution for* x is a y such that $(x, y) \in R$. An enumeration problem R induces a computational problem as follows: Given an input x, output all its solutions. We denote the set of solutions to x with $R(x, \_)$.

Unlike in the database context, we do not assume enumeration problem to be a finite relation. Since in most of the presented results it is going to be the case, we implicitly assume finiteness. In the rare cases when we deal with infinite objects, we explicitly say so.

Remark 4.2.1 *We can define* CONSTANT-DELAY$_{lin}$ *class in terms of abstract enumeration problems.*

An enumeration problem is in the class CONSTANT-DELAY$_{lin}$ *if its computational problem can be solved by a RAM algorithm which, on input x, can be decomposed into two steps:*

- *a precomputation phase that is performed in time $O(|x|)$,*
- *followed by an enumeration phase that outputs all the solutions for x with no repetition and a constant delay between two consecutive outputs. The enumeration phase has full access to the output of the precomputation phase but can use only a constant total amount of extra memory.*

In particular, if R is in CONSTANT-DELAY$_{lin}$*, then the evaluation problem R can be solved in time $O(|x| + |\{y : R(x, y)\}|)$.*

This abstract definition was in fact the original one as introduced in [26].

For the purpose of this thesis, since our main focus remains in the problems of querying databases, we stick to the definition of CONSTANT-DELAY$_{lin}$ *as presented in Section 2.7.3.*

Following [65] we now define a series of enumeration classes:

4.2.2 Polynomial total time

The notion of a polynomial total time was introduced in [45].

We say that an enumeration problem $R(x, y)$ is solvable in polynomial total time (denoted with TotalP) if there exists a polynomial $Q(x)$ and an algorithm that, given x, outputs $R(x, \_)$ in time $Q(|x|)$.

In other words, the class TotalP contains problems that can be evaluated in polynomial time.

An example of a problem from TotalP is:

Theorem 4.2.1 ([34]) *For any k the problem of enumerating all the transversals of any k-uniform hypergraph is in* TotalP.

4.2.3 Incremental polynomial time

The notion of incremental polynomial time was also introduced in [45].

We say that an enumeration problem $R(x, y)$ is solvable in incremental polynomial time (denoted with IncP) if there exists a polynomial $Q(x, n)$ and an algorithm enumerating $R(x, \_)$ in such a way that the time between outputting the i-th and $(i + 1)$-th element from $R(x, \_)$ is bounded by $Q(|x|, i)$.

Example 4.2.1 *Consider for example a polynomial $Q(x, n) = nx$. If Q would be the polynomial witnessing that some enumeration problem R is in* IncP*, then we would be able to get the first solution in time $|x|$, the second in $2|x|$ and in general the i-th in $i|x|$.*

In general it is easy to see that IncP $\subseteq$ TotalP. The equality is rather unlikely, since:

Theorem 4.2.2 ([65]) *If* IncP $=$ TotalP*, then $P = co$*NP $\cap$ NP.

An example of a problem from IncP is:

Theorem 4.2.3 ([49]) *The problem of enumerating the circuits of a matroid is in* IncP.

4.2.4 Polynomial delay

The notion of polynomial delay also this time originates from [45].

We say that an enumeration problem $R(x, y)$ is solvable in polynomial delay (denoted with DelayP) if there exists a polynomial $Q(x)$ and an algorithm enumerating $R(x, \_)$ in such a way that the time between outputting the i-th and $(i + 1)$-th element from $R(x, \_)$ is bounded by $Q(|x|)$.

Clearly DelayP is more restrictive than IncP, since the delay between outputting consecutive solutions now needs to be uniform.

An interesting *infinite* problem from DelayP is:

Theorem 4.2.4 ([38]) *Let θ be an almost-sure* FO *sentence and $\mathcal{G}_\theta$ the almost-sure first-order family of graphs induced by θ. Then the enumeration of the graphs from $\mathcal{G}_\theta$ is in* DelayP*. Moreover, the algorithm works in polynomial space.*

The result of [38] on the first glance seems to be quite promising when searching for some techniques that could be adapted for the database setting (since we encounter sort of a query language for the first time). Unfortunately, this eventually does not seem to be the case. The reason is that the techniques used to deal with the infinite family $\mathcal{G}_\theta$ have a totally different flavor then the ones used in finite cases. The methods used in [38] build on zero-one law (see [31]) and the method of pesimistic estimators (see [59]) to get the so called *extended method of pessimistic estimators*. This techniques being designed for infinite objects, we most likely will not be able to reuse them in the database context. And there is in general no clear view on how any limit-based reasoning can be transferred to a finite space.

Another enumeration algorithms from DelayP are:

Theorem 4.2.5 ([37]) *The problem of, given $n \in \mathbb{N}$ (written in unary), enumerate one representative from each isomorphism class of the set of n-vertex graphs, is in* DelayP*. Moreover, the algorithm works in polynomial total space.*

Theorem 4.2.6 ([64, 15, 68]) *The problem of enumerating all minimal a-b separators of graph is in* DelayP*. Similarly, the problem of enumerating all minimal vertex separators is in* DelayP.

Theorem 4.2.7 ([45]) *The problem of enumerating all the maximal (in terms of inclusion) independent sets of a graph in lexicographical order is in* DelayP.

The algorithm of [45] lists the independent sets in a lexicographical order, but the drawback is that although the delay is polynomial, the space used by the algorithm is exponential. In [71] another solution to the above problem was presented. It does not list the output in the lexicographical order and the delay is not polynomial, but it on the other hand works in polynomial space.

4.2.5 Strong Polynomial delay

We now present a class of problems that was introduced in [65] to add some memory constraints which disallow the space-blowup problem that we faced in Theorem 4.2.7.

We say that an enumeration problem $R(x, y)$ is solvable in strong polynomial delay (denoted with SDelayP) if for every x there is a total order $<_x$ such that the problem of finding first (with respect to $<_x$) element of $R(x, \_)$ can be solved in polynomial time and given $y \in R(x, \_)$ the problem of finding next (with respect to $<_x$) element of $R(x, \_)$ after y is in polynomial time too.

An example problems from SDelayP are:

Theorem 4.2.8 ([6, 72]) *The problems of enumerating all the minimal spanning trees and of enumerating all the maximal matchings of a weighted graph are in* SDelayP.

Theorem 4.2.9 ([24]) *The problem of enumerating all the solutions to* SAT *instances is in* SDelayP, *when the class of the allowed* SAT *instances is limited to any of the following classes:*

- *Horn formulas,*
- *anti-Horn formulas,*
- *affine formulas,*
- *bijunctive formulas.*

4.2.6 Probabilistic enumeration algorithms

In [66] a probabilistic approach to the enumeration algorithms was introduced. The result obtained there is as follows:

Theorem 4.2.10 ([66]) *Let P be a multilinear polynomial with n variables, t monomials and total degree D. There exists an algorithm that computes the set of monomials of P with probability $1-\epsilon$ for any ϵ. The delay between outputting two consecutive monomials is bounded in time by $O(D^2 n^2 \log(n)(n + \log(\epsilon^{-1})))$ and by $O(nD(n + log(\epsilon^{-1})))$ oracle calls. The whole algorithm performs $O(tnD(n + log(\epsilon^{-1})))$ oracle calls on points of size $O(\log(D))$.*

More details on the above theorem and on probabilistic enumeration classes can also be found in [65].

4.2.7 Impact of the order and separation of the enumeration classes

In some of the mentioned algorithms we explicitly said that the output is listed for example in lexicographical order. In general, one might consider classes dual to TotalP, DelayP, etc. where the problem comes with a desired order on the output. For more details on this, the interested reader is referred to [65].

We presented a single theorem separating two of the mentioned enumeration classes (cf. Theorem 4.2.2), but this does not exhaust all the possible relations among those classes. There is a very promising ongoing attempt to separate IncP and DelayP (see [67]), but we are not aware whether the result can already be claimed. For more details on the possible attempts to obtain other separation results, the interested reader is again referred to [65].

4.3 Conclusions

This chapter was dedicated to the enumeration problem in its two aspects:

- in the database scenario we are interested in efficient enumeration of solutions to a query,
- in a more general setting we are interested in efficient enumeration of tuples of a binary relation, where we fix one of the components (but the binary relation can be understood in a very general way, for instance it may contain pairs of graphs, where the first component is an arbitrary graph and the second is one of the minimal spanning tree of the graph from first component).

For more details on the enumeration in the general sense, the interested reader is referred to the thesis [65].

For more details on selected fragments of the database scenario, the interested reader is referred to Chapters 5, 6 and 7 of this thesis.

Thesis [9] also gives an interesting perspective on the database scenario (especially for the parts concerning conjunctive queries, but also for the different approaches on the MSO queries over the classes of graphs with bounded treewidth and FO queries over the classes of graphs with bounded degree).

Of course not all the question concerning enumeration problems have already been answered. For an overview of the interesting open problems see Chapter 8.

5

FO over classes of structures with bounded degree

Contents

5.1 Introduction

As we explained in Chapter 4, without restricting the class of allowed databases, one can hope for efficient enumeration algorithms only for very weak query languages (see Section 4.1.1 for details). The strongest known result is about free-connex acyclic conjunctive queries (see Theorem 4.1.3) and in view of Theorem 4.1.4 it is unlikely that the result of Theorem 4.1.3 can be strengthen. In order to increase the power of the query language to be the full first-order logic (FO), we restrict the allowed databases to the ones with bounded degree.

As we said earlier, the starting point for the potential enumeration algorithm is always the model checking problem. In the case of first-order logic over the class of structures of bounded degree, the linear model checking solution was first shown by Seese [61], and then the result was revisited in [36] to give precise bounds on the constants (see Theorem 5.2.2).

It turns out that not only this result can be lifted to an enumeration algorithm, but all the other problems from Section 2.5 have very efficient solutions too.

The goal of this chapter is to prove the following series of theorems.

Theorem 5.1.1 ([26, 46]) *The enumeration of* FO *queries over the class of structures of bounded degree is in* CONSTANT-DELAY$_{lin}$. *Moreover, the output is returned in a lexicographical order.*

Theorem 5.1.2 ([26, 52]) *The solution testing for* FO *queries over the class of structures of bounded degree is in* CONSTANT-TIME$_{lin}$.

Theorem 5.1.3 ([12]) *The counting problem for* FO *queries over the class of structures of bounded degree is in* LINEAR-TIME.

Theorem 5.1.4 ([12]) *The j-th solution problem for* FO *queries over the class of structures of bounded degree is in* CONSTANT-TIME$_{lin}$.

For Theorem 5.1.1, we follow the line of the proof of [46]. The other proof [26] relies on a quantifier-elimination procedure (see also [52] for a similar argument). Our proof builds on the Gaifman Locality Theorem 5.2.1. The total query evaluation induced by the enumeration procedure of Theorem 5.1.1 is in time $2^{2^{2^{O(|\phi|)}}}(\|\mathbf{D}\| + |\phi(\mathbf{D})|)$ thus matching the model checking complexity of Theorem 5.2.2. Note that it is not clear from the proof of [26] that their algorithm is triply exponential in the size of the formula.

We then show how the approach of [46] can further be exploited to also obtain new proofs of Theorems 5.1.2, 5.1.3 and 5.1.4, with the involved constant also being triply exponential in the size of the input formula. Also in this case it is not clear whether the respective algorithms given in [52] and [12] have this properties.

5.2 Preliminaries

We say that a relational structures has d-bounded degree if the underlying Gaifman graph has d-bounded degree. Note that with respect to the discussions from Section 3.3 it is equivalent (modulo increasing the constant from d to d^2) to defining this notion with respect to the underlying adjacency graph (since the query is assumed to be fixed).

Similarly, we say that a class of databases has d-bounded degree if the underlying class of Gaifman graphs has d-bounded degree.

5.2.1 Gaifman locality

Recall the definition of a Gaifman graph from Chapter 2. We now elaborate a bit more about the Gaifman graph in order to define the Gaifman Locality Theorem for FO.

Let $\mathbf{D}$ be a relational structure and $G(\mathbf{D})$ its Gaifman graph. Given $a, b \in A$, the *distance* between a and b, denoted $\delta(a, b)$, is the length of a shortest path between a and b in $G(\mathbf{D})$ or ∞ if a and b are not connected. The *distance* between two tuples $\bar{a} = (a_1, \dots, a_k)$ and $\bar{b} = (b_1, \dots, b_l)$ of $\mathbf{D}$, denoted $\delta(\bar{a}, \bar{b})$, is the $\min\{\delta(a_i, b_j) : 1 \le i \le k, 1 \le j \le l\}$. For a given $r \in \mathbb{N}$ and a given tuple of elements $\bar{a}$ of some structure $\mathbf{D}$, we denote by $N_r(\bar{a})$ the set of all elements in A such that their distance from $\bar{a}$ is less or equal to r. The *width* of $N_r(\bar{a})$ is the maximal distance between two elements from $N_r(\bar{a})$. The *r-neighborhood* of $\bar{a}$, denoted as $\mathcal{N}_r(\bar{a})$, is the substructure of $\mathbf{D}$ induced by $N_r(\bar{a})$ and expanded with one constant for each element of $\bar{a}$. Given two tuples of elements $\bar{a}$ and $\bar{b}$ we say that they have *the same r-neighborhood type*, written $\mathcal{N}_r(\bar{a}) \simeq \mathcal{N}_r(\bar{b})$, if there is an isomorphism between $\mathcal{N}_r(\bar{a})$ and $\mathcal{N}_r(\bar{b})$.

We consider first-order logic (FO) as defined in Chapter 2. Throughout this chapter, when writing $\phi(\bar{x})$, we always mean that $\bar{x}$ are exactly the free variables of ϕ. As usual $|\phi|$ denotes the size of ϕ.

We are now ready to state Gaifman locality for FO.

Theorem 5.2.1 (Gaifman Locality Theorem [51]) *For any first-order formula $\phi(\bar{x})$, for every structure $\boldsymbol{D}$ and tuples $\bar{a}$, $\bar{b}$, we have $\mathcal{N}_r(\bar{a}) \simeq \mathcal{N}_r(\bar{b})$ implies $\boldsymbol{D} \models \phi(\bar{a})$ iff $\boldsymbol{D} \models \phi(\bar{b})$, where $r = 2^{|\phi|}$.*

To be precise, the bound on r proved in [51] is $\frac{3^{k+1}-1}{2}$, where k is the quantifier rank of the query ϕ. Since in the sequel we are going to use the big O notation (see for instance the statement of Theorem 5.4.1), we allow for this little confusion and leave the statement of Theorem 5.2.1 as it is.

5.2.2 Model checking

We do not give a proof of the model checking problem for FO over classes of structures with bounded degree. A possible way of proving it is via Theorem 5.2.1 (for details see [61] or [36]).

Theorem 5.2.2 ([61, 36]) *Fix $d \in \mathbb{N}$. The problem of whether a given d-degree-bounded structure $\boldsymbol{D}$ satisfies a given first-order sentence ϕ is decidable in time $2^{2^{2^{O(|\phi|)}}} \|\boldsymbol{D}\|$.*

5.2.3 Connectivity, partitions and splits

In this section we assume $d \in \mathbb{N}$ to be fixed and all our structures to be d-degree bounded.

Fix $r \in \mathbb{N}$ and a FO formula $\phi(\bar{x})$ with k free variables $\bar{x} = x_1 \dots x_k$.

A tuple $\bar{x} = x_1 \dots x_k$ of nodes is *r-connected around* x_1 if there exists a permutation π of $\{2, \dots, k\}$ such that for each $j \in \{2, \dots, k\}$ the r-neighborhood of $x_{\pi(j)}$ intersects with the r-neighborhood of $(x_1, x_{\pi(2)}, \dots, x_{\pi(j-1)})$. Equivalently, $\bar{x}$ is *r-connected around* x_1 if the r-neighborhood of $\bar{x}$ is connected. Note that if $\bar{x}$ is r-connected around x_1, then it is also connected around x_i for any $1 \leq i \leq k$.

Formula ϕ is said to be *r-connected around* x_1 if it logically implies that $\bar{x}$ is r-connected around x_1. In the sequel when we write that a formula is r-connected around x, we implicitly assume that x is the most significant free variable of that formula.

Let $\mathcal{T}_s$ be the set of all isomorphism types of s-neighborhoods of single elements, i.e. the isomorphism types of structures of the form $\mathcal{N}_s(a)$ for some element a of some structure $\mathbf{D}$. By *s-neighborhood type* of an element a we mean the isomorphism type of its s-neighborhood. Because our structures are d-degree-bounded each s-neighborhood has at most d^s elements. For each $\tau \in \mathcal{T}_s$ we denote by $\mu_\tau(x)$ the fact that the s-neighborhood type of x is τ. For each type in $\mathcal{T}_s$ we fix a representative for the corresponding s-neighborhood and fix a linear order among its elements. This way, we can speak of the first, second,..., element of an s-neighborhood. For technical reasons, we actually fix a linear order for each l-neighborhood for $l \leq s$ such that (i) it is compatible with the distance from the center of the neighborhood: the center is first, then come all the elements at distance 1, then all elements at distance 2 and so on... and (ii) the order of a $(l+1)$-type is consistent with the order on the induced l-type.

For some sequence $F = \{\alpha_2, \dots, \alpha_m\}$ of $(m-1)$ elements from $[1, \dots, d^s]$, we write $\bar{x} = F(x_1)$ for the fact that, for $j \in \{2, \dots, m\}$, x_j is the α_j-th element of the s-neighborhood of x_1. We call each such F a *s-binding of* $\bar{x}$. Given s-neighborhood type τ, we say that a s-binding F of $\bar{x}$ is *r-good for* τ if $F(x_1)$ is r-connected around x_1 for every x_1 of type τ. Let $\mathcal{F}_s^m$ be the set of all possible s-bindings F. Let $\mathcal{F}_s = \bigcup_{1 \leq m \leq k} \mathcal{F}_s^m$.

For a given $\bar{x} = x_1 \dots x_k$ a *r-partition* of $\bar{x}$ is a set of triples $\{(C_1, F_1, \tau_1), \dots, (C_m, F_m, \tau_m)\}$ such that $\emptyset \neq C_i \subseteq \bar{x}$, $\bigcup_{1 \leq i \leq m} C_i = \{x_1, \dots, x_k\}$, $C_i \cap C_j = \emptyset$ for $i \neq j$; $\tau_i \in \mathcal{T}_{r(2k+1)}$; and $F_i \in \mathcal{F}_{r(2k+1)}^{|C_i|}$ is r-good for τ_i. For a given r-partition C of $\bar{x}$ and $(C_i, F_i, \tau_i) \in C$ we write $\bar{x}^i$ to represent variables from C_i, x_1^i to represent the most significant variable from C_i, x_2^i to represent second most significant variable and so on. We call each such triple (C_i, F_i, τ_i) a *C-component*. To avoid clutter, if F_i and τ_i are not important and we only care about C_i, we also write that $\bar{x}^i$ is a *C-component*. We denote with $C_r^k(\bar{x})$ the set of all r-partitions of $\bar{x}$.

For a given r-partition $C = \{(C_1, F_1, \tau_1), \dots, (C_m, F_m, \tau_m)\}$ of $\bar{x}$ by $\mathrm{Div}_r^C(\bar{x})$ we mean a conjunction of formulas saying that $N_r(\bar{x}^i) \cap N_r(\bar{x}^j) = \emptyset$ for all $1 \leq i \neq j \leq m$ and formula

$\bigwedge_{(C_i,F_i,\tau_i)\in C}\left(\bar{x}^i = F_i(x_1^i) \wedge \mu_{\tau_i}(x_1^i)\right)$. Note that the latter part implies that $\bar{x}^i$ is r-connected around x_1^i (as F_i is r-good for τ_i). We call this formula a *C-split of $\bar{x}$*.

We now present a series of observations using the notions introduced above. They are all straightforward consequences of the appropriate definitions.

Observation 5.2.1 *Fix $r \in \mathbb{N}$, a structure $\boldsymbol{D}$ and a tuple $\bar{a}$ of k nodes from $\boldsymbol{D}$. Then there is exactly one r-partition $C \in C_r^k(\bar{x})$ such that $\boldsymbol{D} \models Div_r^C(\bar{a})$.*

As a consequence of Observation 5.2.1 we get:

Observation 5.2.2 *Fix $r \in \mathbb{N}$ and a structure $\boldsymbol{D}$. Then any* FO *formula $\phi(\bar{x})$ with k free variables is equivalent to a formula of the form*

$$\bigvee_{C\in C_r^k(\bar{x})} \left(Div_r^C(\bar{x}) \wedge \phi(\bar{x})\right).$$

Moreover, the above disjunction is mutually exclusive.

Observation 5.2.3 *Fix $r \in \mathbb{N}$, a structure $\boldsymbol{D}$ and two elements a and b from $\boldsymbol{D}$. If $N_r(a) \cap N_r(b) \neq \emptyset$, then $b \in N_{2r}(a)$.*

As a consequence of Observation 5.2.3 we get:

Observation 5.2.4 *Fix $r \in \mathbb{N}$, a structure $\boldsymbol{D}$ and a tuple $\bar{a}$ of k nodes from $\boldsymbol{D}$. If $\bar{a}$ is r-connected around a_1, then $\bar{a} \subseteq N_{2rk}(a_1)$ and $N_r(\bar{a}) \subseteq N_{r(2k+1)}(a_1)$.*

Observation 5.2.5 *Fix $r \in \mathbb{N}$, a structure $\boldsymbol{D}$, a tuple $\bar{a}$ of l nodes from $\boldsymbol{D}$ and a tuple $\bar{b}$ of k nodes from $\boldsymbol{D}$. If $\bar{b}$ is r-connected around b_1 and $N_r(\bar{a}) \cap N_r(\bar{b}) \neq \emptyset$, then $\bar{b} \subseteq N_{2rk}(\bar{a})$ and $N_r(\bar{b}) \subseteq N_{r(2k+1)}(\bar{a})$.*

In the scope of the above observation, if we do not want to fix $\bar{b}$ but rather fix the "witness for its r-connectedness", we have a bounded number possibilities for the choice of b_1 (which has to be in $N_{2rk}(\bar{a})$), which immediately leads to:

Observation 5.2.6 *Fix $r \in \mathbb{N}$, a structure $\boldsymbol{D}$ and a tuple $\bar{a}$ of l nodes from $\boldsymbol{D}$. Assume $|\bar{x}| = k$ and fix a r-partition $C = \{(C_1, F_1, \tau_1)\}$ of $\bar{x}$. There are up to ld^{2rk} tuples $\bar{b}$ such that $N_r(\bar{a}) \cap N_r(\bar{b}) \neq \emptyset$ and $\boldsymbol{D} \models Div_r^C(\bar{b})$.*

Proof Fix a tuple $\bar{b}$ of k nodes. If $\mathbf{D} \models \mathrm{Div}_r^C(\bar{b})$, then $\bar{b}$ is uniquely determined by b_1 and C (to be precise, $\bar{b} = F_1(b_1)$). To conclude recall from Observation 5.2.5 that $N_r(\bar{a}) \cap N_r(\bar{b}) \neq \emptyset$ implies $\bar{b} \subseteq N_{2rk}(\bar{a})$, so in particular $b_1 \in N_{2rk}(\bar{a})$ and we have up to as many choices for b_1 as is the maximal size of $N_{2rk}(\bar{a})$, which in fact is ld^{2rk}.

■

Observation 5.2.7 *Fix $s \in \mathbb{N}$, a structure $\boldsymbol{D}$ and a tuple $\bar{a}$ of m nodes from $\boldsymbol{D}$. If $\bar{a} \subseteq N_s(a_1)$, then there is exactly one binding $F \in \mathcal{F}_s^m$ such that $\bar{a} = F(a_1)$.*

The following is a very useful consequence of Theorem 5.2.1.

Lemma 5.2.1 *Fix a structure $\boldsymbol{D}$. Then any formula $\phi(\bar{x})$ with k free variables is equivalent over $\boldsymbol{D}$ to a formula of the form*

$$\bigvee_{C\in S} Div_r^C(\bar{x}) \tag{5.1}$$

where $r = 2^{|\phi|}$ and $S \subseteq C_r^k(\bar{x})$ is finite. Moreover, the disjunction given by (5.1) is mutually exclusive.

PROOF Let $\phi(\bar{x})$ be a formula with k free variables and $r = 2^{|\phi|}$. For a given partition $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ of $\bar{x}$ let $C_i = \left\{x_1^i, \ldots, x_{|C_i|}^i\right\}$.

From Observation 5.2.2 we see that $\phi(\bar{x})$ is equivalent to:

$$\bigvee_{C \in C_r^k(\bar{x})} \left(\mathrm{Div}_r^C(\bar{x}) \wedge \phi(\bar{x})\right).$$

Let $\bar{a}$ be a tuple of **D** such that $\mathbf{D} \models \phi(\bar{a})$. From Observation 5.2.1, for exactly one $C \in C_r^k(\bar{x})$ we have that $\mathbf{D} \models \mathrm{Div}_r^C(\bar{a}) \wedge \phi(\bar{a})$. Let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ be that r-partition. As Div_r^C induces that for each $1 \leq i \leq m$ component $\bar{a}^i$ is r-connected, by Observation 5.2.4 the r-neighborhood of each $\bar{a}^i$ is completely included into the $(r(2k+1))$-neighborhood of a_1^i.

We now take S as the set of all r-partitions C such that for some tuple $\bar{a}$ we have that $\mathbf{D} \models \mathrm{Div}_r^C(\bar{a}) \wedge \phi(\bar{a})$.

By construction we have $\phi(\bar{x})$ implies (5.1). The reverse inclusion is an immediate consequence of Gaifman Locality Theorem: When $\mathrm{Div}_r^C(\bar{a})$ holds, $\mathcal{N}_r(\bar{a}^i)$ is induced by $\mathcal{N}_{r(2k+1)}(a_1^i) = \tau_i$ and F_i. Moreover, $\mathcal{N}_r(\bar{a})$ is the disjoint union of $\mathcal{N}_r(\bar{a}^i)$ and is therefore induced by C.

The mutual exclusion is a consequence of Observations 5.2.2 and 5.2.7.

■

Corollary 5.2.1 *Fix a structure **D** and query $\phi(\bar{x})$ with k free variables. Set $r = 2^{|\phi|}$ and let $\mathit{Div}_r^C \in C_r^k(\bar{x})$. If for some tuple $\bar{a}$ of k nodes from **D** we have $\mathbf{D} \models \mathit{Div}_r^C(\bar{a}) \wedge \phi(\bar{a})$, then $\mathbf{D} \models \forall\bar{x}\mathit{Div}_r^C(\bar{x}) \rightarrow \phi(\bar{x})$. Similarly, if for some tuple $\bar{a}$ of k nodes from **D** we have $\mathbf{D} \models \mathit{Div}_r^C(\bar{a}) \wedge \neg\phi(\bar{a})$, then $\mathbf{D} \models \forall\bar{x}\neg(\mathit{Div}_r^C(\bar{x}) \wedge \phi(\bar{x}))$.*

5.3 The index structures

The goal of this section is to design index structures capable of dealing with problems mentioned in Theorems 5.1.1–5.1.4. This is split into two objects: the basic index structure finds its use in the proofs of all the Theorems 5.1.1–5.1.4, while the basic index structure with counting is only necessary for Theorems 5.1.3 and 5.1.4.

5.3.1 The basic index structure

Intuitively, the basic index structure determines the disjunction given by (5.1) and precomputes the $(r(2k+1))$-neighborhoods of each element of **D**. It also allows to effectively navigate through elements sharing the same $(r(2k+1))$-neighborhood type.

Fix a formula $\phi(\bar{x})$ with k free variables. Let **D** be the input structure. Let $r = 2^{|\phi|}$. By Lemma 5.2.1, $\phi(\bar{x})$ is equivalent over **D** to a formula of the form given by (5.1). We assume that **D** comes with a linear order over its elements. If not, we use the linear order induced by the encoding of **D**.

The *basic index structure for ϕ and **D*** is the following object:

• It stores the Gaifman graph of **D**.

• For each element a of **D** and each $i \leq r(2k+1)$, it stores a list of nodes at distance i from a.

• For each element a of **D** and each $i \leq d^{r(2k+1)}$, it stores a pointer from a to the i-th element of its $(r(2k+1))$-neighborhood.

• For each element a of **D**, it stores the $(r(2k+1))$-neighborhood type of a.

• For each $\tau \in \mathcal{T}_{r(2k+1)}$ it stores a sorted array of elements of **D** whose $(r(2k+1))$-neighborhood type is τ.

• It stores r-partitions *relevant* for **D**, that is r-partitions such that $\mathbf{D} \models \exists\bar{x}\mathrm{Div}_r^C(\bar{x}) \wedge \phi(\bar{x})$.

We now show the main property of the basic index structure for ϕ and **D**:

Lemma 5.3.1 *Fix query $\phi(\bar{x})$ with k free variables and $d \in \mathbb{N}$. There exists an algorithm that, given a d-degree bounded structure $\boldsymbol{D}$, computes the basic index structure for ϕ and $\boldsymbol{D}$ in time $2^{2^{2^{O(|\phi|)}}} \cdot \|\boldsymbol{D}\|$.*

PROOF

In a first step, for each $i \leq r(2k+1)$ the algorithm precomputes the pairs of nodes at distance i. In other words, for each a in $\mathbf{D}$, it computes the set of elements b such that $\delta(a,b) = i$. This can easily be done in time linear in $r(2k+1) \cdot \|\mathbf{D}\|$ by induction on i: during the base case the algorithm computes the Gaifman graph of $\mathbf{D}$ and then it performs the classical computation of the transitive closure of this graph up to depth $r(2k+1)$.

In a second step, the algorithm computes for each element a of $\mathbf{D}$ its $(r(2k+1))$-neighborhood: for each element a of $\mathbf{D}$, it computes its $(r(2k+1))$-neighborhood type and for all $i \leq d^{r(2k+1)}$ a pointer from a to the i-th element of its $(r(2k+1))$-neighborhood. We use an induction on the radius of the neighborhood to achieve this goal within the desired time constraints:

As 0-neighborhoods all share the same isomorphism type and have just one pointer to their centers, the induction base is obvious. So let's assume that in linear time in the size of $\mathbf{D}$ the algorithm has computed all l-neighborhoods for all nodes. With one more linear pass it is now possible to compute the $(l+1)$-neighborhoods. Fix $a \in \mathbf{D}$. From the first step, we have all the elements of $\mathbf{D}$ at distance $l+1$ from a. As the algorithm has already computed the l-neighborhood, it remains to try all possible orders among those elements and test isomorphism with the ordered types that were initially fixed.

There are at most d^{l+1} nodes at distance $l+1$ and $l < r(2k+1)$. Hence the number of orders the algorithm needs to test is bounded by $(d^{r(2k+1)})!$. Once the order is fixed it tries all possible $(r(2k+1))$-neighborhood types that we have initially fixed (there are $|\mathcal{T}_{r(2k+1)}|$ possibilities) and then tests that the two orders induce an isomorphism (each test simply requires going through all tuples of the neighborhood). Let $s(r,k,d)$ be the maximal size of a $(r(2k+1))$-neighborhood. Thus this step is altogether achieved in time $O((d^{r(2k+1)})! \cdot |\mathcal{T}_{r(2k+1)}| \cdot s(r,k,d))$, which is triply exponential in $|\phi|$ because $r = 2^{|\phi|}$, $|\mathcal{T}_{r(2k+1)}| = O(2^{s(r,k,d)})$ and $s(r,k,d) = O(d^{r(2k+1)|\sigma|})$.

The third step of the precomputation phase orders, for each $\tau \in \mathcal{T}_{r(2k+1)}$, the elements of $\mathbf{D}$ having that particular $(r(2k+1))$-neighborhood type and stores them in an array. To do that, we just need to enumerate through all the elements in $\mathbf{D}$, in the order provided by the linear order on its elements, and, using information obtained in the second step, add each of them to a proper list.

During the last step the algorithm determines the r-partitions that are relevant for $\mathbf{D}$. Fix a r-partition $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ from $C_r^k(\bar{x})$. Recall that r-partition is relevant for $\mathbf{D}$ if $\mathbf{D} \models \exists \bar{x} \text{Div}_r^C(\bar{x}) \wedge \phi(\bar{x})$. The algorithm first finds a tuple $\bar{a}$ such that $\mathbf{D} \models \text{Div}_r^C(\bar{a})$. If such a tuple does not exist, then clearly C is not relevant for $\mathbf{D}$. If such a tuple $\bar{a}$ exists, then we can apply Theorem 5.2.2 in order to test whether $\mathbf{D} \models \phi(\bar{a})$ or not. If it is the case, then C is relevant for $\mathbf{D}$ by Corollary 5.2.1 and if it is not the case, then C is not relevant for $\mathbf{D}$ by the same corollary. Each application of Theorem 5.2.2 works in time $2^{2^{2^{O(|\phi|)}}}$ and we apply it once for each r-partition. The number of r-partitions is bounded by $(|\mathcal{T}_{r(2k+1)}|)^k = 2^{2^{2^{O(|\phi|)}}}$ multiplied by the number of possible splits of k variables into disjoint and nonempty subsets, multiplied again by $(|\mathcal{F}_{r(2k+1)}|)^k$, which altogether is $2^{2^{2^{O(|\phi|)}}}$. This way all these tests take time $2^{2^{2^{O(|\phi|)}}}$. It remains to show how to find a tuple $\bar{a}$ such that $\mathbf{D} \models \phi(\bar{a})$ under the same time constraints. A somewhat brute-force approach is sufficient for that: recall that during the third step the algorithm computed for each $(r(2k+1))$-neighborhood type τ the list of elements sharing that type. Assume now that one of the lists for $\tau_1, \ldots, \tau_m$ has *many* elements (we shall precise the number shortly). Wlog assume that this is the case for τ_m and set $C' = \{(C_1, F_1, \tau_1), \ldots, (C_{m-1}, F_{m-1}, \tau_{m-1})\}$. Note that if we have a tuple $\bar{b}$ such that $\mathbf{D} \models \text{Div}_r^{C'}(\bar{b})$, then the number of nodes c of type τ_m such that $N_r(F_m(c)) \cap N_r(\bar{b}) \neq \emptyset$ is bounded by kd^{2rk} (see Observation 5.2.6). If our *many* is a number greater than this value, then we can just go through the list of elements with type τ_m and extend the tuple $\bar{b}$ to a desired solution to $\text{Div}_r^C(\bar{x})$. This extension being doable in time doubly exponential in $|\phi|$, we can apply the inductive reasoning to C' to compute tuple $\bar{b}$. It remains to show the base of the induction, that is to solve the case when none of the lists has *many* elements. But this is then trivial: we simply test all possible choices. Their number being bounded by $(kd^{r(2k+1)})^k$, we fit into the required time bound of $2^{2^{2^{O(|\phi|)}}}$.

Altogether we have shown an algorithm constructing the basic index structure for ϕ and $\mathbf{D}$ within the desired time constraints: it works in time linear in $\|\mathbf{D}\|$ and triply exponential in $|\phi|$.

■

5.3.2 Towards counting

We are now aiming at an extension of the basic index structure from Section 5.3.1, so that it could also be used to handle the counting and j-th solution problems. Before we move into details, we briefly sketch the ideas behind our upcoming approach.

Fix a database $\mathbf{D}$. Let $|\bar{x}| = k$, $|\bar{y}| = l$ and fix r-partitions $C = \{(C_1, F_1, \tau_1), \dots, (C_m, F_m, \tau_m)\} \in \mathcal{C}_r^k(\bar{x})$ and $C' = \{(\bar{y}, F_{m+1}, \tau_{m+1})\} \in \mathcal{C}_r^l(\bar{y})$. Let $\bar{a}$ be a tuple of nodes from $\mathbf{D}$ such that $\mathbf{D} \models \text{Div}_r^C(\bar{a})$. The information that we are going to need is the value $s_{\bar{a}} = |\{\bar{b} : \mathbf{D} \models \text{Div}_r^{C'}(\bar{b}) \text{ and } N_r(\bar{a}) \cap N_r(\bar{b}) \neq \emptyset\}|$. For the purpose of this section we call a tuple $\bar{b}$ such that $\mathbf{D} \models \text{Div}_r^{C'}(\bar{b})$ and $N_r(\bar{a}) \cap N_r(\bar{b}) \neq \emptyset$ a *disallowed tuple for* $\bar{a}$. If $\bar{a}$ is clear from the context we just write that $\bar{b}$ is a *disallowed tuple*.

Of course having $\bar{a}$ it is easy to compute $s_{\bar{a}}$: if $\mathbf{D} \models \text{Div}_r^{C'}(\bar{b})$, then $\bar{b}$ is r-connected around b_1 and from Observation 5.2.5 it is enough to investigate $N_{2rl}(\bar{a})$ when searching for possible disallowed tuples. From Observation 5.2.6 we know that there are up to kd^{2rl} choices for the possible positions of b_1 to consider and since $\bar{b} = F_{m+1}(b_1)$ we immediately get an algorithm computing $s_{\bar{a}}$ together with a bound $s_{\bar{a}} \leq kd^{2rl}$.

What we are aiming at is a way to refine the notion of the C-split Div_r^C so that it would group the tuples $\bar{a}$ having the same number of disallowed tuples. Fix $0 \leq s \leq kd^{2rl}$. We would like to be able to compute a query s-Div_r^C with the following property: for any tuple $\bar{a}$ we have $\mathbf{D} \models s\text{-Div}_r^C(\bar{a})$ iff $\mathbf{D} \models \text{Div}_r^C(\bar{a})$ and $s_{\bar{a}} = s$.

Let us start with a simple case when $m = 1$. From Observation 5.2.4 we know that $\bar{a} \subseteq N_{2rk}(a_1)$, so $N_{2rl}(\bar{a}) \subseteq N_{2r(l+k)}(a_1)$. We again have a constant bound on the size of $N_{2r(l+k)}(a_1)$, so with a linear pass on the domain of $\mathbf{D}$ we could extract the possible candidates for a_1 and group them together by refining the notion of neighborhood types in such a way that the $(2r(l+k))$-neighborhood type of a_1 would store the value $s_{\bar{a}}$. In fact no real refinement is actually going to be necessary: $\text{Div}_r^{C'}$ only tests the $(2rl)$-neighborhood type of b_1 and the positioning of $\bar{b}$ inside the $N_{2rl}(b_1)$ (to make sure that it agrees with $F_{m+1}(b_1)$). As soon as this $(2rl)$-neighborhood of possible candidates for b_1 is fully contained in some p-neighborhood of a_1 (we are going to see that $p = 2r(l + k + 1)$ is going to suffice), all the required information is already stored by p-neighborhood type of a_1.

The biggest issue with extending the above reasoning to $m > 1$ is that $\bar{a}$ no longer has to be r-connected. The natural approach to overcome this is to separately count the number of disallowed tuples that intersect with each C-component $\bar{a}^i$. This is exactly what we are going to do, but the reader should keep in mind that this needs to be done with care as the same tuple $\bar{b}$ may be a disallowed tuple for different C-components $\bar{a}^i$ and $\bar{a}^j$ of $\bar{a}$ and we need to make sure that we do not count $\bar{b}$ twice in such a case.

Fix for now a tuple $\bar{a}$ such that $\mathbf{D} \models \text{Div}_r^C(\bar{a})$ and a tuple $\bar{b}$ such that $\mathbf{D} \models \text{Div}_r^{C'}(\bar{b})$ and $N_r(\bar{a}^i) \cap N_r(\bar{b}) \neq \emptyset$ and also $N_r(\bar{a}^j) \cap N_r(\bar{b}) \neq \emptyset$ for two different C-components $\bar{a}^i$ and $\bar{a}^j$ of $\bar{a}$. The key to the solution is the following observation: the above assumptions imply that $\bar{a}^i$ and $\bar{a}^j$, although their r-neighborhoods do not intersect, are in fact "close" to each other as witnessed by $\bar{b}$. By "close" we mean that $N_{r(4l+2)}(\bar{a}^i) \cap N_{r(4l+2)}(\bar{a}^j) \neq \emptyset$. This way, instead of looking at a r-partition C of $\bar{a}$, we are rather going to look at its particular $(r(4l+2))$-partition C'' (to be precise, we are actually going to consider a set of $(r(4l+2))$-partitions, see Lemma 5.3.2 for details). This $(r(4l+2))$-partition C'' is going to imply C, but it also has the following nice property: since all its C''-components are at distance of at least $r(8l+4)$ from each other (which is enforced by $(r(4l+2))$-partitions), their r-neighborhoods can no longer intersect with the same disallowed tuple. This way we may count the number of disallowed tuples for each C''-component separately and we already know how to do this.

In the sequel we formalize the above intuition. Later on we define the basic index structure with counting and then we move to Section 5.4 where we show how to use the index structures to effectively solve the enumeration, testing, counting and j-th solution problems.

5.3.3 Increasing the radius

We start with two simple observations that intuitively can be understood in the following way: when r increases, the r-connectedness is preserved and the r-partitioning may only "merge" previously separated components.

Observation 5.3.1 *Fix $r \leq r' \in \mathbb{N}$, a structure $\boldsymbol{D}$ and a tuple $\bar{a}$ of k nodes from $\boldsymbol{D}$. If $\bar{a}$ is r-connected around a_1 then it is also r'-connected around a_1.*

As a consequence of Observations 5.3.1 and 5.2.3 we get:

Observation 5.3.2 *Fix $r \leq r' \in \mathbb{N}$, a structure $\boldsymbol{D}$ and a tuple $\bar{a}$ of k nodes from $\boldsymbol{D}$. Let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ be a r-partition of $\bar{x}$ and $C' = \{(C'_1, F'_1, \tau'_1), \ldots, (C'_{m'}, F'_{m'}, \tau'_m)\}$ be a r'-partition of $\bar{x}$ such that $\boldsymbol{D} \models \exists \bar{x} \left(\mathrm{Div}^C_r(\bar{x}) \wedge \mathrm{Div}^{C'}_{r'}(\bar{x}) \right)$. Then $m' \leq m$ and for each $1 \leq i \leq m$ there exists $1 \leq f(i) \leq m'$ such that $C_i \subseteq C'_{f(i)}$.*

Let us have a closer look at the situation described in Observation 5.3.2. Fix $r \leq r' \in \mathbb{N}$ and a structure $\mathbf{D}$. Let $|\bar{x}| = k$ and let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ be a r-partition of $\bar{x}$ and let $C' = \{(C'_1, F'_1, \tau'_1), \ldots, (C'_{m'}, F'_{m'}, \tau'_{m'})\}$ be a r'-partition of $\bar{x}$ such that for each $1 \leq i \leq m$ there exists $1 \leq f(i) \leq m'$ such that $C_i \subseteq C'_{f(i)}$. Note that there is no reason for the above assumptions to yield any of the implications $\mathbf{D} \models \forall \bar{x} \left(\mathrm{Div}^C_r(\bar{x}) \to \mathrm{Div}^{C'}_{r'}(\bar{x}) \right)$ or $\mathbf{D} \models \forall \bar{x} \left(\mathrm{Div}^{C'}_{r'}(\bar{x}) \to \mathrm{Div}^C_r(\bar{x}) \right)$. Fortunately, with a bit of additional work, we can enforce a certain equivalence as shown in Lemma 5.3.2.

Lemma 5.3.2 *Fix $r \in \mathbb{N}$ and a structure $\boldsymbol{D}$. Let $|\bar{x}| = k$ and let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ be a r-partition of $\bar{x}$. For any $r' \geq r(2k+1)$ there is a set S of r'-partitions of $\bar{x}$ such that the following holds: $\boldsymbol{D} \models \forall \bar{x} \left[\left(\bigvee_{C' \in S} \mathrm{Div}^{C'}_{r'}(\bar{x}) \right) \leftrightarrow \mathrm{Div}^C_r(\bar{x}) \right]$. Moreover, set S has size $O(2^{2^{r'k}})$ and can be computed in time $O(2^{2^{r'k}})$ (in particular, in time independent from the size of $\boldsymbol{D}$).*

Before we get to the proof let us note that there is nothing really deep in this lemma. It is yet another step towards formalizing the approach explained in the beginning of Section 5.3.2. We explained there that counting the number of disallowed tuples needs to be done with care since different components may share disallowed tuples. The simplest solution to avoid such a scenario is to make sure that the components are "far enough" from each other. $r' \geq r(2k+1)$ is the "far enough" threshold that guarantees that the sets of disallowed tuples of different components are disjoint. The proof of Lemma 5.3.2 describes a way of dealing with the emerging technicalities, but there is nothing particularly surprising or revealing in it.
PROOF [of Lemma 5.3.2]
Let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ be a r-partition of $\bar{x}$ and fix $r' \geq r(2k+1)$.

Let $C' = \{(C'_1, F'_1, \tau'_1), \dots, (C'_{m'}, F'_{m'}, \tau'_{m'})\}$ be a r'-partition of $\bar{x}$.

If it is not the case that for each $1 \le i \le m$ there exists $1 \le f(i) \le m'$ such that $C_i \subseteq C'_{f(i)}$, then by Observation 5.3.2 we have $\mathbf{D} \models \forall \bar{x} \neg \left(\text{Div}^{C'}_{r'}(\bar{x}) \wedge \text{Div}^{C}_{r}(\bar{x}) \right)$. This way none of such r'-partitions may belong to the desired set S.

Assume then that C' is such that for each $1 \le i \le m$ there exists $1 \le f(i) \le m'$ such that $C_i \subseteq C'_{f(i)}$.

Fix $1 \le i \le m$.

We now argue that the $(r(2k+1))$-neighborhood of x^i_1 is completely included in the $(r'(2k+1))$-neighborhood of $x^{f(i)}_1$. Indeed, since $\bar{x}^{f(i)}$ is r'-connected around $\bar{x}^{f(i)}_1$ and $\bar{x}^i \subseteq \bar{x}^{f(i)}$, the distance between $x^{f(i)}_1$ and x^i_1 is bounded by $2r'k$. Having $r' \ge r(2k+1)$ concludes the fact that $N_{r(2k+1)}(x^i_1) \subseteq N_{r'(2k+1)}(x^{f(i)}_1)$. Moreover, as $(r'(2k+1))$-neighborhood type of $x^{f(i)}_1$ is $\tau'_{f(i)}$ and the position of x^i_1 is uniquely determined by $F'_{f(i)}$, the $(r(2k+1))$-neighborhood type of x^i_1 is uniquely determined by $\tau'_{f(i)}$ and $F'_{f(i)}$ and so knowing C' we know whether the $(r(2k+1))$-neighborhood type of x^i_1 is implied to be τ_i or not.

If not, then clearly C' cannot be included in S (since each node has a fixed $(r(2k+1))$-neighborhood type).

If this happens to be the case and the $(r(2k+1))$-neighborhood type of x^i_1 is now fixed to be τ_i, then $\tau'_{f(i)}$ and $F'_{f(i)}$ uniquely determine whether the positioning of $\bar{x}^i$ inside the $(r'(2k+1))$-neighborhood of $x^{f(i)}_1$ as given by $F'_{f(i)}$ is in fact in the $(r(2k+1))$-neighborhood of x^i_1 and if it agrees with its positioning as given by F_i and τ_i.

If this is not the case, then again C' cannot be included in S (since the structure of the i-th C-component would not be preserved).

After performing this sieve for each $1 \le i \le m$ we are left with exactly those C', for which $\text{Div}^{C'}_{r'}(\bar{x})$ implies that the $r(2k+1)$-neighborhood types of x^i_1 are τ_i and that $\bar{x}^i = F_i(x^i_1)$ for each $1 \le i \le m$.

It remains to assure that $N_r(\bar{x}^i) \cap N_r(\bar{x}^j) = \emptyset$ for $i \ne j$. If $f(i) \ne f(j)$, then this is immediate since even $N_{r'}(\bar{x}^i) \cap N_{r'}(\bar{x}^j) = \emptyset$. If $f(i) = f(j)$, then similarly as before we see that $N_r(\bar{x}^i), N_r(\bar{x}^j) \subseteq N_{r'(2k+1)}(x^{f(i)}_1)$ and that $\tau'_{f(i)}$ and $F'_{f(i)}$ uniquely determine the pairwise positioning of $N_r(\bar{x}^i)$ and $N_r(\bar{x}^j)$.

If the r-neighborhoods of $\bar{x}^i$ and $\bar{x}^j$ are implied to have a nonempty intersection, then C' cannot be included in S.

If they are implied to have empty intersection for each pair $i \ne j$, then each such r'-partition C' implies r-partition C and we may conclude with S being the set of exactly those remaining r'-partitions.

By construction we have the left to right implication, namely $\mathbf{D} \models \forall \bar{x} \left[\left(\bigvee_{C' \in S} \text{Div}^{C'}_{r'}(\bar{x}) \right) \to \text{Div}^{C}_{r}(\bar{x}) \right]$.

The other direction is a consequence of Observation 5.2.1 and the fact that we only discarded those r'-partitions which violated the requirements of r-partition C: for each tuple $\bar{a}$ such that $\mathbf{D} \models \text{Div}^{C}_{r}(\bar{a})$ we know that there exists a unique r'-partition C' of $\bar{a}$. Since $\text{Div}^{C}_{r}(\bar{a})$, C' cannot violate the requirements of C and so it was not discarded. This way we have $\mathbf{D} \models \forall \bar{x} \left[\text{Div}^{C}_{r}(\bar{x}) \to \left(\bigvee_{C' \in S} \text{Div}^{C'}_{r'}(\bar{x}) \right) \right]$ which concludes the proof that the constructed set S is in fact the desired one.

Note that the above procedure performs a series of operations for each r'-partition. Each such operation being doable in time bounded by the maximal size of a $(r'(2k+1))$-neighborhood, the procedure requires a total time $2^{2^{O(r'k)}}$.

This gives the desired set S within required time constraints.

■

Lemma 5.3.2 actually yields a bit more. As we are about to see, r'-partitions mentioned in its statement enjoy one more property that will be very useful for the counting and j-th solution problems. For technical reasons we are going to require a slightly bigger threshold this time (the factor will be $4k + 2$ rather than $2k + 1$), but just as it was the case for Lemma 5.3.2, there is nothing deep in the following lemma. It just formalizes the fact that if different components are "far enough" from each other and their "big enough" neighborhood types are known, then the number of disallowed tuples (as they were defined in the beginning of Section 5.3.2) is implied by the partition.

Lemma 5.3.3 *Fix $r \in \mathbb{N}$ and a structure $\boldsymbol{D}$. Let $|\bar{x}| \leq k$, $|\bar{y}| \leq k$ and let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ be a r-partition of $\bar{x}$ and $C'' = \{(C_{m+1}, F_{m+1}, \tau_{m+1})\}$ be a r-partition of $\bar{y}$. Fix $r' \geq r(4k + 2)$ and let S be the set of r'-partitions from the application of Lemma 5.3.2 to C and r'. For any r'-partition $C' = \left\{(C'_1, F'_1, \tau'_1), \ldots, (C'_{m'}, F'_{m'}, \tau'_{m'})\right\} \in S$ of $\bar{x}$ we have that:*

there exists a number $s_{C'}$ such that for any $\bar{a}$ for which $\boldsymbol{D} \models Div^{C'}_{r'}(\bar{a})$ we have that $|\{\bar{b} : \boldsymbol{D} \models Div^{C''}_{r}(\bar{b})$ and $N_r(\bar{a}) \cap N_r(\bar{b}) \neq \emptyset\}| = s_{C'}$.

Moreover, value $s_{C'}$ can be computed in time $O(2^{2^{r'k}})$ (in particular, in time independent from the size of $\boldsymbol{D}$).

PROOF

We use all the terminology as introduced in the statement of this lemma. Fix r'-partition $C' = \left\{(C'_1, F'_1, \tau'_1), \ldots, (C'_{m'}, F'_{m'}, \tau'_{m'})\right\} \in S$ of $\bar{x}$.

Let $\bar{a}$ be such that $\mathbf{D} \models \text{Div}^{C'}_{r'}(\bar{a})$. By the fact that $r' \geq r(4k+2)$, the $(r(4k+2))$-neighborhoods of each pair $\bar{a}^i$ and $\bar{a}^j$ of different C'-components do not intersect. This means that their r-neighborhoods are at distance of at least $8rk$ and so for any $\bar{b}$ such that $\mathbf{D} \models \text{Div}^{C''}_{r}(\bar{b})$ the r-neighborhood of $\bar{b}$ cannot intersect with both r-neighborhoods of $\bar{a}^i$ and $\bar{a}^j$ ($\bar{b}$ is r-connected around b_1, so its width is bounded by $2rk$).

Assume now that $N_r(\bar{b}) \cap N_r(\bar{a}^i) \neq \emptyset$ for some tuple $\bar{b}$ such that $\mathbf{D} \models \text{Div}^{C''}_{r}(\bar{b})$. This means that the distance from a^i_1 to b_1 is bounded by $2r'k + 2r + 2rk$ (since width of $\bar{a}^i$ is bounded by $2r'k$, width of $\bar{b}$ is bounded by $2rk$ and the $2r$ part comes from the fact that we are looking at the r-neighborhoods of mentioned tuples) and so it is included in the $(r'(2k + 1))$-neighborhood of a^i_1 (as $r' \geq r(4k + 2)$). In fact the lower bound on r' gives that the $N_{r(2k+1)}(b_1)$ is included in the $(r'(2k + 1))$-neighborhood of a^i_1.

But this basically finishes the proof. It guarantees that the $r'(2k + 1)$-neighborhood type of a^i_1 uniquely determines the number of tuples $\bar{b}$ such that $N_r(\bar{b}) \cap N_r(\bar{a}^i) \neq \emptyset$ and $\mathbf{D} \models \text{Div}^{C''}_{r}(\bar{b})$: the possible candidates for b_1 are only those nodes inside $N_{r'(2k+1)}(a^i_1)$, whose $(r(2k+1))$-neighborhoods are fully included in $N_{r'(2k+1)}(a^i_1)$ and whose induced r-neighborhood type is τ_{m+1}. This allows to extract $\bar{b} = F_{m+1}(b_1)$ and then it remains to check the non-emptiness of the intersection of r-neighborhoods of $\bar{a}^i$ and $\bar{b}$.

Examining every node from $N_{r'(2k+1)}(x^i_1)$ in time $O(2^{2^{r'k}})$ we can compute the value s_i such that for any tuple $\bar{a}$ for which $\mathbf{D} \models \text{Div}^{C'}_{r}(\bar{a})$, we have $|\{\bar{b} : \mathbf{D} \models \text{Div}^{C''}_{r}(\bar{b})$ and $N_r(\bar{a}^i) \cap N_r(\bar{b}) \neq \emptyset\}| = s_i$.

As we explained before, we are sure not to count the same tuple $\bar{b}$ for different s_i.

The desired value $s_{C'}$ is then given by $s_{C'} = \sum_{1 \leq i \leq m'} s_i$.

■

5.3.4 The basic index structure with counting

We finally move to the definition of the basic index structure with counting. We implicitly assume that it is being build on top of the basic index structure.

Fix $r \in \mathbb{N}$ and a structure $\mathbf{D}$. Let $|\bar{x}| \leq k$, $|\bar{y}| \leq k$ and let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\}$ be a r-partition of $\bar{x}$ and $C'' = \{(C_{m+1}, F_{m+1}, \tau_{m+1})\}$ be a r-partition of $\bar{y}$. Fix $r' \geq r(4k+2)$.

The basic index structure with counting for C, C'' and r' stores:

• set S of r'-partitions of $\bar{x}$ as obtained from the application of Lemma 5.3.2 to C and r',

• for each $C' \in S$ it stores the value $s_{C'}$ obtained from the application of Lemma 5.3.3 to C, C'', r' and C'.

Note that from Lemmas 5.3.2 and 5.3.3 the basic index structure with counting for C, C'' and r' can be computed in time $2^{2^{O(r'k)}}$.

5.4 Solving the problems

5.4.1 Enumeration of FO queries

We now turn to the proof of Theorem 5.1.1. It is a consequence of the following theorem, which also gives the precise (in terms of the size of the formula) constants that are hidden in the statement of Theorem 5.1.1.

Theorem 5.4.1 ([46]) *There is an algorithm that for all $d \in \mathbb{N}$, all $\phi \in$ FO and all d-degree-bounded structures $\boldsymbol{D}$ enumerates $\phi(\boldsymbol{D})$ with a precomputation phase taking time $2^{2^{2^{O(|\phi|)}}} \cdot \|\boldsymbol{D}\|$ and a delay during the enumeration phase that is triply exponential in $|\phi|$. Moreover, if the domain of $\boldsymbol{D}$ is linearly ordered, the algorithm enumerates $\phi(\boldsymbol{D})$ in increasing order relative to the induced lexicographical order on tuples.*

PROOF

Let $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\} \in \mathcal{C}^k_r(\bar{x})$ be a r-partition relevant for ϕ. We show how to enumerate (after a linear preprocessing) in lexicographical order, with no repetition, constant memory and constant delay, all the tuples $\bar{a}$ such that $\mathbf{D} \models \mathrm{Div}^C_r(\bar{a})$. The result will then follow from Lemma 5.2.1 and Fact 3.1.4.

The first step of the precomputation phase is building the basic index structure for ϕ and $\mathbf{D}$ as described in Section 5.3.1. Using the basic index structure we now have access to all the r-partitions that are relevant for ϕ. Fix a relevant for ϕ r-partition $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\} \in \mathcal{C}^k_r(\bar{x})$. As we argued earlier, it is enough to show the enumeration procedure for each $\mathrm{Div}^C_r(\bar{x})$ separately.

The proof is an induction on the number k of free variables in $\mathrm{Div}^C_r(\bar{x})$. The base case $k = 1$ is trivial: since $k = 1$ it must be the case that also $m = 1$ and the basic index structure contains the desired list of all elements of type τ_1.

Assume now that we have the desired result for $k-1$ and we want to extend it to k.

Without loss of generality we assume that the most significant variable of $\bar{x}$ is in the first variable of $\bar{x}^1$, that the most significant variable of $\bar{x} \setminus \bar{x}^1$ is the first variable of $\bar{x}^2$ and so on.

Let $\bar{x}' = C_1 \cup \ldots \cup C_{m-1}$ and $\bar{x}'' = C_m$. Consider query $\psi(\bar{x}') = \exists \bar{x}'' \mathrm{Div}^C_r(\bar{x})$. It has less variables than $\mathrm{Div}^C_r(\bar{x})$ and so the inductive hypothesis holds for it.

The precomputation phase, besides building the basic index structure for ϕ and $\mathbf{D}$ as we mentioned earlier, also performs the precomputation as given by the inductive hypothesis for ψ.

We now turn to the enumeration phase.

Let C' be C with (C_m, F_m, τ_m) removed. The algorithm works as follows: by inductive hypothesis it enumerates solutions to ψ. Note that, from the definition of ψ, for each solution $\bar{a}$ that the enumeration procedure returns, we are sure to find at least one tuple $\bar{b}$ such that $\mathbf{D} \models \mathrm{Div}^C_r(\bar{a}\bar{b})$. Moreover, it is always the case that $\mathbf{D} \models \mathrm{Div}^{C'}_r(\bar{a})$. Fix $\bar{a}$ returned by the inductive enumeration procedure. We now show how to nest the enumeration of all matching tuples $\bar{b}$:

Using the basic index structure it is possible to iterate one by one through all elements b_m of $\mathbf{D}$ whose $(r(2k+1))$-neighborhood type is τ_m. For each such element let $\bar{b}^m = F_m(b_m)$. The algorithm now tests whether $N_r(\bar{b}^m)$ intersects with $N_r(\bar{a})$ or not (recall that this information requires only constant time by Observation 5.2.5 and the fact that the appropriate neighborhoods have been computed). If it does not, then we have a solution $\bar{a}\bar{b}^m$ for $\text{Div}_r^C(\bar{x})$. If it does, then we move to the next element with $(r(2k+1))$-neighborhood type τ_m. Notice that the size of $N_r(\bar{a})$ is bounded by kd^r hence the length of false hits is bounded by kd^{2rk} (see Observation 5.2.6). As we said before, for each considered $\bar{a}$ we are sure to find at least one solution. Altogether, we get the desired constant delay for the enumeration process.

By the assumption that each $\bar{x}^i$ for $i < m$ contains a variable more significant than any variable from $\bar{x}^m$, the lexicographical order on the output follows.

There are no repetitions since the list of elements whose $(r(2k+1))$-neighborhood type is τ_m contains no repetitions.

During the enumeration phase we only check if certain neighborhoods of constant size intersect, so the constant memory assumption is clearly satisfied.

We have shown how to enumerate (after a linear preprocessing) in lexicographical order, with no repetition, constant memory and constant delay, all the tuples $\bar{a}\bar{b}$ such that $\mathbf{D} \models \text{Div}_r^C(\bar{a}\bar{b})$.

The enumeration phase needs to process all relevant r-partitions C, i.e. a number of cases triply exponential in $|\phi|$. The depth of the induction is bounded by k so altogether we have a number of cases that is triply exponential in $|\phi|$ raised to the power k, which is still triply exponential in $|\phi|$. Since the disjunction given by Lemma 5.2.1 is mutually exclusive, we could consider consecutive r-partitions C sequentially. Though in order to enforce lexicographical order on the output, we use Fact 3.1.4 instead.

Altogether this yields a procedure linear in the size of the output and triply exponential in $|\phi|$.

■

5.4.2 Testing FO queries

We now turn to the proof of Theorem 5.1.2.

PROOF [of Theorem 5.1.2] The precomputation phase builds the basic index structure for ϕ and $\mathbf{D}$ as described in Section 5.3.1.

The testing phase is then trivial: given a tuple $\bar{a}$ it is enough to test whether the r-partition $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\} \in C_r^k(\bar{x})$ as given by Observation 5.2.1 for $\bar{a}$ is relevant for ϕ (then the result follows from Lemma 5.2.1). This is done in time triply exponential in the size of the formula and independent from the size of the input structure.

■

5.4.3 Counting FO queries

We now turn to the proof of Theorem 5.1.3.

PROOF [of Theorem 5.1.3] We start with the construction of the basic index structure as described in Section 5.3.1.

From Lemma 5.2.1 it is enough to show the counting algorithm for a single r-partition $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\} \in C_r^k(\bar{x})$ relevant for ϕ. (as Lemma 5.2.1 guarantees mutual exclusion of the mentioned queries).

We need to count the number of solutions to a C-split $\text{Div}_r^C(\bar{x})$. This is done by induction on m. Case $m = 1$ is trivial, as it is enough to count the number of elements with type τ_m and this can easily be done with the basic index structure.

We now move to the inductive step: assume the result for values smaller than m and we want to extend it to m.

Set $C'' = \{(C_1, F_1, \tau_1), \ldots, (C_{m-1}, F_{m-1}, \tau_{m-1})\}$. Let $\bar{x}' = C_1 \cup \ldots \cup C_{m-1}$ and fix a tuple $\bar{a}$ such that $\mathbf{D} \models \mathrm{Div}_r^{C''}(\bar{a})$.

Let N be the number of elements b in $\mathbf{D}$ with type τ_m. From Observation 5.2.6 we know that constantly many (up to kd^{2rk}) of them are such that $N_r(F_m(b)) \cap N_r(\bar{a}) \neq \emptyset$. For every $0 \leq s \leq kd^{2rk}$ we show how to count the number $\#_s$ of tuples $\bar{a}$ that are solutions to $\mathrm{Div}_r^{C''}(\bar{x}')$ and such that their r-neighborhoods intersect with exactly s r-neighborhoods of different $F_m(b)$, where b is of type τ_m. Then the solution to the counting problem is $\sum_s \#_s \cdot (N - s)$.

Fix $s \leq kd^{2rk}$ and set $r' = r(4k+2)$. We now construct the basic index structure with counting for C, C'' and r' as defined in Section 5.3.4. Let us consider the disjunction $\bigvee_{C' \in S} \mathrm{Div}_{r'}^{C'}(\bar{x}')$ obtained from the application of Lemma 5.3.2 to $\mathrm{Div}_r^{C''}(\bar{x}')$ and r'. This disjunction is equivalent to $\mathrm{Div}_r^{C''}(\bar{x}')$ and is mutually exclusive, so we can handle each $C' \in S$ one by one.

From Lemma 5.3.3 we know that we need to consider exactly those C' for which $s_{C'} = s$. As the r'-partition C' has strictly less components than m, we can compute $\#_s$ using the inductive hypothesis.

Note that we need to consider $2^{2^{O(r'k)}}$ choices for s and C'. Moreover, each application of the induction causes r' to grow by a factor $(4k+2)$. Since the depth of the induction is bounded by k, the maximal value of r' is bounded by $r(4k+2)^k$. Altogether this yields a procedure working in time $2^{2^{O(r(4k+2)^k)}}$ which is triply exponential in $|\phi|$ since $r = 2^{|\phi|}$ and $k \leq |\phi|$.

This concludes the proof of Theorem 5.1.3.

■

5.4.4 j-th solution problem for FO queries

Finally, we turn to the proof of Theorem 5.1.4. It follows the line of the proof of Theorem 5.1.3.

PROOF [of Theorem 5.1.4] We start with the construction of the basic index structure as described in Section 5.3.1.

From Lemma 5.2.1 it is enough to show the j-th solution algorithm for a single r-partition $C = \{(C_1, F_1, \tau_1), \ldots, (C_m, F_m, \tau_m)\} \in C_r^k(\bar{x})$ relevant for ϕ. (Lemma 5.2.1 guarantees mutual exclusion of the mentioned queries, Theorem 5.1.3 can be used to solve the respective counting problems and we may conclude with the application of Fact 3.1.3).

We now show the solution to the j-th solution problem for a single C-split $\mathrm{Div}_r^{C,\bar{\tau}}(\bar{x})$. This is done by induction on m. Case $m = 1$ is trivial: the preprocessing phase puts all nodes from $\mathbf{D}$ with type τ_1 into an array and then during the j-th solution phase the appropriate solution can be returned in constant time with a direct access to the j-th element of that array.

We now move to the inductive step: assume the result for values smaller than m and we want to extend it to m.

Set $C'' = \{(C_1, F_1, \tau_1), \ldots, (C_{m-1}, F_{m-1}, \tau_{m-1})\}$ and let $\bar{x}' = C_1 \cup \ldots \cup C_{m-1}$. Let N be the number of elements b in $\mathbf{D}$ with type τ_m.

Similarly as it was done for the counting problem, we split the reasoning to the following mutually exclusive sub-queries: for every $0 \leq s \leq kd^{2rk}$ we show how to solve the j-th solution problem for $\mathrm{Div}_r^{C''}(\bar{x}')$ limited to those solutions such that their r-neighborhoods intersect with exactly s different r-neighborhoods of tuples $F_m(b)$, where b is of type τ_m. As we know how to count the number of solutions to such queries (see Theorem 5.1.3), we can use Fact 3.1.3 in this case too.

Fix $s \leq kd^{2rk}$ and set $r' = r(4k+2)$. We now construct the basic index structure with counting for C, C'' and r' as defined in Section 5.3.4. Let us consider the disjunction $\bigvee_{C' \in S} \mathrm{Div}_{r'}^{C'}(\bar{x}')$ obtained from the application of Lemma 5.3.2 to $\mathrm{Div}_r^{C''}(\bar{x}')$ and r'. This disjunction is equivalent to $\mathrm{Div}_r^{C''}(\bar{x}')$ and is mutually exclusive. But more interestingly, the subset of solutions to $\mathrm{Div}_r^{C''}(\bar{x}')$ that we are interested in is exactly the set of solutions to those $\mathrm{Div}_{r'}^{C'}(\bar{x}')$, for which $s_{C'} = s$. As we know how to count the number of solutions to such queries (cf. Theorem 5.1.3), we can use Fact 3.1.3 in this case too and restrict ourselves to a single r'-partition $C' \in S$ for which $s_{C'} = s$.

We conclude the preprocessing phase with performing the preprocessing phase as given by the inductive hypothesis for the r'-partition C' (recall that it has less than m components).

We now turn to the j-th solution phase.

The j-th solution to $\mathrm{Div}_r^{C}(\bar{x})$ is clearly $(j \text{ div } (N - s))$-th solution $\bar{a}$ to $\mathrm{Div}_{r'}^{C''}(\bar{x})$ extended with $(j \bmod (N - s))$-th element b of type τ_m such that $N_r(F_m(b)) \cap N_r(\bar{a}) = \emptyset$. We compute $\bar{a}$ using the inductive hypothesis for $\mathrm{Div}_{r'}^{C''}(\bar{x}')$. To find the desired element b, first compute all elements $b^1 < b^2 < \ldots < b^s$ such that $N_r(F_m(b^i)) \cap N_r(\bar{a}) \neq \emptyset$ (this can be done in the required time constraints by looking at $N_{2r}(\bar{a})$). Now let b be the $(j \bmod (N - s))$-th element on the sorted array of elements of type τ_m and set $i = 1$. As long as $b \geq b^i$ do $i = i + 1$ and replace b with its successor on the array of elements of type τ_m (this way we find smallest i such that $b^{i-1} < b < b^i$ and b is the $((j \bmod (N - s)) + i - 1)$-th element on the list of elements with type τ_m). Clearly this procedure finishes in constant time and the obtained node b has the desired properties, namely tuple $(\bar{a}F_m(b))$ is the required j-th solution.

The precise analysis of the involved constants is similar to the one from the proof of Theorem 5.1.3. Note that we need to consider $2^{2^{O(r'k)}}$ choices for s and C'. Moreover, each application of the induction causes r' to grow by a factor $(4k + 2)$. Since the depth of the induction is bounded by k, the maximal value of r' is bounded by $r(4k + 2)^k$. Altogether this yields a procedure working in time $2^{2^{O(r(4k+2)^k)}}$ which is triply exponential in $|\phi|$ since $r = 2^{|\phi|}$ and $k \leq |\phi|$.

This concludes the proof of Theorem 5.1.4.

■

6

MSO over classes of structures with bounded treewidth

Contents

6.1 Introduction

In Chapter 5 we have shown that the query problems that are of the biggest interest from the point of view of this thesis (that is enumeration, testing, counting and j-th solution, see Section 2.5 for details) admit very good algorithmic properties when are considered with respect to the first-order logic over the classes of databases of bounded degree. In order to extend Theorems 5.1.1–5.1.4, there are two natural directions that we might follow:

- we can give more power to the logic,
- or we can extend the class of structures.

To see what happens when we extend the class of structures, the reader is referred to Chapter 7. In this chapter we investigate the case when the considered logic is MSO instead of FO.

The moment we switch from FO to MSO, we start facing different problem that we had previously. The parametrized model checking problem for MSO over the class of structures of bounded degree is most likely not in FPT (the tiling problem, which can be expressed in MSO in a canonical way, is know to be NP-complete already in the bounded degree case, so unless P $=$ NP, there is no hope for the mentioned FPT algorithm for MSO), so we are "forced" to change the class of structures if we want to retain good algorithmic properties. The path that we are going to follow is not to limit the allowed class of structures, but rather to go in parallel.

The considered class will be a class of structures with bounded treewidth. Just as it is always the case when searching for CONSTANT-DELAY$_{lin}$ algorithms, the starting point is the model checking problem. Over the class of structures with bounded treewidth, its linear time solution is a celebrated result of Courcelle:

Theorem 6.1.1 ([22]) *Fix $d \in \mathbb{N}$. The problem of whether a given structure $\boldsymbol{D}$ of d-bounded treewidth satisfies a given second-order sentence ϕ is decidable in time $O(\|\boldsymbol{D}\|)$.*

The proof of Courcelle's Theorem explains how a tree decomposition can be efficiently encoded into MSO and then relies on the well known connection between MSO and finite tree automatons. Continuing this path, we are also going to rely on an extension of Courcelle's result to an evaluation algorithm from LINEAR-EVAL (see Theorem 6.2.1).

We would like to lift these results to an enumeration algorithm. Although with MSO we could potentially encounter free set variables, recall that our definition of a query is restricted to only first-order free variables (but set quantification inside the formula is of course allowed). The results that we are going to prove in details are as follows:

Theorem 6.1.2 ([8, 48]) *The enumeration of* MSO *queries over the class of structures of bounded treewidth is in* CONSTANT-DELAY$_{lin}$.

Theorem 6.1.3 *The solution testing for* MSO *queries over the class of structures of bounded treewidth is in* CONSTANT-TIME$_{lin}$.

Theorem 6.1.4 ([5]) *The counting problem for* MSO *queries over the class of structures of bounded treewidth is in* LINEAR-TIME.

Theorem 6.1.5 ([8]) *The j-th solution problem for* MSO *queries over the class of structures of bounded treewidth is in* LOGARITHMIC-TIME$_{lin}$.

For Theorem 6.1.2, we follow the lines of the proof of [48]. Another proof of Bagan [8] computes an intricate index structure based on the well known fact that one can compute a finite tree automaton that is equivalent to a given MSO sentence (which accepts the input tree iff this tree is a model for the given MSO formula). Our proof builds on the deterministic factorization forest decomposition theorem of Colcombet [20]: we use it in Section 6.3 to show that it is enough to consider first-order queries with one alternation of quantifiers, which makes the further reasoning a lot simpler.

We then show how the approach of [48] can further be exploited to also obtain new proofs of Theorems 6.1.3, 6.1.4 and 6.1.5.

As for the Theorem 6.1.3, it is in a sense a "side effect" of any of the algorithms for the enumeration problem. The counting solution of [5], just as the algorithm given in [8], also uses the tree automaton describing the MSO formula.

It should be noted that there is also a third solution to the enumeration problem, as described in [23]. The main difference is that the index structure build there has size $O(\|\mathbf{D}\| \log \|\mathbf{D}\|)$ rather than $O(\|\mathbf{D}\|)$. The techniques used in [23], just as it is the case for both [5] and [8], also rely on the connections of MSO with tree automatons.

6.2 Preliminaries

We say that a relational structures has d-bounded treewidth if the underlying Gaifman graph has d-bounded treewidth. Note that with respect to the discussions from Section 3.3 it is equivalent to defining this notion with respect to the underlying adjacency graph (since the query is assumed to be fixed).

Similarly, we say that a class of databases has d-bounded treewidth if the underlying class of Gaifman graphs has d-bounded treewidth.

6.2.1 Trees

We work with finite trees whose nodes are labeled using a finite alphabet. More formally, let $\mathbb{A}$ be a finite alphabet. Trees over $\mathbb{A}$ are generated by the following rules: for all $a \in \mathbb{A}$, a is a tree, furthermore if $a \in \mathbb{A}$ and, for some $k \geq 1$, $t_1, \cdots, t_k$ are trees, then $a(t_1 + \cdots + t_k)$ is a tree. We use standard terminology for trees defining nodes, root, leaves, ancestors and descendants. A *binary tree* is a tree whose every node has either no child (is a leaf) or exactly two children, the *left* child and the *right* child. Given two nodes u, v of a tree $\mathbf{T}$ we write $u < v$ to denote the fact that u is a strict ancestor of v. Over binary trees, we also use $u <_l v$ (resp. $u <_r v$) to express the fact that v is a descendant of the left (resp. right) child of u. Given a tree $\mathbf{T}$, we denote by $|\mathbf{T}|$ the number of nodes of $\mathbf{T}$. Whenever we define a function $f : \mathbf{T} \to \mathbb{N}$ we implicitly assume that the domain of f is the set of nodes of $\mathbf{T}$.

As usual, we view each tree as a relational structure whose domain is its set of nodes. The signature contains a binary symbol E denoting the child relation and a unary symbol P_a per $a \in \mathbb{A}$ denoting the label of each node. For binary trees, E is replaced with two binary symbols E_1 and E_2 denoting respectively the left child and the right child relation. We consider monadic second-order logic (MSO) over these signatures allowing quantification over nodes or sets of nodes of the tree. We will also often use the binary predicates $<$, $<_l$ and $<_r$ denoting the ancestor relationships. Recall that $<$, $<_l$ and $<_r$ are definable in MSO from E or E_1 and E_2.

Recall from Chapter 2 that by *query* we mean an MSO formula whose free variables are all first order variables: we disallow free set variables.

6.2.2 Useful results

The following result has been proved by Flum, Frick and Grohe. It is a generalization of the well known fact that any MSO sentence can be translated into a tree automaton and can therefore be evaluated in time linear in the size of the input tree. Note that we only mention in this chapter *data complexities*, i.e. all hidden constants may depend on the size of the formula, sometime in a dramatic way, like in the result below where the constant factor is non-elementary in the size of the formula.

Theorem 6.2.1 ([32]) *Let $\phi(\bar{x})$ be an* MSO *query. Given a tree $\boldsymbol{T}$, $\phi(\boldsymbol{T})$ can be computed in time $O(|\boldsymbol{T}| + |\phi(\boldsymbol{T})|)$.*

We will use this result only in the simple case when $|\phi(\mathbf{T})| = O(|\mathbf{T}|)$. When this happens, $\phi(\mathbf{T})$ can be computed in time $O(|\mathbf{T}|)$. We fall in this case when ϕ is unary or ϕ is of the form $\phi(x, y)$, but y happens to be a function of x. Notice that the fact that a binary query is the graph of a function is not in general apparent when looking at the syntax of the formula, but will be in the specific cases we will consider.

Corollary 6.2.1 *Let ϕ be an* MSO *query either of the form $\phi(x)$ or of the form $\phi(x, y)$ with y a function of x. Given a tree $\boldsymbol{T}$, $\phi(\boldsymbol{T})$ can be computed in time $O(|\boldsymbol{T}|)$.*

6.3 Simplifying the problem

All of the Theorems 6.1.2–6.1.5 talk about structures of bounded treewidth. In this section we show that it suffices to prove them for simple first-order queries using only one quantifier alternation. This reduction is obtained in several steps. All our reductions are effective, fairly simple and standard except for one, making use of a deep result on MSO queries over trees. Each step of the reduction consists in a transformation of the input structure that can be performed in linear time, together with an effective transformation of the input formula whose complexity may be non elementary. As we focus only on the complexity in the size of the input structure, the effectiveness of the transformation of the formula is only implicit in all the following statements. Moreover, the linear time complexity guaranties that all the reduction steps can be achieved during the preprocessing phases of the dynamic algorithms or as initial steps of the linear time counting algorithm.

6.3.1 It is enough to consider trees

We start with two results that allow us to restrict all the reasoning to trees. They build on a seminal result of Courcelle [22] that the tree decomposition of a graph can be encoded in MSO.

Theorem 6.3.1 ([22]) *Fix $k \in \mathbb{N}$. There exists an algorithm which given an* MSO *formula ϕ and a structure $\boldsymbol{D}$ with treewidth bounded by k, computes in time $O(\|\boldsymbol{D}\|)$ an* MSO *formula ϕ' and a tree $\boldsymbol{T}$ such that $\phi(\boldsymbol{D}) = \phi'(\boldsymbol{T})$.*

The above result heavily relies on the following theorem:

Theorem 6.3.2 ([16]) *Fix $k \in \mathbb{N}$. There exists an algorithm which given a structure $\boldsymbol{D}$, computes in time $O(\|\boldsymbol{D}\|)$ the tree-decomposition of $\boldsymbol{D}$ of width k if such a decomposition exists.*

It should also be said here that the original statement of Theorem 6.3.1 from [22] was for MSO sentences only rather than for MSO queries. This extension being rather straightforward, we attribute Theorem 6.3.1 to [22]. For details on this extension, an interested reader is referred to [5].

Combining the above theorems with Fact 3.1.1 justifies that we can restrict input structures to trees.

6.3.2 It is enough to consider binary trees

We make use of the classical first child – next sibling encoding of unranked trees. It is folklore that this transformation can be done in time linear in the input tree and that it induces a bijection f between the nodes of the tree $\mathbf{T}$ and the nodes of its first child – next sibling encoding $f(\mathbf{T})$. Moreover, this transformation preserves MSO definability: for any MSO query $\phi(\bar{x})$ over unranked trees there is an MSO query $\phi'(\bar{x})$ over binary trees such that for all trees $\mathbf{T}$ we have $f(\phi(\mathbf{T})) = \phi'(f(\mathbf{T}))$. Hence, Fact 3.1.1 again justifies that we can restrict input structures to binary trees.

6.3.3 It is enough to consider queries that also output all least common ancestors

Given a tree T and two nodes u, u' of $\mathbf{T}$, the least common ancestor of u and u', denoted $\text{lca}(u, u')$, is the node v such that v is an ancestor of both u and u' and no strict descendant of v has this property. Note that there is an MSO formula defining the property $z = \text{lca}(x, y)$. An MSO query $\phi(\bar{x})$ is *lca-complete* if for all variables $x_i, x_j \in \bar{x}$ there is a variable $x_k \in \bar{x}$ such that for all $\mathbf{T}$ and all $\bar{u} \in \phi(\mathbf{T})$, $u_k = \text{lca}(u_i, u_j)$. Given an MSO query $\phi(\bar{x})$ we can compute an lca-complete MSO query $\psi(\bar{x}\bar{y})$ such that for all trees $\mathbf{T}$ and tuples $\bar{a} \in \phi(\mathbf{T})$ there is a unique $\bar{b}$, containing all the desired least common ancestors, such that $\bar{a}\bar{b} \in \psi(\mathbf{T})$. Typically ψ is constructed from ϕ by introducing a new free variable per pair of initial free variables, while adding the formula axiomatizing the fact that the new variable is the least common ancestor of the other two variables. For simplicity of the exposition we also assume that there is a free variable in ψ always denoting the root of the tree. Note that for all trees $\mathbf{T}$ we have $|\psi(\mathbf{T})| = |\phi(\mathbf{T})|$ and that all solutions in $\phi(\mathbf{T})$ can be reconstructed from a solution in $\psi(\mathbf{T})$ by projecting out the newly added component. Hence, Fact 3.1.2 justifies that we can restrict to queries that also output all least common ancestors.

6.3.4 It is enough to consider queries with ancestor-typed outputs

We now make sure that two elements of a solution are always related in a same way in all solutions. An *ancestor-type* $o(\bar{x})$ of a set of variables $\bar{x}$ is a maximal consistent conjunction of formulas of the form $x_i = x_j$, $x_i <_l x_j$, $x_i <_r x_j$ or $\neg(x_i < x_j)$. For a fixed $\bar{x}$ there are only finitely many ancestor-types that we denote by $O(\bar{x})$. For an MSO query $\phi(\bar{x})$ it is easy to see that $\phi(\bar{x})$ is equivalent to $\bigvee_{o \in O(\bar{x})} o(\bar{x}) \wedge \phi(\bar{x})$ and that for two different $o_1, o_2 \in O(\bar{x})$ and any tree $\mathbf{T}$ we have $(o_1 \wedge \phi)(\mathbf{T}) \cap (o_2 \wedge \phi)(\mathbf{T}) = \emptyset$. Hence, Fact 3.1.3 justifies that it is enough to show all the results for $o(\bar{x}) \wedge \phi(\bar{x})$ for each o separately rather than for $\phi(\bar{x})$ (formally, for Theorem 6.1.5 we need Theorem 6.1.4 first to be able to use Fact 3.1.3). In the sequel we can assume that there is a fixed ancestor-type $o(\bar{x})$ on the free variables of our MSO queries. Moreover, if the ancestor-type o requires variables x_i and x_j to be equal, we can make use of Fact 3.1.2 and replace all occurrences of x_j in the formula with x_i. Thus we may assume that the ancestor-type o ensures that all variables are distinct. Once $o(\bar{x})$ is fixed we say that two variables $x_i, x_j \in \bar{x}$ are *o-consecutive* if $o(\bar{x})$ implies that $x_i < x_j$ and no variable z of $\bar{x}$ is such that $x_i < z < x_j$.

6.3.5 It is enough to consider o-compatible queries

Given an MSO query $\phi(\bar{x})$, an ancestor-type $o(\bar{x})$, and two o-consecutive variables $x_i < x_j$ of $\bar{x}$, we say that a formula $\alpha(x_i, x_j)$ *has its quantifications relativized to* $[x_i, x_j]$ if it is defined from arbitrary unary MSO queries and the descendant relation $<$, using first-order quantifications of the form $\exists y\; x_i \leq y \leq x_j \wedge \alpha$ or $\forall y\; x_i \leq y \leq x_j \rightarrow \alpha$ and, similarly, monadic second-order quantifications restricted to sets of elements within $[x_i, x_j]$. Note that such a formula may use *arbitrary unary* MSO subqueries. In particular, they can test whether a node on the path from x_i to x_j is a left or right child. Moreover, they can test for an arbitrary MSO property of the subtree rooted at a sibling of any node on the path from x_i to x_j.

A formula $\phi(\bar{x})$ is said to be *o-compatible* if it is a conjunction of formulas $\alpha(x_i, x_j)$, where x_i, x_j are o-consecutive variables and α has its quantifications relativized to $[x_i, x_j]$. Recall that two queries $\phi_1(\bar{x})$ and $\phi_2(\bar{x})$ are mutually exclusive if for any binary tree $\mathbf{T}$ we have $\phi_1(\mathbf{T}) \cap \phi_2(\mathbf{T}) = \emptyset$.

The following result is a classical consequence of the Compositional Method developed by Shelah [63]. It essentially says that the MSO type of the tree can be derived from the MSO types of the elements covering the tree. It can be proved using a simple Ehrenfeucht-Fraïssé game argument, see also Lemma 1 and Lemma 2 in [20].

Theorem 6.3.3 *Over binary trees, for every ancestor-type o and every* MSO *query $\phi(\bar{x}) \wedge o(\bar{x})$, there is an equivalent query which is a union of mutually exclusive o-compatible queries.*

PROOF Let k be the quantifier rank of ϕ. We start with some useful notation. Given a tree $\mathbf{T}$, its MSO-k-type is the set of MSO sentences of quantifier rank k that hold in $\mathbf{T}$. It is well known that there are only finitely many MSO-k-types and that each of them can be described by a sentence of MSO. Given a tree $\mathbf{T}$ and two nodes x, y of $\mathbf{T}$, such that $x < y$, we denote by $\mathbf{T}_x$ the subtree of $\mathbf{T}$ rooted at x and by $\mathbf{T}[x, y]$ the nodes of $\mathbf{T}$ that are descendants of x but not descendants of y. Given a tuple $\bar{a}$ of nodes of $\mathbf{T}$ we denote by $\mathbf{T}(\bar{a})$ a recoloring of $\mathbf{T}$ such that the nodes of $\bar{a}$ have been distinguished.

A tuple such that $o(\bar{x})$ holds, induces a covering of the tree whose components are: $\mathbf{T}_{x_i}$ — if x_i has no o-consecutive variable, and $\mathbf{T}[x_i, x_j]$ — if x_i and x_j are o-consecutive.

The Composition Method tells us that the MSO-k-types of each component taken separately induces the MSO-k-type of the whole tree. In other words, if we know the MSO-k-types of each of the $\mathbf{T}_{x_i}$ and $\mathbf{T}[x_i, x_j]$, then we know the MSO-k-type of $\mathbf{T}(\bar{x})$. This property is folklore and can be proved by a simple Ehrenfeucht-Fraïssé game argument. From the initial remarks, the MSO-k-type of $\mathbf{T}_{x_i}$ can be described by a unary MSO formula $\xi(x_i)$ and the MSO-k-type of $\mathbf{T}[x_i, x_j]$ by an MSO formula $\xi(x_i, x_j)$. Hence the corresponding MSO-k-type of $\mathbf{T}(\bar{x})$ is a conjunction of MSO formulas that are either unary or are binary and involve two o-consecutive variables.

Because the MSO-k-type of $\mathbf{T}(\bar{x})$ implies whether $\phi(\bar{x})$ holds or not, $\phi(\bar{x})$ is a union of conjunctions as above. The union is finite because there are only finitely many MSO-k-types. The union is mutually exclusive because each conjunct involves two different MSO-k-types for at least one component.

It remains to show that each formula $\xi(x_i, x_j)$ can be chosen with its quantifications relativized to $[x_i, x_j]$. This is exactly the Composition Theorem of Shelah [63]. The path from x_i to x_j is a linear order and we replace each subtree hanging off this path by its MSO-k-type. By this we mean that we recolor each node on the path from x_i to x_j by a color describing the MSO-k-type of the subtree hanging off that node. The result of Shelah says that $\xi(x_i, x_j)$ is equivalent to an MSO formula describing the resulting path, which has its quantifications relativized to $[x_i, x_j]$ by construction. The desired formula is then obtained by replacing the colors with the unary MSO formulas describing the MSO-k-types of each removed subtree.

■

As the o-compatible queries mentioned in Theorem 6.3.3 are mutually exclusive, Fact 3.1.3 justifies that we can restrict ourselves to a single o-compatible MSO query (note that for the j-th solution problem mentioned in Theorem 6.1.5 the use of Fact 3.1.3 has impact on the order as described after the proof of that fact).

6.3.6 It is enough to consider o-compatible queries definable in $\Sigma_2(<)$

This is the nontrivial step of our reductions. It exploits the following result of Colcombet based on a deterministic factorization forest theorem which can be seen here as an index structure for trees.

Theorem 6.3.4 (Implicit in [20]) *Over binary trees, for every ancestor-type o and every o-compatible* MSO *query $\phi(\bar{x})$ there is an equivalent query which is still o-compatible but each of its conjuncts is of the form $\exists\bar{y}\forall\bar{z}\ \theta$, where θ is a disjunction of conjunctions of atomic predicates or* MSO *queries with one free variable or atoms using $<$.*

The result of Colcombet essentially shows that each of the conjuncts in o-compatible queries obtained from the decomposition lemma could be assumed to be definable with one alternation of first-order quantifiers, but using the ancestor relationship and unary MSO queries.

We denote by $\Sigma_2(<)$ the formulas of the form $\exists\bar{y}\forall\bar{z}\varphi$, where φ is a quantifier free formula using only atoms based on the relational predicates $<$ and P_a for all a ranging over a finite alphabet.

From Corollary 6.2.1 we know that each MSO query with one free variable can be evaluated in time linear in the size of the input tree. Therefore the unary MSO queries mentioned in Theorem 6.3.4 can be computed during the preprocessing phase of the dynamic algorithms or as a part of the counting algorithm, obtaining a tree over a new alphabet, where the new labels denote both the old ones and whether each unary query holds or not at that node. We can therefore assume that our query is o-compatible and such that all its conjuncts are in $\Sigma_2(<)$. We denote by *o-simple $\Sigma_2(<)$ queries* these queries.

Note that the queries in $\Sigma_2(<)$ only have access to the labels and $<$. In particular, they do not have access to the successor relation and don't distinguish between left and right children. This information is now part of the relabeling of the tree.

Altogether we have shown that it is enough to prove Theorems 6.1.2–6.1.5 for o-simple $\Sigma_2(<)$ queries over binary trees.

6.4 The index structures

The goal of this section is to design index structures capable of dealing with problems mentioned in Theorems 6.1.2–6.1.5. From Section 6.3 we know that to solve these theorems it is enough to provide solutions for o-simple $\Sigma_2(<)$ queries. Recall that $\psi(\bar{x})$ is a o-simple $\Sigma_2(<)$ query if it is a conjunction of queries $\phi_{i,j}(x_i, x_j)$, where $x_i < x_j$ are o-consecutive and $\phi_{i,j}$ has its quantifications relativized to $[x_i, x_j]$.

In Section 6.4.1 we introduce some notation concerning o-simple $\Sigma_2(<)$ queries.

In Section 6.4.2 we show some useful normalization of the formulas $\phi_{i,j}$ mentioned above.

In Section 6.4.3 we define an order with respect to which the solutions to the j-th solution problem are being output.

In Sections 6.4.4 through 6.4.6 we define the index structures mentioned above.

6.4.1 The o-skeleton decomposition

Let $\psi(\bar{x})$ be a o-simple $\Sigma_2(<)$ query. In particular, this query is ancestor-typed and as such is a conjunction of queries $\phi_{i,j}(x_i, x_j)$, where $x_i < x_j$ are o-consecutive variables. Let x_i be a variable from $\bar{x}$. We now introduce a bit of notation:

• With $\bar{x}_{o\text{-succ}(x_i)}$ we denote the subset of $\bar{x}$ that contains exactly the o-successors of x_i.

• With $o_{o\text{-succ}(x_i)}(x_i, \bar{x}_{o\text{-succ}(x_i)})$ we denote the restriction of o to its sub-part containing only variables from $\{x_i, \bar{x}_{o\text{-succ}(x_i)}\}$.

• With $\psi_{o\text{-succ}(x_i)}(x_i, \bar{x}_{o\text{-succ}(x_i)})$ we denote the restriction of ψ to its sub-part containing only variables from $\{x_i, \bar{x}_{o\text{-succ}(x_i)}\}$.

Recall that ψ outputs all least common ancestors. Together with the fact that it is evaluated over binary trees, we get the following:

Observation 6.4.1 *Let $\psi(\bar{x})$ be a o-simple $\Sigma_2(<)$ query. Then each variable can have exactly 0, 1 or 2 o-consecutive variables.*

Extending the above observation we also get:

Observation 6.4.2 *Let $\psi(\bar{x})$ be a o-simple $\Sigma_2(<)$ query such that x_1 has no o-predecessors. Then there are only three cases possible:*

Case 1: $\psi(x_1)$ is a unary query.

Case 2: x_1 has exactly one o-consecutive variable.

Wlog assume that this o-consecutive variable is x_2. Then ψ is of the form $\psi(\bar{x}) = \psi_{o\text{-succ}(x_2)}(x_2, \ldots, x_{n+1}) \wedge \phi_{1,2}(x_1, x_2)$, where $\phi_{1,2}$ is the $\Sigma_2(<)$ conjunct of ψ that describes the path from x_1 to x_2 and $\psi_{o\text{-succ}(x_2)}$ is $o_{o\text{-succ}(x_2)}$-simple $\Sigma_2(<)$ query such that x_2 has no $o_{o\text{-succ}(x_2)}$-predecessors.

Case 3: x_1 has exactly two o-consecutive variables x_l and x_r.

Then $\psi(\bar{x}) = \psi_{o\text{-succ}(x_l)}(x_l, \bar{x}_{o\text{-succ}(x_l)}) \wedge \psi'_l(x_1, x_l) \wedge \psi_{o\text{-succ}(x_r)}(x_r, \bar{x}_{o\text{-succ}(x_r)}) \wedge \psi'_r(x_1, x_r)$, where ψ'_l and ψ'_r are the $\Sigma_2(<)$ conjuncts of ψ that describe the path from x_1 to x_l and from x_1 to x_r respectively and that $\psi_{o\text{-succ}(x_l)}$ is $o_{o\text{-succ}(x_l)}$-simple $\Sigma_2(<)$ query such that x_l has no $o_{o\text{-succ}(x_l)}$-predecessors and similarly $\psi_{o\text{-succ}(x_r)}$ is $o_{o\text{-succ}(x_r)}$-simple $\Sigma_2(<)$ query such that x_r has no $o_{o\text{-succ}(x_r)}$-predecessors.

Recall that for o-consecutive variables $x_i < x_j$, ancestor type o actually says a bit more than the fact that x_j must be in the subtree of x_i: it also restricts the search for x_j to the left or the right subtree of x_i (and to exactly one of them). In the sequel we always assume that x_l is the variable that has to be in the left subtree of x_1 and x_r is the right subtree of x_1.

The recursive decomposition of ψ, as described in Observation 6.4.2, is called *the o-skeleton decomposition of ψ*. As o is always going to be clear from the context, we just write *the skeleton decomposition of ψ*.

We now recursively define the *components* of the skeleton decomposition of ψ:

- $\psi(\bar{x})$ with type o is a component of the skeleton decomposition of ψ,
- components of the skeleton decomposition of $\psi_{o\text{-succ}(x_2)}$ from Case 2 are components of the skeleton decomposition of ψ.
- components of the skeleton decompositions of both $\psi_{o\text{-succ}(x_l)} \wedge \psi'_l$ and of $\psi_{o\text{-succ}(x_r)} \wedge \psi'_r$ from Case 3 are components of the skeleton decomposition of ψ.

6.4.2 From $\Sigma_2(<)$ to polynomials

Let **T** be a binary tree and $\phi(x, y)$ be a $\Sigma_2(<)$ formula that logically implies $x < y$ and has its quantifications relativized to $[x, y]$.

Over words, a *monomial* is a language of the form $A_0^* a_1 A_1^* \ldots a_m A_m^*$, where each a_i is a fixed letter and A_i is a (possibly empty) set of letters. A *polynomial* is a finite union of monomials. In [70] a characterization of $\Sigma_2(<)$ over words was given that was later shown to be effective [4]:

Theorem 6.4.1 ([70, 4]) *Over words, a language is definable in $\Sigma_2(<)$ if and only if it is a polynomial.*

As we are dealing with formulas with free variables, we slightly extend the above definitions. Over words, a formula $\psi(x, y)$ is called a *monomial formula* if it holds for exactly those pairs (x, y), such that the subword between positions x and y (including both ends) matches a regular expression $a_0 A_0^* a_1 A_1^* \ldots a_m A_m^* a_{m+1}$, where each a_i is a fixed letter and A_i is a (possibly empty) set of letters. A *polynomial formula* is a finite disjunction of monomial formulas.

From Theorem 6.4.1 we immediately get

Corollary 6.4.1 *Over words, any $\Sigma_2(<)$ formula $\psi(x, y)$ that logically implies $x < y$ and has its quantifications relativized to $[x, y]$ is equivalent to a polynomial formula.*

In our case $\phi(x, y)$ is indeed a $\Sigma_2(<)$ formula, but it talks about trees. Fortunately, it also implies that $x < y$ and has its quantifications relativized to $[x, y]$, so for any two nodes $u < v$ of the input tree, $\phi(u, v)$ holds if and only if a word formed by the labels of the nodes on a path from u to v satisfies ϕ. Hence Corollary 6.4.1 implies:

Corollary 6.4.2 *Let $\phi(x, y)$ be a $\Sigma_2(<)$ formula that logically implies $x < y$ and has its quantifications relativized to $[x, y]$. Then there is a polynomial formula $\hat{\phi}$ such that for all trees $\boldsymbol{T}$, $\phi(\boldsymbol{T}) = \hat{\phi}(\boldsymbol{T})$.*

Let $\hat{\phi} = \bigvee_{1 \leq j \leq k} \psi_j$ be the polynomial formula corresponding to ϕ as described by Corollary 6.4.2, where each ψ_j is a monomial formula. It is tempting at this point to handle separately each $\psi_j(\mathbf{T})$. Unfortunately, this will for example not fulfill the "no duplicate" constraint of the enumeration process as a tuple may be present in $\psi_j(\mathbf{T})$ for several values of j. We will cope with this problem by considering all the monomial formulas at the same time using a product construction.

In order to jump quickly from one solution to another it will be enough to remember all the subparts of each of the monomial formulas that are currently satisfied at the current node. This is done as follows.

Fix for the moment an arbitrary $j \leq k$. As ψ_j is a monomial formula, it describes paths of the form $a_0 A_0^* a_1 A_1^* \dots a_m A_m^* a_{m+1}$. Hence if $(u, v) \in \psi_j(\mathbf{T})$, the sequence of labels of the nodes on the path from u to v (denoted $[u, v]$) matches the regular expression $a_0 A_0^* a_1 A_1^* \dots a_m A_m^* a_{m+1}$. We define the following suffixes of this regular expression: $e_0 = a_0 A_0^* a_1 A_1^* \dots a_m A_m^* a_{m+1}$ and for each $1 \leq i \leq m + 1$ let $e_i = A_{i-1}^* a_i A_i^* \dots a_m A_m^* a_{m+1}$. Let also $\mathcal{S} = \{e_0, e_1, \dots, e_{m+1}\}$. For two nodes $u < v$ of $\mathbf{T}$, the j-type of (u, v) is exactly the set of regular expressions of $\mathcal{S}$ matched by $[u, v]$. We say that a j-type is *good* if it contains e_0. Clearly, $(u, v) \in \psi_j(\mathbf{T})$ if and only if the j-type of (u, v) is good.

We do this for all $j \leq k$ and define for two nodes $u < v$ of T the *type* of (u, v) as the tuple formed from all its j-types. Hence $(u, v) \in \phi(T)$ iff for some $1 \leq j \leq k$ its j-type is good. Notice that we have finitely many types and we call *good* those for which at least one component is good.

The following simple claim illustrates the key motivation for using types.

Claim 6.4.1 *Let $\boldsymbol{T}$ be a binary tree and $u < v < w, w'$ be nodes in $\boldsymbol{T}$ such that (u, w) has type τ_1 and both (v, w) and (v, w') have type τ_2. Then (u, w') has type τ_1.*

PROOF As a type is a tuple containing all j-types, it is enough to show the claim for a fixed j-type. By symmetry it is enough to show that the j-type of (u, w) is included in the j-type of (u, w').

Assume that the j-type of (u, w) contains some suffix s. Hence there is a matching of s in the path from u to w, i.e. witnesses for the existentially quantified nodes. Fix such a matching α. α induces a matching of some suffix s' of s in the path from v to w. Since the j-types of (v, w) and (v, w') are the same, s' also matches the path from v to w'. Combining this later matching with α on the path from u to v (excluding v) provides a proper matching of s on the path from u to w'.

■

In particular, in the scenario described in Claim 6.4.1, if $(u, w) \in \phi(\mathbf{T})$ (that is τ_1 is good), then $(u, w') \in \phi(\mathbf{T})$.

We say that a node u of a tree $\mathbf{T}$ is *valid for* x if there exists a node v such that $(u, v) \in \phi(\mathbf{T})$. Similarly a node v is *valid for* y if there exists a node u such that $(u, v) \in \phi(\mathbf{T})$. For a type τ and a node u we say that the pair (u, τ) is *interesting* if there exists a node v such that the type of (u, v) is τ and τ is good (in particular, u is valid for x). Note that all these properties are definable in MSO via unary queries, hence computable in time $O(|\mathbf{T}|)$.

6.4.3 The τ-order

Before defining index structures used by the algorithms from the proofs of Theorems 6.1.2–6.1.5, we first define the order in which the solutions to j-th solution problem from Theorem 6.1.5 are going to be output.

Fix o-simple $\Sigma_2(<)$ query $\psi(\bar{x})$. For every conjunct $\phi_{i,j}(x_i, x_j)$ from ψ that describes the path between o-consecutive variables $x_i < x_j$ let $\hat{\phi}_{i,j}$ be the polynomial formula from the application of

Corollary 6.4.2 to $\phi_{i,j}$. For each such i, j fix an order on the types $\tau_1^{i,j} <_{i,j} \tau_2^{i,j} <_{i,j} \ldots$ induced by polynomial $\hat{\phi}_{i,j}$.

Let $\{\psi_c\}_{c \in C}$ be the set of all components of the skeleton decomposition of ψ. We now inductively define a family of orders $\{\prec_c\}_{c \in C}$ over solutions to ψ_c:

Definition 6.4.1 *Let $\psi_c(\bar{x})$ (with induced ancestor type o_c) be a component of the skeleton decomposition of ψ, and let $\bar{v}$, $\bar{u}$ be two tuples of nodes of **T** such that **T** $\models \psi_c(\bar{v}) \wedge \psi_c(\bar{u})$. Wlog assume that x_1 has no o_c-predecessors.*

Following Observation 6.4.2 there are three cases possible:

Case 1: ψ_c is unary.

*We then set $v \prec_c u$ iff v preceds u in the depth-first traversal of **T**.*

Case 2: x_1 has exactly one o_c-consecutive variable.

Denote with $\psi_{c'}$ the subcomponent of ψ_c after removing x_1 and let $\phi_{i,j}$ be the conjunct of ψ describing the path from x_1 to x_2. Moreover, denote with τ_1 the type of $[v_1, v_2]$ and with τ_2 the type of $[u_1, u_2]$ as they are induced by $\hat{\phi}_{i,j}$.

We then set $\bar{v} \prec_c \bar{u}$ iff:

- *v_1 preceds u_1 in the depth-first traversal of **T**,*
- *or $v_1 = u_1$ and $\tau_1 <_{i,j} \tau_2$,*
- *or $v_1 = u_1$ and $\tau_1 = \tau_2$ and $\bar{v}' \prec_{c'} \bar{u}'$, where $\bar{v}'$ is $\bar{v}$ with v_1 removed and $\bar{u}'$ is $\bar{u}$ with u_1 removed.*

Case 3: x_1 has exactly two o_c-consecutive variables x_l and x_r.

Denote with ψ_{c_l} the subcomponent of ψ_c after removing x_r and all its o_c-successors and with ψ_{c_r} the subcomponent of ψ_c after removing x_l and all its o_c-successors. Moreover, let $\bar{v}_l$ be $\bar{v}$ with v_r and all its successors in $\bar{v}$ removed, let $\bar{v}_r$ be $\bar{v}$ with v_l and all its successors in $\bar{v}$ removed and similarly let $\bar{u}_l$ be $\bar{u}$ with u_r and all its successors in $\bar{u}$ removed and let $\bar{u}_r$ be $\bar{u}$ with u_l and all its successors in $\bar{u}$ removed.

We then set $\bar{v} \prec_c \bar{u}$ iff:

- *v_1 preceds u_1 in the depth-first traversal of **T**,*
- *or $v_1 = u_1$ and $\bar{v}_l \prec_{c_l} \bar{u}_l$,*
- *or $v_1 = u_1$ and $\bar{v}_l = \bar{u}_l$ and $\bar{v}_r \prec_{c_r} \bar{u}_r$.*

Definition 6.4.2 *We use all the notation as introduced in this section.*

We call family $\{\prec_c\}_{c \in C}$ a τ-order *for ψ.*

Given τ-order $\{\prec_c\}_{c \in C}$ and two solutions $\bar{v}$ and $\bar{u}$ to a component ψ_c of the skeleton decomposition of ψ it is always clear which (namely $\prec_c$) order should be used to compare $\bar{v}$ and $\bar{u}$. That is why in the sequel we always skip the subscript c and simply use $\prec$ as an order on the solutions to any single one of the components. The reader should though keep in mind that the τ-order $\prec$ is in fact a family of orders $\{\prec_c\}_{c \in C}$. We overload the notion of τ-order $\prec$ a bit more. Let τ_1 and τ_2 be two types induced by a polynomial $\hat{\phi}_{i,j}$ from the application of Corollary 6.4.2 to $\phi_{i,j}$, where $\phi_{i,j}$ is a conjunct of ψ describing the path between o-consecutive variables $x_i < x_j$. We write $\tau_1 \prec \tau_2$ iff $\tau_1 <_{i,j} \tau_2$.

We now describe the index structures required by Section 6.5.

6.4.4 The basic index structure

Let **T** be a binary tree and $\phi(x, y)$ be a $\Sigma_2(<)$ formula that logically implies $x < y$ and has its quantifications relativized to $[x, y]$. We apply to ϕ the normalization procedure described in Section 6.4.2 and use the terminology introduced in that section.

We are ready to define:

Definition 6.4.3 *The* basic index structure *of* ***T*** *for $\phi(x, y)$ is the following directed graph:*

Its vertices are the nodes of ***T*** *valid for y and the interesting pairs of* ***T****.*

Its edges are defined as follows:

• *We have an edge between v and (u, τ) when the type of (u, v) is τ, and u is the "bottom-most" node of* ***T*** *with $(u, v) \in \phi(\boldsymbol{T})$ (i.e, $\forall w, u < w < v$ implies $(w, v) \notin \phi(\boldsymbol{T})$).*

• *We have an edge between (u, τ) and (u', τ') when $u' < u$ and there is a node $v > u$ with the type of (u, v) being τ and the type of (u', v) being τ' and u' is the "bottom-most" node with this property: for all nodes w with $u' < w < u$ there is no $v > u$ with the type of (u, v) being τ and $(w, v) \in \phi(\boldsymbol{T})$.*

The key properties of this structure are summarized in the following lemma:

Lemma 6.4.1 *The basic index structure of* ***T*** *for $\phi(x, y)$ has the following properties:*

1. *It is computable in time $O(|\boldsymbol{T}|)$.*

2. *It is a forest with leaves being nodes valid for y and internal nodes being interesting pairs.*

3. *For any node u valid for x and two different interesting pairs (u, τ) and (u, τ'), they occur in different trees inside the basic index structure.*

4. *(*Completeness*) For all nodes u, v of the tree* ***T****, if $(u, v) \in \phi(\boldsymbol{T})$ and τ is the type of (u, v), then v is a leaf in the subtree of (u, τ) inside the basic index structure.*

5. *(*Soundness*) For all nodes u, v of the tree* ***T****, if τ is the type of (u, v) and v is a leaf in the subtree of (u, τ) inside the basic index structure, then $(u, v) \in \phi(\boldsymbol{T})$.*

PROOF We start with Property (1).

From the the discussions from the end of Section 6.4.2, the nodes of the basic index structure are definable in MSO and therefore they can be computed in time linear in $|\mathbf{T}|$ using Corollary 6.2.1. For the first kind of edges notice that each v uniquely determines a u and that the relation between u and v can also be described in MSO. Similarly, for the second kind of edges, each interesting pair (u, τ) uniquely determines a u' and their relationship can be described in MSO. Hence Corollary 6.2.1 can be invoked again to compute the edges in time $O(|\mathbf{T}|)$.

We continue with Property (2).

Clearly the structure is acyclic as a node can only point to an ancestor in $\mathbf{T}$. Moreover, each node v valid for y has a u such that $(u, v) \in \phi(\mathbf{T})$ and hence nodes of this kind are leaves and have unique parents, corresponding to the bottom-most u such that $(u, v) \in \phi(\mathbf{T})$.

Similarly, an interesting pair (u, τ) either has no parent or a parent of the form (u', τ'). Recall from the previous point that u' is uniquely determined by (u, τ). It remains to show that τ' is also uniquely determined by (u, τ). Assume this is not the case and that (u, τ) is associated to both (u', τ'_1) and (u', τ'_2). Then, by construction, there exist v_1, v_2 such that τ is the type of both (u, v_1) and (u, v_2), τ'_1 is the type of (u', v_1) and τ'_2 is the type of (u', v_2). Claim 6.4.1 implies that $\tau'_1 = \tau'_2$ and therefore our basic index structure is a forest.

Property (3) is immediate as we already know that the basic index structure is in fact a forest and that a necessary condition for (u', τ') to be a parent of (u, τ) is that $u' < u$.

We now move to Property (4). Assume $(u, v) \in \phi(\mathbf{T})$, τ is the type of (u, v) and that $u = u_0 < u_1 < \ldots < u_t < v$ are all the nodes of $\mathbf{T}$ on the path from u to v such that for each $0 \leq i \leq t$ the pair $(u_i, v) \in \phi(\mathbf{T})$. Let τ_i denote the type of (u_i, v). From the construction of the basic index structure we know that v is a child of (u_t, τ_t).

We show that for all $0 < i \leq t$, (u_i, τ_i) is the child of (u_{i-1}, τ_{i-1}) in the basic index structure. If this was not the case, then there exist nodes w, v', with $u_{i-1} < w < u_i < v'$, the type of (u_i, v') being

τ_i and $(w, v') \in \phi(\mathbf{T})$. As τ_i is also the type of (u_i, v), Claim 6.4.1 implies that the type of (w, v) is the same as the type of (w, v') and therefore $(w, v) \in \phi(\mathbf{T})$, a contradiction.

It remains to show Property (5). We show that if v is in the subtree of (u, τ) inside the basic index structure, then the type of (u, v) is τ. As (u, τ) is interesting pair, τ is (by definition) good and thus $(u, v) \in \phi(\mathbf{T})$. The proof is a simple induction on the distance between v and (u, τ) inside the basic index structure. If (u, τ) is a parent of v, then the type of (u, v) is τ by the definition of the parent relation of leaves. The inductive step is again a direct consequence of Claim 6.4.1.

■

6.4.5 The full index structure

The basic index structure as defined in Definition 6.4.3 is the core of index structures used in the proofs of Theorems 6.1.2–6.1.5. It turns out that to be fully efficient, the basic index structure still lacks some navigational and counting powers. We now show how to handle the navigational part.

In the sequel $\mathbf{T}$ and ϕ are just as defined in Section 6.4.4.

Definition 6.4.4 *The* full index structure *of $\boldsymbol{T}$ for $\phi(x, y)$ is the basic index structure of $\boldsymbol{T}$ for $\phi(x, y)$ with the following enhancements:*

• *Recall that by Property (2) of Lemma 6.4.1 the basic index structure is a forest. We start with computing for each node of $\boldsymbol{T}$ a pointer (called* root-pointer*) to the root of the tree of the basic index structure that it belongs to.*

• *It will be also important that we have access to the descendant relation inside this forest in constant time. To do this we perform a depth-first traversal (where we always visit the left subtree of a node before its right subtree) of the underlying forest and compute a* dfs number *to each node (corresponding to the last time we have visited it).*

• *We enrich furthermore the basic index structure by associating to each leaf the next leaf in the dfs traversal (we call these pointers* next-leaf *pointers). This will allow us to jump from one leaf to the next one in constant time.*

• *Moreover, to each internal node (u, τ) we add an additional pointers to the first (called the* first-leaf *pointer) and to the last (called the* last-leaf *pointer) leafs of its subtree.*

• *Finally, for each node u of the original input tree that is valid for x, we keep a pointer to each of the interesting pairs (u, τ) inside the full index structure.*

The key properties of this structure are summarized in the following lemma:

Lemma 6.4.2 *The full index structure of $\boldsymbol{T}$ for $\phi(x, y)$ has the following properties:*

1. *It is computable in time $O(|\boldsymbol{T}|)$.*

2. *It has all the properties (2)–(5) of the basic index structure from Lemma 6.4.1.*

3. *There exists an algorithm that, given two nodes u, v of $\boldsymbol{T}$ and type τ tests in constant time whether v is a leaf in the subtree of (u, τ) inside the full index structure.*

4. *There exists an algorithm that, given two nodes u, v of $\boldsymbol{T}$, tests in constant time whether $\boldsymbol{T} \models \phi(u, v)$.*

5. *There exists an algorithm that, given a node u of $\boldsymbol{T}$, enumerates with constant delay (without any additional preprocessing) all the solutions to $\phi(\boldsymbol{T})$ whose first component is u.*

PROOF

We start with Property (1).

Recall from Lemma 6.4.1, Property (1), that the basic index structure can be computed in time $O(|\mathbf{T}|)$ and has therefore a size linear in $|\mathbf{T}|$. We now need to show that all the additional pointers and values are computable in time $O(|\mathbf{T}|)$. But this is trivial: depth-first traversal is clearly a linear time procedure, so the dfs numbers are computed in linear time. During this traversal the next-leaf pointers can easily be computed. The root-pointers are trivially computable in linear time and for for first-leaf and last-leaf pointers we can use Corollary 6.2.1. Computing for each node u of the original input tree that is valid for x the pointer to each of the interesting pairs (u, τ) inside the full index structure is clearly linear as every such node u requires a number of pointers bounded by the number of different types, which does not depend on T, and we can compute them with a single traversal of the basic index structure.

Altogether the computation time of the full index structure is $O(|\mathbf{T}|)$, just as required.

We continue with Property (2).

This is immediate, as the full index structure only ads additional specific pointers and some integer values to the nodes of the basic index structure and does not modify the underlying forest at all.

We now describe the testing algorithm mentioned in Property (3).

Using root-pointers we check whether v is a leaf in the same tree of the full index structure as the internal node (u, τ). If this is the case, v is in the subtree of (u, τ) iff its dfs number is between the dfs numbers of first and last leafs of the subtree of (u, τ) (because the depth-first traversal algorithm always visits the left subtree of a node before its right subtree). The latter leafs being accessible in constant time using first-leaf and last-leaf pointers of (u, τ), this concludes the proof.

We now describe the testing algorithm for Property (4).

Lemma 6.4.1, namely Properties (5) and (4), justifies that $\mathbf{T} \models \phi(u, v)$ iff there exists a type τ such that v is a leaf in the subtree of (u, τ) of the full index structure. As there are constantly many choices for τ, it remains to show how to perform a single test in constant time. Such tests are exactly handled by Property (3).

We finally move to Property (5).

Let u be a node in $\mathbf{T}$ which is valid for x (otherwise we are done as there are no solutions to $\phi(\mathbf{T})$ having u as the first component). We wish to enumerate all v such that $(u, v) \in \phi(\mathbf{T})$. To do this we consider in turn all the interesting pairs (u, τ) using the appropriate precomputed pointers of u. For each such τ we jump, in constant time using the precomputed first-leaf pointer, to the first leaf v in the subtree of (u, τ) in the full index structure and output (u, v). Now, as long as we remain in the subtree of (u, τ) (this can easily be checked in constant time by Property (3)), we successively go through all the leaves of our index structure in dfs order (again in constant time, using the precomputed next-leaf pointers) outputting the corresponding pair (u, v).

Clearly, between outputting consecutive solutions there is only a constant delay and each pair we output is unique by Property (3) of Lemma 6.4.1. From Property (5) of Lemma 6.4.1 we know that each pair that we output is in $\phi(\mathbf{T})$. Property (4) of Lemma 6.4.1 ensures that we do not skip any solutions.

This completes the proof of Lemma 6.4.2.

■

6.4.6 The full index structure with counting

To deal with counting and j-th solution problems we need to further extend the full index structure, giving it some counting power. This extension is defined below.

In the sequel $\mathbf{T}$ and ϕ are just as defined in Section 6.4.4.

Definition 6.4.5 *Let $\# : \boldsymbol{T} \to \mathbb{N}$ be a computable function.*

The full index structure with counting *of $\boldsymbol{T}$ for $\phi(x, y)$ and $\#$ is the full index structure of $\boldsymbol{T}$ for $\phi(x, y)$ with the following enhancements:*

• *For every tree t of the full index structure, compute the array (called the* leaf-array of t*) containing in consecutive cells pointers to the consecutive leafs of tree t (a cell pointing to leaf v is called the* array copy *of v) in order they appeared in the depth-first traversal of t. For the root of t, store a pointer (called the* leaf-array *pointer) to the leaf array of t. For every leaf v of t store a pointer (called* array-copy *pointer) to its array copy (this way there is a two-way access between a leaf and its array copy).*

• *Let v be a leaf on leaf-array of t for some tree t of the full index structure and we write $u <_t v$ if u is also a leaf on the leaf-array of t and it appears on that array before v. We define* partial_sum(v) $= \sum_{\{u:u<_t v\}} \#(u)$*. Now for each such leaf-array t, for each leaf v appearing in it, the full index structure with counting stores both the value of $\#(v)$ and partial_sum(v).*

The key properties of this structure are summarized in the following lemma:

Lemma 6.4.3 *Let $\# : \boldsymbol{T} \to \mathbb{N}$ be a computable function. The* full index structure with counting *of $\boldsymbol{T}$ for $\phi(x, y)$ and $\#$ has the following properties:*

1. *It is computable in time $O(|\boldsymbol{T}|)$.*
2. *It has all the properties (2)–(5) of the full index structure from Lemma 6.4.2.*
3. *There exists an algorithm that, given a node u of $\boldsymbol{T}$, computes in constant time number $\sum_{\{v:(u,v)\in\phi(u,v)\}} \#(v)$.*
4. *There exists an algorithm that, in time $O(|\boldsymbol{T}|)$, computes a function $\#' : \boldsymbol{T} \to \mathbb{N}$ such that for a given node u of $\boldsymbol{T}$ we have: $\#'(u) = \sum_{\{v:(u,v)\in\phi(u,v)\}} \#(v)$.*
5. *There exists an algorithm that, given a node u of $\boldsymbol{T}$, $j \in \mathbb{N}$ and a τ-order $\prec$, computes in logarithmic time the node v such that $\sum_{\{v':(u,v')\in\phi(u,v'),v'\prec v\}} \#(v') < j \le \sum_{\{v':(u,v')\in\phi(u,v'),v'\preceq v\}} \#(v')$ or responds (also in logarithmic time) that such a node v does not exist.*

PROOF

We start with Property (1).

Recall from Lemma 6.4.2, Property (1), that the full index structure can be computed in time $O(|\mathbf{T}|)$ and has therefore a size linear in $|\mathbf{T}|$. A simple traversal of each tree of the full index structure allows us to create their leaf-arrays together with all the array-copy pointers in total time $O(|\mathbf{T}|)$. Fix leaf-array of t and let $u <_t v$ be two consecutive nodes on it. As partial_sum$(v) =$ partial_sum$(u) + \#(u)$, we can compute all partial sums via a single left-to-right pass of each leaf-array.

Altogether the computation time of the full index structure with counting is $O(|\mathbf{T}|)$, just as required.

Property (2) is immediate, as the full index structure with counting only enriches the underlying full index structure, adding to it some new specific pointers.

We now move to Property (3).

Fix node u of $\mathbf{T}$. Lemma 6.4.1, namely Properties (5) and (4), justifies that for any node v of $\mathbf{T}$, we have $\mathbf{T} \models \phi(u, v)$ iff there exists a type τ such that v is a leaf in the subtree of (u, τ) of the full index structure with counting. As there are constantly many choices for τ and from Property (3) of Lemma 6.4.1 we know that for each v such that $\mathbf{T} \models \phi(u, v)$, v is a leaf in exactly one tree of the full index structure with counting, it remains to show how to compute $\sum_{\{v:v \text{ is a leaf in the subtree of } (u,\tau)\}} \#(v)$ in constant time.

But this is now immediate: we can access the first (v_l) and the last (v_r) leafs of the subtree of (u, τ) using first-leaf and last-leaf pointers. The desired value is then $\#(v_r)+$partial_sum$(v_r)-$partial_sum(v_l) and all these values are stored by the full index structure with counting.

Property (4) is a simple corollary of Property (3): it is enough to apply the algorithm from Property (3) to each node u of tree $\mathbf{T}$ computing this way the desired function $\#'$.

Finally we move to Property (5).

Fix u, τ-order $\prec$ and $j \in \mathbb{N}$.

The test whether such v exists can be done via Property (3) (and so it even requires only constant rather than logarithmic time) as such v does not exist iff $\sum_{\{v:(u,v)\in\phi(u,v)\}} \#(v) < j$.

Lemma 6.4.1, namely Properties (5) and (4), justifies that for any node v of $\mathbf{T}$, we have $\mathbf{T} \models \phi(u,v)$ iff there exists a type τ such that v is a leaf in the subtree of (u,τ) of the full index structure with counting. As there are constantly many choices for τ and from Property (3) of Lemma 6.4.1 we know that for each v such that $\mathbf{T} \models \phi(u,v)$, v is a leaf in exactly one tree of the full index structure with counting, we can handle each τ one by one in $\prec$-increasing order.

Set $S = 0$. Using Property (3) compute $S_\tau = \sum_{\{v:v \text{ is a leaf in the subtree of } (u,\tau)\}} \#(v)$. As long as $S + S_\tau < j$, add S_τ to S and move to the $\prec$-next type τ. This computations each being doable in constant time (as described in Property (3)), in constant time we find the $\prec$-smallest type τ for which $j \leq S + S_\tau$. The desired node v is now such that:

$\sum_{\{v':v' \text{ is a leaf in the subtree of } (u,\tau), v'\prec v\}} \#(v') < j - S \leq \sum_{\{v':v' \text{ is a leaf in the subtree of } (u,\tau), v'\preceq v\}} \#(v')$.

This node v can now easily be found in logarithmic time using a binary search approach on the subarray of the leaf-array of root-pointer$((u,\tau))$ with initial left and right bounds for the binary search given by first-leaf and last-leaf pointers of (u,τ).

This completes the proof of Lemma 6.4.3.

■

6.5 Solving the problems

From the previous section it follows that Theorems 6.1.2–6.1.5 are (respectively) consequences of the following propositions:

Proposition 6.5.1 *The enumeration of o-simple $\Sigma_2(<)$ queries over binary trees is in* CONSTANT-DELAY$_{lin}$.

Proposition 6.5.2 *The solution testing for o-simple $\Sigma_2(<)$ queries over binary trees is in* CONSTANT-TIME$_{lin}$.

Proposition 6.5.3 *The counting problem for o-simple $\Sigma_2(<)$ queries over binary trees is in* LINEAR-TIME.

Proposition 6.5.4 *The j-th solution problem for o-simple $\Sigma_2(<)$ queries over binary trees is in* LOGARITHMIC-TIME$_{lin}$.

In the sequel we one by one show proofs of the above propositions. They are all inductive, but the inductive approaches differ. Propositions 6.5.1 and 6.5.2 use a similar top-down approach, while Propositions 6.5.3 and 6.5.4 are proved in a bottom-up skeleton-based manner.

6.5.1 Enumerating simple $\Sigma_2(<)$ queries

PROOF [of Proposition 6.5.1]

Let $\psi(\bar{x})$ be the fixed o-simple $\Sigma_2(<)$ query.

The proof follows by an induction on the number of free variables in ψ. The base case (unary query) is done via our assumption that this variable denotes the root of the tree. So let us assume that we have the result for n-ary queries and we want to extend it to ones with $n+1$ variables.

The key ingredient for the inductive step is the Property 5 of Lemma 6.4.2.

Let $\psi(x_1, \ldots, x_{n+1})$ be an o-simple $\Sigma_2(<)$ query. Without loss of generality assume that x_n and x_{n+1} are o-consecutive variables with $x_n < x_{n+1}$ and, for all $i < n$, we do not have $x_{n+1} < x_i$ in o. By our choice of x_{n+1} and o-simplicity, $\psi(x_1, \ldots, x_{n+1})$ must be of the form $\psi'(x_1, \ldots, x_n) \wedge \psi''(x_n, x_{n+1})$ where ψ'' is the $\Sigma_2(<)$ conjunct of ψ that describes the path from x_n to x_{n+1}. Let $\phi(x_1, \ldots, x_n) = \psi'(x_1, \ldots, x_n) \wedge \exists z \psi''(x_n, z)$. Note that $\exists z \psi''(x_n, z)$ is a unary query (it describes a property of x_n) and as such can be handled with Corollary 6.2.1. Formula ϕ is o'-simple where o' is the restriction of o to the first n variables. Therefore by induction the enumeration of ϕ is in CONSTANT-DELAY$_{lin}$. Let $\mathbf{T}$ be a binary tree. A CONSTANT-DELAY$_{lin}$ enumeration procedure for ψ is:

The precomputation phase performs both the precomputation phase of enumeration for ϕ as given by the induction hypothesis, and the construction of the full index structure of $\mathbf{T}$ for ϕ. This is done in $O(|\mathbf{T}|)$ as required (by inductive hypothesis and by Property 1 of Lemma 6.4.2).

The enumeration phase nests the enumeration procedure for ψ'' given by Property 5 of Lemma 6.4.2 inside the enumeration procedure for $\phi(\mathbf{T})$ given by the induction hypothesis. Given a solution $(a_1, \ldots, a_n) \in \phi(\mathbf{T})$ we apply Property 5 of Lemma 6.4.2 to node a_n and find in constant delay all nodes a_{n+1} such that $(a_n, a_{n+1}) \in \psi''(\mathbf{T})$. Notice that, by definition of ϕ, we always have at least one such node a_{n+1} and that all such nodes are exactly those where the tuple $(a_1, \ldots, a_n, a_{n+1})$ is in $\psi(\mathbf{T})$. Therefore, for each such node a_{n+1} we output the tuple $(a_1, \ldots, a_n, a_{n+1})$. Once all the nodes a_{n+1} have been found, we continue the simulation of the enumeration procedure for $\phi(\mathbf{T})$ and obtain in constant delay the next tuple $(b_1, \ldots, b_n) \in \phi(\mathbf{T})$ and eventually compute all of $\psi(\mathbf{T})$.

■

6.5.2 Testing simple $\Sigma_2(<)$ queries

Using exactly the same approach as for the Proposition 6.5.1, we now prove Proposition 6.5.2:

PROOF [of Proposition 6.5.2]

Let $\psi(\bar{x})$ be the fixed o-simple $\Sigma_2(<)$ query.

The proof follows by an induction on the number of free variables in ψ. The base case (unary query) is done via our assumption that this variable denotes the root of the tree. So let us assume that we have the result for n-ary queries and we want to extend it to one with $n+1$ variables.

The key ingredient for the inductive step is the Property 4 of Lemma 6.4.2.

Let $\psi(x_1, \ldots, x_{n+1})$ be an o-simple $\Sigma_2(<)$ query. Without loss of generality assume that x_n and x_{n+1} are o-consecutive variables with $x_n < x_{n+1}$ and, for all $i < n$, we do not have $x_{n+1} < x_i$ in o. By our choice of x_{n+1} and o-simplicity, $\psi(x_1, \ldots, x_{n+1})$ must be of the form $\psi'(x_1, \ldots, x_n) \wedge \psi''(x_n, x_{n+1})$ where ψ'' is the $\Sigma_2(<)$ conjunct of ψ that describes the path from x_n to x_{n+1}. A CONSTANT-TIME$_{lin}$ testing procedure for ψ is:

The precomputation phase performs both the precomputation phase of testing for ψ' as given by the induction hypothesis, and the construction of the full index structure of $\mathbf{T}$ for ψ''. This is done in $O(|\mathbf{T}|)$ as required (by inductive hypothesis and by Property 1 of Lemma 6.4.2).

We now turn to the testing phase.

Given tuple $(v_1, \ldots, v_n, v_{n+1})$ we test whether $\mathbf{T} \models \psi'(v_1, \ldots, v_n)$ in constant time using the inductive hypothesis. If it is the case, it remains to test whether $\mathbf{T} \models \psi''(v_n, v_{n+1})$, and this is directly handled by Property 4 of Lemma 6.4.2.

■

6.5.3 Counting problem for simple $\Sigma_2(<)$ queries

Using Property (4) of Lemma 6.4.3 we prove the following lemma, that will find its use both in the proofs of Proposition 6.5.3 and Proposition 6.5.4

Lemma 6.5.1 *Given a binary tree $\boldsymbol{T}$ and a o-simple $\Sigma_2(<)$ query $\psi(\bar{x})$ such that there are no o-predecessors of x_1 we can, in time $O(|\boldsymbol{T}|)$, compute a function $\# : \boldsymbol{T} \to \mathbb{N}$ such that for a given node u of $\boldsymbol{T}$ we have: $\#(u) = |\{\bar{v} : \boldsymbol{T} \models \psi(u, \bar{v})\}|$.*

PROOF Note that in order to perform an inductive reasoning, throughout this proof we do NOT assume that one of the free variables necessarily denotes root. We now show how to compute in time $O(|\mathbf{T}|)$ the desired function $\#$.

The proof follows by an induction on the number of free variables in ψ. The base case (unary query) is done via a simple iteration through nodes of $\mathbf{T}$ and setting $\#(u) = 1$ if $\mathbf{T} \models \psi(u)$ and $\#(u) = 0$ otherwise. So let us assume that we have the result for queries of arities not greater than n and we want to extend it to ones with $n + 1$ variables.

Wlog assume that ψ has variables $x_1, \ldots, x_{n+1}$ and recall that x_1 has no o-predecessors. Using the skeleton decomposition as defined in Observation 6.4.2, we have at this points two cases.

Case 1: x_1 has exactly 1 o-consecutive variable.

Without loss of generality assume this variable is x_2. This means that $\psi(\bar{x}) = \psi'(x_2, \ldots, x_{n+1}) \wedge \psi''(x_1, x_2)$, where ψ'' is the $\Sigma_2(<)$ conjunct of ψ that describes the path from x_1 to x_2.

Using the inductive hypothesis for ψ' we compute $\#' : \mathbf{T} \to \mathbb{N}$ such that $\#'(u) = |\{\bar{v} : \mathbf{T} \models \psi'(u, \bar{v})\}|$.

Note that $\#(u) = \sum_{\{v:(u,v)\in\psi''(u,v)\}} \#'(v)$ and that we may conclude this case by constructing the full index structure with counting of $\mathbf{T}$ for ψ'' and $\#'$ and the function $\#$ is then computed by the algorithm from Property (4) of Lemma 6.4.3. Properties (1) and (4) of Lemma 6.4.3 guarantee that this is altogether done in time $O(|\mathbf{T}|)$ as required.

Case 2: x_1 has exactly 2 different o-consecutive variables x_l and x_r.

Let $\bar{x}^l$ denote the subset of $\bar{x}$ that contains variables which are o-successors of x_l and $\bar{x}^r$ denote the subset of $\bar{x}$ that contains variables which are o-successors of x_r. As all the variables denote different nodes and ψ talks about all least common ancestors of the variables, we have $\bar{x} = \{x_1\} \cup \{x_l\} \cup \{x_r\} \cup \bar{x}^l \cup \bar{x}^r$ and all mentioned sets are disjoint. This means that $\psi(\bar{x}) = \psi'_l(x_l, \bar{x}^l) \wedge \psi''_l(x_1, x_l) \wedge \psi'_r(x_r, \bar{x}^r) \wedge \psi''_r(x_1, \bar{x}^r)$ where ψ''_l and ψ''_r are the $\Sigma_2(<)$ conjuncts of ψ that describe the path from x_1 to x_l and from x_1 to x_r respectively and that x_l has no o-predecessors in $\psi'_l(x_l, \bar{x}^l)$ and x_r has no o-predecessors in $\psi'_r(x_r, \bar{x}^r)$.

Using the inductive hypothesis for ψ'_l we compute function $\#'_l : \mathbf{T} \to \mathbb{N}$ such that $\#'_l(u) = |\{\bar{v} : \mathbf{T} \models \psi'_l(u, \bar{v})\}|$ and similarly for ψ'_r we compute function $\#'_r : \mathbf{T} \to \mathbb{N}$ such that $\#'_r(u) = |\{\bar{v} : \mathbf{T} \models \psi'_r(u, \bar{v})\}|$.

Note that $\#(u) = \left(\sum_{\{v:(u,v)\in\psi''_l(u,v)\}} \#'_l(v)\right) \cdot \left(\sum_{\{v:(u,v)\in\psi''_r(u,v)\}} \#'_r(v)\right)$. We may then conclude this case by constructing the full index structure with counting of $\mathbf{T}$ for ψ''_l and $\#'_l$ and constructing it also for ψ''_r and $\#'_r$ and then using the algorithm from Property (4) of Lemma 6.4.3 to obtain functions $\#_l(u) = \sum_{\{v:(u,v)\in\psi''_l(u,v)\}} \#'_l(v)$ and $\#_r(u) = \sum_{\{v:(u,v)\in\psi''_r(u,v)\}} \#'_r(v)$. As said earlier, the desired function is $\#(u) = \#_l(u) \cdot \#_r(u)$.

Properties (1) and (4) of Lemma 6.4.3 guarantee that this is again done in time $O(|\mathbf{T}|)$, just as required.

■

Having Lemma 6.5.1 we are ready to prove Proposition 6.5.3:

PROOF [of Proposition 6.5.3]

This proposition is an immediate consequence of Lemma 6.5.1.

Let $\psi(\bar{x})$ be the fixed o-simple $\Sigma_2(<)$ query and without loss of generality assume that x_1 denotes the root of $\mathbf{T}$ (in particular, x_1 has no o-predecessors). Apply Lemma 6.5.1 to $\mathbf{T}$ and ψ and in time $O(|\mathbf{T}|)$ compute a function $\# : \mathbf{T} \to \mathbb{N}$ such that for a given node u of $\mathbf{T}$ we have $\#(u) = |\{\bar{v} : \mathbf{T} \models \psi(u, \bar{v})\}|$.

The value storing the number of solutions to ψ over $\mathbf{T}$ is simply $\#(\text{root}(\mathbf{T}))$.

■

6.5.4 j-th solution problem for simple $\Sigma_2(<)$ queries

Finally we move to the proof of Proposition 6.5.4:
PROOF [of Proposition 6.5.4]

This proof heavily relies on Lemma 6.5.1 and Property (5) of Lemma 6.4.3 and is based on a similar induction to the one seen in the proof of Lemma 6.5.1.

For the inductive argument to work, we do NOT assume that one of the free variables denotes root of the input tree.

Let $\psi(\bar{x})$ be the fixed o-simple $\Sigma_2(<)$ query and without loss of generality assume that x_1 has no o-predecessors and let $\prec$ be a τ-order. We show how to solve the j-th solution problem with respect to the fixed τ-order.

The proof follows by an induction on the number of free variables in ψ. The base case (unary query) is done in the following way: in the preprocessing phase iterate through all nodes of tree $\mathbf{T}$ and put into an array the ones such that $\mathbf{T} \models \psi(u)$. The j-th solution phase is then trivial (and works even in constant rather than logarithmic time): given j output the j-th element of the above array or respond NO SOLUTION if the size of the array is smaller than j.

So let us assume that we have the result for queries of arities not greater than n and we want to extend it to the ones with $n+1$ variables.

Following the lines of the proof of Lemma 6.5.1 (that is using the skeleton decomposition as defined in Observation 6.4.2), we have at this points two cases.

Case 1: x_1 has exactly 1 o-consecutive variable.

Without loss of generality assume this variable is x_2.

This means that $\psi(\bar{x}) = \psi'(x_2, \ldots, x_{n+1}) \wedge \psi''(x_1, x_2)$ where ψ'' is the $\Sigma_2(<)$ conjunct of ψ that describes the path from x_1 to x_2. A LOGARITHMIC-TIME$_{lin}$ j-th solution procedure for ψ is:

The preprocessing phase consists in this case of the preprocessing phase as given by the inductive hypothesis for ψ', of computing function $\# : \mathbf{T} \to \mathbb{N}$ such that $\#(u) = |\{\bar{v} : \mathbf{T} \models \psi'(u, \bar{v})\}|$ which is doable in linear time via Lemma 6.5.1 and of constructing the full index structure with counting of $\mathbf{T}$ for ψ'' and $\#$. It also computes function $\#' : \mathbf{T} \to \mathbb{N}$ such that $\#'(u) = |\{\bar{v} : \mathbf{T} \models \psi(u, \bar{v})\}|$ which is again doable in linear time via Lemma 6.5.1. The algorithm then iterates through nodes of $\mathbf{T}$ and puts into an array A (in $\prec$-ascending order) those nodes u, for which $\#'(u) > 0$. Going through array A from left to right it also computes *partial_sum* for each node u that sums-up the values of $\#'$ for nodes $\prec$-smaller than u, that is $partial\_sum(u) = \sum_{v \prec u} \#'(u)$. In total the preprocessing phase takes time $O(|\mathbf{T}|)$ as required.

The j-th solution phase is now as follows. Fix $j \in \mathbb{N}$. In logarithmic time (using binary search on array A) find node u such that $\text{partial\_sum}(u) < j \leq \text{partial\_sum}(u) + \#'(u)$. If such node u does not exist, output NO SOLUTION. If it does, then clearly u has to play the role of x_1 in the j-th solution to ψ. Then set $j = j - \text{partial\_sum}(u)$. Using Property 5 of Lemma 6.4.3 find in logarithmic time node v such that $\psi''(u, v)$ and $S = \sum_{\{v' : (u,v') \in \psi''(u,v'), v' \prec v\}} \#(v') < j \leq S + \#(v)$. The desired j-th solution is then $(u, v, \bar{w})$ where $\bar{w}$ is the $(j-S)$-th solution to ψ' over the subtree of $\mathbf{T}$ rooted at v. By inductive hypothesis, $\bar{w}$ is computed in logarithmic time and we are done with Case 1.

Case 2: x_1 has exactly 2 different o-consecutive variables x_l and x_r.

Let $\bar{x}^l$ denote the subset of $\bar{x}$ that contains variables which are o-successors of x_l and $\bar{x}^r$ denote the subset of $\bar{x}$ that contains variables which are o-successors of x_r. As all the variables denote different nodes and ψ talks about all least common ancestors of the free variables, we have $\bar{x} = \{x_1\} \cup \{x_l\} \cup$

$\{x_r\} \cup \bar{x}^l \cup \bar{x}^r$ and all mentioned sets are disjoint. This means that $\psi(\bar{x}) = \psi'_l(x_l, \bar{x}^l) \wedge \psi''_l(x_1, x_l) \wedge \psi'_r(x_r, \bar{x}^r) \wedge \psi''_r(x_1, \bar{x}^r)$ where ψ''_l and ψ''_r are the $\Sigma_2(<)$ conjuncts of ψ that describe the path from x_1 to x_l and from x_1 to x_r respectively and that x_l has no o-predecessors in $\psi'_l(x_l, \bar{x}^l)$ and x_r has no o-predecessors in $\psi'_r(x_r, \bar{x}^r)$. A LOGARITHMIC-TIME$_{lin}$ j-th solution procedure for ψ is:

The preprocessing phase consists in this case of the preprocessing phases as given by the inductive hypothesis for $\psi'_l \wedge \psi''_l$ and for $\psi'_r \wedge \psi''_r$ (note that both these queries match Case 1, so for example the preprocessing phase for $\psi'_r \wedge \psi''_r$ computes the function $\#_r(u) = |\{\bar{v} : \mathbf{T} \models (\psi'_r \wedge \psi''_r)(u, \bar{v})\}|$). It also computes function $\# : \mathbf{T} \to \mathbb{N}$ such that $\#(u) = |\{\bar{v} : \mathbf{T} \models \psi(u, \bar{v})\}|$ which is again doable in linear time via Lemma 6.5.1. It iterates through nodes of $\mathbf{T}$ and puts into an array A (in $\prec$-ascending order) those nodes u, for which $\#(u) > 0$. Going through array A from left to right it computes *partial_sum* for each node u that sums the values of $\#$ for nodes $\prec$-smaller than u, that is $partial\_sum(u) = \sum_{v \prec u} \#(u)$. In total the preprocessing phase takes time $O(|\mathbf{T}|)$ as required.

The j-th solution phase is now as follows. Fix $j \in \mathbb{N}$. In logarithmic time (using binary search on array A) find node u such that $\text{partial\_sum}(u) < j \leq \text{partial\_sum}(u) + \#(u)$. If such node u does not exist, output NO SOLUTION. If it does, clearly u has to play the role of x_1 in the j-th solution. Then set $j = j - \text{partial\_sum}(u)$.

It remains to observe that if we have any solution $(u, v_l, \bar{v}^l)$ to $\psi'_l(v_l, \bar{v}^l) \wedge \psi''_l(u, v_l)$ and any solution $(u, v_r, \bar{v}^r)$ to $\psi'_r(v_r, \bar{v}^r) \wedge \psi''_r(u, v_r)$, then it is always the case that $\mathbf{T} \models \psi(u, v_l, \bar{v}^l, v_r, \bar{v}^r)$. This simple observation leads to a fact that j-th solution to ψ is in fact composition of (j div $\#_r(u)$)-th solution to $\psi'_l(v_l, \bar{v}^l) \wedge \psi''_l(u, v_l)$ and of (j mod $\#_r(u)$)-th solution to $\psi'_r(v_r, \bar{v}^r) \wedge \psi''_r(u, v_r)$. Both this cases match exactly the procedure described in the j-th solution phase of Case 1 and as the appropriate preprocessing phases had in fact been performed, the desired solutions to $\psi'_l \wedge \psi''_l$ and to $\psi'_r \wedge \psi''_r$ can be computed in logarithmic time.

Altogether we have shown a solution to j-th solution problem that requires preprocessing in time $O(|\mathbf{T}|)$ and j-th solution phase that works in time $O(\log(|\mathbf{T}|))$, just as required. As the inductive arguments match exactly the skeleton decomposition of ψ, presented solution to the j-th solution problem is in fact with respect to the fixed τ-order $\prec$.

This finishes the proof of Proposition 6.5.4.

■

6.6 Discussions

Throughout this section we focus only on the enumeration problem.

The reader might find it quite surprising that something as simple as polynomial formulas require a non trivial basic index structure to be handled effectively. When inspecting its core, the enumeration algorithm as presented in Section 6.5.1 follows a rather natural approach and it essentially boils down to the algorithm from Property 5 of Lemma 6.4.2. It seems almost obvious that having a node u, one should rather easily be able to enumerate all the pairs (u, v) such that $u < v$ and that the path between u and v matches a given polynomial expression.

Example 6.6.1 below somehow justifies that this is not as simple (and this example even mentions a single monomial rather than a polynomial). The "suffix" oriented basic index structure defined in this chapter is a construction able to handle Example 6.6.1 (and actually able to handle any MSO query on trees), but of course there still might be some room for improvements.

It is worth mentioning that the difficulties with simplifying the basic index structure are related to the possible strengthening of Claim 6.4.1. For example the following lemma is true over words (but not over trees, as explained in Example 6.6.1) and it greatly simplifies the combinatorics over words.

Lemma 6.6.1 *Let $\phi(x, y)$ be a monomial formula, **W** a word and $u < v_1 \leq v' \leq v_2$ positions in **W** such that $\mathbf{W} \models \phi(u, v_1) \wedge \phi(u, v_2) \wedge (\exists x \phi(x, v'))$. Then $\mathbf{W} \models \phi(u, v')$.*

Before moving to the proof, we show how the above lemma can be used to provide the power of Property 5 of Lemma 6.4.2 in an easy way. Fix a monomial formula $\phi(x, y)$. Using Corollary 6.2.1 we may find all nodes v *valid for* y (that is nodes such that $\mathbf{W} \models \exists x \phi(x, v)$) and put them on a list. We similarly extract nodes *valid for* x and for each such node u valid for x we compute v_1 and v_2 that are the first and the last node on the list of nodes valid for y such that $\mathbf{W} \models \phi(u, v_1) \wedge \phi(u, v_2)$ (again using Corollary 6.2.1). This altogether takes time linear in the size of **W** and then the algorithm from Property 5 of Lemma 6.4.2 can be replaced with a simple traversal of the list of nodes valid for y with the bounds given by v_1 and v_2. Lemma 6.6.1 justifies that all the enumerated pairs are in fact solutions and the definitions of v_1 and v_2 guarantee that we do not skip any solutions.

As it might not be completely obvious, we now move to the proof of Lemma 6.6.1.

PROOF [of Lemma 6.6.1]

Assume ϕ looks for a pattern of the form: $a_0 A_0^* a_1 A_1^* \ldots a_m A_m^* a_{m+1}$. Let W be the input word and as in the statement of this lemma, fix positions $u < v_1 \leq v' \leq v_2$ such that $W \models \phi(u, v_1) \wedge \phi(u, v_2) \wedge (\exists x \phi(x, v'))$.

We know that there exists a position u' such that $\mathbf{W} \models \phi(u', v')$. There are two cases now: $u' < u$ or $u < u'$ ($u = u'$ immediately yields the result). The reasoning in both cases is similar, so we present in details only one of them.

Case 1: $u' < u$

We show that if $\mathbf{W} \models \phi(u', v') \wedge \phi(u, v_1)$ for positions $u' < u < v_1 < v'$, then $\mathbf{W} \models \phi(u, v')$.

The proof goes by an induction on m.

The base case $m = 0$ is trivial: if all the letters at positions between u' and v' are in A_0, then clearly $\mathbf{W} \models \phi(u, v')$.

Assume now we have the result for all $i \leq m$ and we want to extend it to $m + 1$.

As $\mathbf{W} \models \phi(u', v') \wedge \phi(u, v_1)$, there exist two *witnessing sequences* of positions $u = u_0 \leq u_1 \leq \ldots \leq u_m \leq u_{m+1} = v_1$ and $u' = u'_0 \leq u'_1 \leq \ldots \leq u'_m \leq u'_{m+1} = v'$ such that for each i letters at position u_i and at position u'_i are a_i and the set of letters at positions between u_i and u_{i+1} is included in A_i and the same holds for the set of letters at positions between u'_i and u'_{i+1}. We now have two cases:

• there exists $1 \leq j \leq m$ such that $u_j < u'_j$. In that case we may apply the inductive hypothesis to the polynomial ϕ' that searches for the pattern $a_0 A_0^* a_1 A_1^* \ldots a_{j-1} A_{j-1}^* a_j$ which is a proper prefix of the pattern mentioned by ϕ (as we have $\mathbf{W} \models \phi'(u, u_j) \wedge \phi'(u', u'_j)$ for $u' < u < u_j < u'_j$). The inductive hypothesis proves that $\mathbf{W} \models \phi'(u, u'_j)$ and the fact that $\mathbf{W} \models \phi(u, v')$ immediately follows, as $[u'_j, v']$ matches pattern $a_j A_j^* \ldots a_m A_m^* a_{m+1}$.

• for each $1 \leq j \leq m$ we have that $u'_j < u_j$. In particular this is the case for $j = m$ and so the letters at positions between u'_m and v' are in A_m (as $\mathbf{W} \models \phi(u', v')$). Since $u'_m < u_m$, the above holds for the set of letters at positions between u_m and v' and so $\mathbf{W} \models \phi(u, v')$ also in this case.

Case 2: $u < u'$

It is enough to show that if $\mathbf{W} \models \phi(u', v') \wedge \phi(u, v_2)$ for positions $u < u' < v' < v_2$, then $\mathbf{W} \models \phi(u, v')$.

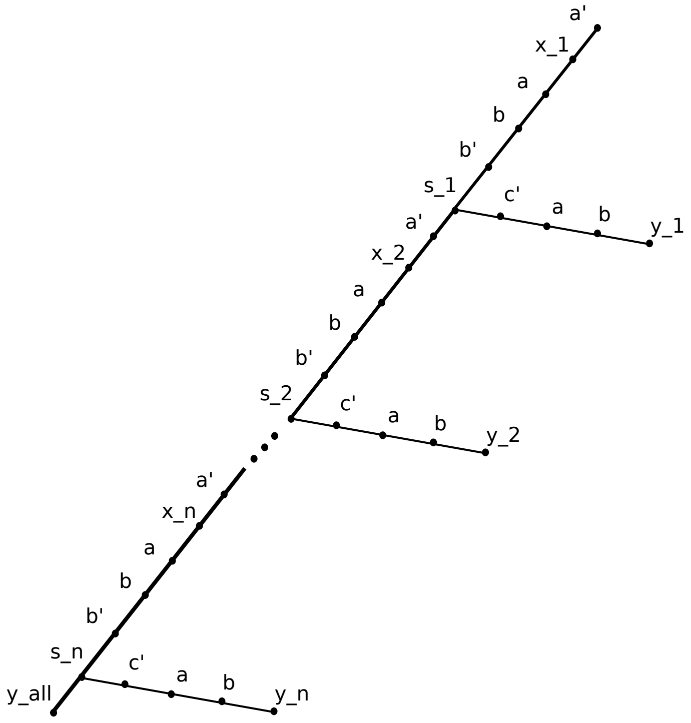
This case follows from an inductive argument that is symmetric to the one from Case 1.

As we said earlier, this finishes the proof of Lemma 6.6.1.

■

We now move to the promised "difficult" example.

Example 6.6.1 *Consider monomial $x(\mathbb{A} \setminus \{a'\})^* a(\mathbb{A} \setminus \{b'\})^* b(\mathbb{A} \setminus \{c'\})^* y$ and the input tree as represented by the figure below.*



The subscripts of some labels (like y_{all}, x_2, etc.) are added purely for the sake of readability: this way a single symbol contains the label (like y, x, etc.) and also distinguishes a particular node with that label.

A quick analysis allows us to extract the solutions to the input monomial over this tree. They are exactly: (x_i, y_{all}) and (x_i, y_i) (where i ranges from 1 to n). The reason for that is the following:

• the monomial seeks for a witnessing sequence with consecutive labels x, a, b, y but such that between x and a there is no a', between a and b there is no b' and between b and y there is no c'.

• consider now fixed x_i. Immediately below in the tree we see label a and we may include it in the witnessing sequence or not:

◦ if we do, we then have to include to the witnessing sequence the following b (as just after that comes b' which cannot separate a and b). Now any "deviation" to the right is no longer an option as each such brunch starts with a disallowed label c'. The only y that we can reach is then y_{all} and we have a solution (x_i, y_{all}).

∘ if we do not, then we cannot include anything all the way down to s_i (as there are no more a-s on that path). The left sub-tree of s_i starts with a', so we cannot go there and thus we are forced to move right and eventually reach y_i. This way we have a solution (x_i, y_i).

The power of Lemma 6.6.1 was that we were able to easily provide the algorithm as described in Property 5 of Lemma 6.4.2. Assume now that we would aim at such an algorithm that, given x_i, returns each matching y with respect to the DFS *order. The above example shows that this algorithm for each i would have to start from pair (x_i, y_{all}) and then find y_i in constant time. While in Lemma 6.6.1 we were always following a unique pointer to go to the next solution, this approach will not work here as we do not have a bound on the number of pointers that would have to originate from y_{all}. It is thus not clear how a structure of size $O(|\boldsymbol{T}|)$ could handle this kind of "pathological" example while directly mimicking the approach from Lemma 6.6.1. It is exactly the kind of issues for which we introduced the basic index structure which, using the suffix approach, handles the above example rather smoothly.*

6.7 Conclusions

It should be noted that our enumeration algorithm, as well as the one of Bagan [8], are non elementary in the size of the formula. Already for FO queries over unranked trees this non elementary blow-up cannot be avoided unless P=NP [36]. In our case the non elementary constants are hidden in Theorem 6.2.1, the Ehrenfeucht-Fraïssé argument of Theorem 6.3.3 and in the result of Colcombet, Theorem 6.3.4.

Our notion of linear time and constant delay requires furthermore that the total extra memory used during the enumeration phase is constant. It is not clear that the enumeration algorithm of Bagan has this extra property as it uses nested subprocess pushing (in constant time) pointers to the parent process on a stack. The nesting depth of this subprocess seems to be arbitrary.

However the result of Bagan also deals with MSO formulas containing free monadic second-order variables. In this case the delay is linear in the size of the solution being output. This cannot be avoided as the algorithm needs at least the time to output the solution. It is not clear how our technique can be lifted in order to take care of monadic second-order variables.

7

FO over classes of structures with bounded expansion

Contents

7.1 Introduction

In Chapter 5 we have shown that the query problems that are of the biggest interest from the point of view of this thesis (that is enumeration, testing, counting and j-th solution, see Section 2.5 for details) admit very good algorithmic properties when are considered with respect to the first-order logic over the classes of databases of bounded degree. In order to extend Theorems 5.1.1–5.1.4, there are two natural directions that we might follow:

- we can give more power to the logic,
- or we can extend the class of structures.

To see what happens when we extend the logical part, the reader is referred to Chapter 6. In this chapter we investigate the case when the considered logic is still FO, but the class of allowed structures is enlarged.

The considered class will be a class of structures with bounded expansion. The results we are going to prove in details are as follows:

Theorem 7.1.1 ([29, 42, 47]) *The model checking of* FO *sentences over the class of structures with bounded expansion is in* LINEAR-TIME.

Theorem 7.1.2 ([47]) *The enumeration of* FO *queries over the class of structures with bounded expansion is in* CONSTANT-DELAY$_{lin}$. *Moreover, the output is returned in a lexicographical order.*

Theorem 7.1.3 ([47]) *The solution testing for* FO *queries over the class of structures with bounded expansion is in* CONSTANT-TIME$_{lin}$.

Theorem 7.1.4 ([56, 47]) *The counting problem for* FO *queries over the class of structures with bounded expansion is in* LINEAR-TIME.

All the proofs that we are going to present here follow the lines of [47].

In the Discussions Section 7.7 we also prove the following theorem:

Theorem 7.1.5 *The j-th solution problem for* FO *queries over the class of structures with bounded expansion is in* LOGARITHMIC-TIME$_{lin}$.

The reason for putting this result a bit aside is that we do not handle this problem "directly". Using low treewidth colorings definition of bounded expansion, we reduce the j-th solution problem to Theorem 6.1.5. For more details, see Section 7.7.

Let us focus on the Theoreom 7.1.1 for a while. The key ingredient that is common for all the three proofs of this theorem is that they all perform some kind of quantifier elimination procedure.

In [29] and in [47] this is realized in the following way:

• First a procedure is shown which, given a FO query $q(\bar{x}) := \exists y \psi(\bar{x}, y)$ where ψ is quantifier free and given a structure $\mathbf{D}$, computes in time $O(\|\mathbf{D}\|)$ a structure $\mathbf{D}'$ and a quantifier free query $q'(\bar{x})$ such that $q(\mathbf{D}) = q'(\mathbf{D}')$.

• The model checking solution works then as follows: given a structure $\mathbf{D}$ and a sentence q the above procedure is applied recursively starting from the inner most quantification inside q and moving outside, which in the end produces a quantifier free sentence q' and a structure $\mathbf{D}'$ such that $\mathbf{D} \models q$ iff $\mathbf{D}' \models q'$. q' turns out to have a constant size and also $\|\mathbf{D}'\| = O(\|\mathbf{D}\|)$, which basically finishes the proof that the model checking is in fact in LINEAR-TIME.

In [42] the approach is slightly different:

• First a procedure is shown which, given a FO query $q(\bar{x}) := \forall \bar{y} \exists \bar{z} \psi(\bar{x}, \bar{y}, \bar{z})$ where ψ is quantifier free and given a structure $\mathbf{D}$, computes in time $O(\|\mathbf{D}\|)$ a structure $\mathbf{D}'$ and a query $q'(\bar{x}) := \exists \bar{y}' \psi'(\bar{x}, \bar{y}')$ such that ψ' is quantifier free and $q(\mathbf{D}) = q'(\mathbf{D}')$.

• The model checking solution works then as follows: given structure $\mathbf{D}$ and a sentence q the above procedure is applied recursively starting from the inner most quantification inside q and moving outside, which in the end produced sentence $q' := \exists \bar{x} \psi'(\bar{x})$ where ψ' is quantifier free and a structure $\mathbf{D}'$ such that $\mathbf{D} \models q$ iff $\mathbf{D}' \models q'$. It is then shown how existential sentences like q' can be verified in linear time over a class of graphs with bounded expansion. Also this time q' turns out to have a constant size and together with the fact that $\|\mathbf{D}'\| = O(\|\mathbf{D}\|)$, the proof that the model checking is in fact in LINEAR-TIME follows.

Despite these similarities, there is a major difference between the proofs of [29, 42] and the one of [47]. While the first ones are based on the low tree depth coloring characterization (which is yet another definition of bounded expansion, cf. [53]) the latter is based on transitive fraternal augmentations. We argue that the use of transitive fraternal augmentations gives a simpler proof. The reason is that it gives a useful normal form on quantifier-free formulas that will be the core of not only the quantifier elimination procedure algorithm, but also the algorithms for constant delay enumeration and for counting the number of solutions.

What should also be noted is that the proofs of Theorems 7.1.1– 7.1.4 that we present here work on the functional representation of the input structures rather than their purely relational equivalents. Without this functional representations we would not be able to eliminate first-order quantifiers. Indeed, with this functional representation we can speak of a node at distance 2 from x using the quantifier-free term $f(f(x))$, avoiding the existential quantification of the middle point. This idea was already taken in [26] for eliminating first-order quantifiers over structures of bounded degree. Our approach differs from theirs in the fact that in the bounded degree case the functions can be assumed to be permutations (in particular they are invertible) while this is no longer true in our setting, complicating significantly the combinatorics.

Before moving to the technical details of the proofs of Theorems 7.1.1– 7.1.4, we should point out that the structure of this chapter is slightly different than the very much alike structures of Chapters 5 and 6. In a very coarse description, the approach there was as follows:

• we always start with some necessary terminology,

• which is followed by descriptions of index structures that are the key tools for obtaining the main results,

• and then come the proofs which, while not that demanding themselves, heavily rely on the mentioned above index structures.

The main difference in this chapter is that the index structures no longer play the key role. From our perspective, the biggest impact should be annotated to the normal form for quantifier-free queries (see Proposition 7.2.1) and to the quantifier elimination procedure (see Proposition 7.3.1).

We are now ready to move to the more technical parts of this chapter.

7.2 Preliminaries

We first introduce our running examples. We focus there only on the model checking and the enumeration parts as they seem to be the most instructive in this case.

Example A-1 *The first query has arity 2 and returns pairs of nodes at distance at most two in a graph. We use the classical notion of distance that ignores the possible orientation of the edges. The query is of the form $\exists z E(x,z) \wedge E(z,y)$, where E is the symmetric closure of the input relation.*

Testing the existence of a solution to this query can be easily done in time linear in the size of the database. For instance one can go trough all nodes of the database and check whether the current node has degree at least two. The degrees of all nodes can be computed in linear time by going through all edges of the database and incrementing the degree counters associated with its endpoints.

Example B-1 *The second query has arity 3 and returns triples (x,y,z) such that y is connected to both x and z via an edge and x is connected to z (in other words, x, y and z form a triangle). The query is of the form $E(x,y) \wedge E(y,z) \wedge E(x,z)$, where E is the symmetric closure of the input relation.*

It is not clear at all how to test the existence of a solution to this query in time linear in the size of the database. The best known algorithm for finding a triangle in a graph has complexity even slightly worse than matrix multiplication [3]. If the degree of the input structure is bounded by a constant d, we can test the existence of a solution in linear time by the following algorithm. We first go through all edges (x,y) of the database and add y to a list associated with x and x to a list associated with y. It remains now to go through all nodes y of the database, consider all pairs (x,z) of nodes in the associated list (the number of such pairs is bounded by d^2) and then test whether there is an edge between x and z (by testing whether x is in the list associated with z).

We aim at generalizing this kind of reasoning to structures with bounded expansion.

Example A-2 *Over the class of all graphs, we cannot enumerate pairs of nodes at distance 2 with constant delay unless the Boolean Matrix Multiplication problem can be solved in quadratic time [11]. However, over the class of graphs of degree d, there is a simple constant delay enumeration algorithm. During the preprocessing phase, we associate with each node the list of all its neighbors at distance 2. This can be done in time linear in the database as in Example B-1. We then color in blue all nodes having a non empty list and make sure each blue node points to the next blue node (according to the linear order on the domain). This also can be done in time linear in the database and concludes the preprocessing phase. The enumeration phase now goes through all blue nodes x using the pointer structure and, for each of them, outputs all pairs (x, y) where y is in the list associated with x.*

Example B-2 *Over the class of all graphs, the query of this example most likely cannot be enumerated in constant delay because, as mentioned in Example B-1, testing whether there is one solution is already not known to be linear. Over the class of graphs of bounded degree, there is a simple constant delay enumeration algorithm, similar to the one from Example A-2.*

7.2.1 Graphs with bounded expansion and augmentation

Recall the definitions of a class of graphs with bounded expansion from Section 2.8.2. As already pointed out before, rather than going through the "initial" characterization of this class (cf. Definition 2.8.5), we will use the one exploiting the notion of augmentations (see Point 2 of Theorem 2.8.1).

Recall that the notion of 1-transitive fraternal augmentation is not a deterministic operation. Although transitivity induces precise edges, fraternity implies nondeterminism and thus there can possibly be many different 1-transitive fraternal augmentations.

Following [54] we fix a deterministic algorithm computing a "good" choice of orientations of the edges induced by the fraternity property. The precise definition of the algorithm is not important for us, it only matters here that it runs in time linear in the size of the input graph (see Lemma 7.2.1 below). With this algorithm fixed, we can now speak of **the** 1-transitive fraternal augmentation of $\mathbf{G}$.

Let $\mathbf{G}$ be a graph. The *transitive fraternal augmentation* of $\mathbf{G}$ is the sequence $\mathbf{G} = \mathbf{G}_0 \subseteq \mathbf{G}_1 \subseteq \mathbf{G}_2 \subseteq \dots$ such that for each $i \geq 1$ the graph $\mathbf{G}_{i+1}$ is the 1-transitive fraternal augmentation of $\mathbf{G}_i$. We will say that $\mathbf{G}_i$ is the i-th augmentation of $\mathbf{G}$.

The following lemma shows that within a class $\mathcal{C}$ of bounded expansion the i-th augmentation of $\mathbf{G} \in \mathcal{C}$ can be computed in linear time.

Lemma 7.2.1 *[54] Let $\mathcal{C}$ be a class of bounded expansion. For each $\mathbf{G} \in \mathcal{C}$ and each i, $\mathbf{G}_i$ is computable from $\mathbf{G}_{i-1}$ in time $O(\|\mathbf{G}_{i-1}\|)$.*

In particular Lemma 7.2.1 implies that for each i, given $\mathbf{G} \in \mathcal{C}$, $\mathbf{G}_i$ is computable from $\mathbf{G}$ in time $O(\|\mathbf{G}\|)$.

7.2.2 Graphs of bounded in-degree as functional structures

For the rest of this section we fix a class $\mathcal{C}$ of graphs with bounded expansion and let $\Gamma_{\mathcal{C}}$ be the function given by Point 2 of Theorem 2.8.1. For any graph $\mathbf{G} \in \mathcal{C}$ its transitive fraternal augmentation $\mathbf{G} = \mathbf{G}_0 \subseteq \mathbf{G}_1 \subseteq \mathbf{G}_2 \subseteq \dots$ is such that for all i, $\mathbf{G}_i$ has in-degree bounded by $\Gamma_{\mathcal{C}}(i)$. From the definition of bounded expansion it follows that the maximal in-degree of the graphs we will manipulate is always bounded by a number independent of the graph. We will use this property by constantly referring to the $1^{st}, 2^{nd} \dots$ predecessor of a node. It will therefore be convenient for us to represent the graphs $\mathbf{G}_i$ as functional structures where this predecessors are images of the current node via some suitable functions. This functional representation is also useful for eliminating some quantifiers.

A *functional signature* is a tuple $\sigma = (f_1, \dots, f_l, P_1, \dots, P_m)$, each f_i being a functional symbol of arity 1 and each P_i being an unary predicate. A *functional structure* over σ is then defined as for relational structures. FO is defined as usual over the functional signature. In particular, it can use atoms of the form $f(f(f(x)))$, which is crucial for the quantifier elimination step of Section 7.3 as the usual relational representation would require existential quantification for denoting the same element. A graph $\mathbf{G}$ of in-degree l and colored with m colors can be represented as a functional structure $\vec{\mathbf{G}}$, where the unary predicates encode the various colors and $v = f_i(u)$ if v is the i^{th} element (according to some arbitrary order that will not be relevant in the sequel) such that (v, u) is an edge of $\mathbf{G}$. We call such node v the i^{th} *predecessor* of u (where "i^{th} predecessor" should really be viewed as an abbreviation for "the node v such that $f_i(u) = v$" and not as a reference to the chosen order). If we do not care about i and we only want to say that v is the image of u under some function, we call it a *predecessor* of u. It is possible that some nodes may have less than l predecessors. To handle this case we allow f_i to be partial functions. As usual, if $f_i(u)$ is not defined for some node u, any atomic expression containing it is evaluated to false. Given $\mathbf{G} \in \mathcal{C}$ we define $\vec{\mathbf{G}}$ to be the functional representation of $\mathbf{G}$ as described above. Note that $\vec{\mathbf{G}}$ is computable in time linear in $\|\mathbf{G}\|$ and that for each first order query $\phi(\bar{x})$ one can easily compute a first order query $\psi(\bar{x})$ such that $\phi(\mathbf{G}) = \psi(\vec{\mathbf{G}})$.

Example A-3 *With the functional point of view, the query computing nodes at distance at most two is of the form:*

$$\bigvee_{f,g\in\sigma} \begin{array}{r} f(g(x)) = y \ \vee \ g(f(y)) = x \ \vee \ f(x) = g(y) \vee \\ \exists z \ \ f(z) = x \wedge g(z) = y \end{array}$$

where there is one disjunct per possible orientation of the edges on a path from x to y. We have removed the inner node z whenever this was possible.

Example B-3 *Similarly, the query of Example B-1 is equivalent to:*

$$\bigvee_{f,g,h\in\sigma} \begin{array}{l} (f(x) = y \wedge g(y) = z \wedge h(x) = z) \\ \vee(f(x) = y \wedge g(y) = z \wedge x = h(z)) \\ \vee(f(x) = y \wedge y = g(z) \wedge h(x) = z) \\ \vee(f(x) = y \wedge y = g(z) \wedge x = h(z)) \\ \vee(x = f(y) \wedge g(y) = z \wedge h(x) = z) \\ \vee(x = f(y) \wedge g(y) = z \wedge x = h(z)) \\ \vee(x = f(y) \wedge y = g(z) \wedge h(x) = z) \\ \vee(x = f(y) \wedge y = g(z) \wedge x = h(z)) \end{array}$$

Recall that the augmentation steps only introduce new edges and do not affect the vertex set. It will be convenient for us to be able to recover $\mathbf{G}_i$ from $\mathbf{G}_{i+1}$. For this we use extra function symbols denoting the edges resulting from an augmentation step. The definition of bounded expansion guarantees that the number of required new symbols is bounded by $\Gamma_\mathcal{C}(i+1)$ and does not depend on the graph.

From this it follows that we have functional signatures $\sigma_\mathcal{C}(0) \subseteq \sigma_\mathcal{C}(1) \subseteq \sigma_\mathcal{C}(2) \subseteq \ldots$, where $\sigma_\mathcal{C}(0)$ is the initial signature and $\sigma_\mathcal{C}(i+1)$ is $\sigma_\mathcal{C}(i)$ plus the $\Gamma_\mathcal{C}(i+1)$ extra symbols needed for the extra augmentation step, such that for any graph $\mathbf{G} \in \mathcal{C}$ and for all i:

1. $\vec{\mathbf{G}}_i$ is a functional structure over $\sigma_\mathcal{C}(i)$,
2. $\vec{\mathbf{G}}_i \subseteq \vec{\mathbf{G}}_{i+1}$ and $\vec{\mathbf{G}}_{i+1}$ is computable in linear time from $\vec{\mathbf{G}}_i$,
3. for every FO query $\phi(\bar{x})$ over $\sigma_\mathcal{C}(i)$ and every $j \geq i$ we have that $\phi(\vec{\mathbf{G}}_i) = \phi(\vec{\mathbf{G}}_j)$.

We denote by $\alpha_\mathcal{C}(i)$ the number of function symbols of $\sigma_\mathcal{C}(i)$. It follows from the discussion above that $\alpha_\mathcal{C}(i) = \Sigma_{j\leq i}\Gamma_\mathcal{C}(j)$. It would be tempting to reduce this number by reusing function symbols, but that would then be problematic to enforce property 3 (see Example 7.2.1 from the end of this section).

We say that a functional signature σ' is a *recoloring* of σ if it extends σ with some extra unary predicates (colors), while the functional part remains unchanged. Similarly, a functional structure $\vec{\mathbf{G}}'$ over σ' is a *recoloring* of $\vec{\mathbf{G}}$ over σ if σ' is a recoloring of σ and $\vec{\mathbf{G}}'$ is a σ'-expansion of $\vec{\mathbf{G}}$ (i.e. it does not differ from $\vec{\mathbf{G}}$ on the predicates in σ). We write *ϕ is over a recoloring of σ* if ϕ is over σ' and σ' is a recoloring of σ.

For each $p \geq 0$ we define $\mathcal{C}_p$ to be the class of all recolorings $\vec{\mathbf{G}}_p'$ of $\vec{\mathbf{G}}_p$ for some $\mathbf{G} \in \mathcal{C}$. In other words $\mathcal{C}_p$ is the class of functional representations of all recolorings of all p-th augmentations of graphs from $\mathcal{C}$. Note that all graphs from $\mathcal{C}_p$ are recolorings of a structure in $\sigma_{\mathcal{C}}(p)$, hence they use at most $\alpha_{\mathcal{C}}(p)$ function symbols.

From now on we assume that all graphs are from $\mathcal{C}$ and all queries are in their functional representation. It follows from the discussion above that this is without loss of generality.

Example 7.2.1 *It would be tempting to set $\sigma_{\mathcal{C}}(i)$ to be the functional structure with $\Gamma_{\mathcal{C}}(i)$ functional symbols that would then be used to encode up to $\Gamma_{\mathcal{C}}(i)$ predecessors of each node. We could then easily have properties 1 and 2, but it would not be the case for property 3. To see this consider the following simple example:*

$\mathcal{C}$ is such that $\Gamma_{\mathcal{C}}(i) = 2$ for all i and $\mathbf{G} \in \mathcal{C}$ is defined as $\mathbf{G} = (V = \{u, v, w\}, E = \{(u, w), (v, w)\})$. Wlog assume that the functional structure describing $\mathbf{G}$ is $\vec{\mathbf{G}}_1 = (V = \{u, v, w\}, \{f_1(w) = u\}, \{f_2(w) = v\})$ and so we need to show a transitive fraternal augmentation $\vec{\mathbf{G}} = \vec{\mathbf{G}}_0 \subseteq \vec{\mathbf{G}}_1 \subseteq \vec{\mathbf{G}}_2 \subseteq \dots$ with the desired properties 1, 2 and 3.

Note that (u, v) is a fraternal pair of nodes in $\vec{\mathbf{G}}_1$ and so $\vec{\mathbf{G}}_2$ must describe an edge between u and v (in at least one of the directions). To match property 2, $\vec{\mathbf{G}}_2$ must contain $\vec{\mathbf{G}}_1$ and wlog we may assume that $\vec{\mathbf{G}}_2$ contains $(V = \{u, v, w\}, \{f_1(w) = u, f_1(u) = v\}, \{f_2(w) = v\})$.

Consider now the following query ϕ over $\sigma_{\mathcal{C}}(0)$:

$\phi(x, y) \equiv f_1(x) = y \vee f_2(x) = z$.

Clearly $(u, v) \in \phi(\vec{\mathbf{G}}_2)$, but $(u, v) \notin \phi(\vec{\mathbf{G}}_1)$ and although $\Delta^-(\vec{\mathbf{G}}_2) \leq 2$, two functional symbols in $\sigma_{\mathcal{C}}(1)$ are not enough to retain property 3.

The general idea behind the above example is that in order to have property 3, we cannot "re-use" functions used in $\vec{\mathbf{G}}_i$ to encode edges that appeared in $\vec{\mathbf{G}}_{i+1}$.

7.2.3 From structures to graphs

Recall the definition of the adjacency graph from Section 2.8.1. As we already pointed out in that section, the definition given there was slightly imprecise, as it did not distinguish colors of edges. While this was enough before (as so far we only cared about the "structure" of the adjacency graph, that is the way nodes are connected), we would like to make it fully precise now. As this will be a lot more convenient, we directly do it in a functional manner.

The *adjacency graph* of a relational structure $\mathbf{D}$, denoted by Adjacency($\mathbf{D}$), is a functional graph defined as follows. The set of vertices of Adjacency($\mathbf{D}$) is $D \cup T$ where T is the set of tuples occurring in some relation of $\mathbf{D}$. For each relation R_i in the schema of $\mathbf{D}$, there is a unary symbol P_{R_i} coloring the elements of T belonging to R_i. For each tuple $t = (a_1, \cdots, a_{r_i})$ such that $\mathbf{D} \models R_i(t)$ for some relation R_i of arity r_i, we have an edge $f_j(t) = a_j$ for all $j \leq r_i$.

Observation 7.2.1 *It is immediate to see that for every relational structure $\mathbf{D}$ we can compute Adjacency($\mathbf{D}$) in time $O(\|\mathbf{D}\|)$.*

Let $\mathcal{C}$ be a class of relational structures. We say that $\mathcal{C}$ has *bounded expansion* if the class $\mathcal{C}$' of adjacency graphs of structures from $\mathcal{C}$ has bounded expansion.

Remark 7.2.1 *In the literature, for instance [29, 42], a class $\mathcal{C}$ of relational structures is said to have bounded expansion if the class of their Gaifman graphs has bounded expansion. Our definition is equivalent to the usual one as shown in Theorem 3.3.3. As it gives directly an oriented graph, it is more convenient for us.*

Let $\Gamma_{\mathcal{C}'}$ be the function given by Point 2 of Theorem 2.8.1 for $\mathcal{C}$'. The following lemma is immediate.

Lemma 7.2.2 *Let $\mathcal{C}$ be a class of relational structures with bounded expansion and let $\mathcal{C}$' be the underlying class of adjacency graphs. Let $\phi(\bar{x}) \in$ FO. In time linear in the size of ϕ we can find a query $\psi(\bar{x})$ over $\sigma_{\mathcal{C}'}(0)$ such that for all $\boldsymbol{D} \in \mathcal{C}$ we have $\phi(\boldsymbol{D}) = \psi(Adjacency(\boldsymbol{D}))$.*

As a consequence of Lemma 7.2.2 it follows that the model checking, enumeration and counting of first-order queries over relational structures reduce to the graph case. Therefore in the rest of this chapter we will only concentrate on the graph case (viewed as a functional structure), but the reader should keep in mind that all the results stated over graphs extend to relational structures via this lemma.

7.2.4 Normal form for quantifier-free first-order queries

We conclude this section by proving a normal form on quantifier-free FO formulas. This normal form will be the ground for all our algorithms later on. It basically says that, modulo performing some extra augmentation steps, a quantifier-free formula has a very simple form.

Fix class $\mathcal{C}$ of graphs with bounded expansion. Recall that we are now implicitly assuming that graphs are represented as functional structures.

A formula is *simple* if it does not contain atoms of the form $f(g(x))$, i.e. it does not contain any compositions of functions. Observe that, modulo augmentations, any formula can be transformed into a simple one.

Lemma 7.2.3 *Let $\psi(\bar{x})$ be a formula over a recoloring of $\sigma_{\mathcal{C}}(p)$. Then, for $q = p + |\psi|$, there is a simple formula $\psi'(\bar{x})$ over a recoloring of $\sigma_{\mathcal{C}}(q)$ such that:*

for all $\vec{\mathbf{G}} \in \mathcal{C}_p$ there is a $\vec{\mathbf{G}}' \in \mathcal{C}_q$ computable in time linear in $\|\vec{\mathbf{G}}\|$ such that $\psi(\vec{\mathbf{G}}) = \psi'(\vec{\mathbf{G}}')$.

PROOF This is a simple consequence of transitivity. Any composition of two functions in $\vec{\mathbf{G}}$ represents a transitive pair of edges and becomes a single edge in the 1-augmentation $\vec{\mathbf{H}}$ of $\vec{\mathbf{G}}$. Then $f(g(x))$ over $\vec{\mathbf{G}}$ is equivalent to $h(x) \wedge P_{f,g,h}(x)$ over $\vec{\mathbf{H}}$, where the newly introduced color $P_{f,g,h}$ holds for those nodes v, for which the $f(g(v)) = h(v)$. As the nesting of compositions of functions is at most $|\psi|$, the result follows. The linear time computability is immediate from Lemma 7.2.1.

■

We make one more observation before proving the normal form:

Lemma 7.2.4 *Let $\vec{\mathbf{G}} \in \mathcal{C}_p$. Let u be a node of $\vec{\mathbf{G}}$. Let S be all the predecessors of u in $\vec{\mathbf{G}}$ and set $q = p + \Gamma_{\mathcal{C}}(p)$. Let $\vec{\mathbf{G}}' \in \mathcal{C}_q$ be the $(q-p)$-th augmentation of $\vec{\mathbf{G}}$. There exists a linear order $<$ induced on S by $\vec{\mathbf{G}}'$, such that for all $v, v' \in S$, $v < v'$ implies $v' = f(v)$ is an edge of $\vec{\mathbf{G}}'$ for some function f from $\sigma_{\mathcal{C}}(q)$.*

PROOF This is because all nodes of S are fraternal and the size of S is at most $\Gamma_{\mathcal{C}}(p)$. Hence, after one step of augmentation, all nodes of S are pairwise connected and, after at most $\Gamma_{\mathcal{C}}(p) - 1$ further augmentation steps, if there is a directed path from one node u of S to another node v of S, then there is also a directed edge from u to v. By induction on $|S|$ we show that there exists a node $u \in S$ such that for all $v \in S$ there is an edge from v to u. If $|S| = 1$ there is nothing to prove. Otherwise fix $v \in S$ and let $S' = S \setminus \{v\}$. By induction we get a u in S' satisfying the properties. If there is an edge from v to u, u also works for S and we are done. Otherwise there must be an edge from u to v. But then there is a path of length 2 from any node of S' to v. By transitivity this means that there is an edge from any node of S' to v and v is a node having all the desired properties.

We then set u as the minimal element of our order on S and we repeat this argument with $S \setminus \{u\}$.

■

Note that there might possibly be many linear orders $<$ as described in the statement of the above lemma (the main reason for this is that two different nodes might be predecessors of each other), but the main focus of this lemma is that there always is at least one such linear order.

Lemma 7.2.4 justifies the following definition.

Definition 7.2.1 *A p-type $\tau_p(x)$ is a quantifier-free conjunctive formula expressing all the relations between predecessors of a node x in some graph $\vec{\mathbf{G}} \in \mathcal{C}_p$ in the $(q-p)$-th augmentation $\vec{\mathbf{G}}'$ of $\vec{\mathbf{G}}$, where q is given by Lemma 7.2.4. More precisely, for every functions $f_i, f_j \in \sigma_{\mathcal{C}}(p)$, $\tau_p(x)$ contains at least one of the conjuncts $h_{i,j}(f_i(x)) = f_j(x)$ or $h_{j,i}(f_j(x)) = f_i(x)$, where $h_{i,j}$ and $h_{j,i}$ are function symbols from $\sigma_{\mathcal{C}}(q)$.*

In particular, a p-type τ induces a linear order on the predecessors of x as described by Lemma 7.2.4 ($f_i(x) < f_j(x)$ whenever $h_{i,j}(f_i(x)) = f_j(x)$ is a conjunct of τ) and moreover specifies all the relations between these predecessors in $\vec{\mathbf{G}}'$. Note that for a given p there are only finitely many possible p-types and that each of them can be specified with a conjunctive formula over $\sigma_{\mathcal{C}}(q)$.

We now state the normal form result.

Proposition 7.2.1 *Let $\phi(\bar{x}y)$ be a simple quantifier-free query over a recoloring of $\sigma_{\mathcal{C}}(p)$. There exists q that depends only on p and ϕ and a quantifier-free query ψ over a recoloring of $\sigma_{\mathcal{C}}(q)$ that is a disjunction of conjunctive formulas:*

$$\psi_1(\bar{x}) \wedge \tau(y) \wedge \Delta^{=}(\bar{x}y) \wedge \Delta^{\neq}(\bar{x}y), \tag{7.1}$$

where $\tau(y)$ contains a p-type of y; $\Delta^{=}(\bar{x}y)$ is either empty or contains one clause of the form $y = f(x_i)$ or one clause of the form $f(y) = g(x_i)$ for some suitable i, f and g; and $\Delta^{\neq}(\bar{x}y)$ contains arbitrarily many clauses of the form $y \neq f(x_i)$ or $f(y) \neq g(x_j)$. Moreover, ψ is such that:

for all $\vec{\mathbf{G}} \in \mathcal{C}_p$ there is a $\vec{\mathbf{G}}' \in \mathcal{C}_q$ computable in time linear in $\|\vec{\mathbf{G}}\|$ with $\phi(\vec{\mathbf{G}}) = \psi(\vec{\mathbf{G}}')$.

PROOF

Set q as given by Lemma 7.2.4. We first put ϕ into a disjunctive normal form (DNF) and in front of each such disjunct we add a big disjunction over all possible p-types of y (recall that a p-type can be specified as a conjunctive formula). Let ϕ' be the resulting formula.

We deal with each disjunct of ϕ' separately. The rest of the proof heavily relies on the fact that part of each such disjunct specifies the p-type of y. Recall that a fixed p-type determines all the relations among the predecessors of a node and in fact enforces a certain linear order among them (see Definition 7.2.1 and Lemma 7.2.4).

Note that each disjunct is a query over $\sigma_{\mathcal{C}}(q)$ of the form:

$$\psi_1(\bar{x}) \wedge \tau(y) \wedge \Delta^{=}(\bar{x}y) \wedge \Delta^{\neq}(\bar{x}y),$$

where all sub-formulas except for $\Delta^{=}$ are as desired. Moreover, $\psi_1(\bar{x})$, $\Delta^{=}(\bar{x}y)$ and $\Delta^{\neq}(\bar{x}y)$ are in fact queries over $\sigma_{\mathcal{C}}(p)$. At this point $\Delta^{=}$ contains arbitrarily many clauses of the form $y = f(x_i)$

or $f(y) = g(x_i)$. If it contains at least one clause of the form $y = f(x_i)$, we can replace each other occurrence of y by $f(x_i)$ and we are done.

Assume now that $\Delta^=$ contains several conjuncts of the form $f_i(y) = g(x_k)$. Assume wlog that the p-type contained in τ is such that $f_1(y) < f_2(y) < \cdots$, where $f_1(y), f_2(y), \ldots$ are all the predecessors of y from $\sigma_{\mathcal{C}}(p)$ and $<$ is the linear order as described in Lemma 7.2.4. Let i_0 be the smallest index i such that a clause of the form $f_i(y) = g(x_k)$ belongs to $\Delta^=$. We have $f_{i_0}(y) = g(x_k)$ in $\Delta^=$ and observe that τ specifies for $i < j$ a function $h_{i,j}$ in $\sigma_{\mathcal{C}}(q)$ such that $h_{i,j}(f_i(y)) = f_j(y)$. Then, as y is of type τ, a clause of the form $f_j(y) = h(x_{k'})$ with $i_0 < j$ is equivalent to $h_{i_0,j}(g(x_k)) = h(x_{k'})$.

Let ψ be the result of performing this operation on each disjunct of ϕ'.

Now, given $\vec{\mathbf{G}} \in \mathcal{C}_p$, let $\vec{\mathbf{G}}' \in \mathcal{C}_q$ be the $(q-p)$-th augementation of $\vec{\mathbf{G}}$. It is computable in time linear in $\vec{\mathbf{G}}$ by Lemma 7.2.1. By Lemma 7.2.4 we have $\phi(\vec{\mathbf{G}}) = \phi'(\vec{\mathbf{G}}')$. By construction we have $\psi(\vec{\mathbf{G}}') = \phi'(\vec{\mathbf{G}}')$ and the result follows.

■

Example A-4 *Let us see what Lemma 7.2.3 and the normalization algorithm do for $p = 0$ and some of the disjuncts of the query of Example A-3:*

In the case of $f(g(x)) = y$ note that by transitivity, in the augmented graph, this clause is equivalent to one of the form $y = h(x) \wedge P_{f,g,h}(x)$ (this case is handled by Lemma 7.2.3).

Consider now $\exists z \;\; f(z) = x \wedge g(z) = y$. It will be convenient to view this query when z plays the role of y in Proposition 7.2.1. Notice that in this case it is not in normal form as $\Delta^=$ contains two elements. However, the two edges $f(z) = x$ and $g(z) = y$ are fraternal. Hence, after one augmentation step, a new edge is added between x and y and we either have $y = h(x)$ or $x = h(y)$ for some h in the new signature.

Let $\tau_{h,f,g}(z)$ be a 0-type stating that $h(f(z)) = g(z)$ and $\tau_{h,g,f}(z)$ be a 0-type stating that $h(g(z)) = f(z)$. It is now easy to see that the query $\exists z \;\; f(z) = x \wedge g(z) = y$ is equivalent, in the augmented graph, to

$$\exists z \bigvee_h \; y = h(x) \wedge \tau_{h,f,g}(z) \wedge f(z) = x \quad \vee$$
$$x = h(y) \wedge \tau_{h,g,f}(z) \wedge f(z) = x$$

7.3 Model checking

In this section we show that the model checking problem of FO over a class of structures with bounded expansion can be done in time linear in the size of the structure. Recall that by Lemma 7.2.2 it is enough to consider oriented graphs viewed as functional structures and so Theorem 7.1.1 follows from Theorem 7.3.1 below:

Theorem 7.3.1 ([29, 42, 47]) *Let $\mathcal{C}$ be a class of graphs with bounded expansion and let ψ be a sentence of* FO. *Then, for all $\vec{\mathbf{G}} \in \mathcal{C}$, testing whether $\vec{\mathbf{G}} \models \psi$ can be done in time $O(\|\vec{\mathbf{G}}\|)$.*

The proof of Theorem 7.3.1 is done using a quantifier elimination procedure: given a query $\psi(\bar{x}y)$ with $|\bar{x}| \geq 1$ we can compute a quantifier-free query $\phi(\bar{x})$ that is "equivalent" to $\exists y \psi(\bar{x}y)$. Again, the equivalence should be understood modulo some augmentation steps for a number of augmentation steps depending only on $\mathcal{C}$ and $|\psi|$. When starting with a sentence ψ we end-up with ϕ being a boolean combination of formulas with one variable. Those can be easily tested in linear time in the size of the augmented structure, which in turns can be computed in time linear from the initial structure by Lemma 7.2.1. The result follows. We now state precisely the quantifier elimination step:

Proposition 7.3.1 *Let $\mathcal{C}$ be a class of graphs with bounded expansion witnessed by the function $\Gamma_{\mathcal{C}}$. Let $\psi(\bar{x}y)$ be a quantifier-free formula over a recoloring of $\sigma_{\mathcal{C}}(p)$ with $|\bar{x}| \geq 1$. Then one can compute a q and a quantifier-free formula $\phi(\bar{x})$ over a recoloring of $\sigma_{\mathcal{C}}(q)$ such that:*

for all $\vec{\mathbf{G}} \in \mathcal{C}_p$ there is a $\vec{\mathbf{G}}' \in \mathcal{C}_q$ such that:

$$\phi(\vec{\mathbf{G}}') = (\exists y \psi)(\vec{\mathbf{G}})$$

Moreover, $\vec{\mathbf{G}}'$ is computable in time $O(\|\vec{\mathbf{G}}\|)$.

Before going into details, we start with an outline of the proof. The reasoning is going to be as follows:

• Using Lemma 7.2.3 and Proposition 7.2.1 we argue that it suffices to show the quantifier elimination procedure only for $\psi(\bar{x}y)$ being of the special form given by (7.1), that is:

$$\psi_1(\bar{x}) \wedge \tau(y) \wedge \Delta^{=}(\bar{x}y) \wedge \Delta^{\neq}(\bar{x}y).$$

• In order to eliminate the existentially quantified variable y we somehow need to encode its existence in terms of the properties of $\bar{x}$.

• In the easy case when ψ contains conjunct of the form $f(x_i) = y$, we can replace each occurrence of y with $f(x_i)$ and we are done.

• The most interesting case is when ψ contains conjunct of the form $f(y) = g(x_i)$. Then the algorithm proceeds as follows:

∘ it iterates through all nodes v of the graph (think of v as of a candidate for substituting the existentially quantified variable y) and in a sense "registers" its existence to node $f(v)$,

∘ given tuple $\bar{u}$ to be substituted for for $\bar{x}$ it is enough to only check nodes from the "list of registrants" of $g(u_i)$ for the possible candidates for y,

∘ unfortunately the above procedure could produce "lists of registrants" of arbitrary lengths, so we have to be more careful,

∘ therefore we limit the "registration" process and allow new nodes to register only if they are "different enough" (in terms of the sets of their predecessors) from the nodes that already registered,

∘ this way we define so called WITNESS sets that are of constant (i.e. independent from the size of $\vec{\mathbf{G}}$) sizes and such that if there exists a valid node for y, there also exists such a node inside WITNESS$(g(u_i))$,

∘ the rest of the argument is a way of encoding WITNESS sets by only recoloring the structure and not altering its functional part.

We now formalize the above approach:

PROOF [of Proposition 7.3.1] Wlog (modulo augmentations, see Lemma 7.2.3 for details) we assume that ψ is simple.

We apply Proposition 7.2.1 to ψ and p and obtain a q and an equivalent formula in DNF, where each disjunct has the special form given by (7.1). As disjunction and existential quantification commute, it is enough to treat each part of the disjunction separately.

We thus assume that $\psi(\bar{x}y)$ is a quantifier-free conjunctive formula over a recoloring of $\sigma_{\mathcal{C}}(q)$ of the form (7.1):

$$\psi_1(\bar{x}) \wedge \tau(y) \wedge \Delta^{=}(\bar{x}y) \wedge \Delta^{\neq}(\bar{x}y).$$

We assume wlog that τ contains a p-type enforcing $f_1(y) < f_2(y) < \cdots$, where $f_1(y), f_2(y), \cdots$ are all the images of y by a function from $\sigma_{\mathcal{C}}(p)$. Moreover, for each $i < j$, τ contains an atom of the form $h_{i,j}(f_i(y)) = f_j(y)$ for some function $h_{i,j} \in \sigma_{\mathcal{C}}(q)$.

If $\Delta^{=}$ is $y = g(x_k)$ for some function g and some k, then we replace y with $g(x_k)$ everywhere in $\psi(\bar{x}y)$ resulting in a formula $\phi(\bar{x})$ having obviously the desired properties.

Assume now that $\Delta^{=}$ is $f(y) = g(x_i)$. Wlog assume that f is f_{i_0} in the order specified by the p-type τ and that $i = 1$. Hence we have $f_{i_0}(y) = g(x_1)$ in $\Delta^{=}$.

We will introduce extra colors in order to simulate all interactions between y and $\bar{x}$.

Let $\vec{\mathbf{G}}''$ be the $(q-p)$-th augmentation of $\vec{\mathbf{G}}$. We construct in time linear in $\|\vec{\mathbf{G}}''\|$ a set $\textsc{Witness}(v)$ for each v of $\vec{\mathbf{G}}''$ such that for all tuples $\bar{v}$ of $\vec{\mathbf{G}}''$, if $\vec{\mathbf{G}}'' \models \psi(\bar{v}u)$ for some node u, then there is a node $u' \in \textsc{Witness}(g(v_1))$ such that $\vec{\mathbf{G}}'' \models \psi(\bar{v}u')$. Moreover, for all v, $|\textsc{Witness}(v)| \leq N$, where N is a number depending only on p. We then encode these witness sets using suitable extra colors.

Computation of the Witness function

We start by initializing $\textsc{Witness}(v) = \emptyset$ for all v.

We then successively investigate all nodes u of $\vec{\mathbf{G}}''$ and do the following. If $\vec{\mathbf{G}}'' \models \neg\tau(u)$ then we move on to the next u. If $\vec{\mathbf{G}}'' \models \tau(u)$ then let $u_1, \cdots, u_l$ be the current value of $\textsc{Witness}(f_{i_0}(u))$.

Let β_p be $\alpha_{\mathcal{C}}(p)(\alpha_{\mathcal{C}}(p)+1)|\bar{x}|+1$.

Let i be minimal such that there exists j with $f_i(u_j) = f_i(u)$ and set $i = \alpha_{\mathcal{C}}(p)+1$ if such an i does not exists. Let $S_i = \{f_{i-1}(u_j) \mid f_i(u_j) = f_i(u)\}$, where $f_0(u_j)$ is u_j in the case where $i = 1$. If $|S_i| \leq \beta_p$ then we add u to $\textsc{Witness}(f_{i_0}(u))$.

The algorithm is linear time and the size of $\textsc{Witness}(v) \leq (\beta_p+1)^{\beta_p+1}$. It remains to show that it has the desired properties.

Analysis of the Witness function

Assume $\vec{\mathbf{G}}'' \models \psi(\bar{v}u)$. If $u \in \textsc{Witness}(g(v_1))$, then we are done. Otherwise note that $f_{i_0}(u) = g(v_1)$ and that $\vec{\mathbf{G}}'' \models \tau(u)$. Let i and S_i be as described in the algorithm when investigating u. As u was not added to $\textsc{Witness}(f_{i_0}(u))$, we must have $|S_i| > \beta_p$. Let $S_i = \{u_{i_1}, \cdots, u_{\beta_p}, \cdots\}$ be the corresponding elements of $\textsc{Witness}(g(v_1))$. Among these data values, for each j at most $\alpha_{\mathcal{C}}(p)$ of them may be a predecessor of v_j. Similarly, for each $i' \leq i$ and each j, at most $\alpha_{\mathcal{C}}(p)$ of them may be such that their image by $f_{i'}$ is a predecessor of v_j. For each $i' > i$ their image is exactly $f_{i'}(u)$ and it does not falsify any inequality conjuncts of ψ. Hence, at most $\alpha_{\mathcal{C}}(p)(\alpha_{\mathcal{C}}(p)+1)|\bar{v}|$ of them may falsify at least one of the inequality conjuncts of ψ. We can therefore find in $\textsc{Witness}(g(v_1))$ at least one element satisfying the formula, as $|S_i| > \alpha_{\mathcal{C}}(p)(\alpha_{\mathcal{C}}(p)+1)|\bar{v}|$.

Recoloring of $\vec{\mathbf{G}}''$

Based on $\textsc{Witness}$ we recolor $\vec{\mathbf{G}}''$ as follows. Let $\gamma_p = (\beta_p+1)^{\beta_p+1}$. For each $v \in \vec{\mathbf{G}}''$ we order $\textsc{Witness}(v)$. We can now speak of the i^{th} witness of v.

For each $i \leq \gamma_p$ we introduce a new unary predicate P_i and for each $u \in \vec{\mathbf{G}}''$ we set $P_i(u)$ if $\textsc{Witness}(u)$ contains at least i elements.

For each $i \leq \gamma_p$ and each $h, h' \in \alpha_{\mathcal{C}}(q)$ we introduce a new unary predicate $P_{i,h,h'}$ and for each $v \in \vec{\mathbf{G}}''$ we set $P_{i,h,h'}(v)$ if the i^{th} witness of $h(v)$ is an element u with $h'(u) = v$.

For each $i \leq \gamma_p$ and each $h \in \alpha_{\mathcal{C}}(q)$ we introduce a new unary predicate $Q_{i,h}$ and for each $v \in \vec{\mathbf{G}}''$ we set $Q_{i,h}(v)$ if the i^{th} witness of $h(v)$ is v.

We denote by $\vec{\mathbf{G}}'$ the resulting graph and notice that it can be computed in linear time from $\vec{\mathbf{G}}''$.

Finally, note that if y is the i^{th} witness of $g(x_1)$, the equality $f_j(y) = h(x_k)$ with $j < i_0$ is equivalent over $\vec{\mathbf{G}}'$ to $h_{j,i_0}(h(x_k)) = g(x_1) \wedge P_{i,h_{j,i_0},f_j}(h(x_k))$ and the equality $y = h(x_k)$ is equivalent over $\vec{\mathbf{G}}'$ to $f_{i_0}(h(x_k)) = g(x_1) \wedge Q_{i,f_{i_0}}(h(x_k))$. From the definition of p-type, the equality $f_j(y) = h(x_k)$ with $j > i_0$ is equivalent to $h_{i_0,j}(g(x_1)) = h(x_k)$.

Computation of ϕ

In view of the analysis above, $\psi(\bar{x}y)$ is equivalent to a formula:

$$\bigvee_{i\leq\gamma_p} \psi_1(\bar{x}) \wedge \psi^i(\bar{x})$$

where $\psi^i(\bar{x})$ checks that the i^{th} witness of $g(x_1)$ makes the initial formula true. In view of the above, this formula $\psi^i(\bar{x})$ is defined by

$$\begin{aligned} P_i(g(x_1)) \quad \wedge & \bigwedge_{\substack{f_j(y)\neq h(x_k)\in\Delta^{\neq} \\ j<i_0}} \neg\big(h_{j,i_0}(h(x_k)) = g(x_1) \wedge P_{i,h_{j,i_0},f_j}(h(x_k))\big) \\ \wedge & \bigwedge_{\substack{f_j(y)\neq h(x_k)\in\Delta^{\neq} \\ j\geq i_0}} h_{i_0,j}(g(x_1)) \neq h(x_k) \\ \wedge & \bigwedge_{y\neq h(x_k)\in\Delta^{\neq}} \neg\big(f_{i_0}(h(x_k)) = g(x_1) \wedge Q_{i,f_{i_0}}(h(x_k))\big) \end{aligned}$$

The special case when $\Delta^=$ is empty is a simpler version of the previous case, only this time it is enough to construct a set WITNESS which does not depend on v. This is done as follows:

It is constructed as in the previous case and verifies: for all tuples $\bar{v}$ of $\vec{\mathbf{G}}''$, if $\vec{\mathbf{G}}'' \models \psi(\bar{v}u)$ for some node u, then there is a node $u' \in$ WITNESS such that $\vec{\mathbf{G}}'' \models \psi(\bar{v}u')$. Moreover, $|\text{WITNESS}| \leq \gamma_p$.

Recoloring of $\vec{\mathbf{G}}''$

Based on WITNESS we recolor $\vec{\mathbf{G}}''$ as follows. Let $\gamma_p = (\beta_p + 1)^{\beta_p+1}$. We order WITNESS and we can now speak of the i^{th} witness.

For each $i \leq \gamma_p$ we introduce a new unary predicate P_i and for each $v \in \vec{\mathbf{G}}''$ we set $P_i(v)$ if WITNESS contains at least i elements.

For each $i \leq \gamma_p$ and each $h \in \sigma_{\mathcal{C}}(q)$ we introduce a new unary predicate $P_{i,h}$ and for each $v \in \vec{\mathbf{G}}''$ we set $P_{i,h}(v)$ if the i^{th} witness is a element u with $h(u) = v$.

For each $i \leq \gamma_p$ and each $h \in \sigma_{\mathcal{C}}(q)$ we introduce a new unary predicate Q_i and for each $v \in \vec{\mathbf{G}}''$ we set $Q_i(v)$ if the i^{th} witness is v.

We denote by $\vec{\mathbf{G}}'$ the resulting graph and notice that it can be computed in linear time from $\vec{\mathbf{G}}$.

Finally, note that if y is the i^{th} witness, the equality $f_j(y) = h(x_k)$ is equivalent over $\vec{\mathbf{G}}'$ to $P_{i,f_j}(h(x_k))$ and the equality $y = h(x_k)$ is equivalent over $\vec{\mathbf{G}}'$ to $Q_i(h(x_k))$.

The desired formula ϕ is computed as for the previous case when $\Delta^=$ was not empty.

This finishes the proof of the last case among all the possible compositions of $\Delta^=$ and thereby concludes the proof of Proposition 7.3.1.

■

Example A-5 *Consider one of the quantified formulas as derived by Example A-4:*

$$\exists z \;\; y = h(x) \wedge \tau_{h,f,g}(z) \wedge f(z) = x$$

The resulting quantifier-free query has the form:

$$P(x) \wedge h(x) = y$$

where $P(x)$ is a newly introduced color saying "$\exists z \ \ \tau_{h,f,g}(z) \wedge f(z) = x$". The key point is that this new predicate can be computed in linear time by iterating through all nodes z, testing whether $\tau_{h,f,g}(z)$ is true and, if this is the case, coloring $f(z)$ with color P.

Applying the quantifier elimination process from inside out using Proposition 7.3.1 for each step and then applying Lemma 7.2.3 to the result yields:

Theorem 7.3.2 *Let $\mathcal{C}$ be a class of graphs with bounded expansion. Let $\psi(\bar{x})$ be a query of* FO *over a recoloring of $\sigma_{\mathcal{C}}(0)$ with at least one free variable. Then one can compute a p and a simple quantifier-free formula $\phi(\bar{x})$ over a recoloring of $\sigma_{\mathcal{C}}(p)$ such that:*
for all $\vec{\mathbf{G}} \in \mathcal{C}$, we can construct in time $O(\|\vec{\mathbf{G}}\|)$ a graph $\vec{\mathbf{G}}' \in \mathcal{C}_p$ such that

$$\phi(\vec{\mathbf{G}}') = \psi(\vec{\mathbf{G}})$$

We will make use of the following useful consequence of Theorem 7.3.2:

Corollary 7.3.1 *Let $\mathcal{C}$ be a class of graphs with bounded expansion and let $\psi(\bar{x})$ be a formula of* FO *with at least one free variable. Then, for all $\vec{\mathbf{G}} \in \mathcal{C}$, after a preprocessing in time $O(\|\vec{\mathbf{G}}\|)$, we can test, given $\bar{u}$ as input, whether $\vec{\mathbf{G}} \models \psi(\bar{u})$ in constant time.*

PROOF By Theorem 7.3.2 it is enough to consider quantifier-free simple queries. Hence it is enough to consider a query consisting in a single atom of either $P(x)$ or $P(f(x))$ or $x = f(y)$ or $f(x) = g(y)$ or $x = y$ or of a negation of any of the above.

During the preprocessing phase we associate with each node v of the input graph a list $L(v)$ containing all the predicates satisfied by v and all the images of v by a function symbol from the signature. This can be computed in linear time by enumerating all relations of the database and updating the appropriate lists with the corresponding predicate or the corresponding image.

Now, because we use the RAM model, given u we can in constant time recover the list $L(u)$. Using those lists it is immediate to check all atoms of the formula in constant time.

■

Theorem 7.3.1 is a direct consequence of Theorem 7.3.2 and Corollary 7.3.1: Starting with a sentence, and applying Theorem 7.3.2 for eliminating quantifiers from inside out we end up with a Boolean combination of formulas with one variable. Each such formula can be tested in $O(\|\vec{\mathbf{G}}\|)$ by iterating through all nodes v of $\vec{\mathbf{G}}$ and in constant time (using Corollary 7.3.1) checking if v can be substituted for the sole existentially quantified variable.

On top of Theorem 7.3.1 the following corollary is immediate from Theorem 7.3.2 and Corollary 7.3.1:

Corollary 7.3.2 *Let $\mathcal{C}$ be a class of graphs with bounded expansion and let $\psi(x)$ be a formula of* FO *with one free variable. Then, for all $\vec{\mathbf{G}} \in \mathcal{C}$, computing the set $\psi(\vec{\mathbf{G}})$ can be done in time $O(\|\vec{\mathbf{G}}\|)$.*

7.4 Testing

In this section we prove Theorem 7.1.3. In fact there is almost nothing left to do: we already have the solution to the testing problem for classes of graphs with bounded expansion (cf. Corollary 7.3.1) and Lemma 7.2.2 justifies that it is all that we need.

7.5 Enumeration

In this section we consider first-order formulas with free variables and show that we can enumerate with constant delay answers to those queries over any class of databases with bounded expansion. Moreover, assuming a linear order on the domain of the input structure, we will see that the answers can be output in the lexicographical order. As before we only state the result for graphs, but it immediately extends to arbitrary structures by Lemma 7.2.2 and so proves Theorem 7.1.2. Recall that we assumed (without loss of generality) the presence of a linear order on the domain.

Theorem 7.5.1 *Let $\mathcal{C}$ be a class of graphs with bounded expansion and let $\phi(\bar{x})$ be a first-order query over $\sigma_{\mathcal{C}}(0)$. Then the enumeration problem of ϕ over $\mathcal{C}$ is in* CONSTANT-DELAY$_{lin}$*. Moreover, the answers to ϕ can be output in lexicographical order.*

Before going into details, we start with an outline of the proof. The reasoning is going to be as follows:

• The proof is by induction on the number of free variables.

• The case $k = 1$ is done by Corollary 7.3.2.

• For $k > 1$, using the normalization and quantification procedures of the previous sections, it is enough to consider quantifier-free queries $\psi(\bar{x}y)$ of the form:

$$\psi_1(\bar{x}) \wedge \tau(y) \wedge \Delta^{=}(\bar{x}y) \wedge \Delta^{\neq}(\bar{x}y).$$

We further set $\psi''(\bar{x})$ to be the formula $\exists y \psi(\bar{x}y)$.

• In the easy case when ψ contains conjunct of the form $f(x_i) = y$, we enumerate $\psi''(\bar{x})$ by induction and append $f(x_i)$ to each resulting tuple.

• The most interesting case is when ψ contains conjunct of the form $f(y) = g(x_i)$. Then the algorithm proceeds as follows:

◦ It enumerates all the solutions to $\psi''(\bar{x})$ by induction and appends to it all the relevant y.

◦ For this it computes, during the preprocessing phase, several successor functions among nodes, such that for each $\bar{x}$, at least one of them will enumerate the associated y.

◦ The key point is that only finitely many successor functions need to be precomputed and that the suitable one can be found by looking only at $\bar{x}$.

We now formalize the above approach:

PROOF [of Theorem 7.5.1]

Fix a class $\mathcal{C}$ of graphs with bounded expansion and a query $\phi(\bar{x})$ with k free variables. Let $\vec{\mathbf{G}}$ be the input graph and V be its set of vertices.

The proof is by induction on the number of free variables. The case $k = 1$ is done by Corollary 7.3.2.

Assume now that $k > 1$ and that $\bar{x}$ and y are the free variables of ϕ, where $|\bar{x}| = k - 1$.

We apply Theorem 7.3.2 to get a simple quantifier-free query $\varphi(\bar{x}y)$ and a structure $\vec{\mathbf{G}}' \in \mathcal{C}_p$, for some p that does not depend on $\vec{\mathbf{G}}$, such that $\varphi(\vec{\mathbf{G}}') = \phi(\vec{\mathbf{G}})$ and $\vec{\mathbf{G}}'$ can be computed in linear time from $\vec{\mathbf{G}}$.

We normalize the resulting simple quantifier-free query using Proposition 7.2.1, and obtain an equivalent quantifier-free formula ψ and a structure $\vec{\mathbf{G}}'' \in \mathcal{C}_q$, where q depends only on p and φ, $\vec{\mathbf{G}}''$ can be computed in linear time from $\vec{\mathbf{G}}'$, $\varphi(\vec{\mathbf{G}}') = \psi(\vec{\mathbf{G}}'')$ and ψ is a disjunction of formulas of the form (7.1):

$$\psi_1(\bar{x}) \wedge \tau(y) \wedge \Delta^{=}(\bar{x}y) \wedge \Delta^{\neq}(\bar{x}y),$$

where $\Delta^{=}(\bar{x}y)$ is either empty or contains one clause of the form $y = f(x_i)$ or one clause of the form $f(y) = g(x_i)$ for some suitable i, f and g; and $\Delta^{\neq}(\bar{x}y)$ contains arbitrarily many clauses of the form $y \neq f(x_i)$ or $f(y) \neq g(x_j)$.

By Fact 3.1.4 it is enough to show that we can enumerate each disjunct separately. In the sequel we then assume that ψ has the form described in (7.1). We let $\psi'(y)$ be the formula $\exists\bar{x}\psi(\bar{x}y)$ and $\psi''(\bar{x})$ be the formula $\exists y\psi(\bar{x}y)$.

If $\Delta^=$ contains an equality of the form $y = f(x_i)$ then we replace y by $f(x_i)$ in $\psi(\bar{x}y)$, enumerate by induction the formula ψ'' and replace each of its output $\bar{a}$ with $(\bar{a}f(a_i))$ in order to obtain the desired constant delay enumeration algorithm. We therefore now assume that $\Delta^=$ does not contain such equality.

We now define two functions $L : V \to 2^V$ and $W : V^{k-1} \to V$ depending on whether $\Delta^=$ is empty or consists of a single clause of the form $f(y) = g(x_i)$. If $\Delta^=$ is empty we pick an arbitrary node w in $\vec{\mathbf{G}}''$ and set $L(w) = \psi'(\vec{\mathbf{G}}'')$, $L(v) = \emptyset$ for $v \neq w$, and $W(\bar{v}) = w$ for all tuples $\bar{v}$. If $\Delta^= = \{f(y) = g(x_i)\}$ we set $W(\bar{v}) = g(v_i)$ for all tuples $\bar{v}$ and define L using the following procedure. We initialize $L(v)$ to $\emptyset$ for each $v \in V$. Then, for each $v \in \psi'(\vec{\mathbf{G}}'')$, we add v to the set $L(f(v))$.

Notice that L can be computed in time linear in $\|\vec{\mathbf{G}}''\|$ (using Corollary 7.3.2), that each list $L(v)$ is sorted with respect to the linear order on the domain and that, given $\bar{v}$, $W(\bar{v})$ can be computed in constant time. Moreover, for each $\bar{v}u$, $\vec{\mathbf{G}}'' \models \psi(\bar{v}u)$ implies $u \in L(W(\bar{v}))$ and if $u \in L(W(\bar{v}))$ then $\Delta^=(\bar{v}u)$ is true.

By induction we can enumerate $\psi''(\bar{x})$ with constant delay.

On top of the linear time preprocessing necessary for enumerating ψ'' we do the following extra preprocessing. We first compute $L(v)$ for all $v \in V$. Then, for each $v \in V$, we perform the following procedure on $L(v)$. Each procedure will work in time linear in the size of $L(v)$, hence the total preprocessing will take time $O(|V|)$.

Fix v and set $L = L(v)$. We denote by $<$ the order on L. (Recall that this order is consistent with the initial order on the domain.)

For $S_1, \ldots, S_{\alpha_{\mathcal{C}}(q)} \subseteq V$ we define $\text{NEXT}_{f_1,S_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$ to be the first element $w \geq u$ of L such that $f_1(w) \notin S_1, \ldots,$ and $f_{\alpha_{\mathcal{C}}(q)}(w) \notin S_{\alpha_{\mathcal{C}}(q)}$. If such w does not exist, the value of $\text{NEXT}_{f_1,S_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$ is NULL. When all S_i are empty, we write $\text{next}_\emptyset(u)$ and by the above definitions we always have $\text{next}_\emptyset(u) = u$. We denote such functions as *shortcut pointers of* u. We write $\text{NEXT}_{f_1,S'_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S'_{\alpha_{\mathcal{C}}(q)}}(u) \preceq \text{NEXT}_{f_1,S_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$ if for each $1 \leq i \leq \alpha_{\mathcal{C}}(q)$ we have $S'_i \subseteq S_i$. Note that for a given u the $\preceq$ relation is a partial order on the set of shortcut pointers of u. A trivial observation is that if $\text{NEXT}_{f_1,S'_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S'_{\alpha_{\mathcal{C}}(q)}}(u) \preceq \text{NEXT}_{f_1,S_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$, then $\text{NEXT}_{f_1,S'_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S'_{\alpha_{\mathcal{C}}(q)}}(u) \leq \text{NEXT}_{f_1,S_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$. The *size* of a shortcut pointer $\text{NEXT}_{f_1,S_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$ is the sum of sizes of the sets S_i.

In order to avoid writing too long expressions containing shortcut pointers, we introduce the following abbreviations:

- $\text{NEXT}_{f_1,S_1,\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$ is denoted with $\text{NEXT}_{\vec{S}}(u)$,
- $\text{NEXT}_{f_1,S_1,\ldots,f_i,S_i\cup\{u_i\},\ldots,f_{\alpha_{\mathcal{C}}(q)},S_{\alpha_{\mathcal{C}}(q)}}(u)$ is denoted with $\text{NEXT}_{\vec{S}[S_i+=\{u_i\}]}(u)$.

Set $\beta_q = (k-1) \cdot \alpha_{\mathcal{C}}(q)^2$.

Computing all shortcut pointers of size β_q would take more than linear time. We therefore compute a subset of those, denoted SC_L, that will be sufficient for our needs. SC_L is defined in an inductive manner. For all u, $\text{next}_\emptyset(u) \in \text{SC}_L$. Moreover, if the shortcut pointer NULL $\neq \text{NEXT}_{\vec{S}}(u) \in \text{SC}_L$ and has a size smaller than β_q, then, for each i, $\text{NEXT}_{\vec{S}[S_i+=\{u_i\}]}(u) \in \text{SC}_L$, where $u_i = f_i(\text{NEXT}_{\vec{S}}(u))$. We then say that $\text{NEXT}_{\vec{S}}(u)$ is the *origin* of $\text{NEXT}_{\vec{S}[S_i+=\{u_i\}]}(u)$. Note that SC_L contains all the shortcut pointers of the form $\text{NEXT}_{f_i,\{f_i(u)\}}(u)$ for $u \in L$ and these are exactly the shortcut pointers of u of size 1. By $\text{SC}_L(u) \subseteq \text{SC}_L$ we denote the shortcut pointers of u that are in SC_L.

The set SC_L has the following properties:

Claim 7.5.1 *Let* $\text{NEXT}_{\vec{S}}(u)$ *be a shortcut pointer of size not greater than* β_q*. Then there exists* $\text{NEXT}_{\vec{S'}}(u) \in \text{SC}_L$ *such that* $\text{NEXT}_{\vec{S}}(u) = \text{NEXT}_{\vec{S'}}(u)$*. Moreover, such* $\text{NEXT}_{\vec{S'}}(u)$ *can be found in constant time.*

PROOF If $\text{NEXT}_{\vec{S}}(u) \in \text{SC}_L$, then we have nothing to prove. Assume then that $\text{NEXT}_{\vec{S}}(u) \notin \text{SC}_L$. Let $\text{NEXT}_{\vec{S'}}(u) \in \text{SC}_L$ be any maximal, in terms of its size, shortcut pointer of u such that $\text{NEXT}_{\vec{S'}}(u) \preceq \text{NEXT}_{\vec{S}}(u)$ (recall that this means that for $1 \leq i \leq \alpha_{\mathcal{C}}(q)$ we have $S'_i \subseteq S_i$). Such a shortcut pointer always exists as $\text{next}_\emptyset(u) \preceq \text{NEXT}_{\vec{S}}(u)$ and $\text{next}_\emptyset(u) \in \text{SC}_L$. Note that the size of $\text{NEXT}_{\vec{S'}}(u)$ is strictly smaller than the size of $\text{NEXT}_{\vec{S}}(u)$, so it is strictly smaller than β_q. Clearly, $\text{NEXT}_{\vec{S'}}(u)$ can be found in constant time. We claim that $\text{NEXT}_{\vec{S}}(u) = \text{NEXT}_{\vec{S'}}(u)$.

Let $v = \text{NEXT}_{\vec{S'}}(u)$. We know that $v \leq \text{NEXT}_{\vec{S}}(u)$. Assume now that there would exist $1 \leq i \leq \alpha_{\mathcal{C}}(q)$ such that $u_i = f_i(v) \in S_i$. Then $u_i \notin S'_i$ and as the size of $\text{NEXT}_{\vec{S'}}(u)$ is smaller than β_q, we have that $\text{NEXT}_{\vec{S}[S_i += \{u_i\}]}(u) \in \text{SC}_L$. But $\text{NEXT}_{\vec{S}[S_i += \{u_i\}]}(u)$ has size strictly greater than $\text{NEXT}_{\vec{S'}}(u)$ and $\text{NEXT}_{\vec{S}[S_i += \{u_i\}]}(u) \preceq \text{NEXT}_{\vec{S}}(u)$, which contradicts the maximality of $\text{NEXT}_{\vec{S'}}(u)$. This means that such an i does not exist and concludes the fact that $\text{NEXT}_{\vec{S}}(u) = \text{NEXT}_{\vec{S'}}(u)$.

■

Claim 7.5.2 *There exists a constant* $\zeta(q,k)$ *such that for every node* u *we have* $|\text{SC}_L(u)| \leq \zeta(q,k)$*.*

PROOF Fix u. Note that there is exactly 1 shortcut pointer of u of size 0 ($\text{next}_\emptyset(u)$) and $\alpha_{\mathcal{C}}(q)$ shortcut pointers of u of size 1. By the definition of SC_L, any shortcut pointer $\text{NEXT}_{\vec{S}}(u)$ can be an origin of up to $\alpha_{\mathcal{C}}(q)$ shortcut pointers of the form $\text{NEXT}_{\vec{S}[S_i += \{u_i\}]}(u)$, where $u_i = f_i(\text{NEXT}_{\vec{S}}(u))$ and the size of $\text{NEXT}_{\vec{S}[S_i += \{u_i\}]}(u)$ is either the same as the size of $\text{NEXT}_{\vec{S}}(u)$ (if $u_i \in S_i$) or greater by 1. This way we see that $\text{SC}_L(u)$ contains up to $\alpha_{\mathcal{C}}(q)^2$ shortcut pointers of size 2 and, in general, up to $\alpha_{\mathcal{C}}(q)^s$ shortcut pointers of size s. As the maximal size of a computed shortcut pointer is bounded by β_q, we have $|\text{SC}_L(u)| \leq \sum_{0 \leq i \leq \beta_q} \alpha_{\mathcal{C}}(q)^i$. Both $\alpha_{\mathcal{C}}(q)$ and β_q depend only on q and k, which concludes the proof.

■

The following claim guarantees that SC_L can be computed in linear time and has therefore a linear size.

Claim 7.5.3 SC_L *can be computed in time linear in* $|L|$*.*

PROOF In linear time we set $\text{next}_\emptyset(u) = u$ for $u \in L$.

We first show how to compute shortcut pointers of size 1 of each node $u \in L$. We do it in an inductive manner, starting from the last node of L and moving backwards. Recall that these shortcut pointers are of the form $\text{NEXT}_{f_i,\{f_i(u)\}}(u)$. If u is the last node on L, then all these values are NULL. We now assume that u is not last on L and that for all $v > u$ all the shortcut pointers of v of size 1 were computed. We show how to compute shortcut pointers of u of size 1.

For each $1 \leq i \leq \alpha_{\mathcal{C}}(q)$ we compute $\text{NEXT}_{f_i,\{f_i(u)\}}(u)$. Let v be the node successor of u in L. If $f_i(u) \neq f_i(v)$, then $\text{NEXT}_{f_i,\{f_i(u)\}}(u) = v$. If $f_i(u) = f_i(v)$, then $\text{NEXT}_{f_i,\{f_i(u)\}}(u) = \text{NEXT}_{f_i,\{f_i(\text{next}(v))\}}(\text{next}(v))$ and the later shortcut pointer has already been computed.

Clearly all the shortcut pointers of size 1 are computed in time linear in the size of L.

We now turn to the computation of arbitrary $\text{NEXT}_{\vec{S}}(u) \in \text{SC}_L$ for $u \in L$. We again do it in an inductive manner starting from the last node on L and move backwards. If u is the last node on L then we are already done as all the shortcut pointers of u of size 1 are NULL and by definition there are no shortcut pointers of u of greater sizes in SC_L. We now assume that u is not last on L and that for all $v > u$ set $\text{SC}_L(v)$ is computed. We show how to compute $\text{SC}_L(u)$.

Consider now $\text{NEXT}_{\vec{S}}(u)$. If for all i we have $f_i(u) \notin S_i$, then we are done as $\text{NEXT}_{\vec{S}}(u) = u$. Otherwise there exists i such that $f_i(u) \in S_i$. Let $v = \text{NEXT}_{f_i,\{f_i(u)\}}(u)$. Clearly $v \le \text{NEXT}_{\vec{S}}(u)$ and $\text{NEXT}_{\vec{S}}(u) = \text{NEXT}_{\vec{S}}(v)$. We can conclude this case with observation that $\text{NEXT}_{\vec{S}}(v) = \text{NEXT}_{\vec{S'}}(v)$, where $\text{NEXT}_{\vec{S'}}(v) \in \text{SC}_L(v)$ is the shortcut pointer of v from the application of Claim 7.5.1 to $\text{NEXT}_{\vec{S}}(v)$. Claim 7.5.1 assures that we can find $\text{NEXT}_{\vec{S'}}(v)$ in constant time and since this shortcut pointer is equal to $\text{NEXT}_{\vec{S}}(u)$, thus $\text{NEXT}_{\vec{S}}(u)$ is computed in constant time as well. As Claim 7.5.2 shows that we only need to consider constantly many shortcut pointers for each u, the whole process takes time $O(|L|)$.

■

The computation of SC_L concludes the preprocessing phase and it follows from Claim 7.5.3 that it can be done in linear time. We now turn to the enumeration phase.

We enumerate one by one the solutions to $\psi''(\bar{x})$ by simulating the enumeration algorithm obtained from the induction.

Having a solution $\bar{v}$ to ψ'' by construction we know that all nodes u such that $\vec{\mathbf{G}}'' \models \psi(\bar{v}u)$ are in $L = L(W(\bar{v}))$. Recall also that all elements $u \in L$ make $\tau(u) \wedge \Delta^{=}(\bar{v}u)$ true. For $1 \le i \le \alpha_{\mathcal{C}}(q)$ we set $S_i = \{g(v_j) : g(x_j) \neq f_i(y) \text{ is a conjunct of } \Delta^{\neq}\}$. Starting with u the first node of the sorted list L, we apply the following procedure:

1. If $u =$ NULL, finish the nested enumeration procedure for $\bar{v}$. If not, let $\text{NEXT}_{\vec{S'}}(u)$ be the shortcut pointer from the application of Claim 7.5.1 to $\text{NEXT}_{\vec{S}}(u)$. Set $u' = \text{NEXT}_{\vec{S'}}(u)$. If $u' =$ NULL, finish the nested enumeration procedure for $\bar{v}$.
2. If $\vec{\mathbf{G}}'' \models \psi(\bar{v}u')$, output $(\bar{v}u')$.
3. Reinitialize u to the successor of u' in L and continue with Step 1.

We now show that the algorithm is correct, i.e. that it outputs all $\psi(\vec{\mathbf{G}}'')$ with no repetition.

The algorithm clearly outputs a subset of $\psi(\vec{\mathbf{G}}'')$ as it tests whether $\vec{\mathbf{G}}'' \models \psi(\bar{v}u')$ before outputting tuple $(\bar{v}u')$.

By the definition, list L contains no duplicates and as the algorithm moves only forward on that list, there are no repetitions during the output process.

By the definition of sets S_i and $\text{NEXT}_{\vec{S}}(u)$, for each $u \le w < u'$ there is a suitable i and j such that $g(v_j) = f_i(w)$ and $g(x_j) \neq f_i(y)$ is a conjunct of $\Delta^{\neq}$. This way the algorithm does not skip any solutions at Step 1 and so it outputs exactly $\psi(\vec{\mathbf{G}}'')$.

It remains to show that there is a constant time between any two outputs.

By construction, for each $\bar{v}$, $L = L(W(\bar{v}))$ contains an element u such that $(\bar{v}u)$ is a solution. We therefore need to show that there is a constant time between any two outputs involving an element in L. Step 1 takes constant time due to Claim 7.5.1. From there the algorithm either immediately outputs a solution at Step 2 or jumps to Step 3. This means that $\vec{\mathbf{G}}'' \not\models \psi(\bar{v}u')$, but from the definitions of list L, sets S_i and shortcut pointers $\text{NEXT}_{\vec{S}}(u)$ it is only the $\Delta^{\neq}$ that is falsified and it is because of an inequality of the form $y \neq g(x_j)$ for some suitable g and j (where g may possibly be identity). This implies that $u' = g(v_j)$. As all the elements on L are distinct, the algorithm can skip over Step 2 up to $(k-1) \cdot (\alpha_{\mathcal{C}}(q) + 1)$ times for each tuple $\bar{v}$ (there are up to that many different images of nodes from $\bar{v}$ under $\alpha_{\mathcal{C}}(q)$ different functions and the initial values of $\bar{v}$). This way the delay is bounded by up to $k \cdot (\alpha_{\mathcal{C}}(q) + 1)$ consecutive applications of Claim 7.5.1 and is in fact constant.

As the list L was sorted with respect to the linear order on the domain, it is clear that the enumeration procedure outputs the set of solutions in lexicographical order.

This concludes the proof of the theorem.

■

7.6 Counting

In this section we investigate the problem of counting the number of solutions to a query, i.e. computing $|q(\mathbf{D})|$. As usual we only state and prove our results over graphs but they generalize to arbitrary relational structures via Lemma 7.2.2. This way Theorem 7.6.1 yields Theorem 7.1.4. The proof goes by induction on the number of free variables and follows the same outline as for enumeration. It only replaces the step of the enumeration that was precomputing several successor functions with a combinatorial argument counting their number.

Theorem 7.6.1 *Let $\mathcal{C}$ be class of graphs with bounded expansion and let $\phi(\bar{x})$ be a first-order formula. Then, for all $\vec{\mathbf{G}} \in \mathcal{C}$, we can compute $|\phi(\vec{\mathbf{G}})|$ in time $O(\|\vec{\mathbf{G}}\|)$.*

Proof The key idea is to prove a weighted version of the desired result. Assume $\phi(\bar{x})$ has exactly k free variables and for $1 \leq i \leq k$ we have functions $\#_i : V \to \mathbb{N}$. We will compute in time linear in $\|\vec{\mathbf{G}}\|$ the following number:

$$|\phi(\vec{\mathbf{G}})|_{\#} := \sum_{\bar{u} \in \phi(\vec{\mathbf{G}})} \prod_{1 \leq i \leq k} \#_i(u_i).$$

By setting all $\#_i$ to be constant functions with value 1 we get the regular counting problem. Hence Theorem 7.6.1 is an immediate consequence of the next lemma.

Lemma 7.6.1 *Let $\mathcal{C}$ be class of graphs with bounded expansion and let $\phi(\bar{x})$ be a first-order formula with exactly k free variables. For $1 \leq i \leq k$ let $\#_i : V \to \mathbb{N}$ be functions such that for each v the value of $\#_i(v)$ can be computed in constant time. Then, for all $\vec{\mathbf{G}} \in \mathcal{C}$, we can compute $|\phi(\vec{\mathbf{G}})|_{\#}$ in time $O(\|\vec{\mathbf{G}}\|)$.*

Proof The proof is by induction on the number of free variables.

The case $k = 1$ is trivial: in time linear in $\|\vec{\mathbf{G}}\|$ we compute $\phi(\vec{\mathbf{G}})$ using Corollary 7.3.2. By hypothesis, for each $v \in \phi(\vec{\mathbf{G}})$, we can compute the value of $\#_1(v)$ in constant time. Therefore the value

$$|\phi(\vec{\mathbf{G}})|_{\#} = \sum_{v \in \phi(\vec{\mathbf{G}})} \#_1(v)$$

can be computed in linear time as desired.

Assume now that $k > 1$ and that $\bar{x}$ and y are the free variables of ϕ, where $|\bar{x}| = k - 1$.

We apply Theorem 7.3.2 to get a simple quantifier-free query $\varphi(\bar{x}y)$ and a structure $\vec{\mathbf{G}}' \in \mathcal{C}_p$, for some p that does not depend on $\vec{\mathbf{G}}$, such that $\varphi(\vec{\mathbf{G}}') = \phi(\vec{\mathbf{G}})$ and $\vec{\mathbf{G}}'$ can be computed in linear time from $\vec{\mathbf{G}}$. Note that $|\phi(\vec{\mathbf{G}})|_{\#} = |\varphi(\vec{\mathbf{G}}')|_{\#}$, so it is enough to compute the latter value.

We normalize the resulting simple quantifier-free query using Proposition 7.2.1, and obtain an equivalent quantifier-free formula ψ and a structure $\vec{\mathbf{G}}'' \in \mathcal{C}_q$, where q depends only on p and φ, $\vec{\mathbf{G}}''$ can be computed in linear time from $\vec{\mathbf{G}}'$, $\varphi(\vec{\mathbf{G}}') = \psi(\vec{\mathbf{G}}'')$ and ψ is a disjunction of formulas of the form (7.1):

$$\psi_1(\bar{x}) \wedge \tau(y) \wedge \Delta^{=}(\bar{x}y) \wedge \Delta^{\neq}(\bar{x}y),$$

where $\Delta^{=}(\bar{x}y)$ is either empty or contains one clause of the form $y = f(x_i)$ or one clause of the form $f(y) = g(x_i)$ for some suitable i, f and g; and $\Delta^{\neq}(\bar{x}y)$ contains arbitrarily many clauses of the form $y \neq f(x_i)$ or $f(y) \neq g(x_j)$. Note that $|\varphi(\vec{\mathbf{G}}')|_{\#} = |\psi(\vec{\mathbf{G}}'')|_{\#}$, so it is enough to compute the latter value.

Observe that it is enough to solve the weighted counting problem for each disjunct separately, as we can then combine the results using a simple inclusion-exclusion reasoning. In the sequel we then assume that ψ has the form described in (7.1).

The proof now goes by induction on the number of inequalities in $\Delta^{\neq}$. While the inductive step turns out to be fairly easy, the difficult part is the base step of the induction.

We start with proving the inductive step. Let $g(y) \neq f(x_i)$ be an arbitrary inequality from $\Delta^{\neq}$ (where g might possibly be the identity). Let ψ^- be ψ with this inequality removed and $\psi^+ = \psi^- \wedge g(y) = f(x_i)$. Of course ψ and ψ^+ have disjoint sets of solutions and we have:

$$|\psi(\vec{\mathbf{G}}'')|_{\#} = |\psi^-(\vec{\mathbf{G}}'')|_{\#} - |\psi^+(\vec{\mathbf{G}}'')|_{\#}.$$

Note that ψ^- and ψ^+ have one less conjunct in $\Delta^{\neq}$. The problem is that ψ^+ is not of the form (7.1) as it may now contain two elements in $\Delta^=$. However it can be seen that the removal of the extra equality in $\Delta^=$ as described in the proof of Proposition 7.2.1 does not introduce any new elements in $\Delta^{\neq}$. While the precise statement of this fact is slightly heavy, there is no deep reasoning in the proof. We now precise it in the following claim:

Claim 7.6.1 *There exists a query ψ^+_{NF} such that: its size depends only on the size of ψ^+, ψ^+_{NF} is in the normal form given by (7.1), it contains an inequality conjunct $h(y) \neq g_1(x_i)$ (where h might possibly be identity) iff ψ^+ also contains such conjunct and $\psi^+_{\text{NF}}(\vec{\mathbf{G}}'') = \psi^+(\vec{\mathbf{G}}'')$. Moreover, ψ^+_{NF} can be constructed in time linear in the size of ψ^+.*

PROOF The proof is a simple case analysis of the content of $\Delta^=$ of ψ.

If its empty, then ψ^+_{NF} is already in the desired form.

If it contains an atom of the form $y = h_2(x_j)$, then equality $g(y) = f(x_i)$ is equivalent to $g(h_2(x_j)) = f(x_i)$ and we are done.

If it contains an atom of the form $h_3(y) = h_2(x_j)$ and g is identity, then $h_3(y) = h_2(x_j)$ is equivalent to $h_3(f(x_i)) = h_2(x_j)$. If g is not identity, then $\tau(y)$ ensures us that either $g(y)$ determines $h_3(y)$ or vice versa. If we have $h_4(g(y)) = h_3(y)$, then $h_3(y) = h_2(x_j)$ is equivalent to $h_4(f(x_i)) = h_2(x_j)$. The other case is symmetric.

The fact that ψ^+_{NF} does not contain any additional inequalities, that it can be computed in time linear in the size of ψ^+ and that $\psi^+_{\text{NF}}(\vec{\mathbf{G}}'') = \psi^+(\vec{\mathbf{G}}'')$ follows from the above construction.

■

We can therefore remove the extra element in Δ^+ and assume that ψ^+ has the desired form. We can now use the inductive hypothesis on the size of $\Delta^{\neq}$ to both ψ^- and ψ^+ in order to compute both $|\psi^-(\vec{\mathbf{G}}'')|_{\#}$ and $|\psi^+(\vec{\mathbf{G}}'')|_{\#}$ and derive $|\psi(\vec{\mathbf{G}}'')|_{\#}$.

It remains to show the base of the inner induction. In the following we assume that $\Delta^{\neq}$ is empty. The rest of the proof is a case analysis on the content of $\Delta^=$. We do each case one by one.

Assume then that $\Delta^=$ consists of an atom of the form $y = f(x_1)$.

Note that the solutions to ψ are of the form $(\bar{a}f(a_1))$. We have:

$$\begin{aligned}
|\psi(\vec{\mathbf{G}}'')|_{\#} &= \sum_{(\bar{u}v)\in\psi(\vec{\mathbf{G}}'')} \left(\#_k(v) \prod_{1\leq i\leq k-1} \#_i(u_i) \right) \\
&= \sum_{(\bar{u}f(u_1))\in\psi(\vec{\mathbf{G}}'')} \left(\#_k(f(u_1)) \prod_{1\leq i\leq k-1} \#_i(u_i) \right) \\
&= \sum_{(\bar{u}f(u_1))\in\psi(\vec{\mathbf{G}}'')} \left(\#_1(u_1)\#_k(f(u_1)) \prod_{2\leq i\leq k-1} \#_i(u_i) \right)
\end{aligned}$$

In linear time we now iterate through all nodes u in $\vec{\mathbf{G}}''$ and set

$$\begin{aligned}
\#'_1(u) &:= \#_1(u) \cdot \#_k(f(u)) \\
\#'_i(u) &:= \#_i(u) && \text{for } 2 \leq i \leq k-1.
\end{aligned}$$

Let $\vartheta(\bar{x})$ be ψ with all occurrences of y replaced with $f(x_1)$. We then have:

$$\begin{aligned}
|\psi(\vec{\mathbf{G}}'')|_{\#} &= \sum_{(\bar{u}f(u_1))\in\psi(\vec{\mathbf{G}}'')} \left(\#'_1(u_1) \prod_{2\leq i\leq k-1} \#'_i(u_i) \right) \\
&= \sum_{\bar{u}\in\vartheta(\vec{\mathbf{G}}'')} \prod_{1\leq i\leq k-1} \#'_i(u_i) \\
&= |\vartheta(\vec{\mathbf{G}}'')|_{\#'}
\end{aligned}$$

By induction on the number of free variables, as $\#'_i(u)$ can be computed in constant time for each i and u, we can compute $|\vartheta(\vec{\mathbf{G}}'')|_{\#'}$ in time linear in $\|\vec{\mathbf{G}}''\|$ and we are done.

Assume now that $\Delta^{=}$ consists of an atom $g(y) = f(x_1)$. Let $\psi'(y)$ be the formula $\exists\bar{x}\psi(\bar{x}y)$ and let $\psi''(\bar{x})$ be the formula $\exists y\psi(\bar{x}y)$. We first compute set $\psi'(\vec{\mathbf{G}}'')$ in linear time using Corollary 7.3.2. We now define a function $\#'_k : V \to \mathbb{N}$ as:

$$\#'_k(u) := \sum_{\substack{\{v\in\psi'(\vec{\mathbf{G}}'')\\ g(v)=u\}}} \#_k(v).$$

Note that this function can be easily computed in linear time by going through all nodes v and adding $\#_k(v)$ to $\#'_k(g(v))$.

Finally we set:

$$\begin{aligned}
\#'_1(u) &:= \#_1(u)\#'_k(f(u)) \\
\#'_i(u) &:= \#_i(u) && \text{for } 2 \leq i \leq k-1.
\end{aligned}$$

Let $u_1, u_2 \in \psi'(\vec{\mathbf{G}}'')$ be such that $g(u_1) = g(u_2)$. Because $\Delta^{\neq}$ is empty, observe that $\vec{\mathbf{G}}'' \models \forall\bar{x}(\psi(\bar{x}u_1) \leftrightarrow \psi(\bar{x}u_2))$. Based on this observation we now group the solutions to ψ according to their last $k-1$ values and get:

$$\begin{aligned}
|\psi(\vec{\mathbf{G}}'')|_{\#} &= \sum_{(\bar{u}v)\in\psi(\vec{\mathbf{G}}'')} \left(\#_k(v) \prod_{1\leq i\leq k-1} \#_i(u_i) \right) \\
&= \sum_{\bar{u}\in\psi''(\vec{\mathbf{G}}'')} \sum_{\substack{\{v\in\psi'(\vec{\mathbf{G}}'') \\ g(v)=f(u_1)\}}} \left(\#_k(v) \prod_{1\leq i\leq k-1} \#_i(u_i) \right) \\
&= \sum_{\bar{u}\in\psi''(\vec{\mathbf{G}}'')} \left(\sum_{\substack{\{v\in\psi'(\vec{\mathbf{G}}'') \\ g(v)=f(u_1)\}}} \#_k(v) \right) \prod_{1\leq i\leq k-1} \#_i(u_i) \\
&= \sum_{\bar{u}\in\psi''(\vec{\mathbf{G}}'')} \left(\#'_k(f(u_1)) \prod_{1\leq i\leq k-1} \#_i(u_i) \right) \\
&= \sum_{\bar{u}\in\psi''(\vec{\mathbf{G}}'')} \left(\#_1(u_1)\#'_k(f(u_1)) \prod_{2\leq i\leq k-1} \#'_i(u_i) \right) \\
&= \sum_{\bar{u}\in\psi''(\vec{\mathbf{G}}'')} \prod_{1\leq i\leq k-1} \#'_i(u_i) \\
&= |\psi''(\vec{\mathbf{G}}'')|_{\#'}
\end{aligned}$$

By induction on the number of free variables, as $\#'_i(u)$ can be computed in constant time for each i and u, we can compute $|\psi''(\vec{\mathbf{G}}'')|_{\#'}$ and we are done with this case.

The remaining case when $\Delta^=$ is empty is handled similarly to the previous one. We then have

$$\psi(\bar{x}y) = \psi_1(\bar{x}) \wedge \tau(y).$$

After setting

$$\begin{aligned}
\#'_1(u) &:= \#_1(u) \cdot \sum_{v\in\tau(\vec{\mathbf{G}}'')} \#_k(v) \\
\#'_i(u) &:= \#_i(u) && \text{for } 2\leq i\leq k-1
\end{aligned}$$

we see that

$$|\psi(\vec{\mathbf{G}}'')|_{\#} = |\psi_1(\vec{\mathbf{G}}'')|_{\#'}$$

and we conclude again by induction on the number of free variables.

This finishes the case analysis of the content of $\Delta^=$ and, as we explained before, concludes the proof of Lemma 7.6.1.

■

As we said earlier, Theorem 7.6.1 is an immediate consequence of Lemma 7.6.1.

■

7.7 Discussions

As we have promised in the Introduction Section 7.1, before getting into conclusions, we first give the reduction of Theorem 7.1.5 to Theorem 6.1.5. As usual, we only state and prove the result over graphs but they generalize to arbitrary relational structures via Lemma 7.2.2.

PROOF [of Theorem 7.1.5]
In [53] yet another definition of the class of graphs with bounded expansion was given:

Theorem 7.7.1 ([53]) *Let $\mathcal{C}$ be a class of graphs. The following conditions are equivalent:*

1. *$\mathcal{C}$ has bounded expansion,*
2. *there exists a function $N : \mathbb{N} \to \mathbb{N}$ such that for every $s \in \mathbb{N}$ and every graph $\mathbf{G} \in \mathcal{C}$ the graph $\mathbf{G}$ may be vertex-colored using $N(s)$ colors so that each of the connected components of the subgraph induced by $i \leq s$ colors has treewidth at most $i-1$.*

It was also shown in [53] that given p the coloring $N(p)$ can actually be computed in time linear in the size of the input graph.

Fix a first-order query $\psi(\bar{x})$ and a class $\mathcal{C}$ of graphs with bounded expansion.

Let $\vec{\mathbf{G}}$ be the input graph.

We use Theorem 7.3.2 to get a simple quantifier-free formula $\phi(\bar{x})$ over a recoloring of $\sigma_{\mathcal{C}}(p)$ and a graph $\vec{\mathbf{G}}' \in \mathcal{C}_p$ such that $\phi(\vec{\mathbf{G}}') = \psi(\vec{\mathbf{G}})$.

Assume now that $|\bar{x}| = k$ and we set $s = k + k \cdot \alpha_{\mathcal{C}}(p)$.

By Fact 3.2.1 class $\mathcal{C}_p$ also has bounded expansion. We know that in time $O(\|\vec{\mathbf{G}}'\|)$ we can compute a coloring of $\vec{\mathbf{G}}'$ with $N(s)$ colors such that each subgraph of this recoloring induced by any s colors has treewidth bounded by $s-1$. (See Point 2 of Theorem 7.7.1 and the discussions below). We do exactly that and for each color C of this coloring we introduce a new unary predicate P_C to the structure describing graph $\vec{\mathbf{G}}'$ (let COL be the set of all these colors) obtaining its recoloring $\vec{\mathbf{G}}''$. It remains to observe that $\phi(\vec{\mathbf{G}}') = \phi(\vec{\mathbf{G}}'')$ (because ϕ does not talk about colors from COL) and that if $\bar{v} \in \phi(\vec{\mathbf{G}}'')$, then each v_i and every of its predecessors has exactly one of the colors from the newly introduced palette COL. Thus:

$$\phi(\vec{\mathbf{G}}'') = \phi'(\vec{\mathbf{G}}''),$$

where $\phi'(\bar{x}) = \bigvee_{C_1,\ldots,C_k,C_1^1,\ldots,C^1_{\alpha_{\mathcal{C}}(p)},\ldots,C_1^k,\ldots,C^k_{\alpha_{\mathcal{C}}(p)} \in \textsc{Col}} \phi(\bar{x}) \wedge \bigwedge_{1\leq i\leq k} \Big(P_{C_i}(x_i) \wedge \bigwedge_{1\leq j\leq \alpha_{\mathcal{C}}(p)} P_{C_j^i}(f_j(x_i))\Big)$.

Moreover, the big disjunction is clearly mutually exclusive from the fact that every node in the graph has exactly one color from COL.

This way (as Theorem 7.1.4 allows us to use Fact 3.1.3), it is enough to handle each disjunct separately. We choose a disjunct, denote it with $\phi''(\bar{x})$ and set ζ to be exactly the set of colors from COL that ϕ'' mentions.

This basically finishes the proof. ϕ'' is simple so it only talks about x_i or their predecessors of the form $f(x_i)$ and can test their unary properties or equalities/inequalities between them. This way, instead of over $\vec{\mathbf{G}}''$, we may evaluate it over $\vec{\mathbf{G}}''_\zeta$ which is the subgraph of $\vec{\mathbf{G}}''$ induced by colors from ζ. From Point 2 of Theorem 7.7.1 we know that $\vec{\mathbf{G}}''_\zeta$ has treewidth bounded by s and as FO queries are in particular MSO queries, we may conclude with Theorem 6.1.5.

■

The above reasoning can of course be mimicked to obtain proofs of Theorems 7.1.2– 7.1.4 without having to go through the more direct reasoning as we presented in Sections 7.4– 7.6. There are a couple of reasons why we did not follow this path.

• The main reason is the lexicographical order on the enumerated tuples. The MSO approach simply does not ensure this property.

• Moreover, we believe that the direct approach to the proofs gives us a better insight into understanding both the implications of the normal form (7.1) and the "augmentational" point of view on the concept of bounded expansion.

• Lastly come the "implementation" difficulties. While the direct approach, as we presented it, seems to be feasible for a "real life" use, the results for MSO rely on some theoretically deep machinery (see for example the Ehrenfeucht-Fraïssé argument of Theorem 6.3.3 and the result of Colcombet, Theorem 6.3.4) which might be quite challenging to handle.

On the other hand, the reduction to the MSO case obviously has its advantages too.

• It also gives us some insight into the notion of bounded expansion. In a sense, it emphasizes the fact that if we consider first-order logic, then the main difficulties are contained in the quantifier elimination procedure. The moment we are left with just quantifier-free queries, we can use the low treewidth representation. While the notion of bounded expansion is rather new, the notion of treewidth and the use of bounded treewidth in the context of obtaining FPT algorithms has been known for quite some time now and has been extensively studied over the past years. As an effect, there is a big variety of tools and results of different sort that one might use in that context. Theorems 6.1.2– 6.1.5 are just a few examples of what class of graphs with bounded treewidth has to offer.

• What we should also admit is that it is not clear on how to use the normal form (7.1) to get the j-th solution in logarithmic time.

It should also be noted that the proof of Theorem 7.1.4 from [56] follows exactly the above approach of reduction to the MSO case of Theorem 6.1.4.

7.8 Conclusions

Queries written in first-order logic can be efficiently processed over the class of structures having bounded expansion. We have seen that over these classes the problems investigated in this thesis can be computed in time linear in the size of the input structure. The constant factor however is not very good. The approach taken here, as well as the ones of [29, 42], yields a constant factor that is a tower of exponentials whose height depends on the size of the query. This nonelementary constant factor is unavoidable already on the class of unranked trees, assuming FPT$\neq$AW$[*]$ [36]. In comparison, this factor can be triply exponential in the size of the query in the bounded degree case, as we have explained in Chapter 5.

It is possible that the results presented here can be generalized to a larger class of structures. In [55] the class of nowhere dense graphs was introduced and it generalizes the notion of bounded expansion. It seems that nowhere dense graphs do enjoy good algorithmic properties. However, we do not know yet whether the model checking problem of first-order logic can be done in linear time over nowhere dense structures. Actually, we do not even know whether the model checking problem is Fixed Parameter Tractable (FPT) over nowhere dense graphs.

The class of nowhere dense structures seems to be the limit for having good algorithmic properties for first-order logic. Indeed, it is known that the model checking problem of first-order logic over a class of structures that is not nowhere dense cannot be FPT [50] (modulo some complexity assumptions and closure of the class under substructures).

For structures of bounded expansion, an interesting open question is whether a sampling of the solutions can be performed in linear time. For instance: can we compute the j-th solution in constant time after a linear preprocessing? This can be done in the bounded degree case (see Theorem 5.1.4) but we do not know if it is also possible in this setting. We leave the bounded expansion case for future research.

8

Discussions

Contents

The results presented in this thesis definitely do not cover all the different aspects of constant delay enumeration. In this chapter we outline a few possible paths that one might follow in order to extend the current knowledge on this topic.

This discussions are divided into two parts:

- Section 8.1 is dedicated to the problem of lower bounds in the context of constant delay enumeration. We already mentioned some results on this in the State of the Art Chapter 4, but this time we look at the problem from a slightly different angle.

- In Section 8.2 we look at the possibilities of extending results from Chapters 5 and 7. We look at both natural directions, that is extending the logic beyond FO and the class of databases beyond bounded expansion.

8.1 Lower bounds

All the lower bounds results that we have presented so far are modulo some complexity assumption. Recall for instance Theorem 4.1.4: for it to work we assume that the matrix multiplication problem cannot be solved in quadratic time. For most of the remaining results saying that certain problem cannot be enumerated with constant delay we made some assumptions on the complexity classes, like for instance that W[1] $\neq$ FPT.

But what all these results have in common is that they actually yield something more. They in fact state that certain problem cannot even be evaluated in LINEAR-EVAL. Then Remark 2.7.1 implies that enumeration is also not possible.

But this rises a very natural question: how does classes CONSTANT-DELAY$_{lin}$ and LINEAR-EVAL relate to each other? We have the obvious inclusion CONSTANT-DELAY$_{lin}$ $\subseteq$ LINEAR-EVAL by Remark 2.7.1, but what about the other direction? Is it the case that CONSTANT-DELAY$_{lin}$ $=$ LINEAR-EVAL?

It turns out that we can answer the last question in a negative way, meaning that we can show a problem that is in LINEAR-EVAL, but not in CONSTANT-DELAY$_{lin}$. Moreover, we can do this without any complexity assumptions. A drawback is that this construction is fully artificial and we do not really learn too much from the example separating the mentioned two classes. No new proof techniques or structural facts about the enumeration as such are revealed. The reason for this is that the example greatly relies on the time hierarchy theorem (we just need the knowledge that there are problems solvable in quadratic time, but not in linear time) and the reasoning then follows a rather natural approach.

We now proceed as follows:

• In Section 8.1.1 we describe the example problem that separates classes LINEAR-EVAL and CONSTANT-DELAY$_{lin}$.

• In Section 8.1.2 we introduce a weakened version of CONSTANT-DELAY$_{lin}$ class called WEAK-CONSTANT-DELAY$_{lin}$ (which still does not catch LINEAR-EVAL due to the same example), which we conjecture is strictly between CONSTANT-DELAY$_{lin}$ and LINEAR-EVAL).

• In Section 8.1.3 we introduce an example that possibly might be used to separate CONSTANT-DELAY$_{lin}$ and WEAK-CONSTANT-DELAY$_{lin}$.

8.1.1 LINEAR-EVAL $\neq$ CONSTANT-DELAY$_{lin}$

We now present an example that separates LINEAR-EVAL from CONSTANT-DELAY$_{lin}$. As we said earlier, the drawback is that this example is rather artificial and does not give us any new insight into constant delay enumeration.

Example 8.1.1 *From the time hierarchy theorem (cf. [44]) we know that there are decision problems that are computable in quadratic time, but not in linear time.*

Let f be a decision problem such that on input I of size n the value $f(I)$ cannot be computed in time $O(n)$ but can be computed in $O(n^2)$.

Consider now the following problem ψ: for the input I of size n we want to compute all the triples $(i, j, f(I))$ such that $1 \leq i, j \leq n$.

Clearly, $\psi(I)$ contains n^2 triples and we can compute it in $O(n^2)$: we first compute in $O(n^2)$ the value of $f(I)$ and then simply list the desired triples.

So the evaluation problem of ψ is in LINEAR-EVAL.

On the other hand it is immediate to see that the enumeration problem of ψ is not in CONSTANT-DELAY$_{lin}$. *In the constant delay enumeration scenario we always output the first solution in linear time. Since each tuple in the solution contains $f(I)$, we cannot compute any single tuple from $\psi(I)$ in $O(n)$.*

As a corollary of the above example and Remark 2.7.1 we get:

Theorem 8.1.1 CONSTANT-DELAY$_{lin}$ $\subsetneq$ LINEAR-EVAL.

8.1.2 WEAK-CONSTANT-DELAY$_{lin}$ class

Recall the definition of CONSTANT-DELAY$_{lin}$ class from Section 2.7.3. We now consider an extension of CONSTANT-DELAY$_{lin}$, which we get by dropping the assumption that a constant delay algorithm may use only a constant amount of write memory during the enumeration phase. To be precise, we define:

We say that the enumeration problem of q is in the class WEAK-CONSTANT-DELAY$_{lin}$ if it can be solved by a RAM algorithm which, on input $\mathbf{D}$, can be decomposed into two steps:

- a *precomputation* phase that is performed in time $O(\|\mathbf{D}\|)$,
- followed by an enumeration phases that outputs $q(\mathbf{D})$ with no repetitions and a constant delay between two consecutive outputs.

A natural question that arises from the above definition is: how does the weakened class relate to the initial one? Of course we have CONSTANT-DELAY$_{lin}$ $\subseteq$ WEAK-CONSTANT-DELAY$_{lin}$, but is this inclusion strict?

One could also consider intermediate classes WEAK-n-CONSTANT-DELAY$_{lin}$, which for input $\mathbf{D}$ limit the write memory used by the enumeration phase to $O(\|\mathbf{D}\|^n)$ or perhaps limit it to $O(\log(\|\mathbf{D}\|)^n)$, etc. Of course more memory always results in at least as general class as the one with stronger restrictions, but would these classes form a strict hierarchy?

Unfortunately, we do not know the answers to the above questions. In fact, we do not even know whether WEAK-CONSTANT-DELAY$_{lin}$ = CONSTANT-DELAY$_{lin}$ or not. We conjecture that the two classes are different, but for now the question remains open:

Open Problem 8.1.1 *Decide whether* WEAK-CONSTANT-DELAY$_{lin}$ = CONSTANT-DELAY$_{lin}$.

8.1.3 On separation of WEAK-CONSTANT-DELAY$_{lin}$ and CONSTANT-DELAY$_{lin}$

In this section we present an example that we believe could be used to separate CONSTANT-DELAY$_{lin}$ from WEAK-CONSTANT-DELAY$_{lin}$. We use it to illustrate a technique of "hiding a computation" that is available in the WEAK-CONSTANT-DELAY$_{lin}$ scenario, but is a lot more limited inside CONSTANT-DELAY$_{lin}$. We find this technique quite promising with respect to possibly answering the question from Open Problem 8.1.1 negatively. We now turn to the example.

Black-white trees

Throughout this section the input database is going to be a full binary tree $\mathbf{T}$ with nodes having either white or black colors. We call such trees the *black-white trees*. Given a node $v \in \mathbf{T}$ we write $W(v)$ if v has white color and $B(v)$ otherwise. We write $\mathrm{col}(v)$ to denote the color of v. A *level* of a tree is the set of nodes that are at the same distance from the root. As usual, root is at level 0, its children are at level 1, children of children of the root are at level 2 and so on. Given a node $v \in \mathbf{T}$ we write $\mathrm{lvl}(v)$ to denote its level. It is easy to see that a full binary tree with n levels (level 0, level 1, ..., level $n-1$) has exactly $2^n - 1$ nodes and that there are 2^i nodes at level i. Given a node $v \in \mathbf{T}$ we write $\mathbf{T}_{\downarrow v}$ to denote the subtree of $\mathbf{T}$ rooted at v.

The $\#$ function

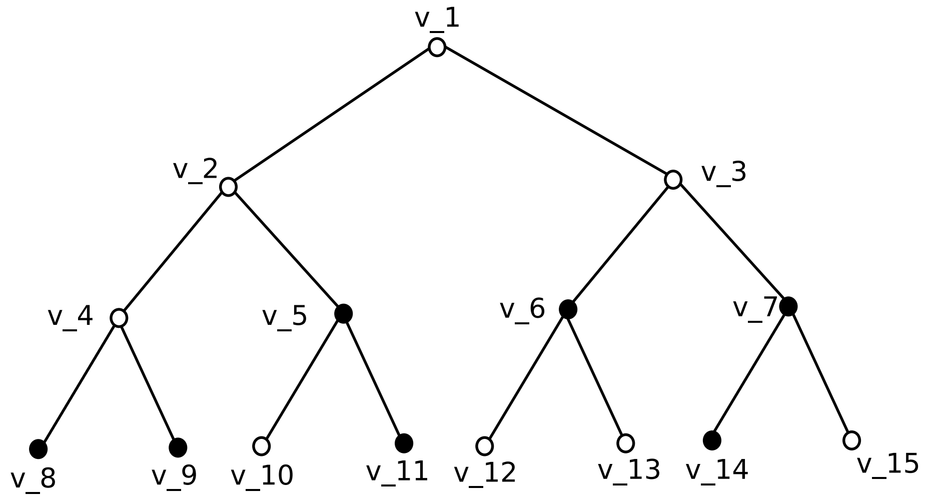
We are going to assign a natural value to each node of the input tree $\mathbf{T}$. Assume $\mathbf{T}$ has n levels. We inductively define a function $\# : \mathbf{T} \to \mathbb{N}$ starting from leaves and moving up. Let $v \in \mathbf{T}$. Then:

- if $\mathrm{lvl}(v) = n - 1$, then $\#(v) = 0$;
- else $\#(v) = |\{(u, w) \in \mathbf{T}_{\downarrow v} : \mathrm{col}(u) = \mathrm{col}(w)$ and $\#(u) = \#(w)$ and $\mathrm{lvl}(u) = \mathrm{lvl}(w)$ and u is to the left of $w\}|$.

Given node v, we call set $\{(u, w) \in \mathbf{T}_{\downarrow v} : \mathrm{col}(u) = \mathrm{col}(w)$ and $\#(u) = \#(w)$ and $\mathrm{lvl}(u) = \mathrm{lvl}(w)$ and u is to the left of $w\}$ the *accounting set for* v. When we say *a value of node* v, we mean the value of $\#(v)$.

Let us consider the following example to have a better intuition about the $\#$ function.

Example 8.1.2 *Consider the following input tree* $\boldsymbol{T}$*:*



From the definition of $\#$ function we have $\#(v_i) = 0$ for $i = 8, 9, \dots, 15$.

For nodes on level 2, we have $\#(v_4) = \#(v_6) = 1$, since leaves in their subtrees share a color (and all leaves have value 0). It is not the case for v_5 and v_7, so $\#(v_5) = \#(v_7) = 0$.

Let us now do the analysis for v_2.

There is just one white leaf in the subtree of v_2 and three black ones. Those black form in total three pairs of nodes that are on the same level and that have the same values of $\#$. v_4 and v_5 have different colors, so the pair (v_4, v_5) does not belong to the accounting set for v_2. In total we get $\#(v_2) = 3$.

For v_3 we have three pairs from a similar analysis of leaves (only the colors are inverted in this case). This time v_6 and v_7 have the same color, but their $\#$ values differ, so the pair (v_6, v_7) does not belong to the accounting set for v_3. Altogether we get $\#(v_3) = 3$.

Finally we compute $\#(v_1)$. It has in its subtree:

• *4 black leaves that result in 6 pairs in the accounting set for v_1. Similarly it has 4 white leaves implying 6 more pairs.*

• *from level 2 only a single pair of nodes (v_5, v_7) that share both their color (black in this case) and their value (which is 0 for both) is in the accounting set for v_1.*

• *pair (v_2, v_3) from level 1 of white nodes with value 3 also is in the accounting set for v_1.*

In total we get $\#(v_1) = 14$.

The $\phi_{\mathbf{BW}}$ query

We are interested in enumerating solutions to the following query:

Definition 8.1.1 *Query $\phi_{BW}(x, y)$ takes as input a full binary black-white tree $\boldsymbol{T}$ and returns the accounting set for the root of $\boldsymbol{T}$.*

Simple facts

We now present a series of basic, but useful facts that are going to be used for designing an algorithm that, given a black-white tree $\mathbf{T}$, enumerates $\phi_{\mathrm{BW}}(\mathbf{T})$.

Fact 8.1.1 *Let $\boldsymbol{T}$ be a full binary tree. There are in total $O(|\boldsymbol{T}|^2)$ different pairs of nodes such that both nodes from each pair share the same level.*

Proof Assume $\mathbf{T}$ has n levels, that is $\mathbf{T}$ has $2^n - 1$ nodes. More precisely, there is 1 node at level 0, 2 nodes at level 1, 4 nodes at level 2, . . . , 2^{n-1} nodes at level $n - 1$.

There are altogether $(2^{n-1})^2$ pairs of nodes at level $n - 1$, $(2^{n-2})^2 = \frac{1}{4}(2^{n-1})^2$ pairs of nodes at level $n - 2$, . . . , $(2^{n-i})^2 = \frac{1}{4^{i-}}(2^{n-1})^2$ pairs of nodes at level $n - i$, which summed up is bounded by $(2^n)^2 = |\mathbf{T}|^2$.

■

As a corollary of Fact 8.1.1 we get:

Corollary 8.1.1 *For any black-white tree $\boldsymbol{T}$ we have $|\phi_{BW}(x, y)| \leq |\boldsymbol{T}|^2$.*

Proof Recall from the definition of $\#$ function that an accounting set for any node (in particular for the root) contains only pairs of nodes at the same level.

■

Fact 8.1.2 *Let $\boldsymbol{T}$ by a black-white tree and let $u \neq w \in \boldsymbol{T}$ be two of its nodes such that $col(u) = col(w)$, $\#(u) = \#(w)$ and $lvl(u) = lvl(w)$ and u is to the left of w. Then the pair (u, w) is exactly in the accounting sets of the following nodes:*

- *node v which is the least common ancestor (lca) of u and w,*
- *any node v' that is an ancestor of the lca of u and w.*

Proof This is a direct consequence of the definition of the accounting set. Assumptions $col(u) = col(w)$, $\#(u) = \#(w)$ and $\mathrm{lvl}(u) = \mathrm{lvl}(w)$ and u is to the left of w are exactly as present in the definition of $\#$ function. The only missing part is that for v to have (u, w) in its accounting set it is necessary that $u, w \in \mathbf{T}_{\downarrow v}$ and all the nodes having that property are exactly the $\mathrm{lca}(u, w)$ and its ancestors.

■

Fact 8.1.3 *Let $\boldsymbol{T}$ by a full binary tree. We can store $\boldsymbol{T}$ on* RAM *machine in such a way that, given $u, v \in \boldsymbol{T}$ with $lvl(u) = lvl(v)$ we can compute $lca(u, v)$ in constant time.*

PROOF This fact is a special case of the general result working for arbitrary binary trees (not necessarily full ones) proved in [14] (see also [18] for the use of the result from [14]).

But in this case we may also follow a more direct approach.

We can use the well know technique of storing a full binary tree in an array. The root is in the first cell, followed by its left and right child and, in general, followed by lists of nodes (starting from the left-most one and going right) from consecutive levels. An example of this numbering can be seen on the tree from Example 8.1.2.

This way, given node v at position i, we have for example:

• children of v are at positions $2i$ (left child) and $2i+1$ (right child),

• parent of v is at position i div 2.

One can verify that given nodes u and v at positions respectively i and $(i+k)$ and such that $\mathrm{lvl}(u) = \mathrm{lvl}(v)$, there exists an equation that returns a position of $\mathrm{lca}(u, v)$ and depends only on i and k.

■

Fact 8.1.4 *Let* $\boldsymbol{T}$ *by a black-white tree. Then* $|\phi_{BW}(\boldsymbol{T})| \geq \frac{|T|^2}{32}$.

PROOF Note that if two leaves of $\mathbf{T}$ share a color, then they are included in the accounting set for the root of $\mathbf{T}$. Since there are $\frac{|\mathbf{T}|}{2}$ leaves, at least half of them share a color and the result follows.

■

Computation of the # function

Relying on Facts 8.1.1, 8.1.2 and 8.1.3 we get the following algorithm for computing # function:

Lemma 8.1.1 *There is an algorithm that, given a black-white tree* $\boldsymbol{T}$ *as input, computes its # function in time* $O(|\boldsymbol{T}|^2)$.

PROOF Fix input black-white tree $\mathbf{T}$ with n levels. The algorithm works as follows:

Set $\#(v) := 0$ for all nodes (note that this implies that for all the leaves the value of # is already correct). Set $l = n-1$ (l represents the currently processed level). Until l is 0 perform the following loop (the invariant of this procedure is that at each point of time the # function has already been computed for all nodes at levels $\geq l$):

1. For every pair of nodes (u, v) from level l, if $\mathrm{col}(u) = \mathrm{col}(v)$ and $\#(u) = \#(v)$ and u is to the left of v, set $\#(\mathrm{lca}(u, v)) := \#(\mathrm{lca}(u, v)) + 1$.

2. For every node u at level l set $\#(\mathrm{parent}(u)) := \#(\mathrm{parent}(u)) + \#(u)$.

3. Set $l := l - 1$.

It is now easy to see that at the first step of the loop each pair (u, v) with $\mathrm{col}(u) = \mathrm{col}(v)$ and $\#(u) = \#(v)$ in a sense "informs" its least common ancestor about the fact that it actually exists. At step two this information is propagated upwards, one level at a time (so that it would be accounted by exactly those nodes as given by Fact 8.1.2).

It follows from Fact 8.1.1 that the running time of the above algorithm is $O(|\mathbf{T}|^2)$.

■

The $\#$-level-split structure

Definition 8.1.2 *Let $\boldsymbol{T}$ be a black-white tree. A $\#$-*level-split structure for $\mathbf{T}$ *is a structure that for each level l contains lists $W_1, \ldots, W_{w_l}$ and $B_1, \ldots, B_{b_l}$ such that each node from $\boldsymbol{T}$ at level l appears on exactly one of those lists and:*

- *each list W_i contains only white nodes and similarly each list B_i contains only black nodes,*
- *for every list W_i and every two elements u, v on that list it is the case that $\#(u) = \#(v)$ and lists B_i have a similar property.*

*We call each list W_i and B_i the $\#$-*grouping list.

As a corollary of Lemma 8.1.1 we get:

Lemma 8.1.2 *There is an algorithm that, given a black-white tree $\boldsymbol{T}$ as input, computes $\#$-level-split structure for $\boldsymbol{T}$ in time $O(|\boldsymbol{T}|^2)$.*

PROOF The first step is the computation of $\#$ function using Lemma 8.1.1. Then it remains to sort lists of nodes at each level and split them among desired lists in a natural way (splitting by both the colors of nodes and their values).

■

Evaluation of $\phi_{\mathbf{BW}}$

Theorem 8.1.2 *There is an algorithm that takes as input a black-white tree $\boldsymbol{T}$ and computes in time $O(|\boldsymbol{T}| + |\phi_{BW}(\boldsymbol{T})|)$ set $\phi_{BW}(\boldsymbol{T})$. In other words, the evaluation problem of ϕ_{BW} over the class of black-white trees is in* LINEAR-EVAL.

PROOF By Fact 8.1.4 we have $|\phi_{\mathrm{BW}}(\mathbf{T})| = O(|\mathbf{T}|^2)$.

The algorithm first computes $\#$-level-split structure for $\mathbf{T}$ in time $O(|\mathbf{T}|^2)$ using Lemma 8.1.2. Then, for each level l and each $\#$-grouping list from that level it outputs pairs of nodes from the currently considered $\#$-grouping list in such a way, that the first component of each output pair is to the left from the second component of that pair.

It is immediate to see that the above algorithm is correct and that it works in the desired time constraints.

■

Weak enumeration of $\phi_{\mathbf{BW}}$

As a corollary of Theorem 8.1.2 and Fact 8.1.4 we get:

Theorem 8.1.3 *The problem of enumerating solutions to query $\phi_{BW}(x, y)$ over the class of black-white trees is in* WEAK-CONSTANT-DELAY$_{lin}$.

PROOF The reason for which this theorem is a corollary of Theorem 8.1.2 is the fact that we have “easy access” to “many” solutions.

From Fact 8.1.4 and Corollary 8.1.1 we know that $O(|\phi_{\mathrm{BW}}(\mathbf{T})|) = O(|\mathbf{T}|^2)$. From the proof of Fact 8.1.4 we know that there are at least $\frac{|\mathbf{T}|^2}{32}$ solutions to ϕ_{BW} on the leaves level. Let us now assume that K is a constant such that Lemma 8.1.2 computes the $\#$-level-split structure for $\mathbf{T}$ in time $K|\mathbf{T}|^2$. We now construct an algorithm from WEAK-CONSTANT-DELAY$_{lin}$ enumerating ϕ_{BW}.

The preprocessing phase creates two lists: list of white leaves of $\mathbf{T}$ and list of black leaves of $\mathbf{T}$. This is clearly done in $O(|\mathbf{T}|)$.

We now turn to the enumeration phase.

It starts with enumerating all the pairs of leaves that share a color. This is easily doable with constant delay and we know from Fact 8.1.4 that there are at least $\frac{|\mathbf{T}|^2}{32}$ such pairs. Before outputting each single pair, the algorithm also performs $32K$ (i.e. a constant number) consecutive steps of the algorithm from Lemma 8.1.2 that computes the $\#$-level-split structure for $\mathbf{T}$. For simplicity, we slightly modify that algorithm, so that it stores only those $\#$-grouping lists that have at least 2 elements.

Then, for each level starting from the one just above leaves and moving up, for each $\#$-grouping list from that level, the algorithm outputs all pairs of distinct nodes from the currently processed $\#$-grouping list.

The constant delay assumption is clearly satisfied and it follows from previous discussions that this way we get the whole set $\phi_{\mathrm{BW}}(\mathbf{T})$ as desired.

■

Conclusions

In the proof of Theorem 8.1.3 we have shown a technique of "hiding a computation" inside the enumeration phase of an algorithm from WEAK-CONSTANT-DELAY$_{lin}$.

The idea was as follows: if we can easily access set of n solutions to the problem, then while outputting them we can perform any additional computations with $O(n)$ steps in total. In our case these additional computations allowed us to construct the $\#$-level-split structure for the input black-white tree $\mathbf{T}$.

We could of course do a similar reasoning for an algorithm from CONSTANT-DELAY$_{lin}$ class, but those algorithms can only use a constant amount of additional write memory during the enumeration process, which seems to be dramatically limiting the powers of the "hidden computation".

We leave the problem of whether ϕ_{BW} can be enumerated with an algorithm from CONSTANT-DELAY$_{lin}$ open. We conjecture that it cannot.

Open Problem 8.1.2 *Decide whether the enumeration problem of ϕ_{BW} over the class of black-white trees is in* CONSTANT-DELAY$_{lin}$.

What should be mentioned is that the idea behind the "hiding of computation" technique was to split the set of solutions into the "easy" and "difficult" ones. Having a lot of easy solutions would allow to perform some additional computation.

In view of the above, the example separating CONSTANT-DELAY$_{lin}$ (and also WEAK-CONSTANT-DELAY$_{lin}$) from LINEAR-EVAL relied on the fact that all the solutions are "difficult" and that we are unable to get even the first one within the required time constraints.

8.2 Bounded degree, bounded expansion and beyond

8.2.1 Stronger logic over bounded degree

In this section we argue that any query language, that admits constant delay enumeration over the class of databases of bounded degree, cannot be much more expressive than the first-order logic.

To see this, consider the following reduction:

Example 8.2.1 *Let* **G** *be an arbitrary graph. We now show how to construct a graph* ***H*** *of degree* 4 *that encodes* **G**. *For simplicity of the presentation we assume that all the nodes of* **G** *share the same color. This is done without loss of generality. Our construction is as follows.*

H *is going to have three-colored nodes: blue* (B) edge nodes*, red* (R) cycle nodes *and black* (C) real nodes. ***H*** *is constructed from* **G** *in the following way:*

- *in the middle of each edge of* **G** *a new blue edge node is added;*
- *each node* v *of* **G** *of degree* n *is replaced with a cycle of* n *red cycle nodes in such a way that each former neighbor of* v *(recall that they are all blue edge nodes at this point) is now connected with exactly one red cycle node from the added cycle (in an arbitrary way). Moreover, a black real node* v *is added to* ***H*** *and connected with exactly one (arbitrarily chosen) node from the* n*-cycle that initially replaced* v*.*

In other words, what this construction does is:
• *We start with a node* v *of degree* n*.*
• *We color it black and then surround it with a cycle of* n *red nodes and give a unique access from* v *to that cycle. If we had an edge* (u, v) *in* **G***, it is replaced with an edge* (r_v^i, r_u^j)*, where* r_v^i *and* r_u^j *are some arbitrarily chosen nodes from red cycles surrounding* v *and* u *respectively. The "arbitrarily chosen" is limited to only those choices, for which each red node surrounding* v *has exactly one neighbor among red cycles around nodes different from* v*.*
• *To distinguish between old edges from graph* **G** *and the new ones that emerged from this construction, we put blue nodes in the middle of each initial edge. This way the red cycles are connected with other red cycles via paths leading through single blue nodes.*
The way we traverse graph ***H*** *is: starting from a black real node* v *we have a unique access to the surrounding ring of red nodes. We may then freely cycle around it until we finally decide to go, via a blue node, to another red cycle and then we can reach its inner black real node* u *or jump to another red cycle via a blue node and so on.*

It is easy to verify that there is an edge between u *and* v *in* **G** *iff* u *and* v *are black in* ***H*** *and there is a path between* u *and* v *such that the labels on that path match expression* R^*BR^**.*

Moreover, notice that the size of ***H*** *is polynomial in the size of* **G** *(and that it can be constructed in polynomial time).*

The above example justifies:

Remark 8.2.1 *If a query language* $\mathcal{L}$ *has the power of* FO *and can express a property that two nodes are linked by a path with the word formed from their colors matching expression* R^*BR^**, then the model checking problem of* $\mathcal{L}$ *over the class of graphs with bounded degree is not in* FPT *(assuming* AW$[*] \neq$ FPT*).*
In particular, both the evaluation and enumeration problem of $\mathcal{L}$ *over the class of graphs of bounded degree are not in* LINEAR-EVAL *and* CONSTANT-DELAY$_{lin}$ *respectively.*

As an immediate corollary we get the following result for first-order logic with transitive closure (FO + TC):

Theorem 8.2.1 *The enumeration problem of* FO + TC *over the class of graphs of bounded degree is not in* CONSTANT-DELAY$_{lin}$ *(assuming* AW$[*] \neq$ FPT*).*

We are not aware whether some intermediate logic between FO and FO + TC can be enumerated over the class of graphs with bounded degree. A possible candidate could be $\mathrm{FO}^* = \mathrm{FO}(E, E^*)$, where E^* is the transitive closure of the edge relation E. Although in FO^*, just as it is the case for FO + TC, one can test whether two nodes are connected, but as an opposite to FO + TC, logic FO^* does not provide any tools to restrict the colors of nodes on paths of arbitrary lengths. Hence Remark 8.2.1 does not apply to FO^*. We believe that even this weak version of transitive closure is already too powerful to admit CONSTANT-DELAY$_{lin}$ enumeration, but we leave this question open for now.

Open Problem 8.2.1 *Decide whether the enumeration of* FO^* *queries over the class of graphs of bounded degree is in* CONSTANT-DELAY$_{lin}$.

8.2.2 Bounded expansion

A conclusion from the previous section is that even if there is an extension of first-order logic such that it would admit constant delay enumeration over the class of graphs of bounded degree, then this extension cannot be too powerful (for instance FO + TC is out of the equation as seen by Theorem 8.2.1).

On the other hand, what we already know is that going beyond bounded degree, while sticking to first-order logic, results in some viable constant delay enumeration algorithms. In fact we can go even as far as to the classes of structures with bounded expansion, as seen in Theorem 7.1.2. A natural question arises: can we go even beyond bounded expansion?

Nowhere dense

In [55] a following generalization of the class of graphs with bounded expansion was introduced.

Recall from Section 2.8.2 that $\mathbf{G}\nabla r$ is the set of all r-minors of the graph $\mathbf{G}$. Given a class $\mathcal{C}$ of graphs, denote with $\mathcal{C}\nabla_r$ the set of all r-minors of all graphs from $\mathcal{C}$. In the context of a class of graphs with bounded expansion we were interested in having a global bound on the greatest reduced average density (grad) of each graph from the class. Recall that the mentioned grad of $\mathbf{G}$ with rank r is defined as:

$$\nabla_r(\mathbf{G}) = \max_{\mathbf{H}\in\mathbf{G}\nabla r} \frac{|\mathbf{H}|_{\mathrm{EDGE}}}{|\mathbf{H}|_{\mathrm{VERT}}}.$$

This time we are going to considere a "flattened" version of the right side, where instead of looking at the number of edges and vertices, we are going to look at their log values. We associate to $\mathcal{C}$ the following number:

$$\lim_{r\to\infty} \limsup_{\mathbf{H}\in\mathcal{C}\nabla_r} \frac{\log(|\mathbf{H}|_{\mathrm{EDGE}})}{\log(|\mathbf{H}|_{\mathrm{VERT}})} \tag{8.1}$$

For any graph $\mathbf{H}$ we have $|\mathbf{H}|_{\mathrm{EDGE}} \leq |\mathbf{H}|^2_{\mathrm{VERT}}$, so the number defined in (8.1) is bounded by 2. It turns out that for any class $\mathcal{C}$ this number can take exactly three possible values: $\{0, 1, 2\}$ [55]. Since finite classes of graphs are in our case trivial, in the sequel we implicitly assume that $\mathcal{C}$ is infinite.

We follow [55] and define as *nowhere dense* those classes of graphs, for which the associated number is 0 or 1. It is rather easy to see that this definition extends the notion of bounded expansion:

Fact 8.2.1 *Let $\mathcal{C}$ be a class of graph with bounded expansion. Then $\mathcal{C}$ is nowhere dense.*

PROOF Let $\mathcal{C}$ be a class of graphs with bounded expansion and let $f(r)$ be the function bounding $\nabla_r(\mathbf{G})$ for all $\mathbf{G} \in \mathcal{C}$. This yields that for every graph $\mathbf{H} \in \mathcal{C}\nabla_r$ we have $|\mathbf{H}|_{\mathrm{EDGE}} \leq f(r)|\mathbf{H}|_{\mathrm{VERT}}$ and hence:

$$\frac{\log(|\mathbf{H}|_{\mathrm{EDGE}})}{\log(|\mathbf{H}|_{\mathrm{VERT}})} \leq 1 + \frac{\log(f(r))}{\log(|\mathbf{H}|_{\mathrm{VERT}})}$$

Since $f(r)$ is a constant and $|\mathbf{H}|_{\text{VERT}}$ can be arbitrarily large, we can infer that:

$$\limsup_{\mathbf{H}\in\mathcal{C}\nabla_r} \frac{\log(|\mathbf{H}|_{\text{EDGE}})}{\log(|\mathbf{H}|_{\text{VERT}})} \le 1$$

which in turn concludes the result.

■

It turns out that nowhere dense is a proper extension of bounded expansion, meaning that there is a class $\mathcal{C}$ that is nowhere dense, but it does not have bounded expansion. As for such classes, we can list for example classes of graphs that locally exclude a minor or ones that have local bounded treewidth (see [25] and [35] for the appropriate definitions). For all the possible details, as well as the current state of the art concerning the notion of nowhere dense, the reader is referred to [56].

We know that classes of graph with bounded expansion are nowhere dense. It is thus natural to ask whether the results of Theorems 7.1.1– 7.1.5 can be extended to the nowhere dense case. It turns out that already a linear time model checking solution for FO is not known (which is a mandatory intermediate step for possible extension of Theorems 7.1.2– 7.1.5). In fact, there is even no known FPT solution to the model checking problem.

Open Problem 8.2.2 *Decide whether the model checking problem of* FO *over classes of nowhere dense graphs is in* FPT.

A direct application of the techniques developed for the bounded expansion case gives a solution to the model checking problem that works in time $O(n^{1+\epsilon})$ (where n is the size of the input graph and ϵ is an arbitrary positive value), but only for the *existential fragment of first-order logic* (where all formulas are of the form $\phi(\bar{x}) = \exists\bar{y}\psi(\bar{x}\bar{y})$ for some quantifier-free ψ) [56].

On the other hand, if a class of graphs is not nowhere dense (we then say it is *somewhere dense*), then there is no FPT algorithm solving FO model checking problem for that class (under an assumption that W$[1] \neq$ FPT).

Theorem 8.2.2 ([30]) *Let $\mathcal{C}$ be an infinite class of somewhere dense graphs that is closed under subgraphs. Then the model checking of existential fragment of* FO *is* W*[1]-hard.*

If we assume the *effectiveness* of class $\mathcal{C}$ (see [50] for details on this definition), we can even get a stronger result:

Theorem 8.2.3 ([50]) *Let $\mathcal{C}$ be an infinite class of effectively somewhere dense graphs that is closed under subgraphs. Then the model checking of* FO *is* AW*[*]-complete.*

In view of the above and the fact that it is strongly believed that FPT $\subsetneq$ W$[1]$ $\subsetneq$ AW$[*]$ (see Section 2.3 for details), it is most likely that nowhere dense is the maximal class of graphs where we can hope for FPT model checking solution and CONSTANT-DELAY$_{lin}$ enumeration. As we said earlier, we leave it as an open problem.

8.2.3 Other properties of bounded expansion

We hope that the reader is convinced that the notion of classes of graphs with bounded expansion is a very robust one. Those classes admit very good algorithmic properties for FO query evaluation (cf. Theorems 7.1.1– 7.1.5), but also the notion itself can be characterized in many different ways (see [53] for details), making it very flexible and easy to use. Depending on the context, one can switch between these characterizations, as we for example saw in the proof of Theorem 3.3.3.

But we think that it is good to be aware of one more argument that emphasizes the importance of the notion of bounded expansion: the graph isomorphism problem. The graph isomorphism test takes as input two graphs and decides whether they are isomorphic or not. This is one of the very few problems that is in NP but is neither not known to be in P nor known to be NP-complete. Current state of the art gives a polynomial time algorithm solving the graph isomorphism problem for classes of graphs excluding at least one minor (which is a very deep result of Martin Grohe, see [41] for details). Without any restrictions, the best known algorithm can test isomorphism of two arbitrary graphs with n vertices in time $2^{O(\sqrt{n \log n})}$ (cf. [7]).

We now argue that if there exists a polynomial time procedure that solves the graph isomorphism problem for classes of graphs with bounded expansion, then this procedure in fact solves the graph isomorphism problem in full generality.

Let $\mathcal{C}$ be a class of all graphs and let $\mathbf{G} \in \mathcal{C}$ be a graph with n nodes. Consider the following transformation of $\mathbf{G}$:

Graph $\mathbf{G}_{BE}$ is obtained from $\mathbf{G}$ by adding n new nodes along each of the edges of $\mathbf{G}$ (more formally, each edge (u, v) from $\mathbf{G}$ is replaced in $\mathbf{G}_{BE}$ with a path $(u, w_1^{u,v}, \dots, w_n^{u,v}, v)$, where $w_i^{u,v}$ are newly introduced nodes). When talking about $\mathbf{G}_{BE}$, we call nodes u, v *the real nodes* and nodes $w_i^{u,v}$ *the edge nodes*.

The following fact is an immediate consequence of the above construction:

Fact 8.2.2 *For any two graphs $\mathbf{G}, \mathbf{G}' \in \mathcal{C}$, they are isomorphic if and only if $\mathbf{G}_{BE}$ and $\mathbf{G}'_{BE}$ are isomorphic.*

PROOF [Sketch] Assume $\mathbf{G}$ and $\mathbf{G}'$ both have n vertices.

It is immediate to extend an isomorphism $i(\mathbf{G}) = \mathbf{G}'$ into an isomorphism $i'(\mathbf{G}_{BE}) = \mathbf{G}'_{BE}$: set i' to agree with i on real nodes and for each edge $(u, v) \in \mathbf{G}$ and each $1 \leq j \leq n$ it is enough to set $i'(w_j^{u,v}) = w_j^{i(u),i(v)}$.

The other direction requires slightly more reasoning. Let $i'(\mathbf{G}_{BE}) = \mathbf{G}'_{BE}$ be an isomorphism of graphs.

If it happens to be the case that i' maps all real nodes of $\mathbf{G}_{BE}$ into reals nodes of $\mathbf{G}'_{BE}$, then it is enough to set i to be i' projected to the set of real nodes. If the above does not hold (that is some real node v is mapped by i' into an edge node), then one can verify that v belongs to a component of $\mathbf{G}_{BE}$ that is a simple cycle and similarly $i'(v)$ belongs to simple cycle of the same length inside $\mathbf{G}'_{BE}$. But these two cycles originate from simple cycles of equal lengths inside $\mathbf{G}$ and $\mathbf{G}'$ respectively and they can safely be mapped to each other in an obvious way.

■

Let $\mathcal{C}_{BE}$ be the class of all graphs $\mathbf{G}_{BE}$ such that $\mathbf{G} \in \mathcal{C}$. We now present the key property of the class $\mathcal{C}_{BE}$.

Lemma 8.2.1 *Class $\mathcal{C}_{BE}$ has bounded expansion.*

PROOF We claim that $\mathcal{C}_{BE}$ has bounded expansion as witnessed by function $f(r) = (4r+1)^3 + 2$. Indeed:

Fix $\mathbf{G}_{BE} \in \mathcal{C}_{BE}$. We show that $\nabla_r(\mathbf{G}_{BE}) \leq f(r)$. This goes by the case analysis on the graph $\mathbf{G}$ from which $\mathbf{G}_{BE}$ originated. There are two cases now:

• if $\mathbf{G}$ has at least $4r+1$ nodes, then each edge from $\mathbf{G}$ was replaced with a path of length $4r+1$ when switching to $\mathbf{G}_{BE}$. This yields that for any r-minor $\mathbf{H}$ of $\mathbf{G}_{BE}$ there are no adjacent ball nodes that both contain real nodes from $\mathbf{G}$ (real nodes of $\mathbf{G}_{BE}$ are at distance of at least $4r+1$ from each other and by contracting balls of diameters bounded by $2r$ into single ball nodes, one needs at least three ball nodes on each path of length $4r+1$). This on the other hand implies that at least one of the endpoints of each edge in $\mathbf{H}$ is a ball node containing only edge nodes of $\mathbf{G}$ and so this endpoint has degree bounded by 2 in $\mathbf{H}$. Having this, we see that $\frac{|\mathbf{H}|_{\text{EDGE}}}{|\mathbf{H}|_{\text{VERT}}} \leq 2$ and since $\mathbf{H}$ was an arbitrary r-minor of $\mathbf{G}_{BE}$, we also have $\nabla_r(\mathbf{G}_{BE}) \leq 2 \leq f(r)$.

• if $\mathbf{G}$ has up to $4r$ nodes, then it has up to $(4r)^2$ edges. When moving to $\mathbf{G}_{BE}$, each of this edges is replaced with $4r+1$ new edges and $4r$ new nodes. Altogether $\mathbf{G}_{BE}$ has up to $(4r+1)^3$ edges, which is also the case for any r-minor $\mathbf{H}$ of $\mathbf{G}_{BE}$. A very coarse approximation shows that $\frac{|\mathbf{H}|_{\text{EDGE}}}{|\mathbf{H}|_{\text{VERT}}} \leq (4r+1)^3$ and since $\mathbf{H}$ was an arbitrary r-minor of $\mathbf{G}_{BE}$, we also have $\nabla_r(\mathbf{G}_{BE}) \leq (4r+1)^3 \leq f(r)$.

The above case analysis shows that it is in fact the case that $\nabla_r(\mathbf{G}_{BE}) \leq f(r)$ for any graph $\mathbf{G}_{BE} \in \mathcal{C}_{BE}$, which is exactly the requirement for $\mathcal{C}_{BE}$ to have bounded expansion.

■

As a consequence of Lemma 8.2.1 and Fact 8.2.2 we get:

Lemma 8.2.2 *If there is a polynomial time solution to the graph isomorphism problem over classes of graphs with bounded expansion, then there is a polynomial time solution to the general graph isomorphism problem.*

PROOF Fix graphs $\mathbf{G}, \mathbf{G}' \in \mathcal{C}$.

Since the transformation from $\mathbf{G}$ to $\mathbf{G}_{BE}$ (and similarly from $\mathbf{G}'$ to $\mathbf{G}'_{BE}$) is of polynomial size and a composition of polynomials is still a polynomial, then by Fact 8.2.2 a polynomial time isomorphism test for $\mathbf{G}_{BE}$ and $\mathbf{G}'_{BE}$ yields a polynomial time isomorphism test for $\mathbf{G}$ and $\mathbf{G}'$.

Both $\mathbf{G}_{BE}$ and $\mathbf{G}'_{BE}$ are from a class $\mathcal{C}_{BE}$ which by Lemma 8.2.1 has a bounded expansion. Thus a polynomial time isomorphism test for classes of graphs with bounded expansion would in particular imply such a test for $\mathcal{C}_{BE}$ and, as we explained above, it would also imply existence of a general isomorphism test working in polynomial time.

■

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
See pages: 15

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
See pages: 14

[3] Noga Alon, Raphael Yuster, and Uri Zwick. Color-Coding. *J. ACM*, 42(4):844–856, 1995.
See pages: 91

[4] Mustapha Arfi. Polynomial Operations on Rational Languages. In *Symp. on Theoretical Aspects of Computer Science (STACS)*, pages 198–206, 1987.
See pages: 74

[5] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy Problems for Tree-Decomposable Graphs. *J. of Algorithms*, 12(2):308–340, 1991.
See pages: 45, 68, 69, 70

[6] David Avis and Komei Fukuda. Reverse Search for Enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.
See pages: 51

[7] László Babai and Eugene M. Luks. Canonical labeling of graphs. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *STOC*, pages 171–183. ACM, 1983.
See pages: 124

[8] Guillaume Bagan. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *Conf. on Computer Science Logic (CSL)*, pages 167–181, 2006.
See pages: 44, 45, 46, 68, 69, 88

[9] Guillaume Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques*. PhD thesis, Université de Caen, 2009.
See pages: 39, 43, 45, 52

[10] Guillaume Bagan, Arnaud Durand, Emmanuel Filiot, and Olivier Gauwin. Efficient Enumeration for Conjunctive Queries over X-underbar Structures. In *Conf. on Computer Science Logic (CSL)*, pages 80–94, 2010.
See pages: 41

[11] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Conf. on Computer Science Logic (CSL)*, pages 208–222, 2007.
See pages: 39, 40, 92

[12] Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *RAIRO Theoretical Informatics and Applications*, 42(1):147–164, 2008.
See pages: 43, 54

[13] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.
See pages: 39

[14] Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.
See pages: 118

[15] Anne Berry, Jean Paul Bordat, and Olivier Cogis. Generating all the minimal separators of a graph. In Peter Widmayer, Gabriele Neyer, and Stephan Eidenbenz, editors, *WG*, volume 1665 of *Lecture Notes in Computer Science*, pages 167–172. Springer, 1999.
See pages: 50

[16] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.
See pages: 44, 70

[17] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. *J. of the ACM*, 56(3), 2009.
See pages: 47

[18] Mikołaj Bojańczyk and Paweł Parys. XPath evaluation in linear time. *J. of the ACM*, 58(4), 2011.
See pages: 45, 48, 118

[19] Johann Brault-Baron. *De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.
See pages: 39, 41

[20] Thomas Colcombet. A Combinatorial Theorem for Trees. In *Intl. Coll. on Automata, Languages and Programming (ICALP)*, pages 901–912, 2007.
See pages: 44, 45, 69, 71, 72

[21] Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. *J. on Symbolic Computation*, 9(3):251–280, 1990.
See pages: 40

[22] Bruno Courcelle. Graph Rewriting: An Algebraic and Logic Approach. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 193–242. 1990.
See pages: 44, 68, 70

[23] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.
See pages: 44, 46, 69

[24] Nadia Creignou and Jean-Jacques Hébrard. On Generating All Solutions of Generalized Satisfiability Problems. *ITA*, 31(6):499–511, 1997.
See pages: 51

[25] Anuj Dawar, Martin Grohe, and Stephan Kreutzer. Locally Excluding a Minor. In *Symp. on Logic in Computer Science (LICS)*, pages 270–279, 2007.
See pages: 123

[26] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. on Computational Logic (ToCL)*, 8(4), 2007.
See pages: 14, 42, 43, 46, 49, 54, 91

[27] Arnaud Durand and Stefan Mengel. On Polynomials Defined by Acyclic Conjunctive Queries and Weighted Counting Problems. *CoRR*, abs/1110.4201, 2011.
See pages: 39

[28] Arnaud Durand and Yann Strozecki. Enumeration Complexity of Logical Query Problems with Second-order Variables. In *Conf. on Computer Science Logic (CSL)*, pages 189–202, 2011.
See pages: 46

[29] Zdeněk Dvořák, Daniel Král, and Robin Thomas. Deciding First-Order Properties for Sparse Graphs. In *Symp. on Foundations of Computer Science (FOCS)*, pages 133–142, 2010.
See pages: 46, 47, 89, 90, 95, 97, 111

[30] Zdenek Dvorak, Daniel Král, and Robin Thomas. Testing first-order properties for subclasses of sparse graphs. *CoRR*, abs/1109.5036, 2011.
See pages: 123

[31] Ronald Fagin. Probabilities on Finite Models. *J. Symb. Log.*, 41(1):50–58, 1976.
See pages: 50

[32] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. of the ACM*, 49(6):716–752, 2002.
See pages: 70

[33] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006.
See pages: 16, 25

[34] Michael L. Fredman and Leonid Khachiyan. On the Complexity of Dualization of Monotone Disjunctive Normal Forms. *J. Algorithms*, 21(3):618–628, 1996.
See pages: 49

[35] Markus Frick and Martin Grohe. Deciding first-order properties of locally tree-decomposable structures. *J. of the ACM*, 48(6):1184–1206, 2001.
See pages: 123

[36] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Logic*, 130(1-3):3–31, 2004.
See pages: 42, 43, 47, 53, 55, 88, 111

[37] Leslie Ann Goldberg. Efficient Algorithms for Listing Unlabeled Graphs. *J. Algorithms*, 13(1):128–143, 1992.
See pages: 50

[38] Leslie Ann Goldberg. Listing Graphs That Satisfy First-Order Sentences. *J. Comput. Syst. Sci.*, 49(2):408–424, 1994.
See pages: 50

[39] Georg Gottlob, Christoph Koch, and Klaus U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.
See pages: 41

[40] Etienne Grandjean. Sorting, Linear Time and the Satisfiability Problem. *Annals of Mathematics and Artificial Intelligence*, 16:183–236, 1996.
See pages: 16, 20

[41] Martin Grohe. Fixed-Point Definability and Polynomial Time on Graphs with Excluded Minors. In *LICS*, pages 179–188. IEEE Computer Society, 2010.
See pages: 124

[42] Martin Grohe and Stephan Kreutzer. *Model Theoretic Methods in Finite Combinatorics*, chapter Methods for Algorithmic Meta Theorems. American Mathematical Society, 2011.
See pages: 46, 47, 89, 90, 95, 97, 111

[43] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.
See pages: 25

[44] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.
See pages: 114

[45] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On Generating All Maximal Independent Sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.
See pages: 49, 50, 51

[46] Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science (LMCS)*, 7(2), 2011.
See pages: 11, 42, 43, 45, 54, 63

[47] Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. *Symp. on Principles of Database Systems (PODS)*, 2013.
See pages: 11, 46, 47, 89, 90, 97

[48] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. on Computational Logic (ToCL)*, to appear.
See pages: 11, 44, 45, 68, 69

[49] Leonid G. Khachiyan, Endre Boros, Khaled M. Elbassioni, Vladimir Gurvich, and Kazuhisa Makino. On the Complexity of Some Enumeration Problems for Matroids. *SIAM J. Discrete Math.*, 19(4):966–984, 2005.
See pages: 50

[50] Stephan Kreutzer and Anuj Dawar. Parameterized complexity of first-order logic. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:131, 2009.
See pages: 111, 123

[51] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.
See pages: 55

[52] Steven Lindell. A Normal Form for First-Order Logic over Doubly-Linked Data Structures. *Int. J. Found. Comput. Sci.*, 19(1):205–217, 2008.
See pages: 43, 54

[53] Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion I. Decompositions. *Eur. J. Comb.*, 29(3):760–776, 2008.
See pages: 25, 26, 27, 46, 47, 90, 110, 124

[54] Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion II. Algorithmic aspects. *Eur. J. Comb.*, 29(3):777–791, 2008.
See pages: 25, 26, 92

[55] Jaroslav Nešetřil and Patrice Ossona de Mendez. On nowhere dense graphs. *European J. of Combinatorics*, 32(4):600–617, 2011.
See pages: 111, 122

[56] Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity*. Springer, 2012.
See pages: 34, 47, 90, 111, 123

[57] Christos H. Papadimitriou and Mihalis Yannakakis. On the Complexity of Database Queries. *J. on Computer and System Sciences (JCSS)*, 58(3):407–427, 1999.
See pages: 11, 16, 38

[58] Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.*, 79(6):984–1001, 2013.
See pages: 39

[59] Prabhakar Raghavan. Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs. *J. Comput. Syst. Sci.*, 37(2):130–143, 1988.
See pages: 50

[60] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.
See pages: 25

[61] Detlef Seese. Linear Time Computable Problems and First-Order Descriptions. *Mathematical Structures in Computer Science*, 6(6):505–526, 1996.
See pages: 42, 53, 55

[62] Luc Segoufin. Enumerating with constant delay the answers to a query. In Wang-Chiew Tan, Giovanna Guerrini, Barbara Catania, and Anastasios Gounaris, editors, *ICDT*, pages 10–20. ACM, 2013.
See pages: 13, 38

[63] Saharon Shelah. The monadic theory of order. *Annals of Mathematics*, 102(3):379–419, 1975.
See pages: 71, 72

[64] Hong Shen and Weifa Liang. Efficient Enumeration of all Minimal Separators in a Graph. *Theor. Comput. Sci.*, 180(1-2):169–180, 1997.
See pages: 50

[65] Yann Strozecki. *Enumeration complexity and matroid decomposition*. PhD thesis, Université de Paris 7, 2010.
See pages: 48, 49, 50, 51, 52

[66] Yann Strozecki. Enumeration of the Monomials of a Polynomial and Related Complexity Classes. In *Intl. Symp. on Mathematical Foundations of Computer Science (MFCS)*, pages 629–640, 2010.
See pages: 51

[67] Yann Strozecki. A note on polynomial delay vs. incremental delay. *manuscript*, 2012.
See pages: 52

[68] Ken Takata. Space-optimal, backtracking algorithms to list the minimal vertex separators of a graph. *Discrete Applied Mathematics*, 158(15):1660–1667, 2010.
See pages: 50

[69] Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.
See pages: 39

[70] Wolfgang Thomas. Classifying Regular Events in Symbolic Logic. *J. on Computer and System Sciences (JCSS)*, 25(3):360–376, 1982.
See pages: 74

[71] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A New Algorithm for Generating All the Maximal Independent Sets. *SIAM J. Comput.*, 6(3):505–517, 1977.
See pages: 51

[72] Takeaki Uno. Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs. In *Intl. Symp. on Algorithms and Computation*, pages 92–101, 1997.
See pages: 51

[73] Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *Intl. Conf. on Very Large Databases (VLDB)*, pages 82–94, 1981.
See pages: 39