Query enumeration and nowhere dense graphs

Alexandre Vigny

Université Sorbonne
Paris Cité

THÈSE DE DOCTORAT

Université Sorbonne Paris Cité
Preparée à l'Université Paris Diderot

École doctorale de Sciences Mathématiques de Paris Centre

Discipline : Mathématiques
Spécialité : Logique et Fondement de l'Informatique

présentée par

Alexandre VIGNY

Query enumeration and nowhere dense graphs

Dirigée par Arnaud Durand et Luc Segoufin

Soutenue le *27 Septembre 2018* devant le jury composé de :

Mme.	Claire	David	Institut Gaspard Monge	Examinatrice
M.	Arnaud	Durand	Université Paris Diderot	Directeur
M.	Étienne	Grandjean	Université de Caen Basse-Normandie	Examinateur
M.	Wim	Martens	Universität Bayreuth	Rapporteur
M.	Patrice	Ossona de Mendez	École des Hautes Études en Sciences Sociales	Rapporteur
M.	Luc	Segoufin	ENS Ulm	Directeur
Mme.	Cristina	Sirangelo	Université Paris Diderot	Présidente

Acknowledgments

First of all, I would like to thank my advisors, Arnaud and Luc. Thank you for all of your advice and for introducing me to the topic of enumeration. I have learned a lot from the two of you during these three years.

I would like to thank Wim Martens and Patrice Ossona de Mendez who accepted to review my thesis and made really nice comments about it. Many thanks to Claire David, Étienne Grandjean and Cristina Sirangelo who accepted to by in my jury on such short notice.

I also want to thank the people I have worked with. First Nicole Schweikardt, all of her comments and questions on the article we wrote together helped to write a better one and helped me to write a better thesis. I also want to thank Yann Strozecki and Charles Paperman, even though our collaborations were only brief and did not lead to concrete results it has been a pleasure to work with you. Thanks to Stefan Mengel for all the discussions we had and the help he gave me during my internship in the LSV.

Big thanks to all the PhD students I have met in the past three years, first those who where (or still are) in my office in Sophie Germain: Pablo, Mauro, Vitto, Véronique, Sacha, Daniel, Karim and Gabriel. I also thank every one in Sophie Germain for all the afternoons of *Perudo* and *Avalon*, and all the pastries during the Bourbakettes. Those who already left us: Florent, Baptiste, Victoria, Charles, Kevin, Aurélien, Marco x2, Martin. And those who are still there: Reda, Colin, Mario, Nicolas, Yi, Kevin x2, Théophile, Andreas, Antoine, Wille, Juan Pablo, Élie x2, Léa, Omar, Maude, Alex, Corentin, Charles, Pierre.

I thank my family and friends, those who managed to come and those who wanted to but could not.

Finally my greatest thanks to Sophie, for your unconditional help, love, and support, thank you!

Abstract

Query evaluation is one of the most central tasks of a database system, and a vast amount of literature is devoted to the complexity of this problem. Given a database $\mathbf{D}$ and a query q, the goal is to compute the set $q(\mathbf{D})$ of all solutions for q over $\mathbf{D}$. Unfortunately, the set $q(\mathbf{D})$ might be much bigger than the database itself, as the number of solutions may be exponential in the arity of the query. It can therefore be insufficient to measure the complexity of answering q on $\mathbf{D}$ only in terms of the total time needed to compute the complete result set $q(\mathbf{D})$. One can imagine many scenarios to overcome this situation. We could for instance only want to compute the number of solutions or just compute the k most relevant solutions relative to some ranking function.

In this thesis we mainly consider the complexity of *enumerating* the set $q(\mathbf{D})$, i.e., generating one by one all the solutions for q on $\mathbf{D}$. In this context two parameters play an important role. The first one is the *preprocessing time*, i.e. the time it takes to produce the first solution. The second one is the *delay*, i.e. the maximum time between the output of any two consecutive solutions. An enumeration algorithm is then said to be *efficient* if these two parameters are small. For the delay, the best we can hope for is constant time: depending only on the query and independent from the size of the database. For the preprocessing time an ideal goal would be linear time: linear in the size of the database with a constant factor depending on the query. When both are achieved we say that the query can be enumerated with constant delay after linear preprocessing.

A special case of enumeration is when the query is boolean. In this case the preprocessing computes the answer to the query (yes or no). In order to be able to enumerate queries of a given language efficiently, it is therefore necessary to be able to solve the boolean case efficiently.

It has been shown recently that boolean first-order (FO) queries can be evaluated in pseudo-linear time over nowhere dense classes of databases [43]. The notion of nowhere dense classes was introduced in [60] as a formalization of classes of "sparse" graphs and generalizes many well known classes of databases. Among classes of databases that are closed under subdatabases, the nowhere dense classes are the largest possible classes enjoying efficient evaluation of FO queries [55] (modulo an assumption in parameterized complexity theory). It has also been shown that over nowhere dense classes of databases, counting the number of solutions to a given FO query can be achieved in pseudo-linear time [45].

In this thesis, we show that enumerating FO queries on nowhere dense classes of databases can be done with constant delay after pseudo-linear preprocessing. This completes the picture of the complexity of FO query evaluation on nowhere dense classes and, due to the above mentioned result of [55], on all classes that are closed under subdatabases. We also show that for any nowhere dense class of databases, given a FO query q and a database $\mathbf{D}$ in the class, after a pseudo-linear time preprocessing we can test in constant time whether an arbitrary input tuple belongs to the result set $q(\mathbf{D})$.

Key words: Database theory, First order query, Algorithm, Enumeration, Nowhere dense.

Résumé (in french)

L'évaluation des requêtes est l'une des tâches les plus importantes en base de données et beaucoup de recherche a été consacrée à l'étude de la complexité de ce problème. Étant donné une base de données $\mathbf{D}$ et une requête q, le but est de calculer l'ensemble $q(\mathbf{D})$ des solutions de q sur $\mathbf{D}$. Malheureusement, l'ensemble $q(\mathbf{D})$ peut être beaucoup plus grand que la base de données elle-même, car le nombre de solutions peut être exponentiel en l'arité de la requête. Il n'est donc pas judicieux de mesurer la complexité de l'évaluation de q sur $\mathbf{D}$ uniquement en termes de temps nécessaire pour calculer l'ensemble $q(\mathbf{D})$. On peut imaginer de nombreux scénarios pour contourner cette difficulté. On peut par exemple seulement vouloir calculer le nombre de solutions ou juste les k solutions les plus pertinentes selon un certain ordre.

Dans cette thèse, nous considérons principalement la complexité de l'*énumération* de l'ensemble $q(\mathbf{D})$, c'est-à-dire de générer une par une toutes les solutions de q sur $\mathbf{D}$. Dans ce contexte deux paramètres jouent un rôle important. Le premier est le *pré-calcul*, c'est-à-dire le temps qu'il faut pour produire la première solution. Le deuxième est le *délai*, c'est-à-dire le temps maximum entre la production de deux solutions consécutives. On dit qu'un algorithme d'énumération est *efficace* si ces deux paramètres sont petits. Pour le délai, on ne peut pas espérer mieux qu'un temps constant: dépendant uniquement de la requête et indépendant de la taille de la base de données. Pour le pré-calcul, on souhaite une exécution en temps linéaire: linéaire dans la taille de la base de données avec un facteur constant dépendant de la requête. Lorsque les deux objectifs sont atteints, on dit que la requête peut être énumérée à délai constant après un pré-calcul linéaire.

Un cas particulier d'énumération est lorsque la requête est booléenne. Dans ce cas, le pré-calcul calcule simplement la réponse à la requête (oui ou non). Afin de pouvoir énumérer efficacement les requêtes d'un langage donné, il est donc nécessaire d'être capable de résoudre le cas booléen efficacement.

Il a été montré récemment que les requêtes booléennes du premier ordre (FO) peuvent être évaluées en temps pseudo-linéaire sur les classes de base de données nulle-part denses [43]. La notion de classe nulle-part dense a été introduite dans [60] comme une formalisation des classes de graphes "éparses" et généralise de nombreuse classes de base de données. Parmi les classes de bases de données qui sont closes par sous bases de données, les classes nulle-part dense sont les plus grandes bénéficiant d'une évaluation efficace des requêtes FO [55] (modulo une hypothèse en théorie de complexité paramétrée). Il a également été montré que sur les bases de données nulle-part dense, compter le nombre de solutions d'une requête FO peut être réalisée en temps pseudo-linéaire [45].

Dans cette thèse, nous montrons que l'énumération des questions FO sur les classes de bases de données nulle-part denses peut se faire à délai constant après un pré-calcul pseudo-linéaire. Cela complète l'étude de la complexité de l'évaluation des requêtes FO sur les classes nulle-part denses et, grâce au résultat de [55], sur toutes les classes close par sous bases de données. Nous montrons également que pour toute classe de bases de données nulle-part dense, étant donné une requête FO q et une base de données $\mathbf{D}$ de cette classe, après un pré-calcul pseudo-linéaire, nous pouvons tester en temps constant si un tuple donné appartient à l'ensemble des solutions $q(\mathbf{D})$.

Mots clés: Base de données, Requêtes, Algorithme, Énumération, Nulle-part dense.

Contents

Chapter 1

Introduction

"Is the nearest bakery still open?" "What is the cheapest flight from Paris to Houston?" "How many copies of this book are available at the library?" Those are frequently asked questions and surely someone somewhere can answer them. But how, and where, can we find these answers? On the other hand, you may have some piece of information that someone is desperately searching for. Funny thing is, the person that could have answered your question might be yourself from two weeks ago! If only you had written it down! But if everyone were to write everything down every time, how would we be able to search through this mess? The question would then become "Where is that piece of paper I used yesterday?"

Even if we do not see it, this is essentially what is currently happening. With everything that we share online, intentionally or otherwise, from our favorite social networks, our smart watch or via our personal web page, we create vast amounts of data that is stored, somehow, somewhere. The question remains the same: How do we search through this mess ?

To get a satisfying answer, the problem can be seen differently: How can we store that information in order to retrieve it efficiently? This thesis is just about that: storing and retrieving information in an efficient way.

1.1 Databases and query evaluation

As you might have guessed, database theory is at the core of this document. If databases have dramatically evolved with the rise of computer science, they have been used basically forever. How would you find a book in a library if the books were not indexed by genres and authors? How would you obtain the phone number of a friend if the names and phone numbers were written at random in the phone book?

A database $\mathbf{D}$ stores data in such a way that, when given a query (i.e. a question) q, we can compute the set of all possible solutions $q(\mathbf{D})$. This is what we call the *query evaluation problem*. For example, the goal can be to compute the names of all people that live in Paris and whose phone number ends with '7518'. This seems to be a big set of solutions, but if we look closely it is not that big compared to the original database. Here, by definition, the set of solutions is not bigger than the phone book i.e. our database. In

this case, the set of solution is a subset of the database. However, we can easily find scenarios where the set of solutions is way bigger than the database itself.

Imagine a store with fifty types of items that cost less than 1€ each. A database that stores the name and the corresponding price of each item fits on a single sheet of paper. Then someone asks: "What can I buy with 10€ ?". The number of possible purchases, and therefore the number of solutions is about 50^{10}. This number is bigger than the number of characters ever written in all the books combined![1]

This creates at least two difficulties. First, from a practical point of view, huge sets of solutions cannot easily be displayed on the screen of a cellphone or stored in a computer. Secondly, as theoreticians in computer science, we tend to say that the time needed to compute the answer of a problem is a good measure of its difficulty. It is not the case here.

Going back to our 1€ store, almost instantaneously, we can say how many solutions there are. We can also immediately produce a first solution by picking ten items randomly. If more solutions are needed, we can quite easily produce new ones. This problem should therefore be considered as easy. However, the time needed to compute the full set of solutions is rather huge because the time needed to simply write all answers is too big, among other things.

To overcome these difficulties, we need to introduce new problems beside (simple) query evaluation. For those problems, the number of solutions should not impact the time needed to compute an answer. We introduce such problems in the next section. Before that, we provide additional information about the query evaluation problem.

In this thesis, we look at the evaluation problem from a theoretical point of view. Nonetheless, this problem is more of a practical one. Everyday around the world database management systems answer queries. Depending on the asked query, the evaluation can be performed more or less efficiently, and this is where theory comes in handy. Theoreticians proved that for some query languages (i.e. sets of queries) the evaluation can be performed in an optimal manner. Those techniques has been implemented and are used by the database management systems. However, for queries that fall out of such languages, database management systems have no choice but to perform a naive (and not always optimal) evaluation of the query.

The theoretical search for instances where the query evaluation problem can be solved efficiently is still full of undiscovered results with real life applications.

1.2 Beyond query evaluation

In this thesis, we deal with several problems that do not have the flaw of query evaluation. For these new problems, for a fixed query and database, we can assume that if we need a lot of time to compute the answer, there is some intrinsic difficulty for this query and/or database.

For the *model checking problem* the queries are such that the answer can only be *YES* or *NO*. For example "Is the nearest bakery still open?"

[1] According to the Guinness World Records, the largest existing book is *A la recherche du temps perdu* by Marcel Proust with around 10^7 characters. The number of books ever published has been estimated to 130 million by Google. This leads to around 10^{15} characters, while $50^{10} \simeq 10^{17}$.

The *counting problem* is the problem of computing the number of solutions. Even when the number of solutions is gigantic, the time we need to write this number may be quite short. In our example with a store where everything cost less than 1€, the question could be "How many different sets of 10 items can I purchase with 10€?"

For the *testing problem*, in addition to a database and a query, a possible solution is also given. The goal is then to answer whether this specific tuple is a solution. Again, even if there are thousands of solutions, it may not impact the time we need to answer this question. We view this problem as a two phases procedure. For example, someone wants to ask about open bakeries. After some computation has been made someone gives the name of a bakery, the goal is to answer whether this specific bakery is still open.

Last but not least, we encounter the *enumeration problem*. The goal is to produce solutions one at a time with no repetition and not all solutions at once, which is the case for the query evaluation problem. This leads to the notion of delay. The *delay* is the maximal time between two consecutive outputs i.e. the time needed to produce a new solution. In order to minimize the delay, we authorize some computation to be made on the database before the first solution is produced. This is the preprocessing phase. The time needed to complete this first phase (the preprocessing time) and the delay are used to evaluate the efficiency of an enumeration algorithm.

Our main goal is to find what properties a database and a query must have in order to find an algorithm that enumerates the set of solutions with constant delay after a preprocessing performed in time that does not exceed too much linear time. Here, *constant delay* means that the time needed to produce a new solution should not depend on the size of the database.

1.3 Goals and tools

As we just stated, the main goal of this document is to provide classes of databases and query languages for which the problems around query evaluation can be solved efficiently.

The first element that we fix is the query language. In this thesis, we only work with *first-order queries* (FO for short). We have chosen FO queries for several reasons. First of all, FO queries are equivalent to relational algebra, which is the core of SQL. And SQL is the query language most used by database management systems. Furthermore, we will see in Chapter 3 that we already have a pretty good idea of when query evaluation problems are tractable for more expressive query languages like *monadic second-order* and for more restricted queries like *acyclic conjunctive queries*. It should be mentioned that FO queries is not the only remaining candidate and some other related languages (like guarded logics) also contains many unanswered problems. Nevertheless, FO queries is one of the query languages that have been the most studied in the last decades. This provides many tools that help us to tackle our problems.

The second element that we fix is the classes of structures that we study. Graph theory provides a vast amount of structural properties that, even if designed for graphs, can be transfered to databases. *Nowhere dense* classes of graphs is the notion of structures that we consider for our databases. In comparison to FO queries, nowhere dense

classes of graphs have been introduced recently. This notion has been defined defined by Nešetřil and Ossona de Mendez in [60]. After that, many equivalent definition of nowhere dense graphs have been provided, making this notion very robust considering that it has been introduced less than ten years ago. Furthermore, we show in this thesis that nowhere dense classes of graphs happens to be the answer of the question: "What is the largest classes of structure (closed under substructure) on which the enumeration of FO queries can be performed efficiently?" And we do not get to choose the answer.

We now turn to the techniques we are using to obtain our algorithms. The main properties of FO queries we use is the locality of such queries. Essentially, it means that every FO query only talk about neighborhoods of elements and that two unrelated elements of the database can be treated independently [35]. It has recently been shown that over nowhere dense classes of graphs, every FO query can effectively be decomposed into local queries [45]. Furthermore, this new decomposition has the advantage of preserving some parameter of the generated queries. Concerning nowhere dense graphs, we use several different results. The first one is a two players game that characterize nowhere dense graphs. This is only used to provide the running time analysis and correctness proof of our algorithms. We also use some existing algorithms that deal with nowhere dense graphs, mainly the algorithm for the model checking problem and an algorithm that computes a neighborhood cover of nowhere dense graphs. Both have been presented in [44].

The main achievement of our work is to prove that over nowhere dense classes of graphs, the solutions of queries written in first-order logic can be enumerated with constant delay after a pseudo-linear preprocessing. We also explain how to efficiently solve the counting and testing problems in the same setting.

1.4 Structure of the thesis

We end this introduction by presenting the structure of this document.

- In Chapter 2, we introduce notions and notations that are going to be used in the subsequent chapters.
- In Chapter 3, we present the state of the art for the different problems related to query evaluations.
- Chapter 4 contains results that are needed in the reminder of this document.
- Chapter 5 deals with the testing problem for first-order queries over nowhere dense classes of graphs.
- In Chapter 6, we focus on the enumeration problem in the same setting.
- Chapter 7 is devoted to the counting problem. This problem has already been tackled for first-order queries and nowhere dense classes of graphs in [45]. We provide an alternative proof.
- In Chapter 8, we conclude the thesis with a discussion of our main contributions from Chapters 5, 6 and 7. We also present problems that are still open in this area.

Chapter 2

Preliminaries

Contents

This chapter sets the main definitions that will be used throughout this thesis. We start by providing the definitions of the most common objects we are using: databases and queries in Section 2.1. We then introduce our model of computation in Section 2.2 and some complexity classes in Section 2.3. We provide additional details for the notion of query language in Section 2.4. After that, we spend some time in Section 2.5 talking about the different problems we encounter in this thesis and how they interact with each other. Finally, Section 2.6 is about graphs.

The goal of this Chapter is to define everything needed to understand the State of the Art that is presented in Chapter 3. For some notions that are at the core of this thesis, we provide additional details in Chapter 4.

2.1 Databases and queries

We define databases as relational structures.

Definition 2.1.1. A relational *schema* (or simply schema) is a finite set of relation symbols $\sigma = (P_1, \dots P_l)$, each P_i having an associated arity r_i. A finite relational *structure* $\mathbf{D}$ over a relational schema σ consists of:

- a finite set D called the *domain* of $\mathbf{D}$, and
- for every P_i in σ, a subset of D^{r_i}, denoted $P_i(\mathbf{D})$ or $P_i^{\mathbf{D}}$. This subset is called the interpretation of P_i in $\mathbf{D}$.

We fix a standard encoding of structures as input, see for example [1]. We denote by $\|\mathbf{D}\|$ the size of (the encoding of) $\mathbf{D}$, while $|\mathbf{D}|$ denotes the size $|D|$ of its domain.

Example 2.1.2. *Here is an example of database with three relations. Films has arity* 1*, Actors has arity* 2 *and Plays has arity* 3*.*

Films

Alone trip
Alaska Phantom
Argonauts Town
Alien Center

Actors

Penelope	*Guiness*
Ed	*Chase*
Bob	*Fawcett*
Julia	*MCQueen*

Plays in

Penelope	*Guiness*	*Alone trip*
Penelope	*Guiness*	*Alaska Phantom*
Ed	*Chase*	*Alaska Phantom*
Bob	*Fawcett*	*Argonauts Town*
Julia	*MCQueen*	*Argonauts Town*
Bob	*Fawcett*	*Alien Center*

In this example, the Domain D of the database is: {Alone trip, Alaska Phantom, Argonauts Town, Alien Center, Penelope, Ed, Bob, Julia, Guiness, Chase, Fawcett and MCQueen }.

Definition 2.1.3. A *query* over a schema σ is a function that associates every database over σ to a subset of D^r. The integer r in $\mathbb{N}$ is what we call the *arity* of the query.

Example 2.1.4. *Example of queries over the schema presented in the previous example can be:*

1. *"Is there a Film in which only one actor plays ?"*
2. *"Who are the actors who played in at least two films ?"*
3. *"What are the pairs of actors who have played in the same film ?"*

We often divide queries in different groups. A *boolean query* (or *sentence*) is a query of arity 0. The "set" of solutions is either *"YES"* or *"NO"* and a database either satisfies or does not satisfy a boolean query. We note $\mathbf{D} \models q$ the fact that the database $\mathbf{D}$ satisfies (or models) the boolean query q. The query 1 in example 2.1.4 is boolean.

The queries with arity 1 are named *unary queries*. This is the case for the query 2 in example 2.1.4. In that case, the set of solution is a subset of the database's domain. Note that for the other queries (i.e. when the arity is greater than 2) the set of solutions might easily be way bigger than the domain of the database itself. The query 3 in example 2.1.4 above has arity 2. More details and definitions for query languages are provided in Section 2.4

2.2 Model of computation

In Section 2.5, we define several problems that form the core of this thesis. The algorithms we provide towards solving those problems work in linear time (or so). When dealing with linear time, it is classical to use Random Access Machines (RAM) with addition, multiplication and uniform cost measure as a model of computation. We do not explain in full details what is a RAM and interested readers are referred to [2]. Nonetheless let us briefly explain what we can do with a RAM.

A RAM contains two accumulators A and B and work registers R_i, for every $i \geq 0$. Its program is a sequence $I(1), \ldots, I(r)$ of instructions of the forms:

1. $A \leftarrow j$, for some constant integer $j \geq 0$,
2. $A \leftarrow R_A$,
3. $R_A \leftarrow B$,
4. $B \leftarrow A$,
5. $A \leftarrow A \star B$ where $\star \in \{+, -, \times, /\}$,
6. if $A = B$ then $I(i)$ else $I(j)$,

where R_A is actually R_i with i being the current value of A.

Each instruction can be performed in constant time. When we write algorithms, we do not use such restricted instructions. They are the bricks used for more complicated

instructions. For example, using instruction of type (6), we can simulate *while* and *for* loops. We also use variables in our algorithms. As we only need a constant number of variables (k for example), we can simply stash them into the first k registers.

Instructions (2) and (3) are the instructions that really make this model different from a classical Turing machine. They make read and write operations much faster than with Turing machines, where the cursor need to be moved to the correct cell between each operation. For example we can switch the values stored in the i-th and the j-th register in constant time. For the same task, a Turing machine needs $O(|i-j|)$ operations.

RAM comes in handy when we need to store functions. For example, if $f(\cdot)$ is a finite function that associates integers to integers, we can simply store the value of $f(i)$ in the i-th register. Later on, we can access this value in constant time. In Section 4.1, we provide efficient data structures to store more complicated functions.

RAM is a powerful model of computation. One rather impressive result that we actually use in our algorithms is the following theorem.

Theorem 2.2.1 ([40]). *A RAM can sort a list of n elements, each using $\log(n)$ bits, in time $O(n)$.*

A typical use of this theorem is that we always assume that the database is sorted. More precisely, one can fix an arbitrary order on the domain of the database. Then using Theorem 2.2.1, we can sort each relation of the database in lexicographical order. Since this only takes linear time, it is an implicit first step of most algorithms presented in this thesis. Using sorted data comes in handy to get an enumeration procedure that outputs tuples in a lexicographic manner.

2.3 Parametrized complexity

Given a problem, its *complexity* (or time complexity) is the time needed to answer this problem. In the vast majority of cases, this time depends on the input size. Most problems in this thesis have two inputs: a database $\mathbf{D}$ and a query q. There are several ways of expressing the complexity of such problems.

In the *combined complexity* framework, we would just say that the input size is the sum of the size of the query and the size of the database. In our setting, this might not be the best approach as those two parameters have very different sizes and play very different roles. For example, if the running time of an algorithm is $O(2^{|q|} \cdot \|\mathbf{D}\|)$, we might not want to consider it as exponential.

An other framework suits better: *parametrized complexity*. Even if we heavily use parametrized complexity, it is not the core of this thesis. We only mention the key elements that will be used later. Interested readers are referred to the book [30] for more details.

In parametrized complexity, a problem contains three elements:

- one or several inputs,
- a parameter, i.e. a number computable from the input, and
- a question.

One example that we encounter in this thesis is the model checking problem: For a given database $\mathbf{D}$ and boolean query q, does $\mathbf{D} \models q$? Considering this problem in the parametrized complexity framework, the input is a database $\mathbf{D}$ and a boolean query q, the parameter is the size $|q|$ of the query, and the question is whether $\mathbf{D} \models q$.

The idea is to isolate some part of the input (the query in our case) and consider it's size to be a constant. Hence an algorithm whose running time is $O(2^{|q|} \cdot \|\mathbf{D}\|)$ is considered as linear in the parametrized complexity framework. We now define some complexity classes.

Definition 2.3.1. A parametrized problem is *Fixed Parameter Tractable* (FPT) if there is a computable function f and a constant c such that the question can be answered in time $O(f(k) \cdot n^c)$, where n is the size of the input and k the size of the parameter.

The idea behind this definition is that in many scenarios (when the parameter is way smaller than the input) instead of having an algorithm working in time $O(n^k)$, it is more efficient to have one that works in time $O(2^k \cdot n)$.

Parametrized complexity is a very fruitful framework. There is a suitable notion of reduction, called FPT-reductions and the class FPT is closed under FPT-reductions. There are some hard classes of parametrized problems that are also closed under FPT-reductions.

An important class of hard parametrized problem is W[1]. We can draw a parallel between the classical complexity framework and parametrized complexity. For instance, FPT mirrors P and W[1] mirrors NP. A complete problem for W[1] is the parametrized model checking problem for conjunctive queries [64]. We give some context for this result in Section 3.1.1. Similarly to classical complexity, we have that FPT $\subseteq$ W[1] but we do not know whether this inclusion is strict or if those two classes contains the exact same problems.

Another class that we encounter in this thesis is AW[$*$]. A typical complete parametrized problem for AW[$*$] is the parametrized model checking problem for first-order queries [64]. This leads to some discussion in Section 2.5. AW[$*$] mirrors PSpace in classical complexity. As previously, we have that W[1] $\subseteq$ AW[$*$] but we do not know whether this inclusion is strict or if those two classes contains the exact same problems. However, it is strongly believed that FPT $\subsetneq$ W[1] $\subsetneq$ AW[$*$]. This is an assumption that we often make to provide lower bounds.

In this thesis, we focus on FPT algorithms i.e. algorithms solving parametrized problems in time $O(f(k) \cdot n^c)$. We actually want better algorithms. Ideally, we would like to have *linear* algorithms i.e. algorithms solving parametrized problem in time $O(f(k) \cdot n)$. Note that linear does not require the function f to be linear. Only the impact of the size n of the input needs to be linear.

For most of our problems, we do not provide linear algorithms. However, we often show that we can get really close to linear algorithms.

Definition 2.3.2. A problem can be solved in *pseudo-linear* time if for every $\epsilon > 0$, there is an algorithm solving the problem in time $O(f(\epsilon) \cdot n^{1+\epsilon})$, where n is the size of the input problem.

A parametrized problem can be solved in *pseudo-linear* time if for every $\epsilon > 0$, there is an algorithm solving the problem in time $O(f(\epsilon, k) \cdot n^{1+\epsilon})$, where n is the size of the input and k the parameter.

2.4 Query languages: logics

In Section 2.1 we gave a first definition of queries. We now go a little deeper into this notion and add some formalism. We assume that readers are familiar with the notion of first-order (FO) and monadic second-order (MSO) logics. Therefore we only provide the basic notions. Readers who are interested in other characteristics of those query languages and their use in database theory should take a look at the book [56].

First-order (FO) queries over a schema $\sigma := (P_1, \ldots, P_l)$ are built from atomic queries of the form $x = y$ or $P_i(x_1, \ldots, x_{r_i})$. We can then combine atomic queries with boolean connectors: $(\neg, \vee, \wedge)$. We can also use existential and universal quantification: $(\exists, \forall)$.

Monadic second-order (MSO) is an extension of FO with set variables and set quantifications. It allows to build more expressive queries than FO does.

In Section 3.1.1, we define several other query languages, more restricted than first-order logic.

In the reminder of this document, queries are either denoted by q or Greek letters like φ or ψ. The size of a query $q(\overline{x})$, noted $|q(\overline{x})|$, is the sum of the number of free variables, the number of quantifiers, the number of boolean connectors and the number of atomic queries that appear in q.

Definition 2.4.1. A *k-tuple* is either a list of k variables $(x_1, \ldots, x_k)$ or a list of k elements $(a_1, \ldots, a_k)$ of some database. When k is clear from context, we often simply write $\overline{x}$ or $\overline{a}$.

When we write $q(\overline{x})$, we mean that the tuple $\overline{x}$ is exactly the free variables of q. When I is a subset of $\{1, \ldots, k\}$, the subset of the tuple $\overline{x}$ that only contains the x_i with i in I is noted $\overline{x}_I$. Similarly, the subset of the tuple $\overline{x}$ that only contains the x_i such that i is not in I is noted $\overline{x}_{\setminus I}$.

2.5 Query problems

In this section, we first define the problems that are at the core of this thesis. We discuss about the model checking, the evaluation, the counting, the enumerating, and the testing problems. After that, we look at the key properties of those problems. Finally, we discuss the interactions between the different problems.

2.5.1 Definitions

In the upcoming definitions, $\mathcal{C}$ is a class of databases and $\mathcal{L}$ a query language.

Model checking

Definition 2.5.1. The *model checking problem* of $\mathcal{L}$ over $\mathcal{C}$ is the problem of, given a database $\mathbf{D}$ in $\mathcal{C}$ and a boolean query q in $\mathcal{L}$, deciding whether $\mathbf{D} \models q$.

In this thesis, we consider the model checking problem to be tractable if it can be solved in polynomial time. This means that there exist an integer $c > 0$, a computable function f and an algorithm that, upon input of a database $\mathbf{D}$ in $\mathcal{C}$ and a boolean query q in $\mathcal{L}$, decides whether $\mathbf{D} \models q$ in time $O(f(|q|) \cdot \|\mathbf{D}\|^c)$. In Chapter 3, we present cases where the model checking problem can be solved in linear time (i.e. when $c = 1$). Sometimes it will just be solvable in pseudo-linear time.

Evaluating

Definition 2.5.2. The *evaluation problem* of $\mathcal{L}$ over $\mathcal{C}$ is the problem of, given a database $\mathbf{D}$ in $\mathcal{C}$ and a query q in $\mathcal{L}$, computing the set of solutions $q(\mathbf{D})$.

In this thesis, we consider the evaluation problem to be tractable if it can be solved in polynomial time. By polynomial, we mean polynomial in both the size of the input and the size of the output. This means that there exist integers $c_1 > 0$ and $c_2 > 0$, a computable function f and an algorithm that, upon input of a database $\mathbf{D}$ in $\mathcal{C}$ and a query q in $\mathcal{L}$, computes the set $q(\mathbf{D})$ in time $O\big(f(|q|) \cdot (\|\mathbf{D}\|^{c_1} + |q(\mathbf{D})|^{c_2})\big)$. Note that it is the same definition with $O(f(|q|) \cdot \|\mathbf{D}\|^{c_1} \cdot |q(\mathbf{D})|^{c_2})$ since $a^{c_1} \cdot b^{c_2} \leq a^{c_1+c_2} + b^{c_1+c_2}$. In Chapter 3, we present cases where the evaluation problem can be solved in linear time (i.e. when $c_1 = c_2 = 1$). The two definitions are not the same if we restrict our attention to linear algorithms. If we do so, then the first one is more constrained.

Counting

Definition 2.5.3. The *counting problem* of $\mathcal{L}$ over $\mathcal{C}$ is the problem of, given a database $\mathbf{D}$ in $\mathcal{C}$ and a query q in $\mathcal{L}$, computing the number of solutions $|q(\mathbf{D})|$.

The complexity we want for a counting algorithm is quite similar to the one we want for a model checking algorithm. We consider the counting problem to be tractable if it can be solved in polynomial time. This means that there exist an integer $c > 0$, a computable function f and an algorithm that, upon input of a database $\mathbf{D}$ in $\mathcal{C}$ and a query q in $\mathcal{L}$, computes $|q(\mathbf{D})|$ in time $O(f(|q|) \cdot \|\mathbf{D}\|^c)$. In Chapter 3, we present cases where the counting problem can be solved in linear time (i.e. when $c = 1$). Sometimes it will just be solvable in pseudo-linear time.

Enumerating

Definition 2.5.4. The *enumeration problem* of $\mathcal{L}$ over $\mathcal{C}$ is the problem of, given a database $\mathbf{D}$ in $\mathcal{C}$ and a query q in $\mathcal{L}$, outputting the elements of $q(\mathbf{D})$ one by one and without repetition.

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 exhibit pairs

$\mathcal{L}$, $\mathcal{C}$ such that the enumeration problem of $\mathcal{L}$ over $\mathcal{C}$ admits an enumeration algorithm with constant delay. At the same time we want to bound the time needed to produce the first solution. The first phase, performed before the enumeration phase, is called the *preprocessing*.

An enumeration algorithm with constant delay after a linear preprocessing can be described as follows: Upon input of a database $\mathbf{D}$ in $\mathcal{C}$ and a query q in $\mathcal{L}$, the algorithm performs

- a preprocessing phase in time $O\big(f(|q|) \cdot \|\mathbf{D}\|\big)$, and
- an enumeration phase with constant delay (i.e. in time $O(f(|q|))$). Moreover, no solution is produced twice.

One can view an enumeration algorithm as a compression algorithm that computes a representation of $q(G)$, together with a streaming decompression algorithm. Sometimes only pseudo-linear preprocessing is achieved. This means that for every $\epsilon > 0$, there is an algorithm whose preprocessing works in time $O\big(f(|q|) \cdot \|\mathbf{D}\|^{1+\epsilon}\big)$. However, the delay remains constant (it only depends on $|q|$ and ϵ).

Remark 2.5.5. An interesting subtlety is the uses of the space during the enumeration phase. Each step only use a constant amount of time to produce a new solution. And therefore only a constant amount of new space is used each step. However, the total amount of space that is used might increase while enumerating solutions. More important, this extra space might be needed. It is not known whether any enumeration algorithm with constant delay can be re-written in an algorithm that only requires a constant amount of space for the whole enumeration phase. An example of problem that might separate those two notions can be found in [48] (Section 8.3.1).

In the reminder of this thesis, we only consider enumeration algorithms with the extra property of only using a constant amount of space during the enumeration phase.

Testing

Definition 2.5.6. The *testing problem* 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 $\mathbf{D} \in \mathcal{C}$, answering the following question: given a tuple $\overline{a}$ of k elements from $\mathbf{D}$, does $\mathbf{D} \models q(\overline{a})$?

The parameters of an algorithm that solves the testing problem are quite similar to the parameters of an enumeration algorithm. A testing algorithm with constant time answering after a linear preprocessing can be described as follows: Upon input of a database $\mathbf{D}$ in $\mathcal{C}$ and a query q in $\mathcal{L}$, the algorithm performs

- a preprocessing phase in time $O\big(f(|q|) \cdot \|\mathbf{D}\|\big)$, and
- upon input of a tuple $\overline{a}$ in D^k (where k is the arity of q), answers in constant time whether $\mathbf{D} \models q(\overline{a})$.

Sometimes only pseudo-linear preprocessing is achieved. This means that for every $\epsilon > 0$, there is an algorithm whose preprocessing works in time $O\big(f(|q|) \cdot \|\mathbf{D}\|^{1+\epsilon}\big)$. However, the enumeration time remains constant (only depending on $|q|$ and ϵ).

2.5.2 Key properties and examples

We now give an example of such algorithms. We consider that our query language $\mathcal{L}$ only contains the query $q(x, y) = \neg P(x, y)$. The input for the different problems is a database $\mathbf{D}$ that contains a binary relation P. The way this input is given is simply the list of all pair (a, b) of elements that are in $P^{\mathbf{D}}$. Moreover, we can assume this list to be lexicographically sorted as this operation can be performed in linear time by Theorem 2.2.1.

The easiest of the problems in this setting is the counting problem because

$$|q(\mathbf{D})| = |D|^2 - |P^{\mathbf{D}}|.$$

A single pass through the input allows us to compute $|q(\mathbf{D})|$. Hence the counting problem can be solved in linear time.

The evaluation problem is also solved by an easy algorithm: Iterate through all pairs $(a, b) \in D^2$. For every pair, test whether $\mathbf{D} \models P(a, b)$. If it is not the case, then (a, b) is a solution and we can output it. Otherwise we go to the next pair. In the following, L is the ordered list of all pairs (a, b) that are in $P^{\mathbf{D}}$.

Algorithm 1 Evaluation example

```
1: i ← 0
2: for (a, b) ∈ D² do                          ▷ in lexicographical order
3:     if i < |L| ∧ L[i] = (a, b) then          ▷ (a, b) is not a solution
4:         i ← i + 1                            ▷ We move in the list
5:     else
6:         Output (a, b)                        ▷ If (a, b) is not in L, it is a solution
7:     end if
8: end for
```

Every pair (a, b) we encounter is in $P^{\mathbf{D}}$ (hence is a part of the input) or is a solution. Therefore, the total time needed to compute the set of solutions is: $O(\|\mathbf{D}\| + |q(\mathbf{D})|)$. This algorithm solves this evaluation problem in linear time. However, this cannot work for the enumeration problem because we can encounter arbitrarily many pairs (a, b) that are in $P^{\mathbf{D}}$ in a row. For Algorithm 1, we do not have a bound on the delay. We now give an example of an enumeration algorithm. We first describe it and then give a pseudo-code version of it.

In the following, t is always a tuple in D^2. We use $t + 1$ to represent the tuple following t in the lexicographical order. If t is the last tuple, then $t + 1 :=$ Null. For each tuple t that is in $P^{\mathbf{D}}$, we compute the smallest tuple t' such that $t' > t$ and t' is not in $P^{\mathbf{D}}$. We define f the function that associates t to t'. This can be done in linear time by a single pass from the last element of $P^{\mathbf{D}}$ to its first one.

This concludes the preprocessing phase. Having the above structure, the enumeration phase is performed as follows: The algorithm starts with the smallest tuple t_0 and as long as it is not an element of $P^{\mathbf{D}}$, the algorithm outputs t and goes to $t + 1$. If t is in $P^{\mathbf{D}}$, we have in constant time access to $f(t)$. Since $f(t)$ is by definition the smallest solution greater than t, the algorithm outputs it and proceeds with $f(t) + 1$.

The delay remains constant as long as we can test whether t is in $P^{\mathbf{D}}$ and reach $f(t)$ in constant time. To do so, we also need to track the position of t in the list $P^{\mathbf{D}}$. We now give the pseudo-code version.

Algorithm 2 Enumeration example.
Lines 1 to 12 compute f and lines 16 to 25 enumerate $q(\mathbf{D})$.

```
1:  i ← |L| − 1
2:  j ← Null
3:  t ← L[i] + 1                                   ▷ t can be Null
4:  while i ≥ 0 do
5:      if i = |L| − 1 or L[i] + 1 < L[i + 1] then
6:          j ← i + 1
7:          t ← L[i] + 1
8:      end if
9:      R_{2i} ← j                                 ▷ We store the next position in L
10:     R_{2i+1} ← t                               ▷ We store the next solution
11:     i ← i − 1
12: end while
13:
14: − − − − − − − − − − − − − − − − −              ▷ End of the preprocessing
15:
16: t ← t_0
17: i ← 0
18: while t ≠ Null do
19:     if t = L[i] then                           ▷ t ∈ P^D, hence t is not a solution
20:         t ← R_{2i+1}                           ▷ We move to the next solution
21:         i ← R_{2i}                             ▷ We move in the list
22:     end if
23:     Output t
24:     t ← t + 1
25: end while
```

In this setting the testing problem is the hardest to deal with. If we just store the list $L := P^{\mathbf{D}}$, on input of a tuple (a, b) in D^2, we need to go through all of the input before answering the question. If in the preprocessing we sort the list L, we can then answer the question in logarithmic time with a dichotomy search. But can we achieve constant time answering ?

One way to do so, would be to use a quadratic amount of memory to store a matrix representation of $P^{\mathbf{D}}$. This means than with n elements in the domain D of $\mathbf{D}$, if a is the i-th element and b the j-th, we can write a 1 in register R_{ni+j} if (a, b) is in $P^{\mathbf{D}}$ and write a 0 otherwise. Using this representation, we can answer the testing problem in constant time. The downside of the method is that the preprocessing requires a quadratic amount of space. In Section 4.1, we present a more complicated algorithm that answers the testing problem for this query in constant time while only requiring a preprocessing pseudo-linear in time and space.

This concludes the example with the query $q(x, y) = \neg P(x, y)$. We now state a key lemma that is often used in enumeration algorithm.

Lemma 2.5.7. *Let $\boldsymbol{D}$ be a database and φ_1, φ_2 a pair of queries with same arity. If after some preprocessing we can enumerate $\varphi_1(\boldsymbol{D})$ and $\varphi_2(\boldsymbol{D})$ with constant delay and increasing order, then we can enumerate $\varphi_1(\boldsymbol{D}) \cup \varphi_2(\boldsymbol{D})$ with constant delay and increasing order.*

Proof. We give a pseudo-code algorithm that enumerates $\varphi_1(\mathbf{D}) \cup \varphi_2(\mathbf{D})$. The algorithm only describes the enumeration phase. For the preprocessing, we just compute the two preprocessing independently.

Algorithm 3 Enumeration of disjunctions

```
1:  a ← First(φ1(D))
2:  b ← First(φ2(D))
3:  while a ≠ Null ∧ b ≠ Null do
4:      if a < b then
5:          Output a
6:          a ← Next(φ1(D))
7:      else if b < a then
8:          Output b
9:          b ← Next(φ2(D))
10:     else
11:         Output a
12:         a ← Next(φ1(D))
13:         b ← Next(φ2(D))
14:     end if
15: end while
```

□

2.5.3 Comparisons

In this section, we compare the different problems defined in Section 2.5.1. The goal is to show that some problems are easier than others. When comparing these problems, one of the difficulties is that the model checking problem only considers boolean queries. Therefore, we want to work with query languages where the boolean and non-boolean queries are not too different from one another.

Remark 2.5.8. We do the proofs that follow for FO queries as this query language is at the core of this thesis. Some of the proofs works for any query language and this will be mentioned. For some proof to work we need, given any boolean query q in $\mathcal{L}$, the unary query $q'(x) := q$ to also be in $\mathcal{L}$. Here, the solutions for $q'(x)$ can either be, the whole domain of the database if q is satisfied, or the empty set if q is not satisfied.

This is the case for FO and MSO queries, as well as conjunctive queries and most natural restrictions of FO queries.

Counting and model checking

The theorem that follows states that the model checking problem is easier than the counting problem.

Theorem 2.5.9. *Let $\mathcal{C}$ be a class of databases. If the* FO *counting problem over $\mathcal{C}$ can be performed in linear time (resp. pseudo-linear or polynomial time), then the* FO *model checking problem over $\mathcal{C}$ can be performed in linear time (resp. pseudo-linear or polynomial time).*

Proof. Let $\mathcal{C}$ be a class of databases. Assume that the FO counting problem over $\mathcal{C}$ can be performed in linear time (resp. pseudo-linear or polynomial time). Let q be a boolean query in FO and $\mathbf{D}$ a database in $\mathcal{C}$. We define $q'(x) := q$. Since $q'(x)$ is in FO, we can count the number of solutions $|q'(\mathbf{D})|$ in linear time (resp. pseudo-linear or polynomial time). We then have that:

$$\mathbf{D} \models q \quad \text{if and only if} \quad |q'(\mathbf{D})| > 0,$$

where D is the domain of $\mathbf{D}$. Hence, we can answer the model checking problem in linear time (resp. pseudo-linear or polynomial time). □

Evaluation and model checking

The theorem that follows states that the model checking problem is easier than the evaluation problem.

Theorem 2.5.10. *Let $\mathcal{C}$ be a class of databases. If the* FO *evaluation problem over $\mathcal{C}$ can be performed in linear time (resp. pseudo-linear or polynomial time), then the* FO *model checking problem over $\mathcal{C}$ can be performed in linear time (resp. pseudo-linear or polynomial time).*

The proof of Theorem 2.5.10 is quite similar to the proof of Theorem 2.5.9.

Proof. Let $\mathcal{C}$ be a class of databases. Assume that the FO evaluation problem over $\mathcal{C}$ can be performed in linear time (resp. pseudo-linear or polynomial time). Let q be a boolean query in FO and $\mathbf{D}$ a database in $\mathcal{C}$. We define $q'(x) := q$. Since $q'(x)$ is also in FO, we can compute the set of solution $q'(\mathbf{D})$ in linear time (resp. pseudo-linear or polynomial time). We then have that:

$$\mathbf{D} \models q \quad \text{if and only if} \quad q'(\mathbf{D}) \neq \emptyset,$$

where D is the domain of $\mathbf{D}$. Hence, we can answer the model checking problem in linear time (resp. pseudo-linear or polynomial time). □

Enumeration and evaluation

It is a little trickier to compare the enumeration problem to the others as we now have two parameters, the delay and the preprocessing, instead of just the running time. Nonetheless, we can prove that the enumeration problem is harder than the evaluation.

Theorem 2.5.11. *Let $\mathcal{L}$ be any query language and $\mathcal{C}$ a class of databases. If the enumeration problem of $\mathcal{L}$ over $\mathcal{C}$ can be performed with constant delay after a linear (or pseudo-linear) preprocessing, then the evaluation problem of $\mathcal{L}$ over $\mathcal{C}$ can be performed in linear (resp. pseudo-linear) time.*

And this remains true with larger delay.

Theorem 2.5.12. *Let $\mathcal{L}$ be any query language and $\mathcal{C}$ a class of databases. If the enumeration problem of $\mathcal{L}$ over $\mathcal{C}$ can be performed with polynomial delay after a polynomial preprocessing, then the evaluation problem of $\mathcal{L}$ over $\mathcal{C}$ can be performed in polynomial time.*

By Theorem 2.5.10 the FO evaluation problem is harder than the FO model checking problem. Hence we also have that the FO enumeration problem is harder that the FO model checking problem.

Corollary 2.5.13. *Let $\mathcal{C}$ be a class of databases. If the* FO *enumeration problem over $\mathcal{C}$ can be performed with constant delay after a linear (or pseudo-linear) preprocessing, then the* FO *model checking problem over $\mathcal{C}$ can be performed in linear (resp. pseudo-linear) time.*

Proof of Theorem 2.5.11 and 2.5.12. Let $\mathcal{C}$ be a class of databases and $\mathcal{L}$ any query language. Assume that the evaluation problem of $\mathcal{L}$ over $\mathcal{C}$ can be performed with constant delay after a linear (or pseudo-linear) preprocessing. Let $\mathbf{D}$ be a database in $\mathcal{C}$ and q a query in $\mathcal{L}$. In order to compute the set of all solutions, one can simply perform the preprocessing of the enumeration algorithm for q over $\mathbf{D}$ and then enumerate all solutions until there is no solution left.

The time needed to do that is first linear in the size of in input (resp pseudo-linear) and also linear in the number of produced solutions (since we enumerate them with constant delay).

With the same reasoning, if the preprocessing and the delay are polynomial, we get that the set of solutions is produced in time polynomial in both the size of the input and the size of the output. □

Testing, evaluation and model checking

Similarly to the enumeration problem, an algorithm for the testing problem has a preprocessing phase and an answering phase. The testing problem seams to be incomparable with the evaluation problem in general. However, they have the same difficulty when we restrict our attention to unary queries.

Theorem 2.5.14. *Let $\mathcal{L}$ be any query language and $\mathcal{C}$ a class of databases. The testing problem for unary queries in $\mathcal{L}$ over $\mathcal{C}$ can be answered in constant time after a linear (or pseudo-linear) preprocessing if and only if the evaluation problem of unary queries from $\mathcal{L}$ over $\mathcal{C}$ can be performed in linear (resp. pseudo-linear) time.*

The proof is again quite similar to the other proofs of this section.

Proof. Let $\mathbf{D}$ be a database and $q(x)$ a unary query. Assume first that there is an algorithm for the testing problem of q over $\mathbf{D}$ that answer in constant time after a linear (or pseudo-linear) preprocessing. One can then compute the set $q(\mathbf{D})$ as follows: First perform the preprocessing, then for every element a of the domain D of $\mathbf{D}$, test whether $\mathbf{D} \models q(a)$. If so, add a to the set of solutions, otherwise, proceed with the next element. The total amount of time required is linear (resp. pseudo-linear).

Assume now that there is an algorithm for the evaluation problem that compute the set $q(\mathbf{D})$ in linear (or pseudo-linear) time. An example algorithm for the testing problem is the following:

- First, compute the set $q(\mathbf{D})$.
- Then, use a RAM to store the result in the following way: Register R_i contains a 1 if the i-th element of D is in $q(\mathbf{D})$ and a 0 otherwise.

  This ends the preprocessing.
- Finally, when given an element a in D, we can check in the corresponding register whether a is in $q(\mathbf{D})$.

The preprocessing is linear (resp. pseudo-linear) and the answering phase can be performed in constant time. □

We now want to compare the testing problem and the model checking. Fortunately, in the proof of Theorem 2.5.11, we only use the fact that the evaluation problem can be performed for unary queries. Hence, we also have the following Corollary.

Corollary 2.5.15. *Let $\mathcal{C}$ be a class of databases. If the* FO *testing problem over $\mathcal{C}$ can be answered in constant time after a linear (or pseudo-linear) preprocessing, then the* FO *model checking problem over $\mathcal{C}$ can be performed in linear (resp. pseudo-linear) time.*

2.5.4 Conclusion

This concludes this section on the various query problems we encounter in this thesis. What we are looking for are classes of databases $\mathcal{C}$ and query languages $\mathcal{L}$ such that the testing problem, the enumeration problem and the counting problem can be preformed efficiently. By efficiently, we mean in linear (or pseudo-linear) time for the counting problem. For the testing or enumeration problems, we want a linear (or pseudo-linear) preprocessing followed by a answering phase performed in constant time or by an enumeration with constant delay.

Thanks to the previous results, we know that we can restrict our attention to classes $\mathcal{C}$ and $\mathcal{L}$ for which the model checking problem can be performed in linear (or pseudo-linear time). The model checking problem has been extensively studied, which helps us to narrow down the research of such classes.

2.6 Classes of Graphs

The notion of graphs plays an important role in this thesis. While every result is stated over classes of databases, the proofs only talk about classes of graphs. In Section 4.3, we

provide the standard technique used to lift a proof from graphs to databases classes. In Section 2.6.1 we define the notions used in graph theory in general. After that, we talk about specific classes of graphs. In Section 2.6.2 we define classes of trees and graphs with bounded tree-width. Then we introduce graphs with bounded degree in Section 2.6.3 and graphs with bounded expansion in Section 2.6.4. Finally, we present nowhere dense and somewhere dense classes of graphs in Section 2.6.5.

2.6.1 General definitions

A *graph* G is composed of a set of vertices (or nodes) V and a set of edges E. We can see graphs as databases over the specific schema $\sigma := \{E\}$ where E is a binary relation. A graph is said to be *undirected* if for every edge (a, b) where a and b are vertices, the pair (b, a) is also an edge. In the remainder of this document, we only consider undirected classes of graphs.

In an (undirected) graph $G = (V, E)$, a *path* is a sequence of distinct nodes $a_1, \dots a_l$ such that for every $i < l$, the pair (a_i, a_{i+1}) is an edge. Given two specific nodes a and b, an $(a - -b)$ path is a path whose end points are a and b. The distance between two nodes a and b is the length (the number of edges) in the shortest $(a - -b)$ path. If there is no path between a and b, we say that the distance between a and b is infinite. The *connected component* of a node a (in the underlying undirected graph) is the set of all nodes that are at a finite distance from a.

The notion of connected components can be refined to the notion of neighborhoods.

Definition 2.6.1. Given an (undirected) graph $G = (V, E)$ and an integer r, the *r-neighborhood* of a node a, noted $N_r^G(a)$ is the set of vertices that are at distance at most r from a.

Given a graph G, there are two ways of defining subgraphs of G.

Definition 2.6.2. Given a graph $G = (V, E)$, the graph $H = (V', E')$ is a *subgraph* of G if V' is a subset of V and E' is a subset of E.

Definition 2.6.3. Given a graph $G = (V, E)$, the graph $H = (V', E')$ is an *induced subgraph* of G if V' is a subset of V and $E' = \{(a, b) \in V'^2 \mid (a, b) \in E\}$.

In the reminder of this document, we study classes of graphs with some regularity.

Definition 2.6.4. A class of graphs $\mathcal{C}$ is *closed under subgraphs* if for every graph G in $\mathcal{C}$ and every subgraph H of G, we have that H is in $\mathcal{C}$.

Every needed notion about graphs is now defined. We can turn to specific (undirected) classes of graphs. We can see how those classes are included into one another in Figure 3.1

2.6.2 Classes of trees

A *tree* is a graph that does not contain cycles. Trees are heavily used in computer science. One characterization of trees, is that when removing a node from a tree, we are left with

several disconnected subtrees. This property allows the use of "divide and conquer" methods that lead to efficient algorithms. The notion of bounded tree-width generalized the notion of tree and such methods can still be applied.

Definition 2.6.5. Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair $(\mathcal{X}, T)$, where $\mathcal{X}$ is a family of subsets $(X_1, \ldots, X_l)$ of V and T is a tree whose nodes are the X_i and such that:

- $\bigcup_{i \leq l} X_i = V$,
- for every edge (a, b) in G, there is an $i \leq l$ such that a and b are in X_i, and
- for every node a in G the set of all X_i with a in X_i form a connected subtree of T.

The *width* of a decomposition (X, T) with $X = (X_1, \ldots, X_l)$ is the integer $\max_{i \leq l} |X_i| - 1$. The *tree-width* of a graph G is the smallest width amongst all possible tree decompositions. The tree-width of a graph G is noted $tw(G)$.

The tree-width measure how far a given graph is to being a tree. For example a tree has tree-width 1.

Definition 2.6.6. A class of graphs $\mathcal{C}$ has *bounded tree-width* if there is an integer d such that for every graph $G \in \mathcal{C}$, we have that $tw(G) \leq d$.

In Section 3.1.2 we present algorithmic results for classes of graphs with bounded tree-width.

2.6.3 Graphs with bounded degree

Given a graph $G = (V, E)$, and a node a in V the *degree* of a in G is the number of nodes that share an edge with a. The *degree* of a graph G is the maximal degree amongst the nodes of G.

Definition 2.6.7. A class of graphs $\mathcal{C}$ has *bounded degree* if there is an integer d such that for every graph $G \in \mathcal{C}$, we have that the degree of G does not exceed d.

2.6.4 Graphs with bounded expansion

The notion of a class of graphs with bounded expansion was introduced by Nešetřil and Ossona de Mendez in [58]. It generalizes the notions of bounded tree-width and bounded degree. It also generalizes other well known classes such as planar graphs or graphs that exclude a minor. The definition of bounded expansion we give here uses the notion of minor of a graph that we define first.

Definition 2.6.8. Let $G = (V, E)$ be an undirected graph and r be an integer. A graph $H = (V', E')$ is an *r-minor* of G if:

- $V' \subseteq V$,
- for every a_i in V', there is a set S_i in V such that:

- $S_i \subseteq N_r^G(a_i)$,
- for every $i \neq j$ we have $S_i \cap S_j = \emptyset$, and
- for every $i \neq j$ we have (a_i, a_j) is in E' if and only if in G there is an edge from a node in S_i to a node in S_j.

We note $H \in G \triangledown r$ the fact that H is an r-minor of G. We also note $H \in \mathcal{C} \triangledown r$ the fact that there is a graph G in $\mathcal{C}$ such that H is an r-minor of G. The notion of r-minor is also named *shallow minor* or *low depth minor*.

We can now define bounded expansion.

Definition 2.6.9. A class of graphs $\mathcal{C}$ has *bounded expansion* if there is a function f such that for all G in $\mathcal{C}$ and for all r in $\mathbb{N}$ we have:

$$\max_{H \in G \triangledown r} \frac{|E_H|}{|V_H|} \leq f(r).$$

In [58] many equivalent definitions of classes of graphs with bounded expansion are given. To see how those properties lead to efficient algorithms, interested readers are referred to [59].

2.6.5 Nowhere and somewhere dense classes of graphs

The notion of nowhere dense classes of graphs was introduced in [60] as a formalization of classes of "sparse" graphs and generalizes many well known classes of graphs such as graphs with bounded tree-width, graphs with bounded degree or bounded expansion [61]. Here we just give the basic definition of nowhere dense graphs. But since this notion is at the core of this thesis, we provide additional details and information in Section 4.2.

The first definitions of nowhere dense classes of graphs use again the notion of graphs minor.

Definition 2.6.10 ([60])**.** Let $\mathcal{C}$ be a class of graphs. $\mathcal{C}$ is *nowhere dense* if and only if for all $r \in \mathbb{N}$ there is a $N_r \in \mathbb{N}$ such that $K_{N_r} \notin \mathcal{C} \triangledown r$. Here, K_n is the clique with n elements i.e. a graph with n vertices and every possible edges.

Nowhere dense is a very robust notion that can be defined by many other equivalent properties [60]. Moreover, nowhere dense seems to be one of the broadest classes of graphs that admit good algorithmic properties.

In opposition to nowhere dense, we can define somewhere dense classes of graphs.

Definition 2.6.11 ([60])**.** A class of graphs $\mathcal{C}$ is *somewhere dense* if and only if it is not *nowhere dense*.

Note that none of those two definitions imply that a nowhere dense (or somewhere dense) class is closed under subgraphs. However, if a class is nowhere dense (resp. somewhere dense) then its closure by adding every possible subgraph remains nowhere dense (resp. somewhere dense). Furthermore, there are sharp dichotomy results that use those two notions.

Nowhere dense graphs admit rather good algorithmic properties and many otherwise intractable problem can efficiently be solved. We present some of those results in Section 3.1.3 and the purpose of this thesis is to increase the number of such results. On somewhere dense and closed under subgraphs classes of graphs, many different problems (like the FO model checking) are as hard as in the general case [55].

Chapter 3

State of the art

Contents

In this chapter we present the state of the art concerning enumeration algorithms. For the different studied cases, we also mention results concerning other problems such as the model checking, the counting or testing problems. This chapter can be divided into three distinct parts.

Section 3.1 and 3.2 only mention results that lie in the database querying framework. More precisely, Section 3.1 focuses on a quite traditional setting where a query and a database are given as input of the algorithm and are fixed once and for all. The core results of this thesis (presented in the following chapters) fall in this scenario. Section 3.2 focuses on a slightly different and rather new topic: databases subject to updates. Here the input database can be changed while the algorithm is solving one of the problems. The algorithm needs to take care of those changes, if possible without having to restart from scratch.

Section 3.3 contains two kinds of enumeration results. The results presented in the first part lie in a more abstract and theoretical point of view. For example, we mention some definitions of complexity classes for enumeration problem. The results presented in the second part lie in a more concrete framework. The goal of those results is to provide enumeration algorithms with real life applications.

The content of this chapter makes a great use of a survey on constant delay enumeration by Luc Segoufin [67] and of a PhD thesis by Wojciech Kazana [48].

3.1 Query answering for static databases

In this section, we investigate the core scenario of this thesis: For a given class $\mathcal{C}$ of databases and a query language $\mathcal{L}$, is there an algorithm which on input $\mathbf{D}$ in $\mathcal{C}$ and q in $\mathcal{L}$ efficiently enumerates the set $q(\mathbf{D})$ of solutions.

We are also interested in algorithms that count the number of solutions (for the counting problem), test efficiently whether a given tuple is a solution (for the testing problem) or simply say whether there is at least one solution (for the model checking problem). More precise definitions of those problems can be found in Chapter 2.5.

We first consider, in Section 3.1.1, algorithms that work for every relational structure. For this to happen, we need to consider really restricted classes of query languages.

After that, we consider several classes of databases allowing efficient algorithms for more general queries. We consider MSO queries in Section 3.1.2 and FO queries in Section 3.1.3.

3.1.1 Arbitrary relational structures

The first scenarios we consider are those where we do not restrict the database part of the input. Hence the results presented in this section answer the following question:

Which query languages enable efficient algorithms that work for every relational structure?

Conjunctive queries

A *conjunctive query* is a first order query of the form:

$$q(\overline{x}) := \exists y_1, \cdots y_l \bigwedge_i R_i(\overline{z}_i)$$

where for every i, R_i is a relational symbol and $\overline{z}_i$ is a tuple of variables from $\overline{x}$ and $\overline{y}$. Conjunctive queries are denoted CQ.

Conjunctive queries are quite a natural restriction but in our case it is not enough. More precisely we can show [64] that the parametrized model checking problem for CQ is W[1] complete. Since it is strongly believed that FPT $\neq$ W[1], it is highly unlikely that there exist an FPT algorithm for the model checking problem of CQ. Consequently, as having an efficient algorithm for the model checking problem is a mandatory step for the counting, the enumeration and the testing problems (see Section 2.5.3), we need more restricted queries.

Acyclic conjunctive queries

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

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

A conjunctive query is said to be *acyclic* if it has a join tree. For example, the query $q_1(x, y, z) := E(x, y) \wedge E(y, z)$ is acyclic. Here, a join tree can be a tree whose root is labeled "$E(x, y)$", and with only one leaf labeled "$E(y, z)$".
However, the query $q_2(x, y, z) := E(x, y) \wedge E(y, z) \wedge E(z, x)$ is not acyclic.

Acyclic conjunctive queries (or ACQ for short) are sufficiently restricted to obtained quite efficient algorithms.

Theorem 3.1.1 ([74]). *There is an algorithm which, upon input of a database $\boldsymbol{D}$ and an acyclic conjunctive query q, computes the set $q(\boldsymbol{D})$ in time $O(|q| \cdot \|\boldsymbol{D}\| \cdot |q(\boldsymbol{D})|)$.*

Note that the existence of an enumeration algorithm with constant delay and linear preprocessing would have implied a running time of the form $O(f(|q| \cdot (\|\mathbf{D}\| + |q(\mathbf{D})|))$ for the evaluation problem. We actually need even more restricted query languages to obtain algorithms with constant delay. However, there is an enumeration algorithm with longer delay.

Theorem 3.1.2 ([9]). *There is an algorithm which, upon input of a database $\boldsymbol{D}$ and an acyclic conjunctive query q, enumerates the set $q(\boldsymbol{D})$ with constant-time preprocessing and linear-time delay.*

Here, remember that by constant and linear, we implicitly mean "in the size of the database".

As far as we know, the testing problem has not been addressed for this setting. However, it is mentioned in Section 3.2.1, for the more general case of databases subject to updates. Nonetheless, the counting problem has been tackled. The combined complexity for counting ACQ is $\#P$-complete [65]. In [24] a new parameter was introduced to precisely define the classes within ACQ for which the counting problem is tractable. Intuitively, the *quantifier-star size* measures how far free variables are spread in the formula. It leads to the following dichotomy theorem.

Theorem 3.1.3 ([24]). *For every s in $\mathbb{N}$, 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 structure. However, if a class of ACQ does not have a bounded quantifier-star size, then its associated counting problem is $\#\mathrm{W}[1]$ hard. It is therefore not FPT unless $\#\mathrm{W}[1] = \mathrm{FPT}$.*

Free-connex acyclic conjunctive queries

An acyclic conjunctive query $q(\overline{x})$ is said to be *free-connex* if the query $q'(\overline{x}) := q(\overline{x}) \wedge R(\overline{x})$ is acyclic, where R is a new symbol.

Note that for boolean queries and unary queries, being acyclic already implies that they are free-connex. It is not the case for binary queries.

Example 3.1.4. *Consider the query: $q(x, y) = \exists w \exists z,\ E(x, w) \wedge E(y, z) \wedge B(z)$. It is free-connex because the query $q'(x, y) = \exists w \exists z,\ E(x, w) \wedge E(y, z) \wedge B(z) \wedge R(x, y)$ is still acyclic. A possible join tree for q' is a tree with root $R(x, y)$, a left child $E(x, w)$ and a right child $E(y, z)$ who also has a child, $B(z)$.*

Consider now an other query: $\Pi(x, y) = \exists z,\ E(x, z) \wedge E(z, y)$. Since $\Pi'(x, y) = \exists z,\ E(x, z) \wedge E(z, y) \wedge R(x, y)$ is clearly not acyclic, q is not free-connex.

It turns out that free-connex is a sufficient restriction to put on acyclic conjunctive queries in order to obtained constant delay enumeration.

Theorem 3.1.5 ([9]). *There is an algorithm which, upon input of a database $\boldsymbol{D}$ and a free-connex acyclic conjunctive query q, enumerates the set $q(\boldsymbol{D})$ with linear-time preprocessing and constant-time delay.*

The result of Theorem 3.1.5 also holds if the query contains inequalities [16].

Matching lower bounds

In [9], it is also proved that, when looking at conjunctive queries, free-connex is exactly what we need to obtain constant delay enumeration, assuming that boolean matrix multiplication cannot be done in quadratic time.

The boolean matrix multiplication is the problem defined as follow: given two $n \times n$ matrices with boolean entries M and N, the goal is to compute their product MN. The best known algorithms so far (based on the Coppersmith–Winograd algorithm [18]) require more than $n^{2.37}$ steps [38].

Theorem 3.1.6 ([9]). *If the boolean matrix multiplication problem cannot be solved in quadratic time, then for any self-join free, acyclic conjunctive query q the following statements are equivalent:*

- *q is free-connex,*
- *there is an algorithm which, upon input of a database $\mathbf{D}$, enumerates the set $q(\mathbf{D})$ with linear-time preprocessing and constant-time delay,*
- *there is an algorithm which, upon input of a database $\mathbf{D}$, compute the set $q(\mathbf{D})$ in time $O(\|\mathbf{D}\| + |q(D)|)$*

Here, a query is said to be self-join free *if no relation symbol is used more than once.*

The idea in the proof of this theorem is that the acyclic conjunctive query $\Pi(x, y)$ from our last example is actually encoding the boolean matrix multiplication problem. Furthermore, we can see that every acyclic conjunctive query that is not free-connex somehow contains a sub-query of the form of $\Pi(x, y)$.

X-underbar structures

The case of $\underline{X}$-structures does not easily fit in the picture we are trying to paint. The first scenario was about restricted queries and the next section about restricted databases. The case of $\underline{X}$-structures is a little bit different since it uses restrictions on both sides. Its use of conjunctive queries makes it relevant for this section.

A structure $\mathbf{D}$ is an *$\underline{X}$-structure* if there is a total order $<$ on the domain D of $\mathbf{D}$, if $\mathbf{D}$ does not contain relations of arities greater than 2, and if for every binary relation R in $\mathbf{D}$ we have:

$$\mathbf{D} \models \forall x_0, x_1, x_2, x_3 \, \big(R(x_0, x_1) \wedge R(x_2, x_3)\big) \rightarrow R\big(\min(x_0, x_2), \min(x_1, x_3)\big).$$

$\underline{X}$-structures have been introduced in the context of XML documents. When a traversal order for a tree is given, some of the usual axis relations (parent, child, ascendant, descendant, siblings, etc) may satisfy the property of being $\underline{X}$-structures for this order. They prove in [8] that query evaluation and enumeration become algorithmically easier when such property is satisfied.

Theorem 3.1.7 ([8]). *There is an algorithm which, upon input of an $\underline{X}$-structure $\mathbf{D}$ and a conjunctive query q, enumerates the set $q(\mathbf{D})$ with constant-time preprocessing and linear-time delay.*

Theorem 3.1.8 ([8]). *There is an algorithm which, upon input of an $\underline{X}$-structure $\mathbf{D}$ and a acyclic conjunctive query q, enumerates the set $q(\mathbf{D})$ with linear-time preprocessing and delay linear in the size of the domain D of $\mathbf{D}$.*

3.1.2 Highly expressive queries (MSO)

We now investigate cases where the query part of the input problem is rather broad. The class of queries we consider here is monadic second order logic (MSO queries). The precise definition can be found in Section 2.4.

Upper bounds

When dealing with MSO queries, it is quite natural to look at graphs with bounded tree-width (see Section 2.6). The first result is an algorithm that solves the model checking problem, also known as Courcelle's theorem.

Theorem 3.1.9 (Courcelle's theorem [19]). *The model checking problem for MSO queries over classes of structures of bounded tree-width can be solved in linear time.*

Following this first result, other problems where found to be tractable over instances of bounded tree-width for MSO queries.

Theorem 3.1.10 ([4]). *The counting problem for MSO queries over classes of structures of bounded tree-width can be solved in linear time.*

The enumeration problem is also tractable. However with MSO queries we could potentially encounter free set variables. This hardly allows constant delay enumeration since the size of a solution might be as large as the input size. Therefore just writing one solution could require linear time.

Theorem 3.1.11 ([6, 20]). *Let $\mathcal{C}$ be a class of structures with bounded tree-width, there is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an MSO query q, enumerates the set $q(\boldsymbol{D})$ with linear-time preprocessing and delay linear in the size of the input.*

For a shorter delay, we then impose that only first order variables are allowed to be free (set quantification inside the formula is of course allowed).

Theorem 3.1.12 ([6, 51]). *Let $\mathcal{C}$ be a class of structures with bounded tree-width, there is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an MSO query q with only first order free variables, enumerates the set $q(\boldsymbol{D})$ with linear-time preprocessing and constant-time delay.*

Additional details can be found in two theses, [7] and [48].

An other approach consists in having an output tape on which the current output is written. 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 [26].

Lower bounds

As every theorem above only works if the input structure has bounded tree-width, the natural question is then to ask whether the previous results can be extended to classes of structures with unbounded tree-width.

It is believed that Courcelle's theorem is close to optimal.

Conjecture 3.1.13 (Grohe's Conjecture 8.3 from [42]). *Let $\mathcal{C}$ be a class of graphs that is closed under taking subgraphs. Suppose that the tree-width of $\mathcal{C}$ is not poly-logarithmically bounded, that is, there is no constant c such that $tw(G) \leq \log^c |G|$ for every $G \in \mathcal{C}$.*

Then the model checking problem for MSO queries over $\mathcal{C}$ is not FPT.

A similar result has been proven. It relies on the hypothesis that there are no algorithm solving the SAT problem working in sub-exponential time. This is called the *exponential-time hypothesis* (ETH) and is rather classical in this domain.

Theorem 3.1.14 ([54]). *Let $\mathcal{C}$ be a constructible class of Γ-colored graphs, closed under colorings. If the tree-width of $\mathcal{C}$ is strongly unbounded poly-logarithmically, the model checking for MSO queries over $\mathcal{C}$ is not FPT, unless ETH fails.*

For the precise definitions of constructible and strongly unbounded poly-logarithmically in this context, the readers are referred to [54].

3.1.3 FO queries and sparse structures

In this section we consider first-order queries (FO) and study classes of databases for which the various query evaluation problems are tractable. All the classes of databases considered are defined over graphs and are generalized to arbitrary relational structures via their Gaifman or adjacency graphs. More details about going from relational structures to graphs can be found in Section 4.3. Every result we present in this section can be seen in Figure 3.1.

We are going to consider classes of databases that are more and more general, following the history of the research in this area.

Bounded degree

The first restriction that we are going to consider is the bounded degree case. In this scenario, the first result deals with the model checking problem.

Theorem 3.1.15 ([66]). *Fix $d \in \mathbb{N}$. The problem of whether a given structure $\boldsymbol{D}$ of degree bounded by d satisfies a given first-order sentence φ is decidable in time $2^{2^{2^{O(|\varphi|)}}} \|D\|$.*

In order to lift this result to an enumeration algorithm with constant delay, two fundamental properties of structures of degree d can be used.

The first property of those structures is that for a given radius r in $\mathbb{N}$, a node of the graphs can only have only finitely many (up to isomorphism) possible r-neighborhoods. Given query $\varphi \in$ FO, the Gaifman Locality Theorem [35] tells us that only the r-neighborhoods types are relevant, where r depends only on φ. 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 [49].

Before that, another property of structures of degree d that has been used is that they can be encoded using bijective unary functions. Using this bijective encoding, it is then possible to obtain a quantifier elimination procedure for FO queries. This procedure is designed in such a way that it returns the quantifier free formula in a very special form. It has been shown in [23] that this property leads to an enumeration algorithm with constant delay.

Theorem 3.1.16 ([23, 49]). *Fix $d \in \mathbb{N}$, there is an algorithm which, upon input of a structure $\boldsymbol{D}$ of degree bounded by d and an* FO *query q, enumerates the set $q(\boldsymbol{D})$ with constant-time delay after a preprocessing performed in time $2^{2^{2^{O(|\varphi|)}}} \|D\|$.*

Bounded degree databases also enable fast algorithms for the other problems.

Theorem 3.1.17 ([23]). *Fix $d \in \mathbb{N}$, there is an algorithm which, upon input of a structure $\boldsymbol{D}$ of degree bounded by d and an* FO *query q, performs a preprocessing in time $2^{2^{2^{O(|\varphi|)}}} \|D\|$ such that afterwards, upon input of a tuple $\overline{a}$, decides in time $O(1)$ whether $\overline{a} \in q(\boldsymbol{D})$.*

Theorem 3.1.18 ([10]). *Fix $d \in \mathbb{N}$, there is an algorithm which, upon input of a structure $\boldsymbol{D}$ of degree bounded by d and an* FO *query q, computes the number of solutions $|q(\boldsymbol{D})|$ in time $2^{2^{2^{O(|\varphi|)}}} \|D\|$.*

In all of the theorems above, we explicitly mentioned the impact of the size of the query on the preprocessing time. In all of those cases, there is a threefold exponential impact. We can show that it cannot be wildly improved unless FPT $=$ AW$[*]$.

Theorem 3.1.19 ([34]). *Let $\mathcal{C}$ be the class of all graphs with degree at most 2 and p a polynomial. If* FPT $\neq$ AW$[*]$*, then there is no algorithm for the model checking of* FO *queries over $\mathcal{C}$ that works in time $2^{2^{o(|\varphi|)}} \cdot p(\|D\|)$.*

Bounded expansion

We now turn to the class of structures with bounded expansion (see Section 2.6 for necessary definitions). In [58], a number of equivalent definitions of this class were shown, giving evidence that this class is very robust. It turns out that many known families of structures have bounded expansion. We list some notable examples below.

- Class of graphs with bounded degree.
- Class of graphs with bounded tree-width.
- Class of planar graphs.
- Class of graphs excluding at least one minor.

As usual, the starting point is the model checking problem.

Theorem 3.1.20 ([27]). *The model checking problem for* FO *queries over classes of structures of bounded expansion can be solved in linear time.*

See [50] for an alternative proof.

Both proofs use the functional representation of the input graph to obtain the quantifier elimination procedure for FO over the class of graphs with bounded expansion. The major difference between the proofs of [27] and the one of [50] is that while the first one is based on the low tree depth coloring characterization the latter is based on transitive fraternal augmentations. Low tree depth coloring and transitive fraternal augmentations are two very different characterizations of bounded expansion, cf. [58].

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

Theorem 3.1.21 ([50]). *Let $\mathcal{C}$ be a class of graphs with bounded expansion. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, enumerates the set $q(\boldsymbol{D})$ with constant-time delay after a linear-time preprocessing. Moreover, the output is returned in a lexicographical order.*

The proof above is performed in two times. The first part is a quantifier elimination. The authors show that we can compute a new query without quantifier and a new structure, without impacting the set of solutions. The second part is then an enumeration algorithm for quantifier free queries.

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.

Theorem 3.1.22 ([50]). *Let $\mathcal{C}$ be a class of graphs with bounded expansion. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, performs a linear-time preprocessing such that afterwards, upon input of a tuple $\overline{a}$, decides in time $O(1)$ whether $\overline{a} \in q(\boldsymbol{D})$.*

Theorem 3.1.23 ([50, 62]). *Let $\mathcal{C}$ be a class of graphs with bounded expansion. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, computes the number of solutions $|q(\boldsymbol{D})|$ in linear time.*

Additional details can be found in Wojciech Kazana's PhD thesis: [48].

Locally bounded tree-width

Classes of graphs with locally bounded tree-width are class of graphs that generalize classes of graphs with bounded degree or bounded tree-width. It fits in the more global scheme consisting of searching for classes of graphs as wide as possible that still admit good algorithmic properties.

Definition 3.1.24. A class $\mathcal{C}$ of graphs is said to have *locally bounded tree-width* if there is a function f such that for every r in $\mathbb{N}$, every r-neighborhood that occurs in a graph of $\mathcal{C}$ has tree-width bounded by $f(r)$.

More precisely, for all graphs G in $\mathcal{C}$, every r in $\mathbb{N}$ and every a in G, $tw\big(N_r^G(a)\big) \leq f(r)$.

If the notion of locally bounded tree-width might not be relevant for real life databases and applications, it is (from the theoretical point of view) quite a natural restriction to study as explained in the following reasoning:

Since, by Gaifman's theorem, every FO queries only talks about neighborhoods, classes of graphs whose neighborhoods admit efficient query evaluation algorithms should admit good algorithms too.

This reasoning leads to the following results:

Theorem 3.1.25 ([33]). *The model checking problem for* FO *queries over classes of structures with locally bounded tree-width can be solved in pseudo-linear time.*

The definition of pseudo-linear can be found in Section 2.3.

The simplified presentation the above result hides several difficulties. On the intuitive level, the two main pieces are indeed Gaifman's locality theorem [35] and Courcelle's theorem (Theorem 3.1.9) restricted to first-order queries. However, to obtain a linear algorithm, one cannot study every neighborhood of a given graph, since the sum of the sizes of each neighborhood could be quadratic in the size of the input structure. To get around this difficulty, Frick and Grohe made a great use of neighborhood covers, a notion that will also be crucial to us later on.

The task becomes even more difficult when dealing with queries with free variables (for the counting, testing or enumerating problems) as a given solution might be spread across the graphs and aggregate the fragments of solutions is not easy.

In [32], Frick manages to compute this patchwork of pieces of solutions for a slightly more restricted notion than locally bounded tree-width.

Theorem 3.1.26 ([32])**.** *Let $\mathcal{C}$ be a class of graphs nicely locally tree-decomposable. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, performs a linear-time preprocessing such that afterwards, upon input of a tuple $\overline{a}$, decides in time $O(1)$ whether $\overline{a} \in q(\boldsymbol{D})$.*

Theorem 3.1.27 ([32])**.** *Let $\mathcal{C}$ be a class of graphs nicely locally tree-decomposable. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, computes the number of solutions $|q(\boldsymbol{D})|$ in linear time.*

See [32] for the precise definition of nicely locally tree-decomposable. Intuitively, it means that any graph can be decomposed into bags such that:

- every neighborhood is included in at least one bag,
- the bags have bounded tree width, and
- the bags do not overlap too much.

The enumeration problem has not been tackled in this scenario bus it is studied in the following one.

Locally bounded expansion

Following the reasoning that classes of graphs whose neighborhoods admit efficient query evaluation algorithms should admit good algorithms too, and since graphs with bounded expansion generalize graphs with bounded tree-width, it is natural to look at classes of graphs with locally bounded expansion.

Definition 3.1.28. Let $\mathcal{C}$ be a class of graphs. For every r in $\mathbb{N}$, we define:

$$\mathcal{C}_r := \{N_r^G(a) \mid G \in \mathcal{C}, a \in G\}.$$

We say that $\mathcal{C}$ has *locally bounded expansion* if for every r in $\mathbb{N}$, the class $\mathcal{C}_r$ has bounded expansion.

Following the ideas from the locally bounded tree-width case, we obtained the following theorems:

Theorem 3.1.29 ([27, 68]). *The model checking problem for* FO *queries over classes of structures with locally bounded expansion can be solved in pseudo-linear time.*

Theorem 3.1.30 ([68]). *Let $\mathcal{C}$ be a class of graphs with locally bounded expansion. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, performs a pseudo-linear-time preprocessing such that afterwards, upon input of a tuple $\overline{a}$, decides in time $O(1)$ whether $\overline{a} \in q(\boldsymbol{D})$.*

Theorem 3.1.31 ([68]). *Let $\mathcal{C}$ be a class of graphs with locally bounded expansion. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, computes the number of solutions $|q(\boldsymbol{D})|$ in pseudo-linear time.*

The enumeration problem requires some extra work. To be able to jump in constant time from a solution to an other, we developed a "skip pointer" mechanism. It is inspired by similar subroutines that appears in [25] and [50]. It is presented in more details in Chapter 6 that deals with the enumeration problem.

Theorem 3.1.32 ([68]). *Let $\mathcal{C}$ be a class of graphs with locally bounded expansion. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, enumerates the set $q(\boldsymbol{D})$ with constant-time delay after a pseudo-linear-time preprocessing. Moreover, the solutions are produced in lexicographical order.*

Nowhere dense and somewhere dense classes of graphs

The results from the two previous sections provide algorithms for more and more generic classes of graphs, however, the question of matching lower bounds remains unanswered. This is where the notion of nowhere dense and somewhere dense classes of graphs come into play.

Those classes are defined in Chapter 2.6 and more information about nowhere dense classes of graphs can be found in Section 4.2.

In [61], Nešetřil and Ossona de Mendez defined these notions in such a way that every class of graphs that is closed under subgraphs is either nowhere dense or somewhere dense. While somewhere dense classes of graphs are as difficult to deal with than completely generic classes of graphs, nowhere dense classes seem to admit quite efficient algorithms for various problems.

Theorem 3.1.33 ([55]). *The model checking problem over somewhere dense classes of graphs that are closed under subgraphs is* AW[$*$] *complete, and therefore very unlikely in* FPT.

An FPT algorithm for the model checking problem is mandatory for a constant delay enumeration algorithm (and similarly for the testing and counting problem) with the reasoning presented in Section 2.5. There is therefore almost no hope for an efficient algorithm for the enumeration problem (or counting and testing ones) over somewhere dense classes of graphs.

Unlike somewhere dense classes of graphs, nowhere dense ones admit a pseudo linear algorithm for the model checking problem.

Theorem 3.1.34 ([44]). *Let $\mathcal{C}$ be a nowhere dense class of graphs. For every* FO *formula φ and $\epsilon > 0$, there is an algorithm that upon input of a graph G in $\mathcal{C}$ decides whether $G \models \varphi$ in time $O(|G|^{1+\epsilon})$.*

This result and more specifically the tools used to proved it will be really useful to us. Since those tools are going to be heavily used in the subsequent Chapters, we spend some time describing them in details in Chapter 4 (More precisely in Sections 4.2 and 4.4). In this thesis, we extend these results to others problems. We present the testing problem in Chapter 5, the enumeration problem in Chapter 6 and the counting problem in Chapter 7. An other proof for the counting problem can be found in [45].

Low degree

Structures with low degree are not closed under subgraphs. They are neither somewhere or nowhere dense. However, structures with low degree can still be considered as "sparse".

Definition 3.1.35. A class $\mathcal{C}$ of structures has *low degree* if for every $\epsilon > 0$, there is a rank N in $\mathbb{N}$ such that for every G in $\mathcal{C}$, if $|G| > N$ then $d(G) \leq |G|^{\epsilon}$, where $d(G)$ is the degree of G.

The most classical example of structures with low degree is the class of all structures with degree $\log(n)$ (where n is the number of vertex of the structure). Since this class of graphs contains all graphs of the form:

- a clique of size k, and
- 2^k independent nodes,

it is clear that such classes may contains arbitrarily large cliques and therefore are not nowhere dense. However, such classes of graphs still admit good algorithms.

Theorem 3.1.36 ([41]). *Let $\mathcal{C}$ be a class of graphs with low degree. For every* FO *formula φ and $\epsilon > 0$, there is an algorithm that upon input of a graph G in $\mathcal{C}$ decides whether $G \models \varphi$ in time $O(|G|^{1+\epsilon})$.*

This result uses some ideas that where at the core of Theorem 3.1.15 i.e. the locality of first-order logic. The difficulty when generalizing this to structures with low degree is that a given node does not have a neighborhood of bounded size but only of "small size". This becomes even more challenging for the enumeration problem where the goal is to obtain constant delay enumeration (and not "small" or "low" delay). Fortunately, it is possible to work around this main issue.

Theorem 3.1.37 ([25]). *Let $\mathcal{C}$ be a class of graphs with low degree. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, enumerates the set $q(\boldsymbol{D})$ with constant-time delay after a pseudo-linear-time preprocessing. Moreover, the output is returned in a lexicographical order.*

The counting and testing problems follow.

Theorem 3.1.38 ([25]). *Let $\mathcal{C}$ be a class of graphs with low degree. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, performs a pseudo-linear-time preprocessing such that afterwards, upon input of a tuple $\overline{a}$, decides in time $O(1)$ whether $\overline{a} \in q(\boldsymbol{D})$.*

Theorem 3.1.39 ([25]). *Let $\mathcal{C}$ be a class of graphs with low degree. There is an algorithm which, upon input of a structure $\boldsymbol{D}$ in $\mathcal{C}$ and an* FO *query q, computes the number of solutions $|q(\boldsymbol{D})|$ in pseudo-linear time.*

Figure 3.1: FO query answering and sparse structures

3.1.4 Discussions

Before moving on to Section 3.2, we spend some time discussing two things that have only been briefly mentioned in this first Section. First, we make more explicit the impact of the sizes of the queries on the running time of the algorithms. After that we provide more information about the notion of pseudo-linear.

Constant factors

In every algorithm presented in this section, the size of the query is labeled as constant and hidden behind the big O.

In the case of free-connex acyclic conjunctive queries over arbitrary structures, the preprocessing is performed in time $O(f(|q|) \cdot \|\mathbf{D}\|)$, where $f(\cdot)$ is a simply exponential function.

The only other case where the constant factor does not blow up is the case of graphs with bounded degree over FO queries. In this case, the preprocessing is performed in time $O(f(|q|) \cdot \|\mathbf{D}\|)$, where $f(\cdot)$ is a triply exponential function. This makes the algorithm hard to use in practice for queries that are not too small. Unfortunately, it cannot be widely improved.

Theorem 3.1.40 ([34]). *The model checking problem for* FO *queries over classes of graphs with bounded degree cannot be solved in time* $2^{2^{2^{O(|q|)}}} \cdot \|\boldsymbol{D}\|$, *unless* FPT $=$ AW[$*$].

Nonetheless, nothing forbids the existence of an algorithm where the constant factor lies between doubly and triply exponential.

For every other cases, the constant factor is a tower of exponentials whose size depends on the size of the query. This is one of the reasons why those algorithms are not likely to be implemented. Here again, we cannot expect much improvement.

Theorem 3.1.41 ([34]). *The model checking problem for* FO *queries over a class* $\mathcal{C}$ *of graphs that contains all trees cannot be solved in time* $O(f(|q|) \cdot \|\boldsymbol{D}\|)$ *for an elementary function* $f(\cdot)$, *unless* FPT $=$ AW[$*$].

Since nowhere dense classes of graphs, but already classes of graphs with bounded tree-width or bounded expansion have the property of generalizing trees, an algorithm with smaller constants for those classes is not likely to exists.

Pseudo-linear time

The other thing we want to discuss is the use of pseudo-linear time for classes of graphs with locally bounded tree-width and beyond. It should be noted that for smaller classes (i.e. graphs with bounded degree, bounded tree-width ...) the number of edges is linear in the number of vertices. This means that for such structure $\mathbf{D}$, we have $\|\mathbf{D}\| = O(|D|)$, where D is the domain of $\mathbf{D}$. This is also the case for classes of graphs with bounded expansion. This is not the case for locally bounded tree-width, locally bounded expansion and nowhere dense classes of graphs. For those classes, the number of edges can exceed the linear bound. However, the number of edges cannot be more than pseudo-linear in

the number of vertices. For such class $\mathcal{C}$, we have that, for every $\epsilon > 0$, there is a size N_ϵ in $\mathbb{N}$ such that for every $\mathbf{D}$ in $\mathcal{C}$, if $|D| > N_\epsilon$, then $\|\mathbf{D}\| = O(|D|^{1+\epsilon})$.

Therefore, an algorithm whose running time is $O(|D|^{1+\epsilon})$ seems optimal. However, it does not implies that the running time is $O(\|\mathbf{D}\|)$. It would be interesting to see whether the algorithms presented in Section 3.1.3 whose running time are pseudo-linear can actually be improved to be linear in the size of the structure.

3.2 Query answering for databases subject to local updates

In this section, we investigate a scenario that is not exactly the core of this thesis. Assume that you have a class $\mathcal{C}$ of databases and a query language $\mathcal{L}$ that (according to Section 3.1) enable efficient algorithms for the model checking, enumerating, counting or testing problem. One can therefore perform some preprocessing toward solving one of those problems, but once this has been computed, what happens if a small change occurs in the database? If the database remains in class $\mathcal{C}$, one solution would be to recompute the desired preprocessing from scratch. We are going to investigate some scenario where, when a small change occurs, one can reproduce the result (or recompute the index provided by the preprocessing) much quicker.

In this section, by *local updates*, we mean changes in the database that only affect one tuple. It could, be the deletion of a tuple from one of the relations, or the adding of a tuple to one of the relations. This might add or remove one or several elements from the domain of the database. For the specific topic of graphs, a local update can be the creation or deletion of an edge or a vertex, or a color modification of a given node.

In this setting, the *update time* is the time needed to recompute the result (for the counting and model checking problems) or to recompute the index built during the preprocessing time (for the testing and enumerating problems). Ideally, we would like update times to be constant or logarithmic in the size of the input structure.

3.2.1 Arbitrary relational structures

Here, we want to find how much we have to restrict our query language to obtain efficient algorithms that work for every relational structures. However, we also want those algorithms to support local updates. Therefore, we need to restrict our query languages at least as much as in the static case.

Keeping this in mind, we know that we have to look at acyclic conjunctive queries for the model checking problem and free-connex acyclic conjunctive queries for the other problems. It turns out that we actually need to restrict our attention to even simpler queries.

Definition 3.2.1. A conjunctive query φ is *q-hierarchical* if for any two variables x, y of φ, the following is satisfied:

1. atoms $(x) \subseteq$ atoms (y) or atoms $(x) \supseteq$ atoms (y) or atoms $(x) \cap$ atoms $(y) = \emptyset$, and
2. if atoms $(x) \subseteq$ atoms (y) and x is a free variable in φ, then y is a free variable too.

Here, *atoms (x)* refers to the set of atoms in φ where x occurs.

This notion is more restricted than the notion of free-connex acyclic conjunctive queries.

Example 3.2.2. *The query $q_1(w,x,y,z) := E(w,x) \wedge E(x,y) \wedge E(y,z)$ is free-connex, however, the atoms that contains x and those containing y are incomparable and not disjunct. Hence q_1 is not q-hierarchical due to the first rule of the definition above.*

The query $q_2(x) := E(x,y) \wedge T(y)$ is again free-connex, but the set of atoms contains x is strictly included in the set of atoms containing y. Therefore q_2 is not q-hierarchical due to the second rule.

Finally, the query $q_3(x) := T(x) \wedge E(x,y)$ is q-hierarchical. Even though q_2 and q_3 seems quite similar, only the second one can be enumerated on any database subject to local updates.

Theorem 3.2.3 (Model checking with updates [12])**.** *Let φ be a boolean conjunctive query and $\boldsymbol{D}$ a database. If the homomorphic core of φ is q-hierarchical, then the query can be answered with linear preprocessing time and constant update time.*

Otherwise, unless the OMv-conjecture fails, there is no algorithm that answers φ with arbitrary preprocessing time and $O(\|\boldsymbol{D}\|^{1-\epsilon})$ update time.

The *homomorphic core* of a conjunctive query q is, roughly speaking, the simplest conjunctive query to be syntactically equivalent to q. The *OMv-conjecture* says that the boolean matrix multiplication where one of the matrix is given column by column cannot be computed in time truly subcubic. Precise definitions and additional details about those notions can be found in [12].

It appears that the notion of q-hierarchical queries is also the restriction needed for the enumeration and counting problems.

Theorem 3.2.4 (Enumeration with updates [12])**.** *Let φ be a self-join free conjunctive query and $\boldsymbol{D}$ a database. If φ is q-hierarchical, then after a linear time preprocessing phase the query result $\varphi(\boldsymbol{D})$ can be enumerated with constant delay and constant update time.*

Otherwise, unless the OMv-conjecture fails, for every $\epsilon > 0$, there is no algorithm that enumerates arbitrary self-join free conjunctive query φ with arbitrary preprocessing time, and $O(\|\boldsymbol{D}\|^{1-\epsilon})$ delay and update time.

Here again, self-join free means that no relation symbol is used more than once in the query. This restriction is not needed for the upper bound to work but it is heavily used in the lower bound part of the theorem.

Theorem 3.2.5 (Counting with updates [12])**.** *Let φ be a conjunctive query and $\boldsymbol{D}$ a database. If φ is q-hierarchical, then the number $|\varphi(\boldsymbol{D})|$ of tuples in the query result can be computed with linear preprocessing time and constant update time.*

Otherwise, assuming the OMv-conjecture and the OV-conjecture, for every $\epsilon > 0$, there is no algorithm that computes $|\varphi(D)|$ with arbitrary preprocessing time and $O(\|\boldsymbol{D}\|^{1-\epsilon})$ update time.

The precise statement of the OV-conjecture (not too different from the OMv-conjecture's) can be found in [12].

However, the testing problem is not mentioned in [12]. It seems that the testing problem behaves a little bit differently than the other ones. It is quite easy to see that the query: $q(x, y) := S(x) \wedge E(x, y) \wedge T(y)$ can be tested efficiently even though it is not q-hierarchical [14].

This leads to the notion of t-hierarchical conjunctive queries.

Definition 3.2.6. A conjunctive query φ is *t-hierarchical* if the following is satisfied:

1. for any two quantified variables x, y of φ, we have
   atoms $(x) \subseteq$ atoms (y) or atoms $(x) \supseteq$ atoms (y) or atoms $(x) \cap$ atoms $(y) = \emptyset$, and

2. for any free variable x and quantified variable y, we have
   atoms $(x) \supseteq$ atoms (y) or atoms $(x) \cap$ atoms $(y) = \emptyset$,

where *atoms (x)* refers to the set of atoms in φ where x occurs.

This notion generalizes q-hierarchical. The only difference with q-hierarchical is that the first item now only considers quantified variables. Moreover, this new notion is incomparable with the notion of acyclic free-connex that were used in the static case.

Example 3.2.7. *The query $q_1(x, y) := S(x) \wedge E(x, y) \wedge T(y)$ is t-hierarchical while not being q-hierarchical. More generally every quantifier free conjunctive query is t-hierarchical (even not acyclic ones). Thus, some queries like $q_2(x, y, z) := E(x, y) \wedge E(y, z) \wedge E(z, x)$ are t-hierarchical but not acyclic free-connex.*

However, when only looking at acyclic conjunctive queries, t-hierarchical queries are necessarily free-connex. The opposite is not true since $q_3(x, y) := E(x, y) \wedge T(y)$ is not t-hierarchical (due to the second rule) while being acyclic and free-connex.

Theorem 3.2.8 (Testing with updates [14])**.**

1. *There is an algorithm with input a t-hierarchical conjunctive query φ and a database $\boldsymbol{D}$ that computes a data structure within linear time. This data structure can be updated in constant time. The data structure also allows, for an input tuple $\overline{a}$ in $\boldsymbol{D}$, to test whether $\overline{a} \in \varphi(\boldsymbol{D})$ within constant time.*

2. *Let $\epsilon > 0$ and let φ be a conjunctive query that is not equivalent to a t-hierarchical conjunctive query. There is no algorithm with arbitrary preprocessing time and $O(\|\boldsymbol{D}\|^{1-\epsilon})$ update time that can test, for any input tuple $\overline{a}$ in $\boldsymbol{D}$, whether $\overline{a} \in \varphi(\boldsymbol{D})$ within testing time $O(\|\boldsymbol{D}\|^{1-\epsilon})$, unless the OMv-conjecture fails.*

More recently, the same authors looked at queries that are not conjunctive and can therefore fall out of the previous dichotomy theorems. A query φ is in UCQ if it is of the form $\varphi = \varphi_1(\overline{x}_1) \vee \ldots \vee \varphi_d(\overline{x}_d)$ where every $\varphi_i(\overline{x}_i)$ is a conjunctive query. The notions can be extended and a query φ in UCQ is q-hierarchical (or t-hierarchical) if every $\varphi_i(\overline{x}_i)$ is q-hierarchical (or t-hierarchical).

The previous results extend to UCQ (almost) without restricting the query languages.

For the testing problem:

Theorem 3.2.9 ([14])**.**

1. *There is a dynamic algorithm with input a t-hierarchical UCQ φ and a database $\boldsymbol{D}$ that computes a data structure within linear time. This data structure can be updated in constant time. This data structure also allows, for an input tuple $\overline{a}$ in $\boldsymbol{D}$, to test whether $\overline{a} \in \varphi(\boldsymbol{D})$ within constant time.*

2. *Let $\epsilon > 0$ and let φ be a UCQ that is not equivalent to a t-hierarchical UCQ. There is no dynamic algorithm with arbitrary preprocessing time and $O(\|\boldsymbol{D}\|^{1-\epsilon})$ update time that can test for any input tuple $\overline{a}$ in $\boldsymbol{D}$ whether $\overline{a} \in \varphi(\boldsymbol{D})$ within testing time $O(\|\boldsymbol{D}\|^{1-\epsilon})$, unless the OMv-conjecture fails.*

For the enumeration problem:

Theorem 3.2.10 ([14]).

1. *There is a dynamic algorithm that receives a q-hierarchical UCQ φ, a database $\boldsymbol{D}$, and computes within linear time an index that can be updated in constant time and allows to enumerate the set $\varphi(\boldsymbol{D})$ with constant delay.*

2. *Let $\epsilon > 0$ and let φ be a UCQ whose homomorphic core is not q-hierarchical and is a union of self-join free conjunctive query. There is no dynamic algorithm with arbitrary preprocessing time and $O(\|\boldsymbol{D}\|^{1-\epsilon})$ update time that can enumerate the set $\varphi(\boldsymbol{D})$ with delay $O(\|\boldsymbol{D}\|^{1-\epsilon})$, unless the OMv-conjecture fails.*

And for the counting problem:

Theorem 3.2.11 ([14]).

1. *There is a dynamic algorithm that receives an exhaustively q-hierarchical UCQ φ, a database $\boldsymbol{D}$, and computes within linear time an index that can be updated in constant time and computes $|\varphi(\boldsymbol{D})|$ in constant time.*

2. *Let $\epsilon > 0$ and let φ be a UCQ that is not exhaustively q-hierarchical. There is no dynamic algorithm with arbitrary preprocessing time and $O(\|\boldsymbol{D}\|^{1-\epsilon})$ update time that can compute $|\varphi(\boldsymbol{D})|$ in time $O(\|\boldsymbol{D}\|^{1-\epsilon})$, unless the OMv-conjecture or the OV-conjecture fails.*

3.2.2 Highly expressive queries (MSO)

As for query answering in a static setting, we now investigate algorithms that works for MSO queries. Keeping in mind the results presented in section 3.1.2, we know that we have to look at classes of graphs with bounded tree-width. Some of the algorithms that have been developed consider even more restricted classes of structures like trees and words.

Concerning local updates, it seams that the most challenging ones are the insertion or deletion of edges. The first reason is because inserting or deleting edges might change the shape of the structure and that could for instance increase the tree-width of the structure or create cycle in a tree. When working with trees, the updates we consider are:

(i) Replace the current label of a specified node by another label,

(ii) insert a new leaf node after a specified node,

(iii) insert a new leaf node as first child of a specified node, and

(iv) delete a specified leaf node.

Like every scenario in this chapter, everything starts with the model checking.

Theorem 3.2.12 ([11]). *There is a dynamic algorithm which, upon input of an* MSO *query φ and a tree $\mathcal{T}$, computes within linear time a data structure that allows to test in constant time whether $\mathcal{T} \models \varphi$. When $\mathcal{T}$ is subject to local updates of the form (i), (ii), (iii) or (iv), the data structure can be recomputed in time $O(\log(n)^2)$.*

This algorithm can be extended to an enumeration one.

Theorem 3.2.13 ([57]). *There is a dynamic algorithm which, upon input of an* MSO *query φ and a tree $\mathcal{T}$, computes within linear time a data structure that allows to enumerate the set $\varphi(\mathcal{T})$ with delay $O(\log(n)^2)$. When $\mathcal{T}$ is subject to local updates of the form (i), (ii), (iii) or (iv), the data structure can be recomputed in time $O(\log(n)^2)$.*

Two different scenarios allows constant delay enumeration. The first one uses more restricted updates.

Theorem 3.2.14 ([3]). *There is a dynamic algorithm which, upon input of an* MSO *query φ and a tree $\mathcal{T}$, computes within linear time a data structure that allows to enumerate the set $\varphi(\mathcal{T})$ with constant delay. When $\mathcal{T}$ is subject to local updates of the form (i), the data structure can be recomputed in time $O(\log(n))$.*

In the other scenario, the class of structure is more restricted. Instead of trees, we now only talk about words. Since the notion of leaf node is irrelevant in this setting, we change the definitions of the local updates. We now consider the local updates:

(i') Replace the current label of a specified node by another label,

(ii') insert a new node at a specified position in the word, and

(iii') delete a specified node.

Theorem 3.2.15 ([63]). *There is a dynamic algorithm which, upon input of an* MSO *query φ and a word $\boldsymbol{W}$, computes within linear time a data structure that allows to enumerate the set $\varphi(\boldsymbol{W})$ with constant delay. When $\boldsymbol{W}$ is subject to local updates of the form (i'), (ii') or (iii'), the data structure can be recomputed in time $O(\log(n))$.*

Unlike the enumeration problem, the counting and testing problems have not been tackled. In the conclusion of [63], the authors stated that their data structure cannot answer in constant time whether a given tuple is a solution or not. They suggest that, in this setting, there might be a tread-off between answering the testing problem efficiently and being able to support local updates. It is not the case in the next section.

3.2.3 FO queries and sparse structures

We now turn to the scenario of FO queries and sparse structures. In the static setting, it was the most prolific one with many different cases studied and many different results. In the dynamic setting, only the case of structure with bounded degree has been addressed so far.

Here local updates must not insert edges at will, since it might increase the degree of the structures. For a fix d in $\mathbb{N}$, a *d-update* insert, delete or relabel a tuple from a structure $\mathbf{D}_1$ in order to obtain a structure $\mathbf{D}_2$ and the degree of $\mathbf{D}_2$ must be smaller or equal than d.

Theorem 3.2.16 ([13]). *For every integer d, there is a dynamic algorithm which, upon input of an FO+MOD query φ and a structure $\boldsymbol{D}$ of degree d, computes within linear time a data structure that:*

- *allows to test in constant time whether $\boldsymbol{D} \models \varphi$ (when φ is a boolean query),*
- *allows to compute in constant time $|\varphi(\boldsymbol{D})|$,*
- *allows to enumerate the set $\varphi(\boldsymbol{D})$ with constant delay, and*
- *allows, when given a tuple $\overline{a}$ to test in constant time whether $\boldsymbol{D} \models \varphi(\overline{a})$.*

Moreover, when $\boldsymbol{D}$ is subject to a d-update, the data structure can be recomputed within constant time.

In this Theorem, FO+MOD is an extension of first-order logic where it is allowed to use modulo-counting quantifiers. For instance, FO+MOD can express that the number of node of degree at most 1 is even:

$$\exists^{0 \bmod 2} x,\ \forall y_1,\ \forall y_2\ \Big(\big(E(x,y_1) \wedge E(x,y_2)\big) \Longrightarrow y_1 = y_2\Big).$$

One of the reasons this algorithm works is that over structure of bounded degree, we can compute an Hanf normal form for FO+MOD queries [46].

3.2.4 DynFO

When presenting results about query answering for databases subject to local updates, it is hard to avoid the subject of DynFO. Here, the goal is again to compute data structures that can be maintained when the database is subject to a local updates. In the previous few sections, the goal was to maintain the data structure within constant or logarithmic time. In DynFO, the data structure is a relational structure and should be recomputable by a procedure that is equivalent to a first-order query.

We do not provide a lot of details about this notion and interested readers should take a look at the thesis [75]. We will only present some recent results on this topic.

Definition 3.2.17 ([75]). DynFO is the class of all dynamic queries that can be maintained by dynamic programs with first-order update formulas and arbitrary initialization.

We can then show that lots of query results can be maintain with FO updates.

Theorem 3.2.18 ([39]). *All context-free languages are in* DynFO.

Theorem 3.2.19 ([22]). *The reachability problem for directed graphs is in* DynFO.

These results are of the same kind of those previously presented in this chapter. We now point out some key differences between them. The first major difference lies in the preprocessing/initialization time. While, previously, only linear or pseudo-linear preprocessing where used, DynFO allows arbitrary initialization. This frequently leads to procedures whose first steps require polynomial or exponential time. The second one, is that an update written with a first order query might not be computable in constant or even logarithmic time.

Consequently, results from the DynFO cannot be directly used for constant delay enumeration or any of the other mentioned problems. However, the tools and ideas behind the proofs might provide some insights toward solving our problems.

3.3 Enumeration in other contexts

In this section, we investigate other scenarios that mention enumeration algorithms. First, we present results that define complexity classes for enumeration problems. In addition, we mention some algorithms that falls in those complexity classes. Those are larger complexity classes an contains much more problem than the restricted notion of constant delay enumeration. After that, we briefly expose enumeration algorithms that can be implemented with real life applications.

3.3.1 More theoretical results

The enumeration outside of the database context turns out to be a rather fruitful field. Recently there has been a very interesting attempt to classify the complexity of different enumeration problems. Interested readers are referred to the thesis [70]. We do not look into the mentioned problems in full details, since the techniques developed for their solutions seem to be too different from what is used in database context. In general we restrict ourselves to just stating the key results. Interested readers should follow the corresponding references for both the definitions and the proofs.

Nonetheless, we first give few definitions before listing the interesting results.

Definition 3.3.1. Given a relation $R \subseteq \Sigma^\star \times \Sigma^\star$, the *(abstract) enumeration problem* Enum · R takes as input an x in $\Sigma^\star$ and outputs all y in $\Sigma^\star$ such that $(x, y) \in R$.

Given an element x, the set $\{y \in \Sigma^\star \mid R(x, y)\}$ will be more simply noted $R(x, \cdot)$.

Unlike in the database context, we do not necessarily assume enumeration problem to be a finite relation. However, as it is the case for every result that follows, we restrict our attention to finite enumeration problems.

Polynomial delay

The notion of polynomial delay allows more flexibility than the notion of constant delay enumeration that is the core of this thesis.

Definition 3.3.2 ([47]). An enumeration problem $R(x, y)$ is solvable in *polynomial delay* (denoted with DelayP) if there exists a polynomial $Q(\cdot)$ and an algorithm enumerating $R(x, \cdot)$ in such a way that the time between outputting the i-th and $(i + 1)$-th element from $R(x, \cdot)$ is bounded by $Q(|x|)$.

Unlike constant delay enumeration, here, there is no preprocessing before the enumeration phase. Allowing a linear (or even polynomial) preprocessing would not helps in any way because such preprocessing could be recomputed between any two consecutive outputs. Many natural problems happened to be solvable in polynomial delay.

Theorem 3.3.3 ([15, 69, 71]). *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 3.3.4 ([47]). *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 [47] lists the independent sets in a lexicographical order, but the drawback is that while the delay is polynomial, the space used by the algorithm is exponential. Adding restrictions to the space used leads to the notion of Strong polynomial delay that we present next. This subtle distinction already exists for constant delay enumeration (see Remark 2.5.5).

Strong polynomial delay

The notion of strong polynomial delay also originates from [47]. It is more restricted than DelayP as it contains some restrictions about the use of memory space.

Definition 3.3.5 ([47]). An enumeration problem R is computable in *strong polynomial delay*, written SDelayP, if for every x there is a total order $<_x$ such that the following problems can be solved in polynomial time:

- given x, output the first element of $R(x, \cdot)$ for $<_x$
- given x and $y \in R(x, \cdot)$ output the next element of $R(x, \cdot)$ for $<_x$ or "None" if there is none.

This definition does not contains explicit mention to space used, but the computation of element in $R(x, \cdot)$ can be performed knowing only the previous solution. Therefore, the computation of the $(i + 1)$-th element can re-use the space used during the computation of the i-th solution.

Theorem 3.3.6 ([5, 73]). *The problems of enumerating all the minimal spanning trees and of enumerating all the maximal matchings of a weighted graph are in SDelayP.*

Theorem 3.3.7 ([21]). *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.*

Incremental polynomial time

The notion of incremental polynomial time, like the previous one, was first introduced in [47]. It allows the delay to increase as the number of already outputted solutions grows.

Definition 3.3.8 ([47])**.** An enumeration problem $R(x, y)$ is solvable in *incremental polynomial time* (denoted with IncP) if there exists a polynomial $Q(\cdot, \cdot)$ and an algorithm enumerating $R(x, \cdot)$ in such a way that the time between outputting the i-th and $(i + 1)$-th element from $R(x, \cdot)$ is bounded by $Q(|x|, i)$.

Very recently, a slightly different definition has been introduced.

Definition 3.3.9 ([17])**.** An enumeration problem $R(x, y)$ is solvable in *incremental polynomial time* (denoted with IncP) if there exists a polynomial $Q(\cdot, \cdot)$ and an algorithm enumerating $R(x, \cdot)$ in such a way that the time needed to output the m first elements of $R(x, \cdot)$ is bounded by $Q(|x|, m)$.

It has been shown that those two definitions are actually equivalent as long as we can use an exponential amount of space [17].

An example of a problem from IncP is:

Theorem 3.3.10 ([52])**.** *The problem of enumerating the circuits of a matroid is in IncP.*

The authors of [17], in addition to providing an alternative definition of IncP, exhibit a strict hierarchy within IncP.

Definition 3.3.11 ([17])**.** For every a in $\mathbb{N}$, an enumeration problem $R(x, y)$ is solvable in $IncP_a$ if there exists a polynomial $Q(\cdot)$ and an algorithm enumerating $R(x, \cdot)$ in such a way that the time needed to output the m first elements of $R(x, \cdot)$ is bounded by $m^a \cdot Q(|x|)$.

Theorem 3.3.12 ([17])**.** *For every a, b in $\mathbb{N}$ with $a < b$, if ETH holds then $IncP_a \subsetneq IncP_b$.*

The subtleties between DelayP and SDelayP about the use of space also generate interesting questions here.

For example, it is shown in [17] that $IncP_1$ = DelayP. However this requires that we can use an exponential amount of space during the enumeration. The natural question that is stated as an open problem in [17] is:

" Prove or disprove that $IncP_1$ with polynomial space is equal to DelayP with polynomial space. "

The results presented here might seem quite far from those presented in Section 3.1 and 3.2. However there are some links between them. For example, in Section 3.1.1, we presented an algorithm enumerating acyclic conjunctive queries with a linear delay over any structure. This problem is therefore in DelayP. This is also the case for the enumeration of conjunctive queries over $\underline{X}$-structures. It is really plausible that other restrictions lead to enumeration problems that fall in SDelayP, DelayP, or IncP.

3.3.2 More practical results

The results proved in this thesis have a strong theoretical flavor and do not seem to be implementable. However, this is not the case for every enumeration algorithm. More precisely, some recent results exclusively focus on instances that lead to implementable algorithms that could be used in real life. This is the case for document spanners.

We give a definition of document spanner taken from [28]. Here, a *document* is just a string of characters. Given a document d of size n , a *span* identifies a substring of d by specifying its bounding indices. A *span* of d as the form $[i, j\rangle$ where $1 \leq i \leq j \leq n + 1$. Then, a *document spanner* is a function that is associated with a finite set V of variables, and that maps every document d to a (V, d)-relation.

For example, the function that associate to every document d the (possibly empty) set of positions of the substring "nowhere dense graphs" is a document spanner.

Every regular document spanner can be expressed by an MSO formula. The study of document spanners therefore falls in Section 3.1.2 of this chapter. However, we only consider some restricted classes document spanners that will allow enumeration algorithms with hidden constants way smaller than those in Theorem 3.1.12.

Theorem 3.3.13 ([31]). *There is an algorithm that enumerates the results of a* functional VA A *over a document* d *with a delay* $O(|A|^2 \times |d|)$ *after a preprocessing computed in time* $O(|A|^2 \times |d|)$.

Theorem 3.3.14 ([29]). *There is an algorithm that enumerates the results of a* deterministic and sequential eVA A *over a document* d *with constant delay after a preprocessing computed in time* $O(|A| \times |d|)$.

In these theorems, *deterministic and sequential eVA* and *functional VA* are classes of document spanners whose definitions can be found in [29].

Chapter 4

Some results

Contents

This Chapter is a continuation of the Preliminaries Chapter 2. The contents of this Chapter is a deepening of some notion briefly presented in Chapter 2. The last part of this Chapter (Section 4.4) is more independent, it contains some deep results proved in [45] that are going to be used in the subsequent chapters.

4.1 The model of computation, Random Access Machines

In this Section, we provide precisions about our model of computation: Random Access Machines (RAM for short), defined in Section 2.2. We explain how this model can be used to store functions and relations. More precisely we want a process to store some functions such that the space used in the RAM is not way bigger than the domain of the function. At the same time, we also want to be able, given a tuple of elements in the

domain, to retrieve its image in constant time. Some variants may require that the given tuple is known to be in the domain, others also want to be able to test whether the tuple is in the domain of the function.

In the following, the structure has n elements and its universe is noted $[n]$. The domain of a partial function f, noted $\mathrm{Dom}(f)$, is a subset of $[n]^k$ where k is the arity of f. It is the set of all tuples on which f is defined. For example, when a function f has arity 1 and domain $[n]$, we can simply store the value of $f(i)$ in register R_i for all i in $[n]$. This can easily be adapted to the cases of bigger arity when the function is total (i.e. the domain is $[n]^k$). For instance, with a function f of arity two, we can store the value of $f(i,j)$ in register $R_{(i(n-1)+j)}$, using n^2 registers

Here we will investigate cases where the functions are not total. For instance, in [25], the space complexity is bigger than the time complexity because they store some binary functions of only pseudo-linear domain using quadratic space.

Theorem 4.1.1 (Storing Theorem)**.** *For every n and $\epsilon > 0$, for every function f of domain $\mathrm{Dom}(f) \subseteq [n]^k$, there is an algorithm that stores the values of f using total time and space $O(n^\epsilon \cdot |\mathrm{Dom}(f)|)$. Moreover, at any point during and after the storing process, given $\overline{a}$ in $[n]^k$, we can in time $O(1)$ compute $f(\overline{a})$ or determine that the value of $f(\overline{a})$ has not been registered yet.*

Here, in the big $O(\cdot)$, the constants may depends on ϵ and k, the arity of the function.

Remark 4.1.2. The fact that a value $f(\overline{a})$ can be asked during the storing process can be useful as knowing some values can help computing others.

We now present several ways of storing functions. Let f be a binary function f of domain included in $[n]^2$.

- A matrix representation can be described as follows: any element i in $[n]$ is associated to a complete vector of linear size to store the values of $f(i,j)$ for all j. This leads to a need of quadratic space but the lookup time is clearly constant on a RAM.
- A list representation on the other hand, associate to each element i a list (possibly ordered) of all couples $(j, f(i,j))$ such that (i,j) is in the domain of f. This is optimal in terms of space used, however the lookup time is now linear.
- A third option is to represent, for every i in $[n]$, the set $\{f(i,j) \mid j \in [n]\}$ by an AVL tree (a balanced binary search tree). The space used is still optimal and the lookup time is now $O(\log(n))$.

The proof of Theorem 4.1.1 is a refinement of this third option using some ideas found in [72]. The trade-off between space needed vs lookup time will become a trade-off between the degree of the tree and its depth.

We first start by defining our data structure before describing how it can be computed. Given n and ϵ, we define $d := \lceil n^\epsilon \rceil$ and $h := \lceil \frac{1}{\epsilon} \rceil$ respectively the degree and the depth of our trees (the depth will actually be kh). We define the tree T_f representing our function f as follows:

The root of T_f is represented by d consecutive registers, from R_1 to R_d. Then either the j^{th} subtree of the root is empty and $R_j = 0$ or the root of the subtree is again represented with d consecutive registers, from R_l to R_{l+d} with $l = R_j$. The leaves of the tree (at depth h) are still d consecutive registers containing values between 0 and n.

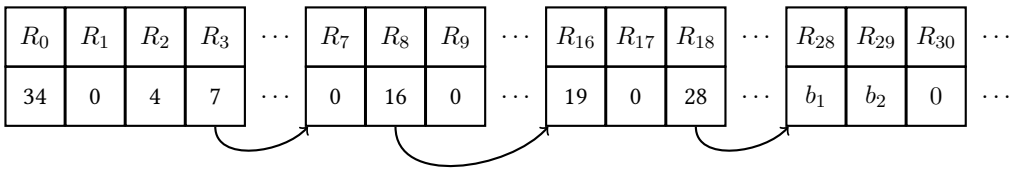
Now, how does the tree store our function? Given $(\overline{a}; b)$ with $\overline{a} := (a_1, \ldots, a_k)$ such that $f(\overline{a}) = b$, b is a leaf of the tree and the path from the root to this leaf only depends on $\overline{a}$. To do so we first decompose each a_i into a word of length h using an alphabet with d letters (we can do that since $d^h \geq n$). We obtain a word $(u_{h(i-1)+1}, \ldots, u_{hi})$ that satisfies: $a_j = \sum_{j=0}^{h-1} u_{(h(i-1)+j+1)} \cdot d^j$. We then do that for every $i \leq k$, which gives us a word $u_1, \ldots, u_{kh}$ of length kh.

We define the search path for $\overline{a}$ in T_f as kh indices of registers $v_1, \ldots, v_{kh}$ such that:

- $v_1 = 1 + u_1$. *(R_0 is used for something else)*
- $v_i = R_{v_{i-1}} + u_i$. For all $2 \leq i \leq kh$. *(We go to the u_i-th child of the subtree)*
- $R_{v_i} \neq 0$. For all i. *(The subtrees cannot be empty, they must contain the next address)*
- $R_{v_{kh}} = b$ *(b is stored at the leaf level)*

Otherwise, if we are given a tuple $\overline{a}$ such that $f(\overline{a})$ is undefined, then by defining $(u_i)_{i \leq kh}$ and $(v_i)_{i \leq kh}$ as previously, we have that there is an i with $R_{v_i} = 0$.

Example 4.1.3. *Consider the case of a binary function f. In the case where the domain of f has less than 9 elements, we can encode each of them by a word of length 2 over the alphabet $\{0, 1, 2\}$. This means that here, $d = 3$ and $h = 2$. Moreover, since f is binary, we have $k = 2$. Let say that element a is represented by the word "21" and a' by "20". In the figure below, we show how to store the facts that $f(a, a') = b_1$ and $f(a, a) = b_2$.*



Here, for $f(a, a') = b_1$, we have $u_1 = 2$, $u_2 = 1$, $u_3 = 2$ and $u_4 = 0$. This leads to $v_1 = u_1 + 1 = 3$, and $v_2 = R_{v_1} + u_2 = R_3 + 1 = 7 + 1 = 8$. After that, $v_3 = R_{v_2} + u_3 = R_8 + 2 = 16 + 2 = 18$, and finally, $v_4 = R_{v_3} + u_4 = R_{18} + 0 = 28 + 0$, thus, b_1 is stored in R_{28}.

We can also deduce from the figure above that there are no element c such that (a', c) is in the domain of f because R_7 contains a 0. With the same kind of reasoning, we can see that there is at least one element c whose decomposition starts with a "0" and such that (a, c) is in the domain of f because R_{16} does not contains a 0.

We can now turn to formal details.

Proof of the Storing Theorem 4.1.1. We will use two different names for registers, R and S, for a better readability. As S is used only for a constant number of registers (kh precisely), we could easily stash them into R by changing some indexes. We assume n and ϵ given to the RAM, as well as $d = \lceil n^\epsilon \rceil$ and $h = \lceil \frac{1}{\epsilon} \rceil$, where $\lceil x \rceil$ is the ceiling of x (i.e. the smallest i such that $x \leq i$). We define several procedures:

Algorithm 4 Decomposition$(a_1, \dots a_k)$

for $i = 1, \quad i \leq k, \quad i++$ **do** ▷ Basically an Euclidean division
  $A \leftarrow a_i$
  for $j = h(i-1), \quad j \leq hi - 1, \quad j++$ **do**
    $B \leftarrow \lfloor \frac{A}{d} \rfloor$ ▷ The quotient of A/d
    $S_j \leftarrow A - d \cdot B$ ▷ The remainder of A/d
    $A \leftarrow B$
  end for
end for

At the end, $\overline{a}$ is decomposed in base d using kh characters. We have:

$$a_1 = \sum_{j=0}^{h-1} S_j \times d^j \quad \text{and more generally} \quad a_i = \sum_{j=h(i-1)}^{hi-1} S_j \times d^{j-h(i-1)}$$

For the main algorithm, we need to have access in constant time to unused registers. We use R_0 for that, at the beginning, $R_0 = d+1$. And every R_i with $i \leq d$ contains a 0.

The next function takes $k+1$ elements and states that $f(\overline{a}) = b$.

Algorithm 5 NEW$(\overline{a}, b)$

Decomposition$(\overline{a})$ ▷ Decompose $\overline{a}$ using register $S_1, \dots, S_{kh}$
ADD$(0, 1, b)$ ▷ see next function

The next and last function (**ADD**(i, l, b)) insert b in a subtree whose root begin in register R_l. Parameter i is used to count how deep we are in the tree.

Algorithm 6 ADD(i, l, b)

if $i = kh - 1$ **then** ▷ We are at the leaves level
  $R_{(l+S_i)} \leftarrow b$ ▷ We put b in the good leaf
else
  if $R_{(l+S_i)} = 0$ **then** ▷ We need to start a new subtree
    $R_{(l+S_i)} \leftarrow R_0$ ▷ We use new registers for that
    for $j = 0, \quad j < d, \quad j++$ **do**
      $R_{R_0+j} \leftarrow 0$ ▷ We clear those registers
    end for ▷ (Useless if initially empty)
    $R_0 \leftarrow R_0 + d$ ▷ We update R_0
  end if
  ADD$(i+1, R_{(l+S_i)}, b)$
end if

This end the computation of the representation of f. Every tuple $(\overline{a})$ in $\mathrm{Dom}(f)$ has a path of length kh in our tree. Each path used at most $d \cdot kh$ registers since a given node use d register. This leads to a total of dkh registers to store the image of $(\overline{a})$. We also use kh registers to decompose the elements (the S registers).

The total number of registers used is:

$$O(kh + dkh \cdot |\mathrm{Dom}(f)|) = O(n^{\epsilon} \cdot |\mathrm{Dom}(f)|).$$

Moreover, every fact is inserted in time: $O(kh + dkh)$, because we use time $O(kh)$ to decompose the tuple, and then time at most $O(d)$ at every depth in the tree. Finally, we have:

$$O(kh + dkh) = O(n^{\epsilon}),$$

where $O(\cdot)$ depends on ϵ and k.

Remark 4.1.4. The factor d that appears in in the time needed to store one fact can be avoid if we can guaranty that every register is initialized to 0 and thus remove lines $6, 7, 8$ from the previous algorithm.

Example 4.1.5. *Consider again case of example 4.1.3. Imagine that this is the state of the memory before you want to store that $f(a, a') = b$. Where a is encoded by "21" and a' by "20".*

S_0	S_1	S_2	S_3
0	2	1	1

R_0	R_1	R_2	R_3	$\cdots$	R_7	R_8	R_9	$\cdots$	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	$\cdots$
13	0	4	7	$\cdots$	0	0	10	$\cdots$	0	0	0	0	0	0	$\cdots$

We then apply ***Decomposition***$(\overline{a})$ *and the two first calls of* ***ADD***$(0, 1, b)$.

S_0	S_1	S_2	S_3
2	1	2	0

R_0	R_1	R_2	R_3	$\cdots$	R_7	R_8	R_9	$\cdots$	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	$\cdots$
16	0	4	7	$\cdots$	0	13	10	$\cdots$	0	0	0	0	0	0	$\cdots$

The first call read 7 *in* R_3 *and proceed by calling* ***ADD***$(2, 7, b)$. *The second one read* 0 *in* R_8 *and then write* R_0 *on* R_8 *and* $R_0 + 3$ *on* R_0. *After that, it calls* ***ADD***$(3, 13, b)$, *which produce what follows:*

R_0	R_1	R_2	R_3	$\cdots$	R_7	R_8	R_9	$\cdots$	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	$\cdots$
19	0	4	7	$\cdots$	0	13	10	$\cdots$	16	0	0	0	0	0	$\cdots$

Because 0 *where written in* R_{13}*, it wrote* R_0 *in it* (16 *at this point) and* $R_0 + 3$ *in* R_0 *(i.e.* 19*). After that it calls* ***ADD***$(4, 16, b)$*, which will just write* b *in* R_{16+S_3} *i.e. in* R_{16}*.*

R_0	R_1	R_2	R_3	$\cdots$	R_7	R_8	R_9	$\cdots$	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	$\cdots$
19	0	4	7	$\cdots$	0	13	10	$\cdots$	16	0	0	b	0	0	$\cdots$

We now need to be able to access $f(\overline{a})$ in constant time (or determine that f is not define here)

Algorithm 7 Access$(\overline{a})$

1: **Decomposition**$(\overline{a})$ ▷ Decompose $\overline{a}$ using register $S_1, \ldots, S_{kh}$
2: $l \leftarrow 1$
3: $i \leftarrow 0$
4: **while** $i \leq kh - 1$ & $l \neq 0$ **do**
5: $\quad l \leftarrow R_{(l+S_i)}$ ▷ We follow the search path
6: $\quad i \leftarrow i + 1$
7: **end while**
8: RETURN(l) ▷ Contains either 0 if it fails or $f(\overline{a})$ if $i = kh$

If while going through the tree T_f as if we wanted to add a new element for $f(\overline{a})$ we never encounter an empty register, then at the end l contains $f(\overline{a})$. Otherwise, an empty register would means an empty subtree, therefore $f(\overline{a})$ is undefined and we can safely output 0.

For the time complexity, the sub-procedure **Decomposition** takes time $O(kh)$. We go through the loop in Line 4 only kh times, each of them takes constant time, leading to a total time complexity in $O(kh) = O(\frac{k}{\epsilon}) = O(1)$.

This ends the proof of the Storing Theorem 4.1.1. □

We can use the same technique to store a relation by using its identity function: if $a \in R$, then $f_R(\overline{a}) = 1$, otherwise $f_r(\overline{a})$ equals 0 or is undefined.

4.2 Nowhere dense graphs

4.2.1 Definitions

The notion of nowhere dense classes of graphs was introduced in [60] as a formalization of classes of "sparse" graphs and generalizes many well known classes of graphs such as bounded degree, planar graphs or graphs with bounded tree width [61].

In this section, we give several definitions of nowhere dense classes of graphs. We mainly focus on the definitions that will be used in the following chapters. Interested readers can read [60, 61] for more information and details.

In Section 2.6.5 we provide a definition of nowhere dense classes of graphs using the notion of graph minors an more precisely r*-minors*. Here we use another equivalent definition of nowhere dense classes of graphs using weak coloring numbers instead.

Definition 4.2.1 ([53]). Let L be a linear order on the vertex set of a graph G, and let x, y be vertices of G. We say y is *weakly r-accessible* from x if $y <_L x$ and there exists an $(x - -y)$ path P of length at most r (i.e. with at most r edges) with minimum vertex y with respect to $<_L$.

We note $wcol_r(G, L, a)$ the set of nodes weakly r-accessible from a.

Definition 4.2.2 ([60]). The *weak r-coloring number* $\mathrm{wcol}_r(G)$ of G is defined by

$$\mathrm{wcol}_r(G) = 1 + \min_L \max_{a \in G} |\mathrm{wcol}_r(G, L, a)|$$

Theorem 4.2.3 ([60]). *A class of graphs $\mathcal{C}$ is* nowhere dense *if there is a function f such that :* $\forall r \in \mathbb{N}, \forall \epsilon > 0, \forall G \in \mathcal{C}, |G| > f(r, \epsilon) \Rightarrow \mathrm{wcol}_r(G) \leq |G|^\epsilon$

This is equivalent to the Definition 2.6.10 from Section 2.6.5 [61].

Remark 4.2.4. We call a class $\mathcal{C}$ *effectively nowhere dense* if there is a computable function f such that such that : $\forall r \in \mathbb{N}, \forall \epsilon > 0, \forall G \in \mathcal{C}, |G| > f(r, \epsilon) \Rightarrow \mathrm{wcol}_r(G) \leq |G|^\epsilon$. All natural nowhere dense classes are effectively nowhere dense, but it is possible to construct artificial classes that are nowhere dense, but not effectively so [44].

The notion of effectively nowhere dense is needed when we want *uniform* results. This means that instead of having one specific algorithm for every query φ and every $\epsilon > 0$, we will be able to obtained one algorithm solving the problem with φ and ϵ as inputs.

This definition of nowhere dense graphs is used to show that neighborhood covers can efficiently be computed over nowhere dense classes of graphs.

Definition 4.2.5 (Neighborhood cover)**.** Given a graph G and a number $r \in \mathbb{N}$, an *(r, s)-neighborhood cover* of G is a collection $\mathcal{X}$ of subsets of V such that:

- $\forall\, a \in V \;\; \exists\, X \in \mathcal{X}$ with $N_r^G(a) \subseteq X$
- $\forall\, X \in \mathcal{X} \;\; \exists\, a \in V$ with $X \subseteq N_s^G(a)$.

The number r is called the *radius* of the neighborhood cover, and the sets $X \in \mathcal{X}$ are called *bags*. An (r, s)-neighborhood cover will sometime just be called an r-neighborhood cover when s depends on r.

The *degree* $\delta(\mathcal{X})$ of the cover is the maximal number of bags that intersect at a given node, i.e.

$$\delta(\mathcal{X}) \quad := \quad \max_{a \in V} |\{X \in \mathcal{X} \mid a \in X\}|.$$

Given a neighborhood cover of radius r of G, for every $a \in V$ we arbitrarily fix a bag containing the r-neighborhood of a and denote it by $\mathcal{X}(a)$.

Such neighborhood cover can be computed efficiently over nowhere dense classes of graphs:

Theorem 4.2.6 ([43]). *Let $\mathcal{C}$ be a nowhere dense class of graphs. There is a function $f_{\mathcal{C}}$ such that, for all $\epsilon > 0$, and $r \in \mathbb{N}$, for every graphs $G \in \mathcal{C}$ whose domain V is larger than $f_{\mathcal{C}}(r, \epsilon)$, there exists an $(r, 2r)$-neighborhood cover of G with degree at most $|V|^{\epsilon}$ that can be computed in time $f_{\mathcal{C}}(r, \epsilon) \cdot |V|^{1+\epsilon}$.*

Moreover, when V is larger than $f_{\mathcal{C}}(r, \epsilon)$, we have that $\|G\| \leq |V|^{1+\epsilon}$.

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, then $f_{\mathcal{C}}$ is computable.

Additionally, using the Storing Theorem 4.1.1, once the neighborhood cover is computed, given $a \in V$, and $X \in \mathcal{X}$, we can test in constant time whether $a \in X$. We also have access to $\mathcal{X}(a)$ in constant time.

For a graph G, a neighborhood cover $\mathcal{X}$ of G, a number $r \in \mathbb{N}$, and a tuple $\overline{a}$ of nodes of G we say that a bag $X \in \mathcal{X}$ *r-covers* $\overline{a}$ if $N_r^G(\overline{a}) \subseteq X$.

The fact that neighborhood covers can efficiently be computed over nowhere dense classes graphs plays an important role in the following theorem.

Theorem 4.2.7 (Grohe, Kreutzer, Siebertz [43]). *Let q be a first-order query with at most one free variable and $\mathcal{C}$ a nowhere dense class of graphs. There is an algorithm that, on input of $G \in \mathcal{C}$ computes $q(G)$ in pseudo-linear time.*

This theorem will often be referenced as the Model Checking Theorem or the Unary Theorem (depending on the arity of the used query being 1 or 0).

Theorem 4.2.6 is not the only tool used in the proof of Theorem 4.2.7. An other crucial ingredient is a characterization of nowhere dense classes of graphs using a two players game.

Definition 4.2.8 (*Splitter game* [43]). Let G be a graph and let $\lambda, r \in \mathbb{N}_{\geq 1}$. The (λ, r)-*splitter game* on G is played by two players called *Connector* and *Splitter*, as follows. We let $G_0 := G$ and $V_0 := V$. In round $i+1$ of the game, Connector chooses a vertex $a_{i+1} \in V_i$. Then Splitter picks a vertex $b_{i+1} \in N_r^{G_i}(a_{i+1})$. If $V_{i+1} := N_r^{G_i}(a_{i+1}) \setminus \{b_{i+1}\} = \emptyset$, then Splitter wins the game. Otherwise, the game continues with $G_{i+1} := G_i[V_{i+1}]$. If Splitter has not won after λ rounds, then Connector wins.

We say that Splitter *wins* the (λ, r)-splitter game on G if she has a winning strategy for the game.

Theorem 4.2.9 (Game characterization on nowhere dense classes of graphs [43]). *A class $\mathcal{C}$ of graphs is nowhere dense if, and only if, for every $r \in \mathbb{N}_{\geq 1}$ there is a $\lambda \in \mathbb{N}_{\geq 1}$, such that for every $G \in \mathcal{C}$, Splitter wins the (λ, r)-splitter game on G.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, then λ can be computed from r.

In the following chapters, we will make use of both notions: neighborhood cover and splitter game. Those are not the only tools from [43] that we are going to use. The others, that are not necessarily related to nowhere dense classes, can be found in Section 4.4.

4.2.2 Examples

In this section, we provide examples of wining strategies for Splitter over some sparse classes of graphs.

Class of Stars

Here we consider graphs that are stars.

Definition 4.2.10. The *star* with n elements is the graph $G = (V, E)$ where $V := [n]$ and $E := \{(1, i) \mid 1 < i \leq n\}$.

We then explain how Splitter easily wins over a class $\mathcal{C}$ of star graphs.

Claim 4.2.11. *Let $\mathcal{C}$ be a class of star graphs, for every r in $\mathbb{N}$, for every G in $\mathcal{C}$, Splitter wins the $(2, r)$-splitter game on G.*

Proof. Let G be the star with n elements for some n in $\mathbb{N}$. The first node that Splitter delete is the center of the star. We are then left with $n-1$ isolated nodes. Then connector picks one of those node and Splitter wins by deleting this final node. □

Class of Paths

Here we consider graphs that are paths.

Definition 4.2.12. The *path* with n elements is the graph $G = (V, E)$ where $V := [n]$ and $E := \{(i, i+1) \mid 1 \leq i < n\}$. The *length* of a path is this number n.

We then explain how Splitter easily wins over a class $\mathcal{C}$ of path graphs.

Claim 4.2.13. *Let $\mathcal{C}$ be a class of path graphs, for every r in $\mathbb{N}$, for every G in $\mathcal{C}$, Splitter wins the $(\log(r), r)$-splitter game on G.*

Proof. The strategy applied by Splitter preserves the following invariant: "After the end of the i-th round, we are left with two paths of length bounded by $r/2^i$." We prove the invariant by induction on i.

During the first round, when connector picks a node, we can restrict our attention to its r-neighborhood: a path of length at most r. Splitter then deletes the middle node. We are left with two disconnected paths of length at most $r/2$.

At the beginning of the $i+1$-th rounds, we have two disconnected paths of length at most $r/2^i$. Connector picks a node and we can restrict our attention to one of the paths. Splitter then delete the node in the middle of that path. We are left with two disconnected paths of length at most $r/2^{(i+1)}$.

After the end of the ℓ-th round, with $\ell > \log(r)$, the graph is empty and Splitter wins the game. □

Class of Trees

Here we consider graphs that are trees i.e. graphs without cycle.

Claim 4.2.14. *For every r in $\mathbb{N}$, for every tree T Splitter wins the (r, r)-splitter game on G.*

Proof. The strategy applied by Splitter preserves the following invariant: "After the end of the i-th round, we are left with several rooted trees of depth bounded by $r - i$." We prove the invariant by induction on i.

During the first round, when connector picks a node, we can restrict our attention to its r-neighborhood: a rooted tree of depth at most r. Splitter then deletes the root, we are left with several disconnected rooted trees of depth at most $r-1$.

At the beginning of the $i+1$-th round, we have several rooted trees of depth at most $r-i$. Connector picks a node and we can restrict our attention to one of the trees. Splitter then deletes the root, we are left with several disconnected rooted trees of depth at most $r-(i+1)$.

After the end of the r-th round, the graph is empty and Splitter wins the game. □

Class of graphs with bounded degree

Here we consider graphs with bounded degree and explain how Splitter easily wins over such classes.

Claim 4.2.15. *Let $\mathcal{C}$ be a class of graphs with degree bounded by d. For every r in $\mathbb{N}$, for every G in $\mathcal{C}$, Splitter wins the (d^r, r)-splitter game on G.*

Proof. When Connector picks the first node, we restrict our attention to the r-neighborhood of this node. Since this neighborhood only contains at most d^r elements, Splitter can just pick nodes randomly and win in less than d^r rounds. □

Classes of cliques

We now consider some classes of graphs where Splitter does not wins. Here we consider graphs that are cliques.

Definition 4.2.16. The *clique* with n elements is the graph $G = (V, E)$ where $V := [n]$ and $E := \{(i,j) \mid 1 \leq i, j \leq n\}$.

We then explain how Connector wins over a class $\mathcal{C}$ of cliques.

Claim 4.2.17. *Let $\mathcal{C}$ be an infinite class of cliques, for every r and λ in $\mathbb{N}$, there is a graph G in $\mathcal{C}$, such that Connector wins the (λ, r)-splitter game on G.*

Proof. Let λ and r be two integers. Let G be a cliques with $n > \lambda$ elements. Regardless of the choices made by Splitter and Connector, we have the following invariant: "After the end of the i-th round, we are left with a clique with $n-i$ elements."

As every pair of nodes are connected, when Connector picks a node, its r-neighborhood is the entire graph. Then when Splitter picks a vertex, we just delete this one node and the adjacent edges.

At the end of the λ-th round, the graphs is not empty and Connector wins the game. □

Class of subdivided cliques

We want to show that there exist classes on which Splitter wins but the number of rounds she needs to win is arbitrarily large with respect to the other parameter: r.

Claim 4.2.18. *For any increasing function $f(\cdot)$ from $\mathbb{N}$ to $\mathbb{N}$, there is a nowhere dense class of graph $\mathcal{C}$, such that:*

- *For every r in $\mathbb{N}_{\geq 1}$, and for every graph G in $\mathcal{C}$, Splitter wins the $(f(r)+r+2, r)$-splitter game on G.*
- *For every r in $\mathbb{N}$, there is a graph G in $\mathcal{C}$ such that Splitter does not win the $(f(r)-1, r)$-splitter game on G.*

Proof. Let $f(\cdot)$ be a fixed increasing function from $\mathbb{N}$ to $\mathbb{N}$. For every r in $\mathbb{N}_{\geq 1}$, we define G_r as the $f(r)$-clique $(r-1)$-subdivided. This means that in the $f(r)$-clique, every edge is replaced by a path of length r. We then define $\mathcal{C} := \{G_r \mid r \in \mathbb{N}_{\geq 1}\}$.

We now prove the first part of Claim 4.2.18. Let r in $\mathbb{N}_{\geq 1}$ and G a graph in $\mathcal{C}$. We want to explain how Splitter wins the $(f(r)+r+2, r)$-splitter game on G. Since G is in $\mathcal{C}$, we have that $G = G_{r'}$ for some $r' \in \mathbb{N}$. We distinguish two cases whether $r' \leq r$ or $r' > r$.

Case 1: $r' \leq r$. In that case, G is the $f(r')$-clique subdivided $r'-1$ times. In the first $f(r')$ rounds, Splitter delete every original node (i.e. that existed before the subdivision of the $f(r')$-clique). We are then left with many disconnected path of length at most r'. When Connector picks a node, we restrict our attention to only one path of length at most r', Splitter then wins in less than r' additional rounds (she actually only needs $\log(r)$ rounds using the strategy described above for path). Since $r' \leq r$, we have that $f(r') \leq f(r)$, the total amongst of rounds is then bounded by $f(r)+r$.

Case 2: $r' > r$. In that case, when Connector picks a first node, only one or two original nodes are in this node's r-neighborhood. Thus, we are either left with a star r-subdivided or two stars r-subdivided that share one path. In one or two rounds, Splitter delete the center of those stars and only remains many disconnected path of length at most r. As previously, Splitter wins using less than r additional rounds.

In both cases, Splitter wins in less than $(f(r)+r+2)$ rounds. Which conclude the first part of the proof. This also shows that $\mathcal{C}$ is nowhere dense.

We now prove the second part of Claim 4.2.18. Let r in $\mathbb{N}_{\geq 1}$, we explained how Connector can win the $(f(r)-1, r)$-splitter game on G_r. At the i-th round, Connector picks one of the original nodes. This node is still connected to $f(r)-i$ other nodes from the original $f(r)$-clique. He can therefore continue to play. After $f(r)-1$ rounds, there is still at least one node in the graphs, making Connector the winner of this game. □

Remark 4.2.19. In the proof above, it is quite interesting to note that in every cases and for both players, all wining strategies are actually the same: "pick randomly a node of maximal degree until the end of the game".

In the following Chapters, we provide algorithms for various tasks over nowhere dense classes of graphs. In each of them, the time need to complete the task depends on the number λ of rounds Splitter needs to win the splitter game where the parameter r depends on the query. By the characterization of nowhere dense graphs using splitter game (Theorem 4.2.9) we now that this number λ does not depends on the size of the graph. What Claim 4.2.18 tells us is that this "constant" can be arbitrarily large.

4.3 From databases to graphs

4.3.1 Representing databases with colored graphs

In the following Chapters, we will present algorithms that work over colored graphs. This extends to all relational structures using standard techniques as we now explain.

Definition 4.3.1 (colored graphs)**.** For every integer c, we define *c-colored graphs* as structures over the schema $\sigma_c := \{E, C_1, \dots, C_c\}$ where E is a binary relation and $(C_i)_{i \leq c}$ are unary relations. A *colored graph* is a c-colored graph for some integer c.

In the previous sections, we provided some results and definitions about classes of graphs or undirected graphs. All of those easily extend to colored graphs by considering the underlying uncolored graphs. For example, the facts that nodes have colors has no impact on the definition of neighborhood covers of in the Splitter game. The notion of undirected is not impacted as well. An *undirected colored graph* is a colored graph whose underlying uncolored graph is undirected.

Theorem 4.3.2 (From databases to graphs)**.** *For every schema σ, every structure $\boldsymbol{D}$ over σ and every first order query q over σ, in time $O(\|\boldsymbol{D}\|)$ we can compute:*

- *an integer c,*
- *a c-colored graph G and*
- *a first order query q' over σ_c*

such that $q(\boldsymbol{D}) = q'(G)$.

Moreover, c only depends on σ, G only depends on $\boldsymbol{D}$ and σ and q' only depends on q and σ.

Here, the constant in the big $O(\cdot)$ may depends on σ and q.

Proof. Given a database $\mathbf{D}$ over a schema σ, we define its *adjacency graph* $A(\mathbf{D})$ as the relational structure whose domain is $D \cup T$ where D is the domain of $\mathbf{D}$ and T is the set of tuples occurring in a relation of $\mathbf{D}$. We have one unary relation P_R per relation R of σ containing all tuples t of $R(\mathbf{D})$. We have k binary relations $E_1, \dots, E_k$ where k is the maximal arity of relations in σ. For $a \in D$ and $t \in T$, we have $A(\mathbf{D}) \models E_i(a, t)$ if and only if the element a is the i-th element of the tuple t. When talking about nodes in $A(\mathbf{D})$, we talk about "tuple nodes" (those in P_R for some R) and "real nodes" (those that are elements of D).

The colored graph version $A'(\mathbf{D})$ of the adjacency graph $A(\mathbf{D})$ is defined as follows.

The colors of $A'(\mathbf{D})$ are the "vertex colors" P_R of $A(\mathbf{D})$ and k further colors $(C_i)_{i \leq k}$, where k is the maximal arity of the relations in σ. The domain of $A'(\mathbf{D})$ is the domain of $A(\mathbf{D})$ plus one node per edge of $A(\mathbf{D})$: For every E_i-edge (a, t) of $A(\mathbf{D})$, we add in $A'(\mathbf{D})$ a new node v of color C_i and such that (a, v) and (v, t) are E-edges in $A'(\mathbf{D})$. In particular, $A'(\mathbf{D})$ is a colored graph of schema $\{E, C_1, \dots, C_k, P_{R_1}, \dots, P_{R_m}\}$ if $\mathbf{D}$ is a database of schema $\sigma = \{R_1, \dots, R_m\}$ consisting of m relations of maximum arity k. Henceforth, we will identify the schema of $A'(\mathbf{D})$ with the schema σ_c of c-colored graphs for $c = k + m$.

The following straightforward lemma concludes the proof of Theorem 4.3.2.

Lemma 4.3.3. *Let $q(\overline{x})$ be a* FO *query over some schema σ and let k be the maximal arity of the relations of σ. In time linear in the size of q, we can compute a* FO *query $\varphi(\overline{x})$ over σ_c, for $c = k + |\sigma|$, such that for every database $\mathbf{D}$ over σ, $q(\mathbf{D}) = \varphi(A'(\mathbf{D}))$.*

The lemma is a consequence of the fact that for every integer j, every relation R of arity j and every tuple $(a_1, \ldots, a_j)$, we have:

$$\mathbf{D} \models R(a_1, \ldots, a_j)$$

$$\iff$$

$$A'(\mathbf{D}) \models \exists t \Big(P_R(t) \wedge \bigwedge_{i \leq j} \exists z \big(C_i(z) \wedge E(a_i, z) \wedge E(z, t) \big) \Big).$$

□

Theorem 4.3.2, is not enough in our case, we will indeed restrict our attention to colored graphs but our algorithms don't works for all colored graphs but only nowhere dense ones. The question is then : "For which classes of databases our algorithms can be extended ?"

The answer to this question is what we call efficiently nowhere dense reducible.

Definition 4.3.4. A class of databases $\mathcal{C}$ over a schema σ is said to be *nowhere dense reducible* if there are:

- an integer c,
- a function f that maps σ-databases to c-colored graphs,
- a function g that maps FO queries to FO queries,
- for every $\mathbf{D}$ in $\mathcal{C}$ and every q in FO, an injective function $h_{\mathbf{D},q}(\cdot)$ that maps k-tuples of elements of the domain D of $\mathbf{D}$ to k'-tuples of vertex of the graph $G := f(\mathbf{D})$. (k and k' being the arity of q and $g(q)$)

That satisfy:

- The class $\mathcal{C}' := \{f(\mathbf{D}) \mid \mathbf{D} \in \mathcal{C}\}$ is a nowhere dense class of colored graphs.
- For every database $\mathbf{D}$ in $\mathcal{C}$, FO query q of arity k, and k-tuple $\overline{a}$ in D^k, with $G := f(\mathbf{D})$ and $q' := g(q)$, we have:

  $$\mathbf{D} \models q(\overline{a}) \iff G \models q'(h_{\mathbf{D},q}(\overline{a}))$$

  and for every tuple $\overline{b} \in G^{k'}$, if $G \models q'(\overline{b})$, there is exactly one tuple $\overline{a} \in \mathbf{D}$ with $\overline{b} = h_{\mathbf{D},q}(\overline{a})$.

Additionally, $\mathcal{C}$ is said to be *efficiently nowhere dense reducible* if c is computable from $\mathcal{C}$, g is a computable function, f is a linear-time computable function and for every database $\mathbf{D}$ in $\mathcal{C}$, FO query q and tuple $\overline{b}$, if there is a k-tuple $\overline{a}$ in D^k such that $\overline{b} = h_{\mathbf{D},q}(\overline{a})$, it can be computed in constant time.

An easy example is any classes of databases whose colored adjacency graphs define a nowhere dense class of graphs.

4.3.2 Gaifman versus adjacency

Most of the time, people use gaifman graphs instead of adjacency ones. However, the algorithms are generally executed over the original structures. Here, we do every proof for graphs and automatically lift them to databases using Theorem 4.3.2. An interesting question is whether choosing one of the two notions allows to lift the results to more general classes of database.

Definition 4.3.5. The *gaifman graph* of database $\mathbf{D}$ of domain D is a graph $G = (V, E)$ with $V := D$ and for every elements (a, b) in D, (a, b) is an edge in E if and only if there is a tuple t in $\mathbf{D}$ where a and b are two elements of t.

We show that those two notions coincide for nowhere dense classes of structures over a fixed schema. When $\mathbf{D}$ is a structure, we note $\mathbf{D}_G$ the gaifman graph of $\mathbf{D}$ and $\mathbf{D}_A$ its adjacency graph. Given a class $\mathcal{C}$ of databases, we want to compare the class $\mathcal{C}_G$ of gaifman graphs of structures from $\mathcal{C}$ with the class $\mathcal{C}_A$ of adjacency graphs of databases from $\mathcal{C}$.

It should be noted that in Section 4.3.1 we use a subdivision of the adjacency graph we note $A'(\mathbf{D})$. We should therefore do the proof for the class of all $A'(\mathbf{D})$ instead of all $\mathbf{D}_A$. However, $\mathbf{D}_A$ is a 1-minor of $A'(\mathbf{D})$ and $A'(\mathbf{D})$ is a 1-subdivision of $\mathbf{D}_A$. Thus we have that the class of all $\mathbf{D}_A$ is nowhere dense if and only if the class of all $A'(\mathbf{D})$ is nowhere dense.

Theorem 4.3.6 (Gaifman versus Adjacency)**.** *Let $\mathcal{C}$ be a class of structures over a fixed schema σ. The class of graphs $\mathcal{C}_G$ is nowhere dense if and only if $\mathcal{C}_A$ is nowhere dense.*

The Theorem is the combination of the following two propositions:

Proposition 4.3.7. *Let $\mathcal{C}$ be a class of structures with $\mathcal{C}_G$ nowhere dense. k such that $K_k \notin \mathcal{C}_G \nabla 0$. Then, for all $\mathbf{D} \in \mathcal{C}$, for all $r \in \mathbb{N}$, $\mathrm{wcol}_r(\boldsymbol{D}_A) \leq k \times \mathrm{wcol}_r(\boldsymbol{D}_G)$.*

Proposition 4.3.8. *Let $\mathcal{C}$ be a class of structures with a fixed signature σ such that $\mathcal{C}_A$ is nowhere dense. Then, $\forall \boldsymbol{D} \in \mathcal{C}, \forall r \in \mathbb{N}$, $\mathrm{wcol}_r(\boldsymbol{D}_G) \leq (k \times \mathrm{wcol}_{2r}(\boldsymbol{D}_A))^{2r}$.*
Here, k is the maximal arity amongst the relations in σ.

We start with a left-to-right direction (Proposition 4.3.7), which does not need to fix the schema.

Proof of Proposition 4.3.7. Let $\mathcal{C}$ be a class of structures such that $\mathcal{C}_G$ is nowhere dense. Let $\mathbf{D} \in \mathcal{C}$ and t a tuple in $\mathbf{D}$ with arity s. Let $(a_1, \ldots, a_s)$ be the elements in the tuple t. By definition, $(a_1, \ldots, a_s)$ is a clique in $\mathbf{D}_G$. Therefore $K_s \in \mathcal{C}_G \nabla 0$. Hence $s \leq N_0$ (given by Definition 2.6.10). Let $\mathbf{D} \in \mathcal{C}$ and let $<_G$ an order on $\mathbf{D}_G$ such that $\max_{a \in \mathbf{D}_G} |\mathrm{wcol}_r(\mathbf{D}_G, <_G, a)| = \mathrm{wcol}_r(\mathbf{D}_G)$. We define an order $<_A$ on $\mathbf{D}_A$ as follow :

- If a and b are "real nodes" in $\mathbf{D}_A$, then $a <_A b \iff a <_G b$.
- If a is a real node and b is a "tuple node", then $a <_A b$.
- If a and b are both "tuples nodes", the order can be chosen arbitrarily.

Let a be a real node in $\mathbf{D}_A$. Let b weakly r-accessible from a in $\mathbf{D}_A$. By definition, $b <_A a$ therefore b is also a real node. Moreover, every real node in the witness path from a to b is also greater than b in $\mathbf{D}_A$. Since two real nodes at distance two in $\mathbf{D}_A$ are connected in $\mathbf{D}_G$ we can conclude that b is also weakly $\frac{r}{2}$-accessible from a in $\mathbf{D}_G$. Therefore $\mathrm{wcol}_r(\mathbf{D}_A, <_A, a) \subseteq \mathrm{wcol}_r(\mathbf{D}_G, <_G, a)$

Let now a be a "tuple node" in $\mathbf{D}_A$. By definition of $\mathbf{D}_A$, a is linked to only k real nodes $(b_1, \ldots, b_k)$. We have that :

$$\mathrm{wcol}_r(\mathbf{D}_A, <_A, a) \subseteq \bigcup_{1 \leq i \leq k} \mathrm{wcol}_{r-1}(\mathbf{D}_A, <_A, b_i)$$

Hence :

$$\mathrm{wcol}_r(\mathbf{D}_A) \leq k \times \mathrm{wcol}_r(\mathbf{D}_G)$$

□

This conclude the proof of Proposition 4.3.7. Let now prove the right-to-left part of Theorem 4.3.6, i.e. Proposition 4.3.8.

Proof of Proposition 4.3.8. Let $\mathbf{D} \in \mathcal{C}$ and let $<_A$ an order on $\mathbf{D}_A$ such that $\max_{a \in \mathbf{D}_A} |\mathrm{wcol}_r(\mathbf{D}_A, <_A, a)| = \mathrm{wcol}_r(\mathbf{D}_A)$. We define an new order $<'_A$ on $\mathbf{D}_A$ as follow :

- If a and b are real nodes in $\mathbf{D}_A$, then $a <_A b \iff a <'_A b$.
- If a is a real node and b is a tuple node, then $a <'_A b$.
- If a and b are both tuples nodes, the order can be chosen arbitrarily.

We also define an order $<_G$ on $\mathbf{D}_G$ that is just the order $<_A$ restricted to the real nodes.

Claim 4.3.9. $\forall a \in \boldsymbol{D}_G, |\mathrm{wcol}_r(\boldsymbol{D}_G, <_G, a)| \leq |\mathrm{wcol}_{2r}(\boldsymbol{D}_A, <'_A, a)|$

Claim 4.3.10. *For all real node a in $\boldsymbol{D}_A$,* $|\mathrm{wcol}_r(\boldsymbol{D}_A, <'_A, a)| \leq (k \times \mathrm{wcol}_r(\boldsymbol{D}_A))^{r-1}$

It is clear that proposition 4.3.8 is just the combination of the two claims.

Proof of claim 4.3.9. Let a be a node in $\mathbf{D}_G$ and $b \in \mathrm{wcol}_r(\mathbf{D}_G, <_G, a)$, let P be a witness path. By construction, there is a path P' of length $2r$ from a to b in $\mathbf{D}_A$ obtained by adding one tuple node between every two consecutive nodes from P. P' witnesses that $b \in \mathrm{wcol}_{2r}(\mathbf{D}_A, <'_A, a)$ which concludes the proof of Claim 4.3.9. □

Proof of Claim 4.3.10. We prove by induction on l that:

$$\forall l \leq r, \forall a \in \mathbf{D}_A, |\mathrm{wcol}_l(\mathbf{D}_A, <'_A, a)| \leq (k \times |\mathrm{wcol}_r(\mathbf{D}_A)|)^{l-1}$$

We start with $l = 2$. Let a a real node in $\mathbf{D}_A$. Let b be a node weakly 2-accessible from a in $\mathbf{D}_A$ with $<'_A$. Then there is a tuple node c such that $a - c - b$ is a path in $\mathbf{D}_A$. Since $b <'_A a$ implies $b <_A a$, we have that either $b <_A c$ and thus b is again weakly 2-accessible from a in $\mathbf{D}_A$ with $<_A$, or $c <_A b <_A a$ and then b is weakly 2-accessible

from a in $\mathbf{D}_A$ with $<_A$. Since c is a tuple node, it can only be connected to k real nodes. It follows that :

$$|\mathrm{wcol}_2(\mathbf{D}_A, <'_A, a)| \leq k \times |\mathrm{wcol}_r(\mathbf{D}_A)|$$

Let now fix any l. We suppose that for any $s \leq l$

$$|\mathrm{wcol}_s(\mathbf{D}_A, <'_A, a)| \leq (k \times \mathrm{wcol}_r(\mathbf{D}_A))^{s-1}$$

Let a a real node in $\mathbf{D}_A$. Let b be a node weakly l-accessible in $\mathbf{D}_A$ with $<'_A$. Let $a - c_1 - d_1 - c_2 - d_2 \cdots c_{\frac{l}{2}} - b$ be a witnessing path.

Remark 4.3.11. Since for all $i \leq l$ we have $b <'_A d_i$ we also have $b <_A d_i$ since every d_i is a real node as for a and b.

- First case, b a real node weakly l-accessible from a with $<_A$.
  There are only $\mathrm{wcol}_l(\mathbf{D}_A, <_A, a)$ of them.
- Second case, there is a tuple node c_i such that $c_i <_A b$ but $b <'_A c_i$. In this case, let j such that for all $i, c_j <_A c_i$. We have that :
  - $c_j \in \mathrm{wcol}_{2j}(\mathbf{D}_A, <_A, a)$. There are only $|\mathrm{wcol}_l(\mathbf{D}_A, <_A, a)|$ of them.
  - Given c_j there are only k possible such d_j since c_j is a tuple node.
  - $b \in \mathrm{wcol}_{l-2j}(\mathbf{D}_A, <'_A, d_j)$. Hence, with our induction hypothesis, given d_j there are only $(k \times \mathrm{wcol}_r(\mathbf{D}_A))^{l-2}$ such b.

By combining this, we get that

$$\mathrm{wcol}_l(\mathbf{D}_A, <'_A, a) \leq \mathrm{wcol}_r(\mathbf{D}_A) \times k \times (k \times \mathrm{wcol}_r(\mathbf{D}_A))^{l-2})$$

Hence

$$\mathrm{wcol}_l(\mathbf{D}_A, <'_A, a) \leq (k \times \mathrm{wcol}_r(\mathbf{D}_A))^{l-1}$$

□

This ends the proof of Proposition 4.3.8 and therefore of Theorem 4.3.6.

□

We have shown that for a class of structures $\mathcal{C}$ (over a fixed schema) σ. The class of graphs $\mathcal{C}_G$ is nowhere dense if and only if $\mathcal{C}_A$ is nowhere dense. Answering the question at the beginning of Section 4.3.2. We also would like to point out two facts.

The first one is that even if choosing gaifman over adjacency define the same notion of nowhere dense databases, it is not clear how the notion of gaifman graphs can help us to efficiently translate the results from graphs to databases. In the example that follows, we show that there exist different databases sharing a common gaifman graph. Contrariwise, the operation that associates to a database its adjacency graph is an injection.

Example 4.3.12. *Let $\sigma := \{R_1, R_2\}$ and two databases $\boldsymbol{D}_1, \boldsymbol{D}_2$ with domain (a, b), where R_1 and R_2 are binary relation symbols. Additionally, if $R_1(\boldsymbol{D}_1) = \{(a, b)\}$, $R_2(\boldsymbol{D}_1) = \emptyset$, $R_1(\boldsymbol{D}_2) = \{(a, b)\}$ and $R_2(\boldsymbol{D}_2) = \{(a, b)\}$, we obtained that $G(\boldsymbol{D}_1) = G(\boldsymbol{D}_2)$.*

The second fact is that fixing the schema is mandatory to get Theorem 4.3.6. Otherwise, we can build an example where only the adjacency class of graphs is nowhere dense (providing another argument for choosing adjacency over gaifman).

Example 4.3.13 (From [48]). *For all i in $\mathbb{N}$, we define $\sigma_i := \{R_i\}$ where R_i is a relation symbol of arity i. We also define $\boldsymbol{D}_i$ as the database with domain $\{a_1, \dots a_i\}$ where $(a_1, \dots a_i)$ is the only tuple in $R_i(\boldsymbol{D}_i)$. Finally, let $\mathcal{C} := \{\boldsymbol{D}_i \mid i \in \mathbb{N}\}$.*

We have that $\mathcal{C}_G$ is the class of all cliques (and thus somewhere dense) while $\mathcal{C}_A$ is the class of all stars (and thus nowhere dense).

4.4 Other tools

This section contains tools that are going to be used in the subsequent chapters.

4.4.1 Rank-Preserving Normal Form

Here and in the remaining, FO$^+$ is a syntactic extension of first-order logic. In FO$^+$ we are allowed to use distance queries as atoms. Allowing to use these distance-atoms does not increase the expressive power of first-order logic, as these atoms can clearly be expressed in FO, but it will lead to a different notion of quantifier-rank.

Definition 4.4.1. A *distance query* is a binary query that only talks about the distance of two nodes in a graph. For all $r \in \mathbb{N}$, the query $\mathrm{dist}_{\leq r}(a, b)$ state that there is a path of length at most r between a and b. It can formally be defined as follows:

$$\begin{array}{lcl} \mathrm{dist}_{\leq 0}(x, y) & := & (x = y) \\ \mathrm{dist}_{\leq r+1}(x, y) & := & \mathrm{dist}_{\leq r}(x, y) \ \vee \ \exists z \left(E(x, z) \wedge \mathrm{dist}_{\leq r}(z, y)\right) \end{array}$$

A slightly different definition can make the quantifier rank of $\mathrm{dist}_{\leq r}$ equal to $\log(r)$ instead of r here, but this is not important to us. Once this is defined, one can also make use of those variations:

$$\begin{array}{lcl} \mathrm{dist}_{=r}(x, y) & := & \mathrm{dist}_{\leq r}(x, y) \wedge \neg \mathrm{dist}_{\leq r-1}(x, y) \\ \mathrm{dist}_{>r}(x, y) & := & \neg \mathrm{dist}_{\leq r}(x, y) \end{array}$$

Following [43], we say that an FO$^+$-query φ has *q-rank* at most ℓ if it has quantifier-rank at most ℓ and each distance-atom $\mathrm{dist}_{\leq d}(x, y)$ in the scope of $i \leq \ell$ quantifiers satisfies $d \leq (4q)^{q+\ell-i}$. We also define (as in [43] and [45])

$$f_q(\ell) \quad := \quad (4q)^{q+\ell}.$$

Example 4.4.2. *Consider the query $\varphi_1(x, y) := \mathrm{dist}_{\leq 10000}(x, y)$. If we write it in first-order logic, its quantifier rank would be quite huge. Using the notion of q-rank, we have for example that φ_1 has 4-rank 0 and 3-rank 1. However, it does not have 3-rank 0, since $(4 \cdot 3)^{3+0} = 1728 < 10000$.*

Consider now a more complicated query where we also impose some restriction on the path from x to y. Let $\varphi_2(x,y) := \exists z\, B(z) \wedge \mathrm{dist}_{\leq 5000}(x,z) \wedge \mathrm{dist}_{\leq 5000}(z,y)$, where B is a unary predicate. Since both distance predicate are on the scope of one quantifier, φ_2 doest not have 3-rank 1 since $(4\cdot 3)^{3+1-1} = 1728 < 5000$. However, it has 3-rank 2 and 4-rank 1. Note that φ_2 does not have q-rank 0 for any q even when $(4\cdot q)^{q-1} > 5000$ because φ_2 has quantifier rank 1.

Definition 4.4.3. An *(r,q)-independence sentence* is an FO^+-sentence of the form

$$\exists z_1 \cdots \exists z_{k'} \Big(\bigwedge_{1\leq i<j\leq k'} \mathrm{dist}_{>r'}(z_i,z_j) \;\wedge\; \bigwedge_{1\leq i\leq k'} \psi(z_i) \Big)$$

where $k' \leq q$ and $r' \leq r$ and $\psi(z)$ is quantifier-free.

Definition 4.4.4. Given a graph G and a tuple $\overline{a} = (a_1,\ldots,a_k) \in V^k$, for all $r \in \mathbb{N}$ we define the *r-distance type* of $\overline{a}$ as the undirected graph $\tau_r^G(\overline{a})$ whose nodes are $[1,k]$, and where (i,j) is an edge if and only if $G \models \mathrm{dist}_{\leq r}(a_i,a_j)$. The set of all possible distance types with k elements is denoted $\mathcal{T}_k$.

Given a distance type τ, a *connected component* I of τ is a minimal subset of $[1,k]$ such that τ does not contain an edge (i,j) for any i in I and j not in I.

Theorem 4.4.5 (Rank-Preserving Normal Form, [45]). *Let $q,k \in \mathbb{N}$ such that $k \leq q$, let $\ell := q-k$, $r := f_q(\ell)$. For every $c \in \mathbb{N}$ we can compute a $c' \geq c$ such that the following is true for $\sigma := \sigma_c$ and $\sigma^\star := \sigma_{c'}$. For every $\mathrm{FO}^+[\sigma]$-formula $\varphi(\overline{x})$ of q-rank at most l where $\overline{x} = (x_1,\ldots,x_k)$, and for each distance type $\tau \in \mathcal{T}_k$ we can compute a number $m_\tau \in \mathbb{N}$ and, for each $i \leq m_\tau$,*

- *a Boolean combination ξ_τ^i of (r,q)-independence sentences of schema $\sigma^\star$ and*
- *for each connected component I of τ an $\mathrm{FO}^+[\sigma^\star]$-formula $\psi_{\tau,I}^i(\overline{x}_I)$ of q-rank at most ℓ*

such that the following holds:

For all c-colored graphs G and all kr-neighborhood covers $\mathcal{X}$ of G, there is a $\sigma^\star$-expansion $G^\star$ of G (which only depends on $G,\mathcal{X},q,\ell$) with the following properties, where V denotes the domain of G and $G^\star$:

(a) For all $\overline{a} \in V^k$ and for $\tau := \tau_r^G(\overline{a})$ we have:
$G \models \varphi(\overline{a})$ iff there is an $i \leq m_\tau$ that satisfies the following condition

- *$(*)_i$: $G^\star \models \xi_\tau^i$ and for every connected component I of τ there is an $X \in \mathcal{X}$ that r-covers $\overline{a}_I$ and $G^\star[X] \models \psi_{\tau,I}^i(\overline{a}_I)$.*

(b) For all $\overline{a} \in V^k$ and for $\tau := \tau_r^G(\overline{a})$, there is at most one $i \leq m_\tau$ such that the condition $(*)_i$ is satisfied.

(c) *For all* $\overline{a} \in V^k$, *all connected components* I *of* $\tau := \tau_r^G(\overline{a})$, *all* $X, X' \in \mathcal{X}$ *that both* r*-cover* $\overline{a}_I$, *and all* $i \leq m_\tau$,

$$G^\star[X] \models \psi^i_{\tau,I}(\overline{a}_I) \iff G^\star[X'] \models \psi^i_{\tau,I}(\overline{a}_I).$$

Furthermore, for every nowhere dense class $\mathcal{C}$ *of colored graphs, there is an algorithm which, when given as input a* $G \in \mathcal{C}$, *a* kr*-neighborhood cover* $\mathcal{X}$ *of* G *of degree at most* $|V|^\epsilon$, *and parameters* $q, \ell \in \mathbb{N}$, *computes* $G^\star$ *in time* $O(|V|^{1+\epsilon})$.

4.4.2 Removal Lemma

The Lemma we are going to present allows us to preserve every needed information when a node is removed from a graph. This will later be used together with the Splitter game.

Lemma 4.4.6 (Removal Lemma [45])**.** *Let* $k, q, \ell \in \mathbb{N}$. *There is an algorithm which takes as input a number* $c \in \mathbb{N}$, *a* c*-colored graph* G, *a* k*-ary query* $\varphi(\overline{z}) \in \mathrm{FO}^+[\sigma_c]$ *of* q*-rank at most* ℓ, *a set of free variables* $\overline{y} \subseteq \overline{z}$, *and a node* $s \in V$, *which produces*

- *a number* $c' \geq c$,
- *a query* $\varphi'(\overline{z} \setminus \overline{y}) \in \mathrm{FO}^+[\sigma_{c'}]$ *of* q*-rank at most* ℓ,
- *a* $\sigma_{c'}$*-expansion* G' *of* $G \setminus \{s\}$,

such that for all tuples $\overline{b}$ *over* V *where* $\{i \leq k \mid b_i = s\} = \{i \leq k \mid z_i \in \overline{y}\} =: I$ *we have*

$$G \models \varphi(\overline{b}) \iff G' \models \varphi'(\overline{b}_{\setminus I}).$$

Moreover, the running time of this algorithm is linear in the size of G, *and*

- c' *only depends on* c, q, ℓ,
- φ' *only depends on* $c, q, \ell, \varphi, \overline{y}$,
- G' *only depends on* c, q, ℓ, G, s.

Proof of Lemma 4.4.6. Let c, q and l be integers, G a c-colored graph, $\varphi(\overline{x})$ a k-ary query in $\mathrm{FO}^+[\sigma_c]$ of q-rank at most ℓ, $\overline{y} \subseteq \overline{z}$ a set of free variables, and s a node in G. Let $r = f_q(\ell)$, and $R_1, \ldots, R_r$ be r fresh unary predicate.

We define $c' := c + r$, and let φ' is a rewriting of the query φ. We introduce our rewriting notion noted $[\varphi]_\Delta$. Where Δ is a set of free variables (it is the set of all variables assigned to the removed node).

- For atom $P(x)$, unary predicate in σ:
  - $[P(x)]_\Delta := P(x)$ when $x \notin \Delta$
  - $[P(x)]_\Delta :=$ True iff $G \models P(s)$ when $x \in \Delta$

- For atom $E(x, y)$ (or equivalently for $x = y$):
  - $[E(x, y)]_\Delta := E(x, y)$ when $x \notin \Delta \wedge y \notin \Delta$
  - $[E(x, y)]_\Delta := R_1(x)$ when $x \notin \Delta \wedge y \in \Delta$ ($R_1(y)$ in the dual case)
  - $[E(x, y)]_\Delta :=$ True iff $G \models E(s, s)$ when $x \in \Delta \wedge y \in \Delta$
- For atom $\mathrm{dist}_{\leq i}(x, y)$ with $i \leq r$:
  - $[\mathrm{dist}_{\leq i}(x, y)]_\Delta := \mathrm{dist}_{\leq i}(x, y) \vee \bigvee_{\substack{0 \leq j, j' \leq r \\ j + j' \leq i}} \Big(R_j(x) \wedge R_{j'}(y) \Big)$ when $x \notin \Delta \wedge y \notin \Delta$.
  - $[\mathrm{dist}_{\leq i}(x, y)]_\Delta := \bigvee_{j \leq i} R_j(x)$ when $x \notin \Delta \wedge y \in \Delta$ ($\bigvee_{j \leq i} R_j(y)$ in the dual case)
  - $[\mathrm{dist}_{\leq i}(x, y)]_\Delta :=$ True when $x \in \Delta \wedge y \in \Delta$.
- Boolean combinations are not modified:

$$\begin{array}{rcl} [\psi(\overline{x}) \wedge \psi'(\overline{y})]_\Delta & := & [\psi(\overline{x})]_\Delta \wedge [\psi'(\overline{y})]_\Delta \\ [\neg\psi(\overline{x})]_\Delta & := & \neg[\psi(\overline{x})]_\Delta \end{array}$$

- For existential quantification:

$$[\exists z \psi(\overline{x}, z)]_\Delta := [\psi(\overline{x}, z)]_{(\Delta \cup \{z\})} \ \vee \ \exists z \, [\psi(\overline{x}, z)]_\Delta$$

Here, we define and return $\varphi'(\overline{x} \setminus \overline{y}) := [\varphi(\overline{x})]_\Delta$, where $\Delta := \overline{y}$ and G' as the unary expansion of $G \setminus \{c\}$ with $R_i(G') := N_i^G(s)$.

The proof that for all $\overline{a} \in G$ where $I := \{i \leq k \mid a_i = s\} = \{i \leq k \mid x_i \in \overline{y}\}$ we have:

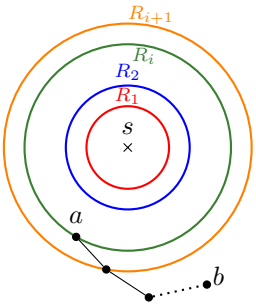
$$G \models \varphi(\overline{a}) \iff G' \models \varphi'(\overline{a}_{\setminus I})$$

is straightforward following the definition of $[\ ]_\Delta$.

This concludes the proof of Lemma 4.4.6. □

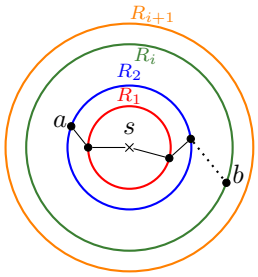
The next figure illustrates the different cases investigated in the Removal Lemma 4.4.6 for the query $\varphi(x, y) := \mathrm{dist}_{\leq r}(x, y)$.

Figure 4.1: How to recolor the graph when a node is removed.



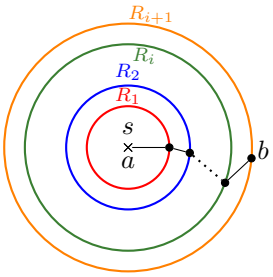
When there is a r-path not using s

the new query is: $\mathrm{dist}_{\leq r}(x, y)$



When s is on the only short path from a to b

the new query is: $\bigvee_{\substack{0 \leq j, j' \leq r \\ j+j' \leq r}} \Big(R_j(x) \wedge R_{j'}(y) \Big)$



When $a = s$ (similarly for $b = s$)

the new query is: $\bigvee_{j \leq r} R_j(y)$

Chapter 5

Testing FO queries

Contents

5.1 Introduction

This introduction is decomposed in two distinct parts. The first part is an introduction to Chapter 5, 6 and 7. The second part is dedicated to the specificity of Chapter 5.

5.1.1 Introduction to Chapters 5, 6 and 7

We have a pretty good idea of what kind of restrictions over structures make the model checking problem for FO queries tractable. More precisely, we have a dichotomy theorem saying that given a class $\mathcal{C}$ of graphs that is closed under subgraphs, the FO model checking over $\mathcal{C}$ is in FPT if and only if $\mathcal{C}$ is nowhere dense (unless FPT $=$ AW[$*$]). However, such precise statement does not exist for the other problems (testing,enumerating and counting). The big open question is then: Is there a precise dichotomy statement

that characterizes the tractability of the testing problem, the enumeration problem or the counting problem?

As we shall see, the answer is positive in all three cases. Moreover, the dichotomy result for each problem states that the tractability frontier is nowhere dense classes of graphs, like for the model checking problem.

For each of them, the lower bound part of the dichotomy theorem is straightforward. Thanks to Section 2.5.3 and 2.5.4, we know that for a fixed class of structures $\mathcal{C}$, the existence of either:

- an algorithm solving the counting problem for FO queries over $\mathcal{C}$ in polynomial time (or faster),
- an algorithm solving the enumeration problem for FO queries over $\mathcal{C}$ with polynomial delay after a polynomial preprocessing (or faster),
- an algorithm solving the testing problem for FO queries over $\mathcal{C}$ with polynomial answering time after polynomial preprocessing (or faster),

implies that the model checking problem for FO queries over $\mathcal{C}$ can be solved in polynomial time. If $\mathcal{C}$ is closed under subgraphs and somewhere dense, it is not the case (unless FPT $=$ AW[$*$]) [55].

Chapters 5, 6, and 7 are devoted to the upper bound concerning respectively the testing problem, the enumeration problem and the counting problem.

5.1.2 Introduction to Chapter 5

In this chapter we deal with the testing problem for FO queries. We recall what the testing problem is: Given a structure $\mathcal{A}$ and a query $q(\overline{x})$ in FO we want, given a tuple $\overline{a} \in A^k$ to decide whether $\overline{a} \in q(\mathcal{A})$.

The idea is to perform first some computation knowing $\mathcal{A}$ and q before $\overline{a}$ is given, and then, when $\overline{a}$ is known, to answer the question as quickly as possible. A testing algorithm for a structure $\mathcal{A}$ and a query q can then be decomposed in two steps:

- a preprocessing phase, and
- on input a tuple $\overline{a}$, an answering phase.

Ideally, we would like the second phase to only take constant time (independent from the size of $\mathcal{A}$). The *preprocessing time* of the testing algorithm is the time needed during the first phase.

The result that we are going to prove in details is that first-order queries can be tested in constant time after a pseudo-linear preprocessing over nowhere dense classes of graphs.

Theorem 5.1.1. *Let $q(\overline{x})$ be a first-order query and $\mathcal{C}$ a nowhere dense class of colored graphs. For every $\epsilon > 0$, there is an algorithm which on input a colored graph $G = (V, E) \in \mathcal{C}$ performs a preprocessing in time $O(|V|^{1+\epsilon})$ such that afterwards, upon input of a tuple $\overline{a}$, decides in time $O(1)$ whether $\overline{a} \in q(G)$.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, there is an algorithm which on input $\epsilon > 0$, a colored graph $G = (V, E) \in \mathcal{C}$, and a first-order query $q(\overline{x})$ performs a preprocessing in time $O(|V|^{1+\epsilon})$ such that afterwards, upon input of a tuple $\overline{a}$, decides in time $O(1)$ whether $\overline{a} \in q(G)$.

Here, the constants in the $O(\cdot)$ may depend on q, $\mathcal{C}$ and ϵ.
We decompose the proof of this theorem in two parts:

- The first one only proves Theorem 5.1.1 for some restricted queries: distance queries.
- The second one is the complete proof of Theorem 5.1.1, using the result of the first part.

The two parts follow the same schema. They are separated because while the second give all details of the proof, the first one allows us to give both a "high level view" of the proof and a detailed use of some tools.

5.2 Distance queries

A distance query is a binary query that only talks about the distance in the graphs of two nodes. For all $r \in \mathbb{N}$, the query $\mathrm{dist}_{\leq r}(a, b)$ state that there is a path of length less than r between a and b. It as been formally defined in Chapter 4 (more precisely in Section 4.4.1).

In this section, we prove Theorem 5.1.1 only for distance queries. This allows us to present the scheme of the more complicated proof that will be presented in the next section. We rephrase Theorem 5.1.1 for distance queries in the following proposition.

Proposition 5.2.1 (Theorem 5.1.1 for distance queries)**.** *Let $\mathcal{C}$ be a nowhere dense class of graphs. For all $\epsilon > 0$ and for all $r \in \mathbb{N}$ there is an algorithm which, upon input of a colored graph $G = (V, E) \in \mathcal{C}$, performs a preprocessing in time $O(|V|^{1+\epsilon})$, such that afterwards, upon input of a tuple $(a, b) \in V^2$, we can test in constant time whether $(a, b) \in \mathrm{dist}_{\leq r}(G)$.*

Here and in the following, V is the domain of G (the set of vertices). As previously, remember that the constants behind the big $O(\cdot)$ may depend on q, ϵ and $\mathcal{C}$.

5.2.1 The idea of the proof

We first give the ideas and ingredients of the proof. To do so, we are going to study the specific query $q(x, y) := \mathrm{dist}_{\leq 2}(x, y)$.

The first thing there is to notice is that given two nodes (a, b), to test whether $(a, b) \in \mathrm{dist}_{\leq 2}(G)$, we only need a small fraction of the graph: the neighborhoods of a and b. It is therefore tempting to precompute the 2-neighborhoods of all nodes. Unfortunately, this cannot be done in pseudo-linear time as the sum of the sizes of those neighborhoods might be too big. To overcome this situation we use a *neighborhood cover* that selects a small but sufficiently representative set of neighborhoods [43].

We recall some parts of Definition 4.2.5. Given a graph G, a number $r \in \mathbb{N}$ and $\epsilon > 0$, an $(r, 2r)$-neighborhood cover of degree $|V|^{\epsilon}$ of G is a collection $\mathcal{X}$ of subsets of V such that:

- $\forall a \in V \ \exists X \in \mathcal{X}$ with $N_r^G(a) \subseteq X$
- $\forall X \in \mathcal{X} \ \exists a \in V$ with $X \subseteq N_{2r}^G(a)$
- $\max_{a \in V} |\{X \in \mathcal{X} \mid a \in X\}| \leq |V|^\epsilon$

Moreover, for every $a \in V$ we arbitrarily fix a bag containing the r-neighborhood of a and denote it by $\mathcal{X}(a)$. We also recall that with Theorem 4.2.6, such neighborhood cover can be computed efficiently over nowhere dense classes of graphs.

Additionally, using the Storing Theorem 4.1.1, once the neighborhood cover is computed, given $a \in V$, and $X \in \mathcal{X}$, we can test in constant time whether $a \in X$. We also have access to $\mathcal{X}(a)$ in constant time.

We now explain how this notion will greatly help us. The radius of the relevant cover depends on the query. For our query $q = \text{dist}_{\leq 2}$, we have for all nodes a, b that

$$G \models q(a,b) \iff \mathcal{N}_2^G(a) \models q(a,b).$$

Therefore if we compute a $(2,4)$-neighborhood cover $\mathcal{X}$ of G, since $N_2^G(a) \subseteq \mathcal{X}(a)$, we have that for all nodes a and b:

$$G \models q(a,b) \iff G[\mathcal{X}(a)] \models q(a,b)$$

This cannot be the end of the preprocessing, when given $(a,b) \in V^2$, if we have that b does not belong to $\mathcal{X}(a)$ we can conclude that $(a,b) \notin q(G)$, otherwise we need some additional preprocessing within each bag X of $\mathcal{X}$.

Remark 5.2.2. What we are aiming at is a preprocessing on each bag X allowing us, given (a,b) in X to decide in constant time whether $G[X] \models \text{dist}_{\leq 2}(a,b)$. To do so, one can use specific properties of those bags. For example, in [68], the family of bags had bounded expansion. In [50], a testing algorithm with constant time answering and linear preprocessing was provided for first-order queries and classes of graphs with bounded expansion. The combination of the latter algorithm together with neighborhoods covers was sufficient.

Here, as we are working with nowhere dense graphs in general, the bags are just nowhere dense graphs with bounded radius. Therefore we cannot use the idea found in [68]. What we will see next is the needed preprocessing in the nowhere dense case.

The game characterization of nowhere dense classes of graphs (Definition 4.2.8), presented in the previous chapter, gives us an inductive parameter that decreases when diving within a bag. Recall that the game in question is a two player game (Splitter and Connector) and that while Connector tries to preserve the graph, Splitter will deconstruct it in finitely many moves. Our inductive parameter is the number of remaining rounds for Splitter before she wins the game with parameters $(\lambda(r'), r')$ when given an $(r'/2, r')$-neighborhood cover of G for a suitable number r'.

In the beginning of the preprocessing, we have computed an $(r, 2r)$-neighborhood cover. This implies that each bag X in $\mathcal{X}$ is included in the $2r$-neighborhood of some node. Our inductive parameter is the number of remaining rounds for Splitter before she wins the game starting with parameters $(\lambda(2r), 2r)$.

For our example, $r = 2$. A $(2, 4)$-neighborhood cover has then be computed. Let then λ be such that Splitter wins the $(\lambda, 4)$-splitter game on G. Assume that $\lambda = 1$. Then G is edgeless hence $G \models \mathrm{dist}_{\leq 2}(a, b)$ if and only if $a = b$.

Assume now that $\lambda > 1$. Let X be a bag of $\mathcal{X}$. By definition, there is an element c such that $X \subseteq N_4^G(c)$. Let s be Splitter's answer if Connector picks c in the game's first round. Then, by definition, Splitter wins the $(\lambda-1, 4)$-splitter game on $G[N_4^G(c) \setminus \{s\}]$. In particular, Splitter wins the $(\lambda-1, 4)$-splitter game on $G' := G[X \setminus \{s\}]$. Hence we can use an inductive argument within G'. For this we compute a new query $q'(x, y)$ such that for all a, b with $\mathcal{X}(a) = X$, $G \models q(a, b)$ if and only if $G' \models q'(a, b)$. Recall that we already know for such pairs a, b that $G \models q(a, b)$ if and only if $G[X] \models q(a, b)$. It therefore remains to encode the removal of the node s, and this is the Removal Lemma 4.4.6. In this case, it can be done by recoloring the nodes adjacent to s. Let R_1 and R_2 be new unary predicates such that:

$$\begin{array}{lll} w \in R_1(G') & \text{iff} & G \models \mathrm{dist}_{\leq 1}(w, s) \\ w \in R_2(G') & \text{iff} & G \models \mathrm{dist}_{\leq 2}(w, s) \end{array}$$

Those two predicates can easily be computed with a breadth first search on X.

The new query $q'(x, y)$ is then defined as a disjunction of the following queries

$$q(x, y) \ \vee \ \big(R_1(x) \wedge R_1(y) \big) \ \vee \ \big(R_2(x) \wedge y = s \big) \ \vee \ \big(R_2(y) \wedge x = s \big).$$

The first disjunct takes care of the general case, the second one for when the unique node connecting x and y is s, the third case when $y = s$ and the last one when $x = s$. As s is not part of G', the case $R_2(x) \wedge y = s$ should be understood as: if $y = s$, then just test whether x is in $R_2(G')$. Similarly for the last case.

The only part of the new query that is not easily computed is the first one: $q(x, y)$. However, we are now on an other graph G' where Splitter needs one less round to win the game. Therefore we can compute the preprocessing given by induction on G' allowing us, upon input of a tuple (a, b), to test in constant time whether $G' \models q(a, b)$.

Finally, we have available all technical notions and results that we need for presenting our testing algorithm restricted to distance queries.

5.2.2 The proof

We now turn into formal details. Let $\mathcal{C}$ be a nowhere dense class of graphs and $r \in \mathbb{N}$. Let $\epsilon > 0$ and G a graph in $\mathcal{C}$.

Until the end of this section, consider the particular number $2r$. For every $\lambda \in \mathbb{N}_{\geq 1}$ let $\mathcal{C}_\lambda$ be the subclass of $\mathcal{C}$ consisting of all graphs G such that Splitter wins the $(\lambda, 2r)$-splitter game on G. Clearly, $\mathcal{C}_\lambda \subseteq \mathcal{C}_{\lambda+1}$ for every λ. Since $\mathcal{C}$ is nowhere dense, by Theorem 4.2.9 there exists a number $\Lambda := \lambda(2r) \in \mathbb{N}$ such that $\mathcal{C} = \mathcal{C}_\Lambda$. Thus,

$$\mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq \cdots \subseteq \mathcal{C}_\Lambda = \mathcal{C}.$$

We proceed by induction on λ such that $G \in \mathcal{C}_\lambda$. The induction base for $\lambda = 1$ follows immediately, since by definition of the splitter game every $G \in \mathcal{C}_1$ has to be edgeless, and thus a naive algorithm works. For the induction step consider a $\lambda \geq 2$ and assume that Proposition 5.2.1 already holds for $\lambda-1$.

Let $n := |V|$ be the size of the domain V of G and let $\delta > 0$ such that $3\delta + 2\delta^2 \leq \epsilon$.

The preprocessing phase

The preprocessing phase is composed of the following steps:

1. Let $f_{\mathcal{C}}(r,\delta)$ be the number provided by the Neighborhood-cover Theorem 4.2.6. If $n \leq f_{\mathcal{C}}(r,\delta)$, we use a naive algorithm to compute the query result $\mathrm{dist}_{\leq r}(G)$ and trivially provide the functionality claimed by Proposition 5.2.1.

   From now on, consider the case where $n > f_{\mathcal{C}}(r,\delta)$. Recall that this implies that $\|G\| \leq O(n^{1+\delta})$.

2. Using the algorithm provided by Theorem 4.2.6, and since $n > f_{\mathcal{C}}(r,\delta)$, we compute a $(r,2r)$-neighborhood cover $\mathcal{X}$ of G with degree at most n^{δ}. Furthermore, in the same way as in [43, 45], we also compute for each $X \in \mathcal{X}$ a list of all $b \in V$ satisfying $\mathcal{X}(b) = X$, and we compute a node c_X such that $X \subseteq N^G_{2r}(c_X)$. All of those relations can be efficiently stored and retrieved with the Storing Theorem 4.1.1.

3. Recall that in step 2 we have already computed for every X in $\mathcal{X}$ a node c_X whose $2r$-neighborhood contains X. Since $G \in \mathcal{C}_\lambda$, we know that Splitter wins the $(\lambda, 2r)$-splitter game on G. For every $X \in \mathcal{X}$ we now compute a node s_X that is Splitter's answer if Connector plays c_X in the first round of the $(\lambda, 2r)$-splitter game on the graph G. From [43] we know that the nodes $(s_X)_{X\in\mathcal{X}}$ can be computed within total time $O(n^{1+\delta})$.

4. For every X in $\mathcal{X}$, we define X' as $G[X \setminus \{s_X\}]$ and we compute for every $i \leq r$:

$$R_i(X') \quad := \quad \{w \mid X \models \mathrm{dist}_{\leq i}(w, s_X)\}$$

   using a simple breadth first search. This recoloring is illustrated in Figure 4.1.

5. By Splitter's choice of the node s_X, we know that she wins the $(\lambda - 1, 2r)$-splitter game on X' for every X in $\mathcal{X}$. Therefore, for each X in $\mathcal{X}$, we spend time $O(|X'|^{1+\delta})$ for the preprocessing obtained by induction on Proposition 5.2.1 for the query $\mathrm{dist}_{\leq r}$. Allowing us, given (a,b) in X' to test in constant time whether $(a,b) \in \mathrm{dist}_{\leq r}(X')$.

This end the preprocessing, let's us show that it took time $O(n^{1+\epsilon})$. Step 1 only took time $O(1)$ and Step 2 and 3 time $O(n^{1+\delta})$. Step 4 and 5 took, for each X' time $O(\|X'\|^{1+\delta})$ leading to a total time:

$$\begin{aligned} O\Big(\sum_{X\in\mathcal{X}} \|X'\|^{1+\delta}\Big) &\leq O\Big(\Big(\sum_{X\in\mathcal{X}} \|X'\|\Big)^{1+\delta}\Big) \leq O\Big((n^{\delta}\|G\|)^{1+\delta}\Big) \\ &\leq O\left((n^{\delta}n^{1+\delta})^{1+\delta}\right) = O\left(n^{1+3\delta+2\delta^2}\right) \leq O(n^{1+\epsilon}). \end{aligned}$$

Answering and correctness

We are now given two nodes a and b. We want to answer in constant time the question: "Is (a,b) in $\mathrm{dist}_{\leq r}(G)$?" As the preprocessing is now complete, in constant time we have access to $\mathcal{X}(a)$ and we can test whether b is in $\mathcal{X}(a)$. If it is not the case, then clearly

$(a,b) \notin \mathrm{dist}_{\leq r}(G)$. Those tests can be performed in constant time using the Storing Theorem 4.1.1.

Otherwise, we have that $(a,b) \in \mathrm{dist}_{\leq r}(G)$ iff $(a,b) \in \mathrm{dist}_{\leq r}(G[\mathcal{X}(a)])$. We also have that $(a,b) \in \mathrm{dist}_{\leq r}(G[\mathcal{X}(a)])$ if and only if one of the following is true:

- $a \neq s_X$ and $b \neq s_X$, and $X' \models \mathrm{dist}_{\leq r}(a,b) \vee \bigvee_{\substack{1 \leq i,j \leq r-1 \\ i+j \leq r}} R_i(a) \wedge R_j(b)$
- $a \neq s_X$ and $b = s_X$, and $X' \models R_r(a)$
- $a = s_X$ and $b \neq s_X$, and $X' \models R_r(b)$
- $a = s_X$ and $b = s_X$.

Where $X' := G[\mathcal{X}(a) \setminus \{s_X\}]$. As we can test all of those in constant time, we are able to decide in constant time whether $(a,b) \in \mathrm{dist}_{\leq r}(G)$.

This ends the section about distance queries. We can now turn to the general case, following a similar schema. Moreover, we can now use Proposition 5.2.1 to simplify the complete upcoming proof.

5.3 The general algorithm

In this section, we are going to prove Testing Theorem 5.1.1. We will first give few extra tools that where not used for distance queries.

5.3.1 Some extra tools

Here, we will deal with queries of arbitrarily big arities (and not just binary distance queries). This more complicated proof works first by induction on the arity of the given query. The Unary Theorem 4.2.7 takes care of the base cases, arity 0 and 1.

As previously, we also use the splitter game characterization of nowhere dense classes of graphs (giving us a second induction parameter). However, the rewriting of the initial query when a node is removed is not as easy as for distance queries. To do so, we use the Removal Lemma 4.4.6.

The last tool that we need is a "local" normal form for FO queries which is rather standard for this kind of proof. However, most a the time, the Gaifman normal form of first-order queries [35] is sufficient. This is not the case for us as this normal form does not control the quantifier-rank of the local formulas it generates. As we would like to use it inductively for the queries derived by the Removal Lemma 4.4.6, the radius of our queries may increases during the induction— and we don't want this to happen. As a remedy, we use a locality theorem based on q-rank instead of quantifier-rank. It has the advantage that it preserves the q-rank within the local formulas. It was first introduced in [43] for unary queries. Here we use a stronger variant for queries of arbitrary arities: Theorem 4.4.5, proved in [45, Theorem 7.1].

We are now ready to turn to formal details.

5.3.2 The complete proof

Let $\mathcal{C}$ be a fixed nowhere dense class of undirected colored graphs. W.l.o.g. we assume that $\mathcal{C}$ is closed under taking subgraphs.

Now let us fix arbitrary numbers $k, q \in \mathbb{N}$ with $q \geq k$, let $\ell := q-k$, and let $r := f_q(\ell)$. Note that this is the same choice of parameters as for the Rank Preserving Normal Form Theorem 4.4.5. Our goal is to show that the statement of Theorem 5.1.1 is true for all k-ary queries φ of q-rank at most ℓ.

Until the end of this section, consider the particular number $2kr$. For every $\lambda \in \mathbb{N}_{\geq 1}$ let $\mathcal{C}_\lambda$ be the subclass of $\mathcal{C}$ consisting of all colored graphs G such that Splitter wins the $(\lambda, 2kr)$-splitter game on G. Clearly, $\mathcal{C}_\lambda \subseteq \mathcal{C}_{\lambda+1}$ for every λ. Since $\mathcal{C}$ is nowhere dense, by Theorem 4.2.9 there exists a number $\Lambda := \lambda(2kr) \in \mathbb{N}$ such that $\mathcal{C} = \mathcal{C}_\Lambda$. Thus,

$$\mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq \cdots \subseteq \mathcal{C}_\Lambda = \mathcal{C}.$$

We proceed by induction on λ such that $G \in \mathcal{C}_\lambda$. The induction base for $\lambda = 1$ follows immediately, since again, by definition of the splitter game every $G \in \mathcal{C}_1$ has to be edgeless, and thus a naive algorithm works. For the induction step consider a $\lambda \geq 2$ and assume that the statement of Theorem 5.1.1 already holds for $\lambda - 1$ and queries of arities not exceeding k.

We fix an $\epsilon > 0$ and an FO$^+$-query $\varphi(\overline{x})$ of arity k and q-rank at most ℓ. Our goal throughout the rest of this section is to provide an algorithm which upon input of a graph $G \in \mathcal{C}_\lambda$ performs a preprocessing phase using time $O(|V|^{1+\epsilon})$, such that afterwards, upon input of a tuple $\overline{a}$ in V^k, we can decide in constant time whether $\overline{a} \in \varphi(G)$.

We first describe the preprocessing phase and then the testing procedure. While describing these, we also provide a runtime analysis and a correctness proof.

The preprocessing phase

1. Let $\delta > 0$ such that $3\delta + 2\delta^2 < \epsilon$ and let $f_{\mathcal{C}}(2kr, \delta)$ be the number provided by the Neighborhood-cover Theorem 4.2.6. If $n \leq f_{\mathcal{C}}(2kr, \delta)$, we use a naive algorithm to compute the query result $\varphi(G)$ and trivially provide the functionality claimed by Theorem 5.1.1.

   From now on, consider the case where $n > f_{\mathcal{C}}(2kr, \delta)$. Recall that this implies that $\|G\| \leq O(n^{1+\delta})$.

2. Using the algorithm provided by Theorem 4.2.6, we compute a $(kr, 2kr)$-neighborhood cover $\mathcal{X}$ of G with degree at most n^δ.

   Furthermore, in the same way as in [43, 45], we also compute for each $X \in \mathcal{X}$ a list of all $b \in V$ satisfying $\mathcal{X}(b) = X$, and we compute a node c_X such that $X \subseteq N^G_{2kr}(c_X)$.

   Additionally, using the Storing Theorem 4.1.1 we are now able, given any node a, to compute $\mathcal{X}(a)$ in constant time. If a bag X is also given, We can test in constant time whether a is in X.

   All this can be done in time $O(n^{1+\delta})$.

3. Let σ be the schema of G, we use the algorithm provided by Rank-Preserving Normal Form Theorem 4.4.5 upon input of $\varphi, G, \mathcal{X}$ to compute in time $O(n^{1+\delta})$ the schema $\sigma^\star$, the $\sigma^\star$-expansion $G^\star$ of G, and for each distance type $\tau \in \mathcal{T}_k$ the number m_τ, the $\mathrm{FO}^+[\sigma^\star]$-sentences ξ^i_τ and the $\mathrm{FO}^+[\sigma^\star]$-formulas $\psi^i_{\tau,I}(\overline{x}_I)$ of q-rank at most ℓ, for each $i \leq m_\tau$ and each connected component I of τ.

   Afterwards, we proceed in the same way as in [43, 45] to compute, for every $X \in \mathcal{X}$ the structure $G^\star[X]$ in total time $O\Big(\sum_{x\in\mathcal{X}} \|X\|^{1+\delta}\Big)$.

   Note that $G^\star[X]$ has domain X and belongs to the class $\mathcal{C}_\lambda$.

4. Using the Model Checking Theorem 4.2.7, we can test in time $O(n^{1+\delta})$ for every distance type $\tau \in \mathcal{T}_k$ and $i \leq m_\tau$ whether $G^\star \models \xi^i_\tau$.

   We continue by performing the following preprocessing steps for all $\tau \in \mathcal{T}_k$ and every $i \leq m_\tau$ such that $G^\star \models \xi^i_\tau$. If there is no such τ and i then we can safely stop as $\varphi(G)$ is empty.

   We now consider $\tau \in \mathcal{T}_k$ and $i \leq m_\tau$ fixed.

5. Using Proposition 5.2.1, we spend time $O(n^{1+\delta})$ to compute on $G^\star$ a preprocessing allowing us, given a tuple $\overline{a} \in V^k$ to test in constant time whether $\tau_r(\overline{a}) = \tau$. This can be done because the query "$\tau_r(\overline{x}) = \tau$" is just a conjunction of distance queries.

   We now consider two different cases. The first case is when τ contains several connected components. If so, we go to Step 6 and then stop. Otherwise, i.e. τ is composed of only one component, we skip Step 6 and proceed.

6. Here, we investigate the case where τ contains several components $I_1, \ldots, I_{k'}$ with $1 < k' \leq k$. Since for every $j \leq k'$ we have that $|I_j| < k$, we can, for every $j \leq k'$ and every $X \in \mathcal{X}$ spend time $O(|X|^{1+\delta})$ to compute the preprocessing of Theorem 5.1.1 given by induction for queries of lower arities.

   After that, we will be able, given $\overline{a}_j$ of size $|I_j|$, to test in constant time whether $\mathcal{X}(\overline{a}_j) \models \psi^i_{\tau,I_j}(\overline{a}_{I_j})$. This can be done in time $O\Big(\sum_{x\in\mathcal{X}} |X|^{1+\delta}\Big)$.

   We then stop the preprocessing.

7. Here and in the following steps, we assume that τ contains only one connected component: I. Hence we cannot use the preprocessing given by induction since $|I| = k$. Therefore we have to make our second parameter decrease.

   What we want now is a preprocessing after which, given $\overline{a}$ in V^k, we can test in constant time whether:

$$G^\star[\mathcal{X}(\overline{a})] \models \psi^i_{\tau,I}(\overline{a}).$$

   As we don't know $\overline{a}$ in advance, we don't know $\mathcal{X}(\overline{a})$ either. The following steps are processed for every bag X in $\mathcal{X}$.

8. Now $\tau \in \mathcal{T}$, $i \leq m_\tau$ and $X \in \mathcal{X}$ are fixed. Let c_X be the node defined in Step 2, we have $X \subseteq N^{G^\star}_{2kr}(c_X)$. Let s_X be Splitter answer if Connector picks c_X. Then we have

that Splitter can win the $(\lambda - 1, 2kr)$-splitter game on $G^\star[X \setminus \{s_X\}]$. It remains to encode the removal of s_X.

For every $\overline{y} \subseteq \overline{x}$, we apply the Removal Lemma 4.4.6 to the colored graph $G^\star[X]$, the query $\psi^i_{\tau,I}(\overline{x})$, the variables $\overline{y}$, and the node s_X. This yields a query $\psi'^i_{\tau,I,\overline{y}}(\overline{x} \setminus \overline{y})$ of q-rank at most ℓ and an expansion $H^\star_X$ of $G^\star[X \setminus \{s_X\}]$ by unary predicates, such that for all k-tuples $\overline{a}$ over X with $\overline{y} = \{x_i \in \overline{x} \mid a_i = s_X\}$ and $\overline{a}' := \{a_i \in \overline{a} \mid a_i \neq s_x\}$ we have:

$$G^\star[X] \models \psi^i_{\tau,I}(\overline{a}) \quad \iff \quad H^\star_X \models \psi'^i_{\tau,I,\overline{y}}(\overline{a}').$$

All this can be achieved in time $O(\|X\|^{1+\delta})$.

9. Since Splitter wins the $(\lambda - 1, 2kr)$-splitter game on $G^\star[X \setminus \{s_X\}]$. We compute, for every $\overline{y} \subseteq \overline{x}$, the preprocessing given by induction for the query $\psi'^i_{\tau,I,\overline{y}}(\overline{x} \setminus \overline{y})$ on the graph $H^\star_X$, allowing us given $\overline{a}'$ of size $k - |\overline{y}|$ to test in constant time whether $\overline{a}' \in \psi'^i_{\tau,I,\overline{y}}(H^\star_X)$.

   This can be achieve in time $O(|X|^{1+\delta})$.

This ends the preprocessing. Let us show briefly that the running time is the promised one. Step 1 took time $O(1)$. Then Steps 2, 3, 4 and 5 took time $O(n^{1+\delta})$ each and Step 7 is just a discussion. The three other Steps: 6, 8 and 9 took at most time:

$$\begin{aligned} O\Big(\sum_{x\in\mathcal{X}} \|X\|^{1+\delta}\Big) &\leq O\left(\Big(\sum_{x\in\mathcal{X}} \|X\|\Big)^{1+\delta}\right) \leq O\left(\big(n^\delta \|G\|\big)^{1+\delta}\right) \\ &\leq O\Big(\big(n^\delta n^{1+\delta}\big)^{1+\delta}\Big) \leq O(n^{1+3\delta+2\delta^2}) \\ &\leq O(n^{1+\epsilon}) \end{aligned}$$

We can now turn to the second phase of our algorithm.

Answering and correctness

We assume that the preprocessing described above has been processed. We are now given a tuple $\overline{a}$ in V^k. Thanks to Step 5, we can compute in constant time its distance type $\tau := \tau_r(\overline{a})$.

We then have that $\overline{a} \in \varphi(G)$ if and only if there is an $i \leq m_\tau$ such that $G^\star \models \xi^i_\tau$ and for all components I of τ, $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)$ where $\overline{a}_I := \{a_i \in \overline{a} \mid i \in I\}$.

Thanks to Step 4, we know whether $G^\star \models \xi^i_\tau$ if this is the case we continue. We then have two distinct cases to investigate. The first one is when τ contains several components. In this first case, Step 6 of the preprocessing allows us to test in constant time whether $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)$ for each I in τ. We can then conclude.

In the second case, let $X := \mathcal{X}(a_1)$ (for instance). We have that X r-covers $\overline{a}$. Hence $\overline{a} \in \varphi(G)$ if and only if $G^\star[X] \models \psi^i_{\tau,I}(\overline{a})$. Let s_X the node chosen by Splitter in this scenario (computed in Step 8). Let $\overline{y} := \{x_i \in \overline{x} \mid a_i = s_x\}$ and $\overline{a}' := \{a_i \in \overline{a} \mid a_i \neq s_X\}$, with the removal lemma 4.4.6 we have:

$$G^\star[X] \models \psi^i_{\tau,I}(\overline{a}) \iff H^\star \models \psi'^i_{\tau,I,\overline{y}}(\overline{a}')$$

Using Step 9, we can decide in constant time whether $H^\star \models \psi'^{i}_{\tau,I,\overline{y}}(\overline{a}')$.

In summary, once the preprocessing is completed, upon input of a tuple $\overline{a}$ we can decide in constant time whether $\overline{a}$ is in $\varphi(G)$.

5.4 Conclusion

In this chapter, we proved that queries written in first-order logic can efficiently be tested over nowhere dense classes of graphs. Since the algorithms presented in Chapters 5, 6, and 7 are quite similar, we group our comments in Chapter 8.

Chapter 6

Enumerating FO queries

Contents

6.1 Introduction

In this chapter we deal with the *enumeration problem* for FO queries. We recall what the enumeration problem is: Given a structure $\mathcal{A}$ and a query $q(\overline{x})$ in FO we want, to enumerate the set $q(\mathcal{A})$.

The idea is to produce solutions one by one without waiting too much between two consecutive outputs. To help us in this task, we are allowed to perform first some computation knowing $\mathcal{A}$ and q. An enumeration algorithm for a structure $\mathcal{A}$ and a query q can be decomposed in two steps:

- a preprocessing phase, and
- an enumerating phase.

The *preprocessing time* of the enumeration algorithm is the time needed in the first phase. The *delay* that is the maximal time between two consecutive outputs during the second phase. Ideally we would like the delay to be constant (independent from the size of $\mathcal{A}$).

The result that we are going to prove in details is that first-order query can be enumerated with constant delay after a pseudo-linear preprocessing over nowhere dense classes of graphs.

Theorem 6.1.1 (Enumeration Theorem). *Let $\mathcal{C}$ be a nowhere dense class of undirected colored graphs and $q(\overline{x})$ a first-order query. For every $\epsilon > 0$, there is an algorithm which on input a graph $G = (V, E) \in \mathcal{C}$ performs a preprocessing in time $O(|V|^{1+\epsilon})$ such that afterwards, we can enumerate $q(G)$ with delay $O(1)$. Moreover, the tuples are enumerated lexicographically.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, there is an algorithm which on input $\epsilon > 0$, a colored graph $G = (V, E) \in \mathcal{C}$, and a first-order query $q(\overline{x})$ performs a preprocessing in time $O(|V|^{1+\epsilon})$ such that afterwards, we can enumerate $q(G)$ with delay $O(1)$. Moreover, the tuples are enumerated lexicographically.

Here, the constants in the $O(\cdot)$ may depend on $q, \mathcal{C}$ and ϵ. We assume that the input graph comes with an order on vertices, leading to a *lexicographical order* over tuples of vertices. We can assume that without loss of generality as a first step of any algorithm could be to fix an arbitrary order on the domain V of the input G.

6.2 The main algorithm

This section is devoted to the complete proof of the Enumeration Theorem 6.1.1. We will make use of notions and results presented in Chapter 4. We are also using the results obtained in the previous Chapter: the Testing Theorem 5.1.1 and the Testing Theorem for distance queries (Proposition 5.2.1).

We decompose this section into two distinct parts. The first one will only consider some queries as examples, while the second one contains the complete proof with all needed details. The first part is just here to introduce some new tools and gives a high level view of the proof.

6.2.1 The idea of the proof

We first give the ideas and ingredients of the proof.

The proof of Theorem 6.1.1 proceeds by induction on the number of free variables. The Unary Theorem 4.2.7 from [43] takes care of the base cases, arity 0 and 1.

Example 6.2.1 (1-A). *Consider the distance 2 query:*

$$q(x, y) \quad := \quad \mathrm{dist}_{\leq 2}(x, y) \quad = \quad \exists z \left(E(x, z) \wedge E(z, y) \right)$$

To enumerate all pairs (a, b) such that $G \models q(a, b)$, we first use the Unary Theorem 4.2.7 to compute the list of all a such that $G \models \exists y\, q(a, y)$. It remains to construct an index structure such that, on input an element a in this list, we can enumerate with constant delay all b such that $G \models q(a, b)$.

It is well-known that first-order queries are local in the sense that whether or not a tuple belongs to the query result $q(G)$ only depends on the r-neighborhood of the tuple, where the radius r depends only on the query, but not on G. It is therefore tempting to precompute the r-neighborhoods of all nodes. Unfortunately, this cannot be done in pseudo-linear time as the sum of the sizes of those neighborhoods might be too big. To

overcome this situation we use a *neighborhood cover* that selects a small but sufficiently representative set of neighborhoods [43].

We recall some parts of Definition 4.2.5. Given a graph G, a number $r \in \mathbb{N}$ and $\epsilon > 0$, an $(r, 2r)$-neighborhood cover of degree $|V|^\epsilon$ of G is a collection $\mathcal{X}$ of subsets of V such that:

- $\forall\, a \in V \;\; \exists\, X \in \mathcal{X}$ with $N_r^G(a) \subseteq X$
- $\forall\, X \in \mathcal{X} \;\; \exists\, a \in V$ with $X \subseteq N_{2r}^G(a)$
- $\max_{a \in V} |\{X \in \mathcal{X} \mid a \in X\}| \leq |V|^\epsilon$

Moreover, for every $a \in V$ we arbitrarily fix a bag containing the r-neighborhood of a and denote it by $\mathcal{X}(a)$. We also recall that with Theorem 4.2.6, such neighborhood cover can be computed efficiently over nowhere dense classes of graphs.

Let's get back to our running example.

Example 6.2.2 (1-B). *The radius of the relevant cover depends on the query. For our query q from Example (1-A), we have for all nodes a, b that*

$$G \models q(a,b) \quad \iff \quad \mathcal{N}_2^G(a) \models q(a,b).$$

Therefore if we compute a $(2,4)$-neighborhood cover $\mathcal{X}$ of G, since $N_2^G(a) \subseteq \mathcal{X}(a)$, we have that for all nodes a and b:

$$G \models q(a,b) \quad \iff \quad G[\mathcal{X}(a)] \models q(a,b).$$

Hence, modulo a pseudo-linear preprocessing, given an element a, to enumerate all b at distance 2 from a, it is enough to restrain our attention to the bag $\mathcal{X}(a)$ of a. We will see later why this is useful.

As illustrated by our running example, we reduce the general problem to enumerating the query results inside a bag of the cover. The game characterization of nowhere dense classes of graphs (Theorem 4.2.9) gives us a second inductive parameter that decreases when diving within a bag (recall that our first inductive parameter is the number of free variables). The splitter game in question is a two player game (Splitter and Connector) and while Connector tries to preserve the graph, Splitter tries deconstruct it. If the graphs comes from a nowhere dense class, then Splitter can win in finitely many moves (Definition 4.2.8 and Theorem 4.2.9).

Our second inductive parameter is the number of remaining rounds for Splitter before she wins the game starting with parameters $(\lambda(r'), r')$ when given an $(r'/2, r')$-neighborhood cover of G for a suitable number r'.

Example 6.2.3 (1-C). *For our running example, we have already computed a $(2,4)$-neighborhood cover $\mathcal{X}$ of G. Let then λ be such that Splitter wins the $(\lambda, 4)$-splitter game on G. Assume that $\lambda = 1$. Then G is edgeless and any naive algorithm can enumerate the solutions. Assume now that $\lambda > 1$. Let X be a bag of $\mathcal{X}$. By definition, there is an element c such that $X \subseteq N_4^G(c)$. Let s be Splitter's answer if Connector picks c in the game's first*

round. Then, by definition, Splitter wins the $(\lambda-1,4)$-splitter game on $G[N_4^G(c)\setminus\{s\}]$. In particular, Splitter wins the $(\lambda-1,4)$-splitter game on $G' := G[X\setminus\{s\}]$. Hence we can use an inductive argument within G'. For this we compute a new query $q'(x,y)$ such that for all a,b such that $\mathcal{X}(a)=X$, $G\models q(a,b)$ iff $G'\models q'(a,b)$. Recall that we already know for such pairs a,b that $G\models q(a,b)$ iff $G[X]\models q(a,b)$. It therefore remains to encode the removal of the node s, and this can be done by recoloring the nodes adjacent to s. Let R_1,R_2 be new unary predicates such that:

$$\begin{array}{lll} w\in R_1(G') & \text{iff} & G\models E(w,s)\\ w\in R_2(G') & \text{iff} & G\models \mathrm{dist}_{\leq 2}(w,s). \end{array}$$

The new query $q'(x,y)$ is then defined as a disjunction of the following queries

$$q(x,y)\ \vee\ \big(R_1(x)\ \wedge\ R_1(y)\big)\ \vee\ \big(R_2(x)\ \wedge\ y=s\big)\ \vee\ \big(R_2(y)\ \wedge\ x=s\big).$$

The first disjunct takes care of the general case, the second one for when the unique node connecting x and y is s, the third case when $y=s$ and the last one when $x=s$. As s is not part of G', the case "$R_2(x)\ \wedge\ y=s$" should be understood as listing all x in G' such that $G'\models R_2(x)$ and then outputting the pair (x,s). Similarly for the last case.

For the general proof, we need to generalize this idea to every first-order query. This is the Removal Lemma 4.4.6, presented in Chapter 4.

For our proof it is important that the radius of the neighborhood cover, and therefore the number of rounds Splitter needs to win the game, are fixed once for all at the beginning of the induction. It is well-known that this radius depends on the quantifier-rank of the formula. Unfortunately, the classical Gaifman Normal Form Theorem [35] does not control the quantifier-rank of the local formulas it generates. As we would like to use it inductively for the queries derived by Lemma 4.4.6, the radius may change during the induction — and we don't want this to happen. As a remedy, we use a locality theorem based on q-rank instead of quantifier-rank. It has the advantage that it preserves the q-rank within the local formulas. It was first introduced in [43] for unary queries. Here, we need a stronger variant for queries of arbitrary arities, proved in [45, Theorem 7.1]. This is the Rank-Preserving Normal Form Theorem 4.4.5.

Example 6.2.4 (2). *Our first query example is limited in the sense that all its solutions satisfy the same distance type requiring that x is close to y. We therefore consider a slightly more complicated query:*

$$q(x,y)\quad :=\quad \mathrm{dist}_{>2}(x,y)\ \wedge\ B(y)$$

where B is interpreted as the set of "blue" nodes of a colored graph. As before, the goal is, given a node a, to enumerate all blue nodes that are at distance greater than 2 from a. As previously, we compute a $(2,4)$-neighborhood cover, and given a node a, we consider the bag $X := \mathcal{X}(a)$.

We then start two concurrent enumeration processes: the first one only enumerates nodes that are within X; the second one enumerates those that are not in X. We can enumerate the union of the two queries by Lemma 2.5.7. The first one uses ideas previously presented,

diving into $G[X \setminus \{s\}]$ using the query constructed by Lemma 4.4.6 and the induction on λ. We now explain how the second one works.

Note that for every node b outside of X, we have $\mathrm{dist}(a,b) > 2$ as $N_2^G(a) \subseteq X$. Therefore, it suffices to enumerate all blue nodes that don't belong to X. To do so, during the preprocessing phase we compute for all nodes b' of G and all bags $X \in \mathcal{X}$ with $b' \in X$, the smallest blue node bigger than b' that is not in X; let us denote this node by $v(b', X)$. As the degree of our cover is pseudo-constant, the domain of the function $v(\cdot,\cdot)$ is pseudo-linear and the computation of the function can be done in pseudo-linear time. We also compute a function $NB(\cdot)$ that on input of a node b outputs the smallest blue node b' greater or equal than b. We can then use the preprocessing as follows:

Algorithm 8 On input a in G

```
X := 𝒳(a)
b := b_0
while b ≠ Null do
    b := NB(b)                          ▷ we only output blue nodes
    if b ∈ X then
        b := v(b, X)                    ▷ the blue node that is not in X
    end if
    Output (b)                          ▷ we are sure that b is blue and not in X
    b := b + 1                          ▷ we continue with the next node in the graph
end while
```

However, a naive extension of this idea for queries of bigger arities does not work: consider the query

$$q(x,y,z) \;:=\; \mathrm{dist}_{>2}(x,z) \wedge \mathrm{dist}_{>2}(y,z) \wedge B(z).$$

Assume that, given a pair (a,b) of nodes, we want to enumerate all nodes c such that $q(a,b,c)$ holds. Given a, b we consider the bags $X := \mathcal{X}(a)$ and $Y := \mathcal{X}(b)$. Now, we have three concurrent processes, one of them being in charge of enumerating all blue nodes that are neither in X nor in Y. The previous algorithm can enumerate all blue nodes that are not in X, but some of those may be in Y. Computing given c in $X \cup Y$ the smallest blue node c' bigger than c which falls out of $X \cup Y$, may require quadratic time and space. Therefore, we need another approach.

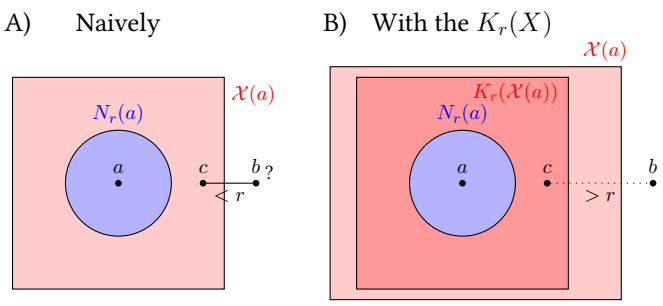
Moreover an other problem arises when diving into X to enumerate all nodes c in X such that $q(a,b,c)$ holds. We might lose information as b is not necessarily in X but some potential assignment for z in X might be close to b. See Figure 6.1

To overcome the second difficulty presented in the previous example, we introduce the notion of kernel.

Definition 6.2.5 (Kernel)**.** If $\mathcal{X}$ is a neighborhood cover of G with radius r, we define for all $X \in \mathcal{X}$ and $p \leq r$, the *p-kernel* of X as the set $K_p(X) := \{a \in V \mid N_p^G(a) \subseteq X\}$.

Lemma 6.2.6 ([43])**.** *Assume $\mathcal{X}$ is a neighborhood cover of G. Given a bag X and a number p, we can compute $K_p(X)$ in time $O(p \cdot \|G[X]\|)$.*

Figure 6.1: The use of $K_r(X)$



Proof. We prove the lemma by induction on p.

- If $p = 1$, let L be a list initialized empty. Then for each element a of X, we go through every neighbor of a. If we find one that is not in A, we add a to L and we go to the following element of X. At the end, we have $K_1(X) = X \setminus L$.
- If $p = i + 1$, we apply an inductive argument using the fact that $K_{i+1}(X) = K_1(K_i(X))$.

In the first case, the number of tests we have to performed is bounded by the number of edges plus the number of vertices in $G[X]$. Therefore the total computation time is bounded by $O(p \cdot \|G[X]\|)$. □

It only remains to extend the idea presented in the last example to queries with arbitrarily large arities.

Lemma 6.2.7 (*Skip pointers* [68]). *For every nowhere dense class $\mathcal{C}$ of undirected colored graphs, for every G in $\mathcal{C}$, for every $r > 0$, every $\epsilon > 0$, and every $k \in \mathbb{N}$, for every r-neighborhood cover $\mathcal{X}$ of G of degree at most $|V|^\epsilon$, and for every set $L \subseteq V$, there is a preprocessing algorithm working in time $O(|V|^{1+k\epsilon})$ allowing us, when given a node b and a set S of at most k bags of $\mathcal{X}$, to compute in constant time the node*

$$\mathrm{SKIP}(b, S) := \min\left\{b' \in L : b' \geq b \wedge b' \notin \bigcup_{X \in S} K_r(X)\right\}.$$

Proof. From now on we fix ϵ, r, G, $\mathcal{X}$ and L as in the statement of the lemma.

We assume that all kernels have been computed. This can be done during the preprocessing in time $O(\|G\|^{1+\epsilon})$ using Lemma 6.2.6.

The main issue here (exposed in the last example) is that the domain of the $\mathrm{SKIP}(\cdot,\cdot)$-function is too big (recall that there can be a linear number of bags). So we cannot compute it during the preprocessing phase. Fortunately, computing only a small part of it will be good enough for our needs. For each node b, we define by induction a set $SC(b)$ of sets of at most k bags. We start with $SC(b) = \emptyset$ and then proceed as follows.

- For all nodes b of G and for all bags X in $\mathcal{X}$ with b in the r-kernel of X (noted $K_r(X)$), we add $\{X\}$ to $SC(b)$.

- For all nodes b of G, for all sets S of bags from $\mathcal{X}$, and all bags X of $\mathcal{X}$, if $|S| < k$ and $S \in SC(b)$ and $\text{SKIP}(b, S) \in K_r(X)$, then we add $\{S \cup \{X\}\}$ to $SC(b)$.

In the preprocessing phase we will compute $\text{SKIP}(b, S)$ for all nodes b of G and all sets $S \in SC(b)$. Before explaining how this can be accomplished within the desired time constraints, we first show that this is sufficient for deriving $\text{SKIP}(b, S)$ in constant time for all nodes b and all sets S consisting of at most k bags of $\mathcal{X}$.

Claim 6.2.8. *Given a node b of G, a set S of at most k bags of $\mathcal{X}$, and $\text{SKIP}(c, S')$ for all nodes $c > b$ of G and all sets $S' \in SC(c)$, we can compute $\text{SKIP}(b, S)$ in constant time.*

Proof. We consider two cases (testing in which case we fall can be done in constant time as the kernels have been computed and S has size bounded by k).

Case 1: $b \in L$ and $b \notin \bigcup_{X \in S} K_r(X)$. In this case, b is $\text{SKIP}(b, S)$ and we are done.

Case 2: $b \notin L$ or $b \in \bigcup_{X \in S} K_r(X)$. In this case, let c be the smallest element of L strictly bigger than b. If there is no such c then $\text{SKIP}(b, S) = \text{Null}$ and we are done. Otherwise, we proceed as follows.

If $c \notin \bigcup_{X \in S} K_r(X)$, then c is $\text{SKIP}(b, S)$ and we are done. Otherwise, we know that $c \in K_r(X)$ for some $X \in S$. Therefore $\{X\} \in SC(c)$. Let S' be a maximal (w.r.t. inclusion) subset of S in $SC(c)$. Since $\{X\} \in SC(c)$, we know that S' is non-empty.

We claim that $\text{SKIP}(c, S') = \text{SKIP}(b, S)$. To prove this, let us first assume for contradiction that $\text{SKIP}(c, S') \in K_r(Y)$ for some $Y \in S$. By definition, this implies that Y is not in S'. Hence $|S'| < |S| \leq k$. Thus, by definition of $SC(c)$ we have $S' \cup \{Y\} \in SC(c)$ and S' was not maximal.

Moreover, by definition of $\text{SKIP}(c, S')$, every point between c and $\text{SKIP}(c, S')$ is either not in L or in some $K_r(Z)$ with $Z \in S'$ (and therefore $Z \in S$). As all nodes between b and c are not in L, the claim follows. □

We conclude by showing that $SC(b)$ is small for all nodes b of G and that we can compute efficiently $\text{SKIP}(b, S)$ for all nodes b and all sets $S \in SC(b)$.

Claim 6.2.9. *For each node b of G, $|SC(b)|$ has size $O(|V|^{k\epsilon})$. Moreover, it is possible to compute $\text{SKIP}(b, S)$ for all nodes b of G and all sets $S \in SC(b)$ in time $O(|V|^{1+k\epsilon})$*

Proof. We start by proving the first statement, and afterwards we use Claim 6.2.8 to show that we can compute these pointers inductively.

By $SC_\ell(b)$ we denote the subset of $SC(b)$ of sets S with $|S| \leq \ell$. Let d be the degree of the cover $\mathcal{X}$, i.e., $d = |V|^\epsilon$. By definition of d, we know that $|SC_1(b)| \leq d$ for all nodes b of G. For the same reason, we have that $|SC_{\ell+1}(b)|$ is of size at most $O(d \cdot |SC_\ell(b)|)$. Therefore, for all $b \in V$, we have:

$$|SC(b)| \;=\; |SC_k(b)| \;\leq\; O(d^k).$$

We compute the pointers for b from b_{max} to b_{min} downwards, where b_{max} and b_{min} are, respectively, the biggest and the smallest element of V. Given a node b in V, assume we

have computed SKIP(c, S') for all $c > b$ and $S' \in SC(c)$. We then compute SKIP(b, S) for $S \in SC(b)$ using Claim 6.2.8.

At each step the pointer is computed in constant time. Since there are $O(|V|^{1+k\epsilon})$ of them, the time required to compute them is as desired. □

Through all of those proofs, we use the Storing Theorem 4.1.1 to efficiently store and retrieve those skip pointers. More precisely, given a node b and a set $S \in SC(b)$, once SKIP(b, S) has been stored, it can be retrieve in constant time.

The combination of those two claims proves Lemma 6.2.7. □

Finally, we have available all technical notions and results that we need for presenting our constant-delay algorithm for enumerating the result of first-order queries.

6.2.2 The proof

Let $\mathcal{C}$ be a fixed nowhere dense class of undirected colored graphs. We prove the Enumeration Theorem 6.1.1 by induction on the arity k of the given query φ. The Unary Theorem 4.2.7 takes care of the induction base for $k = 0$ and $k = 1$, as precomputing the list of solutions is enough for our needs. For the induction step let us consider some arity $k \geq 2$ (that is fixed throughout the remainder of this section) and assume that the statement of Theorem 6.1.1 can be used for all queries of arity strictly smaller than k.

Now let us fix an arbitrary number $q \in \mathbb{N}$ with $q \geq k$, let $\ell := q-k$, and let $r := f_q(\ell) = 4q^{q+\ell}$. Note that this is the same choice of parameters as for the Rank Preserving Normal Form Theorem 4.4.5. Our goal is to show that the statement of Theorem 6.1.1 is true for all k-ary queries φ of q-rank at most ℓ. We prove the following, where $\overline{x} = (x_1, \ldots, x_{k-1})$ is a $(k-1)$-ary tuple of variables and x_k is a further variable.

Proposition 6.2.10. *For all $\epsilon > 0$ and all FO$^+$-queries $\phi(\overline{x}, x_k)$ of arity k and q-rank at most ℓ there exists an algorithm which, upon input of an undirected colored graph $G \in \mathcal{C}$, performs a preprocessing in time $O(|V|^{1+\epsilon})$, such that afterwards, upon input of a tuple $\overline{a} \in V^{k-1}$ we can enumerate with constant delay and increasing order all nodes $a_k \in V$ such that $G \models \varphi(\overline{a}, a_k)$.*

Before turning to the proof of this proposition, let us first argue that the proof of Theorem 6.1.1 is completed by an easy application of the proposition.

Proof of Theorem 6.1.1. Let $\phi(\overline{x}, x_k)$ be a first-order query of arity k and of q-rank at most ℓ. For a fixed $\epsilon > 0$, upon input of a $G \in \mathcal{C}$ perform the preprocessing phase of the algorithm provided by Proposition 6.2.10 for the query $\phi(\overline{x}, x_k)$. Furthermore, since $\psi(\overline{x}) := \exists x_k\, \phi(\overline{x}, x_k)$ is a query of arity $k-1$, by our induction hypothesis the statement of Theorem 6.1.1 already holds for $\psi(\overline{x})$. Thus, after $O(|V|^{1+\epsilon})$ preprocessing time we can enumerate with constant delay and in lexicographical order all $\overline{a} \in V^{k-1}$ such that $G \models \exists x_k\, \phi(\overline{a}, x_k)$. For each such $\overline{a}$ we use the enumeration procedure provided by Proposition 6.2.10. This allows us to enumerate, with constant delay and lexicographical order, all tuples $(\overline{a}, a_k)$ with $G \models \phi(\overline{a}, a_k)$.

□

The remainder of this section is devoted to the proof of Proposition 6.2.10. We consider the particular number $2kr$. For every $\lambda \in \mathbb{N}_{\geq 1}$ let $\mathcal{C}_\lambda$ be the subclass of $\mathcal{C}$ consisting of all graphs G such that Splitter wins the $(\lambda, 2kr)$-splitter game on G. Clearly, $\mathcal{C}_\lambda \subseteq \mathcal{C}_{\lambda+1}$ for every λ. Since $\mathcal{C}$ is nowhere dense, by Theorem 4.2.9 there exists a number $\Lambda := \lambda(2kr) \in \mathbb{N}$ such that $\mathcal{C} = \mathcal{C}_\Lambda$. Thus,

$$\mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq \cdots \subseteq \mathcal{C}_\Lambda = \mathcal{C}.$$

We proceed by induction on λ such that $G \in \mathcal{C}_\lambda$. The induction base for $\lambda = 1$ follows immediately, since by definition of the splitter game every $G \in \mathcal{C}_1$ has to be edgeless, and thus a naive algorithm works. For the induction step consider a $\lambda \geq 2$ and assume that the statement of Proposition 6.2.10 already holds for $\lambda - 1$ and queries of arities not exceeding k.

We fix an $\epsilon > 0$ and an FO$^+$-query $\phi(\overline{x}, x_k)$ of arity k and q-rank at most ℓ. Our goal throughout the rest of this section is to provide an algorithm which upon input of a $G \in \mathcal{C}_\lambda$ performs a preprocessing phase using time $O(|V|^{1+\epsilon})$, such that afterwards upon input of a tuple $\overline{a} \in V^{k-1}$ we can enumerate with constant delay and increasing order all nodes $a_k \in V$ such that $G \models \phi(\overline{a}, a_k)$.

We first describe the preprocessing phase, then the testing procedure, and afterwards the enumeration procedure. While describing these, we also provide a runtime analysis and a correctness proof.

The preprocessing phase

Consider the query $\phi(\overline{x}, x_k)$ with $\overline{x} = (x_1, \ldots, x_{k-1})$. We define $\delta > 0$ such that $3k\delta + 2\delta^2 \leq \epsilon$.

Let $G \in \mathcal{C}_\lambda$ be the input and let $n := |V|$ be the size of the domain V of G. The preprocessing phase is composed of the following steps:

1. Let $f_{\mathcal{C}}(2kr, \delta)$ be the number provided by Theorem 4.2.6. If $n \leq f_{\mathcal{C}}(2kr, \delta)$, we use a naive algorithm to compute the query result $\phi(G)$ and trivially provide the functionality claimed by Proposition 6.2.10.

   From now on, consider the case where $n > f_{\mathcal{C}}(2kr, \delta)$. Recall that this implies that $\|G\| \leq O(n^{1+\delta})$.

2. For every $k' < k$ and every distance type $\tau' \in \mathcal{T}_{k'}$, consider the k'-ary query $\rho_{\tau'}(x_1, \ldots, x_{k'})$ defined as the conjunction of the atoms $\mathrm{dist}_{\leq r}(x_i, x_j)$ for all edges $\{i, j\}$ of τ' and the conjunction of the formulas $\mathrm{dist}_{>r}(x_i, x_j)$ for all $i, j \in \{1, \ldots, k'\}$ with $i \neq j$ for which τ' does not contain the edge $\{i, j\}$. Note that for every $\overline{a} \in V^{k'}$ we have $G \models \rho_{\tau'}(\overline{a})$ iff $\tau' = \tau_r^G(\overline{a})$.

   We spend time $O(|V|^{1+\delta})$ to perform the preprocessing phase provided by the Testing Theorem 5.1.1 for the query $\rho_{\tau'}$. Henceforth, for each $k' < k$ and each $\tau' \in \mathcal{T}_{k'}$, this will enable us upon input of a tuple $\overline{a} \in V^{k'}$, to test in constant time whether $G \models \rho_{\tau'}(\overline{a})$, i.e., whether $\tau_r^G(\overline{a}) = \tau'$. (We actually only need the restriction of the Testing Theorem 5.1.1 to distance queries, i.e. Proposition 5.2.1.)

3. Using the algorithm provided by Theorem 4.2.6, we compute a $(kr, 2kr)$-neighborhood cover $\mathcal{X}$ of G with degree at most n^δ.

   Furthermore, in the same way as in [43, 45], we also compute for each $X \in \mathcal{X}$ a list of all $b \in V$ satisfying $\mathcal{X}(b) = X$, and we compute a node c_X such that $X \subseteq N^G_{2kr}(c_X)$.

   In addition, we use Lemma 6.2.6 to compute for every bag $X \in \mathcal{X}$ the r-kernel of X: $K_r(X) = \{a \in X \mid N^G_r(a) \subseteq X\}$.

   All of this can be efficiently stored and retrieved with the Storing Theorem 4.1.1. This can be done in time $O(n^{1+\delta})$.

4. Let σ be the schema of G and use the algorithm provided by the Rank-Preserving Normal Form Theorem 4.4.5 upon input of $\phi, G, \mathcal{X}$ to compute in time $O(n^{1+\delta})$ the schema $\sigma^\star$, the $\sigma^\star$-expansion $G^\star$ of G, and for each distance type $\tau \in \mathcal{T}_k$ the number m_τ, the FO$^+[\sigma^\star]$-sentences ξ^i_τ and the FO$^+[\sigma^\star]$-formulas $\psi^i_{\tau,I}(\overline{x}_I)$ of q-rank at most ℓ, for each $i \leq m_\tau$ and each connected component I of τ.

   Afterwards, we proceed in the same way as in [43, 45] to compute, within total time $O(n^{1+\delta})$, for every $X \in \mathcal{X}$ the structure $G^\star[X]$, and we let $G^\star_X$ be the expansion of $G^\star[X]$ where the new unary relation symbol K is interpreted by the r-kernel $K_r(X)$.

   Note that $G^\star_X$ has domain X and belongs to the class $\mathcal{C}_\lambda$.

5. By the Rank-Preserving Normal Form Theorem 4.4.5 we have for all $\overline{a} = (a_1, \ldots, a_{k-1}) \in V^{k-1}$ and all $a_k \in V$ that $G \models \varphi(\overline{a}, a_k)$ if and only if there is a distance type $\tau \in \mathcal{T}_k$ and an $i \leq m_\tau$ such that:

   (a) $\tau = \tau^G_r(\overline{a}, a_k)$

   (b) $G^\star \models \xi^i_\tau$

   (c) $G^\star[\mathcal{X}(b)] \models \psi^i_{\tau,J}(\overline{a}_J)$, where J is the connected component of τ with $k \in J$. Note that for a_k the bag $\mathcal{X}(a_k)$ r-covers the tuple $\overline{a}_J$, since $k \in J$, J is a connected component of $\tau = \tau^G_r(\overline{a}, a_k)$, $|J| \leq k$ and $\mathcal{X}$ is a kr-neighborhood cover of G.

   (d) For all connected components I of τ with $k \notin I$ we have $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)$, where $\mathcal{X}(\overline{a}_I)$ is defined to be $\mathcal{X}(a_{\min(I)})$; note that this bag r-covers $\overline{a}_I$.

   Using the Unary Theorem 4.2.7, we can test in time $O(n^{1+\delta})$ for every $\tau \in \mathcal{T}_k$ and $i \leq m_\tau$ whether $G^\star \models \xi^i_\tau$.

   We continue by performing the following preprocessing steps for all distance type $\tau \in \mathcal{T}_k$ and every number $i \leq m_\tau$ such that $G^\star \models \xi^i_\tau$. If there is no such τ and i then we can safely stop as there is no tuple $(\overline{a}, a_k)$ with $G \models \phi(\overline{a}, a_k)$.

6. For every connected component I of τ with $k \notin I$, we spend time $O(|X|^{1+\delta})$ to perform the preprocessing of the Testing Theorem 5.1.1, for the query $\psi^i_{\tau,I}(\overline{x}_I)$.

   Having performed this preprocessing will henceforth enable us, upon input of a tuple $\overline{a}_I$ of elements in X, to test in constant time whether $G^\star[X] \models \psi^i_{\tau,I}(\overline{a}_I)$.

7. Let J be the connected component of τ with $k \in J$ and note that x_k is the last variable in the tuple $\overline{x}_J$. Let $z_1, \ldots, z_{k-|J|}$ be new variables and consider for each $p \in \{0, \ldots, k-|J|\}$ the query

$$\psi^i_{\tau,p}(z_1, \ldots, z_p, \overline{x}_J) \;:=\; \psi^i_{\tau,J}(\overline{x}_J) \;\wedge\; K(x_k) \;\wedge\; \rho_\tau(\overline{x}_J) \;\wedge\; \bigwedge_{p' \in [1,p]} \operatorname{dist}(x_k, z_{p'}) > r$$

   Note that the query $\psi^i_{\tau,p}$ has q-rank at most ℓ.

   The idea of this query is to make sure that $\psi^i_{\tau,J}(\overline{x}_J)$ is satisfied and that the nodes of $\overline{x}$ that are not in $\overline{x}_J$ but fall in the bag $\mathcal{X}(x_k)$ are sufficiently far away from x_k. Since we don't know in advance how many there will be, we anticipate all possibilities (by considering every $p \leq k-|J|$). Note that the arity of the query $\psi^i_{\tau,p}$ is k for $p = k-|J|$, and it is smaller than k for smaller p.

   For every $X \in \mathcal{X}$ we would like to provide the following functionality: Upon input of a tuple of $p+|J|-1$ elements $c_1, \ldots, c_p, \overline{a}_{J\setminus\{k\}}$ we want to be able to enumerate with constant delay all a_k in X such that $G^\star_X \models \psi^i_{\tau,p}(c_1, \ldots, c_p, \overline{a}_{J\setminus\{k\}}, a_k)$.

   But as the query's arity $p+|J|$ might be as large as k, we do not have available the statement of the Enumeration Theorem 6.1.1 for this query. As a remedy, we perform the following steps 8–11 which make use of our second inductive assumption, stating that Proposition 6.2.10 already holds for the class $\mathcal{C}_{\lambda-1}$ and for queries of arity up to k.

8. Recall that in step 3 we have already computed for every X in $\mathcal{X}$ a node c_X whose $2kr$-neighborhood contains X. Since $G \in \mathcal{C}_\lambda$, we know that Splitter wins the $(\lambda, 2kr)$-splitter game on G. For every $X \in \mathcal{X}$ we now compute a node s_X that is Splitter's answer if Connector plays c_X in the first round of the $(\lambda, 2kr)$-splitter game on G. From [43] we know that the nodes $(s_X)_{X\in\mathcal{X}}$ can be computed within total time $O(n^{1+\delta})$.

9. Let $p \in \{0, \ldots, k-|J|\}$, let $k' := p+|J|$ and let $\overline{z} = (z_1, \ldots, z_{k'}) := (z_1, \ldots, z_p, \overline{x}_J)$. Recall that $k \in J$ and x_k is the last variable of the tuple $\overline{x}_J$, hence $x_k = z_{k'}$. For every set $\overline{y}$ of variables from $\overline{z}$, we proceed as follows. For every $X \in \mathcal{X}$ we apply the Removal Lemma 4.4.6 to the colored graph $G^\star_X$, the query $\psi^i_{\tau,p}(\overline{z})$, the variables $\overline{y}$, and the node s_X. This yields a query $\psi'^i_{\tau,p,\overline{y}}(\overline{z} \setminus \overline{y})$ of q-rank at most ℓ and an expansion $H^\star_X$ of $G^\star_X \setminus \{s_X\}$ by unary predicates, such that for all k'-tuples $\overline{b}$ over X where $\{i \leq k' \mid b_i = s_X\} = \{i \leq k' \mid z_i \in \overline{y}\} =: \Delta$ we have

$$G^\star_X \models \psi^i_{\tau,p}(\overline{b}) \quad\Longleftrightarrow\quad H^\star_X \models \psi'^i_{\tau,p,\overline{y}}(\overline{b}_{\setminus\Delta}).$$

   For every $X \in \mathcal{X}$, this takes times $O(\|X\|)$.

10. By our choice of the node s_X we know that Splitter wins the $(\lambda-1, 2kr)$-splitter game on $H^\star_X$. Hence, $H^\star_X$ belongs to $\mathcal{C}_{\lambda-1}$, for every $X \in \mathcal{X}$. Using our induction hypothesis of Proposition 6.2.10 we can thus spend for every X time at most $O(|X|^{1+\delta})$ to perform the preprocessing phase that allows us to enumerate with constant delay the

results of queries on $H^\star_X$, since the queries have arity at mos k and q-rank at most ℓ. Here, we carry this out for the queries $\psi'^i_{\tau,p,\overline{y}}(\overline{z} \setminus \overline{y})$, for all $\overline{y} \subseteq \overline{z}$.

Note that henceforth, this will allow us to do the following for every $X \in \mathcal{X}$:

For any $\overline{y}$ with $x_k \notin \overline{y}$, then when given an assignment $\overline{a}'$ in $X \setminus \{s_X\}$ to the variables in $\overline{z} \setminus (\overline{y} \cup \{x_k\})$, we can enumerate with constant delay and in increasing order all assignments $a_k \in X \setminus \{s_X\}$ to the variable $x_k = z_{k'}$ such that $H^\star_X \models \psi'^i_{\tau,p,\overline{y}}(\overline{a}', a_k)$.

11. In addition of the last step, using the algorithm provided by the Testing Theorem 5.1.1, we can, for every $\overline{y}$ with $x_k \in \overline{y}$ spend time $O(|X|^{1+\delta})$ allowing us, given an assignment $\overline{a}'$ in $X \setminus \{s_X\}$ to the variables in $\overline{z} \setminus \overline{y}$, to test in constant time whether $H^\star_X \models \psi'^i_{\tau,p,\overline{y}}(\overline{a}')$.

    This allows us to test whether s_X is also a node we want to enumerate

The next two steps are only performed when $J = \{k\}$; otherwise the preprocessing stops here. Note that if $J = \{k\}$, then the tuple $\overline{x}_J$ only consists of the variable x_k. Furthermore, for a tuple $\overline{a} = (a_1, \ldots, a_{k-1})$ and a node a_k, the tuple $\overline{a}_J$ consists of the single element a_k.

12. We compute the set

$$L^i_{\tau,J} \quad := \quad \{\, a_k \in V \mid G^\star[\mathcal{X}(a_k)] \models \psi^i_{\tau,J}(a_k) \,\}.$$

    This can be achieved as follows: For each $X \in \mathcal{X}$ use the algorithm provided by the Unary Theorem 4.2.7 to compute in time $O(|X|^{1+\delta})$ the result of the unary query $\psi^i_{\tau,J}$ on $G^\star_X$, and let L_X be the intersection of this query result with the list of all elements b with $\mathcal{X}(b) = X$ (recall that we already precomputed this list in step 3).

    Furthermore, $L^i_{\tau,J}$ is the disjoint union of the sets L_X for all $X \in \mathcal{X}$. It can therefore be computed in time $O(\sum_{X \in \mathcal{X}} |X|^{1+\delta})$.

13. We compute the skip pointers with respect to the set $L := L^i_{\tau,J}$ and $K_r(X)$ for all $X \in \mathcal{X}$ as in Lemma 6.2.7. By Lemma 6.2.7 this is done in time $O(n^{1+k\delta})$.

This concludes the preprocessing phase. We now give a complexity analysis. In the following, remember that the $O(\cdot)$ hides constants that may depends on, $\mathcal{C}$,ϕ or ϵ.

Most of the steps above are clearly doable in time $O(n^{1+\delta})$ or $O(n^{1+k\delta})$ for the last one. But for some of them we spend time up to $O(\|X\|^{1+\delta})$ for every $X \in \mathcal{X}$. Since a given edge (or a node) can only be found in n^δ different bags *(due to the degree of our cover)*, we have that:

$$O\Big(\sum_{X \in \mathcal{X}} \|X\|^{1+\delta}\Big) \;\leq\; O\left(\Big(\sum_{X \in \mathcal{X}} \|X\|\Big)^{1+\delta}\right) \;\leq\; O\left(\big(n^\delta \|G\|\big)^{1+\delta}\right)$$

Moreover, after Step 1, We have that $\|G\| \leq n^{1+\delta}$. Hence:

$$O\left(\big(n^\delta \|G\|\big)^{1+\delta}\right) \;\leq\; O\Big(\big(n^\delta n^{1+\delta}\big)^{1+\delta}\Big) \;\leq\; O(n^{1+3\delta+2\delta^2})$$

Therefore, the overall preprocessing can be performed in time $O(n^{1+3k\delta+2\delta^2})$. And since ϵ is bigger than $3k\delta + 2\delta^2$, we have that the overall preprocessing works in time $O(n^{1+\epsilon})$.

The enumeration phase

We now describe how, upon input of a tuple $\overline{a} = (a_1, \ldots, a_{k-1}) \in V^{k-1}$ we can enumerate with constant delay and increasing order all nodes $a_k \in V$ such that $G \models \phi(\overline{a}, a_k)$. What we actually do is the following: For a given pair (τ, i), we enumerate in increasing order all a_k such that:

(a) $\tau = \tau_r^G(\overline{a}, a_k)$,

(b) $G^\star \models \xi_\tau^i$,

(c) $G^\star[\mathcal{X}(a_k)] \models \psi_{\tau,J}^i(\overline{a}_J)$ where J is the connected component of τ with $k \in J$, and

(d) for all connected components I of τ with $k \notin I$ we have $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi_{\tau,I}^i(\overline{a}_I)$, where $\mathcal{X}(\overline{a}_I)$ denotes the bag $\mathcal{X}(a_{\min(I)})$.

As there are only a constant number of pairs (τ, i), we will enumerate all corresponding solutions concurrently and use this to obtain a global enumeration in lexicographical order with constant delay.

We now consider a fixed pair (τ, i). Let τ' be the subgraph of τ induced on $\{1, \ldots, k-1\}$, and use the functionality provided by step 2 of the preprocessing phase to test in constant time whether $\tau_r^G(\overline{a}) = \tau'$. If this is not the case, we know that for this τ, the condition of item (a) cannot be satisfied by any $a_k \in V$. Therefore, we can safely enumerate the empty set.

Otherwise, i.e., if $\tau_r^G(\overline{a}) = \tau'$, we proceed as follows.

By step 5 of the preprocessing phase, we can look up in constant time whether $G^\star \models \xi_\tau^i$. We thus know if item (b) is satisfied. Furthermore, using the functionality provided in step 6 of the preprocessing phase, we can test in constant time for all connected components I of τ with $k \notin I$, whether $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi_{G,I}^i(\overline{a}_I)$. Afterwards, we know if the item (d) is satisfied.

If one of the items (b) or (d) is not satisfied, we know that there is no matching solution for this $\overline{a}$ and (τ, i), and we can therefore safely enumerate the empty set.

Otherwise, i.e., if the items (b) and (d) are satisfied, we let J be the connected component of τ with $k \in J$, and we proceed with the two following cases.

Case I: $J = \{k\}$.

In this case, every matching solution a_k for this $\overline{a}$ and this (τ, i) has to be of distance greater than r to every element in $\overline{a}$. Consider the bags $\mathcal{X}(a_1), \ldots, \mathcal{X}(a_{k-1})$, let $k' := |\{\mathcal{X}(a_\nu) \mid \nu \in \{1, \ldots, k-1\}\}|$, and let $X_1, \ldots, X_{k'}$ be a list of these bags. Clearly, $k' \leq k-1$, and for each component a_ν of $\overline{a}$, there is exactly one κ such that $\mathcal{X}(a_\nu) = X_\kappa$.

For each $\kappa \leq k'$, we

- let p_κ be the number of elements in $\{a_1, \ldots, a_{k-1}\}$ that belong to X_κ, and we let $\overline{c}_\kappa := (c_{\kappa,1}, \ldots, c_{\kappa,p_\kappa})$ be a list of all these elements.

- Let $\overline{y}$ consist of the variables x_ν for all $\nu \in \{1, \ldots, k-1\}$ such that $a_\nu = s_{X_\kappa}$.
- Let $\overline{c}'_\kappa = (c'_{\kappa,1}, \ldots, c'_{\kappa,p'_\kappa})$ be a list of all elements of $\overline{c}_\kappa$ that are not equal to s_{X_κ}.

We execute simultaneously (using Lemma 2.5.7) the following $2k' + 1$ enumeration procedures:

- For each $\kappa \in \{1, \ldots, k'\}$ we want to enumerate all $a_k \in X_\kappa \setminus \{s_{X_\kappa}\}$ that are far away from all the nodes in the list $\overline{c}_\kappa$. More precisely, we want to enumerate all $b \in X_\kappa \setminus \{s_{X_\kappa}\}$ such that $G^\star_{X_\kappa} \models \psi^i_{\tau,p_\kappa}(\overline{c}_\kappa, a_k)$. Note that these are exactly all the nodes $a_k \in K_r(X_\kappa) \setminus \{s_{X_\kappa}\}$ that satisfy the items (a) and (c).

  From the statement made at the end of step 9 of the preprocessing phase, we know that to enumerate all these nodes b, we can use the functionality provided by step 10 of the preprocessing phase: We enumerate, with constant delay and in increasing order, all $b \in X_\kappa \setminus \{s_{X_\kappa}\}$ such that $H^\star_{X_\kappa} \models \psi'^i_{\tau,p_\kappa,\overline{y}}(\overline{c}'_\kappa, b)$.

- For each $\kappa \in \{1, \ldots, k'\}$ we want to check if $G^\star_{X_\kappa} \models \psi^i_{\tau,p_\kappa}(\overline{c}_\kappa, b)$ holds for the particular node $a_k := s_{X_\kappa}$, and if so, we want to output this node (otherwise, we enumerate the empty set).

  By the statement made at the end of step 9 of the preprocessing phase, this check can be performed by using the functionality provided by step 11 of the preprocessing phase:
  We simply check in constant time if $H^\star_{X_\kappa} \models \psi'^i_{\tau,p_\kappa,\overline{y}\cup\{x_k\}}(\overline{c}'_\kappa)$.

- Using the functionality provided by step 13 of the preprocessing phase, we enumerate with constant delay and in increasing order the set

$$\{\, b \in L \mid b \notin \bigcup_{\kappa \leq k'} K_r(X_\kappa) \,\}$$

  where $L := L^i_{\tau,J}$ is the set computed in step 12 of the preprocessing phase.

It should be clear that every a_k that is enumerated by this process is a matching solution for $\overline{a}$ and (τ, i). Let us now argue that our enumeration process produces *all* matching solutions for $\overline{a}$ and (τ, i): note that any matching solution a_k for $\overline{a}$ and (τ, i) is either in the canonical bag $\mathcal{X}(a_\nu)$ of one of the elements a_ν in $\overline{a}$, or it is outside all these bags. In the first case, b is produced by one of the enumeration procedures of the first two bullets. In the second case, b is enumerated by the skip pointers. Therefore, all matching solutions b will eventually be enumerated.

Case II: $\{k\} \subsetneq J$

W.l.o.g. let us assume that $1 \in J$ and that $\{1, k\}$ is an edge in τ.

Regarding item (c), note that the Rank-Preserving Normal Form Theorem 4.4.5 tells us that instead of the bag $\mathcal{X}(a_k)$ we can use *any* bag X that r-covers $\overline{a}_J$.

We define:

- $X := \mathcal{X}(a_1)$. Note that every $a_k \in V$ that satisfies item 1 belongs to X and, moreover, X r-covers $\overline{a}_J$ for $\overline{a} = (a_1, \ldots, a_{k-1})$.

- Let p be the number of elements of $\{a_1, \dots, a_{k-1}\}$ that belong to X but not to the tuple $\overline{a}_J$. Let $c_1, \dots, c_p$ be the list of all these elements.
- Let $c'_1, \dots, c'_{p'}$ be the elements of $c_1, \dots, c_p$ that are not equal to s_X.
- Let $\Gamma := J \setminus \{k\}$.
- Let $\overline{a}'_\Gamma$ be the tuple obtained from $\overline{a}_\Gamma$ by removing all components whose entry is s_X.
- Let $\overline{y}$ consist of the variables x_ν for all $\nu \in \{1, \dots, k-1\}$ such that $a_\nu = s_X$.

Since all matching a_k must be close to a_1 in this case, it suffices to run in parallel the following two enumeration procedures:

- We want to enumerate all $a_k \in X \setminus \{s_X\}$ that are far from all the nodes $c_1, \dots, c_p$. More precisely, we want to enumerate all $a_k \in X \setminus \{s_X\}$ such that:

  $$G^\star_X \models \psi^i_{\tau,p}(c_1, \dots, c_p, \overline{a}_\Gamma, a_k).$$

  Note that these are precisely the nodes $a_k \in V \setminus \{s_X\}$ that satisfy the items (a) and (c).

  From the statement made at the end of step 9 of the preprocessing phase, we know that to enumerate all these nodes a_k, we can use the functionality provided by step 10 of the preprocessing phase: We enumerate, with constant delay and in increasing order, all $a_k \in X \setminus \{s_X\}$ such that $H^\star_X \models \psi'^i_{\tau,p,\overline{y}}(c'_1, \dots, c'_{p'}, \overline{a}'_\Gamma, a_k)$.

- We want to check if $G^\star_X \models \psi^i_{\tau,p}(c_1, \dots, c_p, \overline{a}_\Gamma, a_k)$ holds for the particular node $a_k := s_X$, and if so, we want to output this node (otherwise, we enumerate the empty set).

  From the statement made at the end of step 9 of the preprocessing phase, we know that this check can be performed by using the functionality provided by step 11 of the preprocessing phase: We simply check in constant time if:

  $$H^\star_X \models \psi'^i_{\tau,p,\overline{y}\cup\{x_k\}}(c'_1, \dots, c'_{p'}, \overline{a}'_\Gamma).$$

This concludes the description of the enumeration procedure. While describing this procedure, we have already verified that it outputs exactly those $a_k \in V$ for which $G \models \phi(\overline{a}, b)$.

As we execute concurrently a constant number of enumeration processes and since all of them produce solutions in increasing order and with constant delay, by Lemma 2.5.7, we obtained a global enumeration with constant delay and increasing order. This completes the proof of Proposition 6.2.10 and hence also completes the proof of Theorem 6.1.1.

6.3 Conclusion

In this chapter, we proved that queries written in first-order logic can efficiently be enumerated over nowhere dense classes of graphs. Since the algorithms presented in Chapters 5, 6, and 7 are quite similar, we group our comments in Chapter 8.

Chapter 7

Counting FO queries

Contents

7.1 Introduction

In this chapter we deal with the *counting problem* for FO queries. We recall what the counting problem is: Given a structure $\mathcal{A}$ and a query $q(\overline{x})$ in FO we want, to compute $|q(\mathcal{A})|$ the size of the set $q(\mathcal{A})$.

The result that we are going to prove in details is that first-order queries can be counted over nowhere dense classes of graphs in pseudo linear time.

Theorem 7.1.1 (Counting Theorem)**.** *Let $q(\overline{x})$ be a first-order query and $\mathcal{C}$ a nowhere dense class of undirected colored graphs. For every $\epsilon > 0$, there is an algorithm which on input a graph $G = (V, E) \in \mathcal{C}$ computes in time $O(|V|^{1+\epsilon})$ the size $|q(G)|$ of the set $q(G)$.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, there is an algorithm which on input $\epsilon > 0$, a colored graph $G = (V, E) \in \mathcal{C}$ and a first-order query $q(\overline{x})$ computes in time $O(|V|^{1+\epsilon})$ the size $|q(G)|$ of the set $q(G)$.

Here, the constants in the $O(\cdot)$ may depend on $q, \mathcal{C}$ and ϵ.

This results has already been proved in [45], where "counting terms" are added in first-order logic, increasing its expressive power enough to express the counting problem as defined above. Here, we use some of their ideas (mainly the Rank-Preserving Normal Form Theorem 4.4.5) but the proof scheme is the one presented in [68] that is more or less a standard inclusion/exclusion argument.

7.2 The main algorithm

This section is devoted to the complete proof of the Counting Theorem 7.1.1. We will make use of notions and results presented in Chapter 4.

We decompose this section into two distinct parts. In the first part, we are going to define the normal form of first-order logic that is going to be used in the second part. We also explained how to compute efficiently this normal form. The second part only contains the main algorithm, promised by the Counting Theorem 7.1.1.

7.2.1 Local normal form

We first give the ideas and ingredients of the proof. The main idea of the proof is to decompose FO$^+$ queries into local queries.

Definition 7.2.1. An FO$^+$ query $\varphi(\overline{x})$ is *r-local* if it is of the form:

$$\varphi(\overline{x}) = \rho_{\tau,r}(\overline{x}) \ \wedge \varphi'(\overline{x})$$

where φ' is an FO$^+$ query and τ is a r-distance type with only one connected component.

The query $\rho_{\tau,r}(\overline{x})$ states that the r-distance type of $\overline{x}$ is exactly τ and is defined as follow:

$$\rho_{\tau,r}(\overline{x}) \ = \bigwedge_{(i,j)\in\tau} \mathrm{dist}_{\leq r}(x_i,x_j) \ \wedge \bigwedge_{(i,j)\notin\tau} \mathrm{dist}_{>r}(x_i,x_j)$$

Definition 7.2.2. Let G be a colored graph, k and r two integers, $\mathcal{X}$ a kr-neighborhood cover and φ an r-local FO$^+$ query of arity k we define:

$$\mathcal{S}(G,\mathcal{X},\varphi) \ := \ \left\{\overline{a}\in V^k \ \mid \ G^\star[\mathcal{X}(\overline{a})] \models \varphi(\overline{a})\right\}$$

$$\#(G,\mathcal{X},\varphi) \ := \ \left|\left\{\overline{a}\in V^k \ \mid \ G[\mathcal{X}(\overline{a})] \models \varphi(\overline{a})\right\}\right|$$

Example 7.2.3. *Consider the query $\varphi(x,y) := \mathrm{dist}_{>r}(x,y) \wedge B(x) \wedge R(y)$, where B and R are unary predicate representing Blue and Red nodes. The goal is then to count the number of Blue-Red nodes that are far apart. The main idea lies in the simple fact that a pair of Blue-Red nodes is not a solution if and only if it satisfies the query $\varphi_2(x,y) := \mathrm{dist}_{\leq r}(x,y) \wedge B(x) \wedge R(y)$. The latter having the advantage of being r-local without increasing the quantifier rank. We then have:*

$$\varphi(G) \ = \ \{a\in V \mid B(a)\} \times \{b\in V \mid R(b)\} \setminus \varphi_2(G)$$

Which leads to:

$$|\varphi(G)| \ = \ |\{a\in V \mid B(a)\}| \cdot |\{b\in V \mid R(b)\}| - |\varphi_2(G)|$$

And all those three queries are r-local which will be needed in the proof.

The next Theorem generalize the idea presented in Example 7.2.3. It allows us to restrict our attention to local queries.

Theorem 7.2.4 (Counting Normal Form). *Let $\mathcal{C}$ be a nowhere dense class of graphs, $\epsilon > 0$ and $\varphi(\overline{x})$ an FO^+ query of arity k and q-rank at most ℓ with $\ell = q - k$. Let $r := f_q(\ell) = 4q^{q+\ell}$. Then for every graph $G = (V, E)$ in $\mathcal{C}$ and every kr-neighborhood cover $\mathcal{X}$, there are:*

- *a unary expansion $G^\star$ of G,*
- *new r-local queries $\varphi_1(\overline{x}_1)$, $\varphi_2(\overline{x}_2)$, ..., $\varphi_\omega(\overline{x}_\omega)$ of q-rank at most ℓ and arities at most k,*
- *and an arithmetic function $\mathcal{F}$ of arity ω*

such that:

$$|\varphi(G)| \quad = \quad \mathcal{F}\Big(\#(G^\star, \mathcal{X}, \varphi_1), \ldots, \#(G^\star, \mathcal{X}, \varphi_\omega)\Big),$$

where ω is an integer that only depends on φ. Moreover, $G^\star$, $\varphi_1(\overline{x}_1)$, ..., $\varphi_\omega(\overline{x}_\omega)$ and $\mathcal{F}$ are computable in time $O(|V|^{1+\epsilon})$ and only $G^\star$ depends on G.

The proof of the Counting Normal Form Theorem mainly use the Rank-Preserving Normal Form Theorem 4.4.5 and the following Lemma.

Lemma 7.2.5. *Let G be a colored graph, k and r two integers, $\mathcal{X}$ a kr-neighborhood cover. Let τ be a r-distance type and ψ_1,...,ψ_m r-local queries of respective arity $k_1, \ldots, k_m$ with $\sum_{i \leq m} k_i = k$ and k_i being the size of the ith connected component of τ. Let q and ℓ be two integers with $\ell = q - k$, assume that every ψ_i has q-rank at most ℓ.*

Then in constant time we can compute an arithmetic function $\mathcal{F}$ and α new r-local queries $\varphi_1(\overline{x}_1), \ldots, \varphi_\alpha(\overline{x}_\alpha)$ of q-rank at most ℓ and arities at most k, such that:

$$\Big|\Big\{\overline{a} \in V^k \;\Big|\; \rho_{\tau,r}(\overline{a}) \wedge \bigwedge_{i \leq m} G[\mathcal{X}(\overline{a}_i)] \models \psi_i(\overline{a}_i)\Big\}\Big| \quad = \quad \mathcal{F}\Big(\#(G, \mathcal{X}, \varphi_1), \ldots, \#(G, \mathcal{X}, \varphi_\alpha)\Big)$$

Moreover, $\mathcal{F}$ and $(\varphi_i(\cdot))_{i \leq \omega}$ do not depend on G.

Remark 7.2.6. In the proof of Lemma 7.2.5, the number α of computed queries might be way bigger than the number m of initial queries. What is important is that α does not depends on the size of the input graph.

What is left of this section is used to prove Lemma 7.2.5 and Theorem 7.2.4.

Proof of Lemma 7.2.5. Let G be a colored graph, k and r two integers, $\mathcal{X}$ a kr-neighborhood cover, τ a r-distance type and ψ_1,...,ψ_m r-local queries .

The proof of Lemma 7.2.5 goes by induction on the number m of connected components in τ.

Case I: $m = 1$.

Here the Lemma is easily satisfiable by choosing $\varphi_1(\overline{x}) := \rho_{\tau,r}(\overline{x}) \wedge \psi_1(\mathcal{X})$ and $\mathcal{F}$ as the identity function.

Case II: $m > 1$. We use an argument similar to the one presented in Example 7.2.3.

We call τ_i the distance type of τ restricted to its ith connected component.

We say that a distance type τ' *strictly extends* τ (we note that $\tau \ll \tau'$) if τ is a subgraph of τ' and τ' as strictly less connected components than τ (and same domain).

Let $\overline{a} := \overline{a}_1, \dots, \overline{a}_m$ a k-tuple satisfying $G[\mathcal{X}(\overline{a}_i)] \models \rho_{\tau_i,r}(\overline{a}_i) \wedge \psi_i(\overline{a}_i)$ for every $i \leq m$. Then $\overline{a}$ does not satisfies $G \models \rho_{\tau,r}(\overline{a})$ iff and only if there is exactly one τ' that strictly extends τ such that $G \models \rho_{\tau',r}(\overline{a})$.

Hence we can also decompose $\overline{a}$ into $\overline{a}'_1, \dots, \overline{a}'_{m'}$ where m' is the number of connected components in τ' (implying that $m' < m$).

We now define a bunch of new queries, for every τ' with $\tau \ll \tau'$:

- For every $i \leq m$, let $\Psi_i(\overline{x}_i) := \rho_{\tau_i,r}(\overline{x}_i) \wedge \psi_i(\overline{x}_i)$.

  Note that $\Psi_i(\overline{x}_i)$ is r-local and that $\overline{a}_i \in \mathcal{S}(G, \mathcal{X}, \Psi_i)$.

- For every $j \leq m'$, let $\psi_{\tau',j}(\overline{x}'_j) := \rho_{\tau'_j,r}(\overline{x}_j) \wedge \bigwedge_{\substack{i \leq m \\ \tau_i \subseteq \tau'_j}} \Psi_i(\overline{x}_i)$.

  Here, even if we are talking about several connected components of τ, we have that $\psi_{\tau',j}(\overline{x}'_j)$ is still r-local because we only talk about the jth connected component of τ'.

From those definitions we derive that $\overline{a}$ is in one of the set:

$$\left\{\overline{b} \in V^k \;\middle|\; \rho_{\tau',r}(\overline{b}) \wedge \bigwedge_{j \leq m'} G[\mathcal{X}(\overline{b}'_j)] \models \psi_{\tau',j}(\overline{b}'_j)\right\}$$

The latter being included in

$$\prod_{i \leq m} \mathcal{S}(G, \mathcal{X}, \Psi_i)$$

By combining all those informations, we obtain the following equality:

$$\left\{\overline{a} \in V^k \;\middle|\; \rho_{\tau,r}(\overline{a}) \wedge \bigwedge_{i \leq m} G[\mathcal{X}(\overline{a}_i)] \models \psi_i(\overline{a}_i)\right\} =$$
$$\prod_{i \leq m} \mathcal{S}(G, \mathcal{X}, \Psi_i) \setminus \biguplus_{\tau' \gg \tau} \left\{\overline{a} \in V^k \;\middle|\; \rho_{\tau',r}(\overline{a}) \wedge \bigwedge_{j \leq m'} G[\mathcal{X}(\overline{a}'_j)] \models \psi_{\tau',j}(\overline{a}'_j)\right\}$$

And from this we derive a similar equation regarding the size of those sets:

$$\left|\left\{\overline{a} \in V^k \;\middle|\; \rho_{\tau,r}(\overline{a}) \wedge \bigwedge_{i \leq m} G[\mathcal{X}(\overline{a}_i] \models \psi_i(\overline{a}_i)\right\}\right| =$$
$$\prod_{i \leq m} \#(G, \mathcal{X}, \Psi_i) - \sum_{\tau' \gg \tau} \left|\left\{\overline{a} \in V^k \;\middle|\; \rho_{\tau',r}(\overline{a}) \wedge \bigwedge_{j \leq m'} G[\mathcal{X}(\overline{a}'_j)] \models \psi_{\tau',j}(\overline{a}'_j)\right\}\right|$$

And since every $\tau' \gg \tau$ as strictly less than m connected components, we can use our induction hypothesis leading to the fact that for every $\tau' \gg \tau$, we have:

- $\alpha_{\tau'}$ new r-local queries $(\varphi_{\tau',j})_{j \leq \alpha_{\tau'}}$ of arities at most k and q-rank at most ℓ

- an arithmetic function $\mathcal{F}_{\tau'}$ of arity $\alpha_{\tau'}$

such that:

$$\left|\left\{\overline{a} \in V^k \;\middle|\; \rho_{\tau',r}(\overline{a}) \wedge \bigwedge_{j \leq m'} G[\mathcal{X}(\overline{a}'_j)] \models \psi_{\tau',j}(\overline{a}'_j)\right\}\right| = \mathcal{F}_{\tau'}\Big(\#(G,\mathcal{X},\varphi_{\tau',1}),\ldots,\#(G,\mathcal{X},\varphi_{\tau',\alpha_{\tau'}})\Big)$$

Therefore, using the last two equations, we obtain our final result:

$$\left|\left\{\overline{a} \in V^k \;\middle|\; \rho_{\tau,r}(\overline{a}) \wedge \bigwedge_{i \leq m} G[\mathcal{X}(\overline{a}_i] \models \psi_i(\overline{a}_i)\right\}\right| = \prod_{I \leq m} \#(G,\mathcal{X},\Psi_i) \;-\; \sum_{\tau' \gg \tau} \mathcal{F}_{\tau'}\Big(\#(G,\mathcal{X},\varphi_{\tau',1}),\ldots,\#(G,\mathcal{X},\varphi_{\tau',\alpha_{\tau'}})\Big)$$

And this is indeed an arithmetic combination of terms of the form $\#(G,\mathcal{X},\varphi)$ for some new queries $(\varphi(\cdot))$. Moreover, everything that we have just done only depends on the inputs queries $\psi_1,\ldots,\psi_m$ and the distance type τ. The size of the input graph G does not impact the computation time, we can safely conclude that it only took time $O(1)$. □

We are now ready to prove the Counting Normal Form Theorem 7.2.4.

Proof of the Counting Normal Form Theorem 7.2.4. The proof of the Counting Normal Form Theorem 7.2.4 is composed of several steps. The first one is the Rank-Preserving Normal Form Theorem 4.4.5.

Let $\mathcal{C}$ be a nowhere dense class of colored graphs, $\epsilon > 0$, and $\varphi(\overline{x})$ an FO$^+$ query of arity k and q-rank at most ℓ with $\ell = q - k$. Let $G = (V,E)$ a colored graph in $\mathcal{C}$ and $\mathcal{X}$ a $(kr, 2kr)$-neighborhood cover of G. Using the Rank-Preserving Normal Form Theorem 4.4.5, in time $O(|V|^{1+\epsilon})$, we get:

- an expansion $G^\star$ of G,
- for each distance type $\tau \in \mathcal{T}_k$ a number m_τ,
- for each $i \leq m_\tau$, an FO$^+[\sigma^\star]$-sentences ξ^i_τ,
- for each connected component I of τ a query $\psi^i_{\tau,I}(\overline{x}_I)$ of q-rank at most ℓ,

such that for every tuple $\overline{a}$ in V^k, $\overline{a}$ is in $\varphi(G)$ if and only if there is exactly one distance type τ and one integer $i \leq m_\tau$ satisfying:

1. $\tau = \tau_r^G(\overline{a})$
2. $G^\star \models \xi^i_\tau$
3. For all connected components $I \sqsubseteq \tau$ we have $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)$, where $\mathcal{X}(\overline{a}_I)$ is defined to be $\mathcal{X}(a_{\min(I)})$; note that this bag r-covers $\overline{a}_I$.

We then have that:

$$\varphi(G) \;=\; \biguplus_{\substack{\tau\in\mathcal{T}_k, i\le m_\tau \\ G^\star\models\xi^i_\tau}} \Big\{\overline{a}\in V^k \;\Big|\; \rho_{\tau,r}(\overline{a}) \wedge \bigwedge_{I\sqsubseteq\tau} G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)\Big\}$$

From which we obtained that:

$$|\varphi(G)| \;=\; \sum_{\substack{\tau,i \\ G^\star\models\xi^i_\tau}} \Big|\Big\{\overline{a}\in V^k \;\Big|\; \rho_{\tau,r}(\overline{a}) \wedge \bigwedge_{I\sqsubseteq\tau} G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)\Big\}\Big|$$

We can now make use of our second tool, Lemma 7.2.5. For every distance type τ and for every $i \le m_\tau$, we can apply Lemma 7.2.5 with input $G^\star$, k, r, $\mathcal{X}$ and the queries $\psi^i_{\tau,I}$ for every $I \sqsubseteq \tau$ that gives us in constant time:

- α_τ new queries $\varphi_{1,\tau,i}, \ldots, \varphi_{\alpha_\tau,\tau,i}$ and
- An arithmetic function $\mathcal{F}_{\tau,i}$ of arity α_τ such that:

$$\Big|\Big\{\overline{a}\in V^k \;\Big|\; \rho_{\tau,r}(\overline{a}) \wedge \bigwedge_{I\sqsubseteq\tau} G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)\Big\}\Big| \;=\; \mathcal{F}_{\tau,i}\Big(\#(G^\star,\mathcal{X},\varphi_{1,\tau,i}),\ldots,\#(G^\star,\mathcal{X},\varphi_{\alpha_\tau,\tau,i})\Big)$$

The last piece of the puzzle is the Model Checking Theorem 4.2.7 allowing us, given τ and i to test in time $O(|V|^{1+\epsilon})$ whether $G^\star \models \xi^i_\tau$. We do that for every τ in $\mathcal{T}_k$ and every $i \le m$, and we compute the set $S := \{\tau \in \mathcal{T}_k, i \le m \mid G^\star \models \xi^i_\tau\}$.

By combining the last two equation and the definition of S, we obtain:

$$|\varphi(G)| \;=\; \sum_{(\tau,i)\in S} \mathcal{F}_{\tau,i}\Big(\#(G^\star,\mathcal{X},\varphi_{1,\tau,i}),\ldots,\#(G^\star,\mathcal{X},\varphi_{\alpha_\tau,\tau,i})\Big)$$

Which has the desired form. Concerning the running time of this computation, the first part (the Rank-Preserving Normal Form Theorem 4.4.5) is performed in time $O(|V|^{1+\epsilon})$. The second part (Lemma 7.2.5) only takes time $O(1)$. We perform this for every τ in $\mathcal{T}_k$ and every $i \le m_\tau$, therefore the total time spent during the second part remains $O(1)$. The third and last part (the Model Checking Theorem 4.2.7) takes time $O(|V|^{1+\epsilon})$. We also perform it for every τ in $\mathcal{T}_k$ and every $i \le m_\tau$. Overall, the total time spent for this construction is $O(|V|^{1+\epsilon})$.

This concludes the proof of Theorem 7.2.4. □

We now have every technical detail to prove the Counting Theorem 7.1.1.

7.2.2 The complete proof

Let $\mathcal{C}$ be a fixed nowhere dense class of undirected colored graphs. Contrary to the other chapters, the proof of the Counting Theorem 7.1.1 does not use an induction on the arity k of the given query φ.

Now let us fix $\epsilon > 0$ and arbitrary numbers $q, l, k \in \mathbb{N}$ with $\ell = q-k$, and let $r := f_q(\ell) = 4q^{q+\ell}$. Note that this is the same choice of parameters as for the Rank Preserving Normal Form Theorem 4.4.5. Our goal is to show that the statement of Theorem 7.1.1 is true for all k-ary queries φ of q-rank at most ℓ.

We consider the particular number $2kr$. For every $\lambda \in \mathbb{N}_{\geq 1}$ let $\mathcal{C}_\lambda$ be the subclass of $\mathcal{C}$ consisting of all graphs G such that Splitter wins the $(\lambda, 2kr)$-splitter game on G. Clearly, $\mathcal{C}_\lambda \subseteq \mathcal{C}_{\lambda+1}$ for every λ. Since $\mathcal{C}$ is nowhere dense, by Theorem 4.2.9 there exists a number $\Lambda := \lambda(2kr) \in \mathbb{N}$ such that $\mathcal{C} = \mathcal{C}_\Lambda$. Thus,

$$\mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq \cdots \subseteq \mathcal{C}_\Lambda = \mathcal{C}.$$

We proceed by induction on λ such that $G \in \mathcal{C}_\lambda$. The induction base for $\lambda = 1$ follows immediately, since by definition of the splitter game every $G \in \mathcal{C}_1$ has to be edgeless, and thus a naive algorithm works. For the induction step consider a $\lambda \geq 2$ and assume that the statement of the counting Theorem 7.1.1 already holds for every G in $\mathcal{C}_{\lambda-1}$.

We fix an FO$^+$-query $\varphi(\overline{x})$ of arity k and q-rank at most ℓ. Then, let $\delta > 0$ such that $\epsilon \geq 3\delta + 2\delta^2$.

Our goal throughout the rest of this section is to provide an algorithm which upon input of a $G \in \mathcal{C}_\lambda$ compute $|\varphi(G)|$. To do so, we apply the following steps.

1. Let $f_{\mathcal{C}}(2kr, \delta)$ be the number provided by Theorem 4.2.6. If $n \leq f_{\mathcal{C}}(2kr, \delta)$, where $n := |V|$, we use a brute-force algorithm to compute the result $|\varphi(G)|$.

   From now on, consider the case where $n > f_{\mathcal{C}}(2kr, \delta)$. Recall that this implies that $\|G\| \leq O(n^{1+\delta})$.

2. Using the algorithm provided by Theorem 4.2.6, we compute a $(kr, 2kr)$-neighborhood cover $\mathcal{X}$ of G with degree at most n^δ.

   Furthermore, in the same way as in [43, 45], we also compute for each $X \in \mathcal{X}$ a list of all $b \in V$ satisfying $\mathcal{X}(b) = X$, and we compute a node c_X such that $X \subseteq N^G_{2kr}(c_X)$.

   All of this can be efficiently stored and retrieved with the Storing Theorem 4.1.1. This can be done in time $O(n^{1+\delta})$.

3. We use the algorithm provided by the Counting Normal Form Theorem 7.2.4 upon input of $\varphi, G, \mathcal{X}$ to compute in time $O(n^{1+\delta})$:

   - a unary expansion $G^\star$ of G,
   - new r-local queries $\varphi_1(\overline{x}_1), \varphi_2(\overline{x}_2), \ldots, \varphi_\omega(\overline{x}_\omega)$ of q-rank at most ℓ and arities at most k,
   - and an arithmetic function $\mathcal{F}$ of arity ω, such that:

   $$|\varphi(G)| = \mathcal{F}\Big(\#(G^\star, \mathcal{X}, \varphi_1), \ldots, \#(G^\star, \mathcal{X}, \varphi_\omega)\Big)$$

   What remains to compute is, for every $i \leq \omega$, the number $\#(G^\star, \mathcal{X}, \varphi_i)$.

4. Recall that in step 2 we have already computed for every X in $\mathcal{X}$ a node c_X whose $2kr$-neighborhood contains X. Since $G \in \mathcal{C}_\lambda$, we know that Splitter wins the $(\lambda, 2kr)$-Splitter game on G. For every $X \in \mathcal{X}$ we now compute a node s_X that is Splitter's answer if Connector plays c_X in the first round of the $(\lambda, 2kr)$-splitter game on G. From [43] we know that the nodes $(s_X)_{X \in \mathcal{X}}$ can be computed within total time $O(n^{1+\delta})$.

5. For a fixed $i \leq \omega$, let k_i be the arity of φ_i and $\overline{x}_i = (x_{i,1}, \ldots, x_{i,k_i})$ its variables. We want to restrict our attention to bags of the cover. The difficulty is that $\#(G^\star, \mathcal{X}, \varphi_i)$ (which is what we want to compute) is not equals to $\sum_{X \in \mathcal{X}} |\varphi_i(G^\star_X)|$ since some tuples can be satisfied in several bags. In order to proceed, we need to add this constraint within the query.

   - For every bag X in $\mathcal{X}$, we define a new unary relation P, and $a \in P$ if and only if $\mathcal{X}(a) = X$. Using Step 2 and the Storing Theorem 4.1.1, this can be done in time $O(n^{1+\epsilon})$.
   - Let $\varphi'_i(\overline{x}_i) := \varphi_i(\overline{x}_i) \wedge P(x_{i,1})$. Note that φ'_i is still r-local. Moreover, we now have: $\#(G^\star, \mathcal{X}, \varphi_i) = \sum_{X \in \mathcal{X}} |\varphi'_i(G^\star_X)|$

   The next two steps compute $|\varphi'_i(G^\star_X)|$.

6. For every set $\overline{y}$ of variables from $\overline{x}_i$, we proceed as follows. For every $X \in \mathcal{X}$ we apply the Removal Lemma 4.4.6 to the colored graph $G^\star_X$, the query $\varphi'_i(\overline{x}_i)$, the variables $\overline{y}$, and the node s_X. This yields a query $\varphi'_{i,y}$ of q-rank at most ℓ and an expansion $H^\star_X$ of $G^\star_X \setminus \{s_X\}$ by unary predicates, such that for all k_i-tuples $\overline{a}_i$ over X where $\Delta := \{j \leq k_i \mid a_{i,j} = s_X\} = \{j \leq k_i \mid x_{i,j} \in \overline{y}\}$ we have

$$G^\star_X \models \varphi'_i(\overline{a}_i) \quad \Longleftrightarrow \quad H^\star_X \models \varphi'_{i,\overline{y}}(\overline{a}_{i \setminus \Delta}).$$

   For every $X \in \mathcal{X}$, this takes times $O(\|X\|)$.

7. By our choice of the node s_X we know that Splitter wins the $(\lambda-1, 2kr)$-Splitter game on $H^\star_X$. Hence, $H^\star_X$ belongs to $\mathcal{C}_{\lambda-1}$, for every $X \in \mathcal{X}$. Using our induction hypothesis of the Counting Theorem 7.1.1 we can thus spend for every X time at most $O(|X|^{1+\delta})$ to compute $|\varphi'_{i,\overline{y}}(H^\star_X)|$ since the query has arity at most k and q-rank at most ℓ, and therefore $r = f_q(\ell)$ does not increase.

   For the special case where $\overline{y}$ equals $\overline{x}_i$, the query $\varphi'_{i,\overline{x}_i}$ is a boolean one. In order to keep the same notation, we say that $|\varphi'_{i,\overline{x}_i}(H^\star_X)|$ equals 1 if $H^\star_X \models \varphi'_{i,\overline{x}_i}$ and 0 otherwise.

   We can then compute $|\varphi'_i(G^\star_X)|$ using:

$$|\varphi'_i(G^\star_X)| = \sum_{\overline{y} \subseteq \overline{x}_i} |\varphi'_{i,\overline{y}}(H^\star_X)|$$

   And then:

$$\#(G^\star, \mathcal{X}, \varphi_i) = \sum_{X \in \mathcal{X}} |\varphi'_i(G^\star_X)| = \sum_{X \in \mathcal{X}} \sum_{\overline{y} \subseteq \overline{x}_i} |\varphi'_{i,\overline{y}}(H^\star_X)|$$

We can then conclude by computing Steps 6 and 7 for every $i \leq \omega$ and apply $\mathcal{F}$ to the obtained results.

We now take some time to look at the time complexity of this entire process. Steps 1 and 5 only take time $O(1)$. Steps 2, 3 and 4 take time $O(n^{1+\delta})$ each.

Steps 6 and 7 are both preformed for every $i \leq \omega$ and every $X \in \mathcal{X}$ and they are both using time bounded by $O(\|(X)\|^{1+\delta})$.

$$O\Big(\sum_{X\in\mathcal{X}} \|X\|^{1+\delta}\Big) \leq O\left(\Big(\sum_{X\in\mathcal{X}} \|X\|\Big)^{1+\delta}\right) \leq O\left(\big(n^{\delta}\|G\|\big)^{1+\delta}\right)$$

Moreover, after Step 1, We have that $\|G\| \leq n^{1+\delta}$. Hence:

$$O\left(\big(n^{\delta}\|G\|\big)^{1+\delta}\right) \leq O\Big(\big(n^{\delta}n^{1+\delta}\big)^{1+\delta}\Big) \leq O(n^{1+3\delta+2\delta^2})$$

Therefore, the overall preprocessing can be performed in time $O(n^{1+3\delta+2\delta^2})$. Since ϵ is bigger than $3\delta + 2\delta^2$, we have that the overall preprocessing works in time $O(n^{1+\epsilon})$.

7.3 Conclusion

In this chapter, we proved that queries written in first-order logic can efficiently be counted over nowhere dense classes of graphs. Since the algorithms presented in Chapters 5, 6, and 7 are quite similar, we group our comments in Chapter 8.

Chapter 8

Conclusion

We have seen in the previous chapters, that, given a query written in first-order logic and a graph that belongs to some nowhere dense class of graphs, we can efficiently count the number of tuples satisfying the query, enumerate those tuples with constant delay and test in constant time whether a given tuple is a solution. This fills the gap between the existing algorithms for smaller classes of graphs and the lower bound for classes of graphs that are closed under subgraphs but not nowhere dense.

We conclude this thesis by discussing some possible improvements of our work and presenting interesting questions that could lead to future works. We first talk about the importance of the constant factors hidden in our algorithms. After that, we provide precisions about the pseudo-linear notion. We then point out the main question that are still open in this area: query evaluation for dense classes of graphs and query evaluation with updates.

Constant factors. We first add some precisions about the constant factors of our algorithms. In Chapter 5, 6, and 7, we made precise the dependency on $|G|$ for the running time of our algorithms. However the impact of $|\varphi|$ remains unclear.

The constant factor produced by each of our algorithms is quite big. More precisely, it is at least tower of exponentials whose height depends on the size of the query. Already for the model checking problem, it is proved in [34] an elementary constant factor is not reachable for first-order queries if the class of structures contains all trees, unless $\mathrm{FPT} = \mathrm{AW}[*]$. What we will do is find in which steps of the algorithm the constant factor blows up.

The base case of our induction (i.e. for queries of arity 0 or 1) already contains a non elementary constant factor, due to the result mentioned above. The second occasion where the constant factor blows up is our second inductive parameter: the number of rounds Splitter needs to win the game. This number ($\lambda(r)$) can be non elementary in r (and r is exponential in $|\varphi|$). An example of classes of graphs where this happens is provided in Claim 4.2.18. However, this is not the case for every nowhere dense classes. For instance, databases with bounded degree allows algorithms with only three nested exponentials in the constant factor. For those classes, Splitter wins with only d^r rounds where d is the degree of the structure.

In contrast, some parts of our algorithms that seems to create such blowup behave

nicely. For instance when doing a disjunction over every distance type $\tau \in \mathcal{T}_k$, we only create 2^{k^2} different cases.

Finally, the impact of some steps remains unclear. For example, the size of m_τ from the Rank Preserving Normal Form Theorem 4.4.5 is not clearly mentioned in [45]. This might result in a disjunction of size non elementary in the size of the query $|\varphi|$. Although we do not think this is the case.

Pseudo-linear time. In our algorithms, the running time of the preprocessing is pseudo linear is the number of vertices. Since the number of edges might also be of that size we can not expect to do better. The algorithm need to read the entire output! But nothing forbids the preprocessing to be linear in the size of the structure i.e. $O(\|G\|)$. However, this is a substantial task as the complexity of most theorems used in our algorithms are also pseudo-linear in the number of vertices without properly addressing the complexity in the global size of the structure.

We believe one of the biggest challenges to obtain a linear algorithm lies in the use of the neighborhood cover. Simply saying that the degree of the cover is bounded by $|G|^\epsilon$ is not enough, as this factor will appear somewhere in the overall complexity. A first step toward a linear algorithm could be to find whether there is a computable neighborhood cover with the following extra property:

$$\sum_{a \in V} |\{X \in \mathcal{X} \mid a \in X\}| = O(\|G\|).$$

Another important challenge is to improve our Storing Theorem 4.1.1. Here, again, we see a factor of the form $|G|^\epsilon$. We would need a strategy that can store a function f, using only a number of registers linear in the size of the domain of f. Furthermore, given a tuple $\overline{a}$, we want to be able to compute $f(\overline{a})$ or determine that $\overline{a}$ is not in the domain of f in constant time. This seems hard to achieve.

Beyond the dichotomy. Our dichotomy theorem only works for classes of graphs that are closed under subgraphs. The next question is then: "Are there dense classes of graphs that admit efficient algorithms for problems related to the query evaluation problem?" Some of those problems have been tackled. We have results for classes of graphs that are FO interpretable into classes of graphs with bounded degree [36] and bounded expansion [37]. However, we still do not know what are the classes that are interpretable into nowhere dense classes of graphs.

Answering queries with updates. Recently, some attention has been given to algorithms that answer the various problems around query evaluation with the extra property that, when some small change occurs in the database, the answer or index can be recomputed efficiently (not from scratch). Section 3.2 of the State of the art is devoted to such results. Even though a lot of work has been produced for acyclic conjunctive queries and tree-like structures, the case of sparse structures and first-order queries still contains many unanswered questions.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on trees under relabelings. In *Intl. Conf. on Database Theory (ICDT)*, 2018.

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

[5] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65(1-3):21–46, 1996.

[6] Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *Conf. on Computer Science Logic (CSL)*, 2006.

[7] Guillaume Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. (Algorithms and complexity of enumeration problems for the evaluation of logical queries)*. PhD thesis, University of Caen Normandy, France, 2009.

[8] 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)*, 2010.

[9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Conf. on Computer Science Logic (CSL)*, 2007.

[10] Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *ITA*, 42(1):147–164, 2008.

[11] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[12] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Symp. on Principles of Database Systems (PODS)*, 2017.

[13] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO + MOD queries under updates on bounded degree databases. In *Intl. Conf. on Database Theory (ICDT)*, 2017.

[14] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. In *Intl. Conf. on Database Theory (ICDT)*, 2018.

[15] Anne Berry, Jean Paul Bordat, and Olivier Cogis. Generating all the minimal separators of a graph. *Int. J. Found. Comput. Sci.*, 11(3):397–403, 2000.

[16] Johann Brault-Baron. *De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre. (The relevance of the list: propositional logic and complexity of the first order).* PhD thesis, University of Caen Normandy, France, 2013.

[17] Florent Capelli and Yann Strozecki. On the complexity of enumeration. *CoRR*, abs/1703.01928, 2017.

[18] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[19] 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. Elsevier, 1990.

[20] Bruno Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.

[21] Nadia Creignou and Jean-Jacques Hébrard. On generating all solutions of generalized satisfiability problems. *ITA*, 31(6):499–511, 1997.

[22] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in dynfo. In *Intl. Coll. on Automata, Languages and Programming (ICALP)*, 2015.

[23] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 2007.

[24] Arnaud Durand and Stefan Mengel. The complexity of weighted counting for acyclic conjunctive queries. *J. Comput. Syst. Sci.*, 80(1):277–296, 2014.

[25] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating answers to first-order queries over databases of low degree. In *Symp. on Principles of Database Systems (PODS)*, 2014.

[26] Arnaud Durand and Yann Strozecki. Enumeration complexity of logical query problems with second-order variables. In *Conf. on Computer Science Logic (CSL)*, 2011.

[27] Zdenek Dvorak, Daniel Král, and Robin Thomas. Testing first-order properties for subclasses of sparse graphs. *J. ACM*, 60(5):36, 2013.

[28] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12:1–12:51, 2015.

[29] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. Constant delay algorithms for regular document spanners. In *Symp. on Principles of Database Systems (PODS)*, 2018.

[30] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer, 2006.

[31] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. Joining extractions of regular expressions. In *Symp. on Principles of Database Systems (PODS)*, 2018.

[32] Markus Frick. Generalized model-checking over locally tree-decomposable classes. *Theory Comput. Syst.*, 37(1):157–191, 2004.

[33] Markus Frick and Martin Grohe. Deciding first-order properties of locally tree-decomposable structures. *J. ACM*, 48(6):1184–1206, 2001.

[34] 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.

[35] Haim Gaifman. On local and non-local properties. In *Proceedings of the Herbrand Symposium*. 1982.

[36] Jakub Gajarský, Petr Hlinený, Jan Obdrzálek, Daniel Lokshtanov, and M. S. Ramanujan. A new perspective on FO model checking of dense graph classes. In *Symp. on Logic in Computer Science (LICS)*, 2016.

[37] Jakub Gajarský, Stephan Kreutzer, Jaroslav Nešetřil, Patrice Ossona de Mendez, Michał Pilipczuk, Sebastian Siebertz, and Szymon Toruńczyk. First-order interpretations of bounded expansion classes. In *Intl. Coll. on Automata, Languages and Programming (ICALP)*, 2018.

[38] Francois Le Gall. Powers of tensors and fast matrix multiplication. In *Intl. Symp. on Symbolic and Algebraic Computation (ISSAC)*, 2014.

[39] Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. The dynamic complexity of formal languages. *ACM Trans. Comput. Log.*, 13(3):19:1–19:36, 2012.

[40] Etienne Grandjean. Sorting, linear time and the satisfiability problem. *Ann. Math. Artif. Intell.*, 16:183–236, 1996.

[41] Martin Grohe. Generalized model-checking problems for first-order logic. In *Symp. on Theoretical Aspects in Computer Science (STACS)*, 2001.

[42] Martin Grohe. Logic, graphs, and algorithms. In *Logic and Automata*, 2008.

[43] Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. Deciding first-order properties of nowhere dense graphs. In *Symp. on Theory of Computing (STOC)*, 2014.

[44] Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. Deciding first-order properties of nowhere dense graphs. *J. ACM*, 64(3):17:1–17:32, 2017.

[45] Martin Grohe and Nicole Schweikardt. First-order query evaluation with cardinality conditions. In *Symp. on Principles of Database Systems (PODS)*, 2018.

[46] Lucas Heimberg, Dietrich Kuske, and Nicole Schweikardt. Hanf normal form for first-order logic with unary counting quantifiers. In *Symp. on Logic in Computer Science (LICS)*, 2016.

[47] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[48] Wojciech Kazana. *Query evaluation with constant delay. (L'évaluation de requêtes avec un délai constant)*. PhD thesis, École normale supérieure de Cachan, Paris, France, 2013.

[49] Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science*, 7(2), 2011.

[50] Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In *Symp. on Principles of Database Systems (PODS)*, 2013.

[51] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.*, 14(4), 2013.

[52] 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.

[53] Hal A. Kierstead and Daqing Yang. Orderings on graphs and game coloring number. *Order*, 20(3):255–264, 2003.

[54] Stephan Kreutzer. On the parameterised intractability of monadic second-order logic. In *Conf. on Computer Science Logic (CSL)*, 2009.

[55] Stephan Kreutzer and Anuj Dawar. Parameterized complexity of first-order logic. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:131, 2009.

[56] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[57] Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In *Conf. on Computer Science Logic & Symp. on Logic in Computer Science (CSL-LICS)*, 2014.

[58] Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion I. decompositions. *Eur. J. Comb.*, 29(3):760–776, 2008.

[59] Jaroslav Nešetřil and Patrice Ossona de Mendez. Grad and classes with bounded expansion II. algorithmic aspects. *Eur. J. Comb.*, 29(3):777–791, 2008.

[60] Jaroslav Nešetřil and Patrice Ossona de Mendez. First order properties on nowhere dense structures. *J. Symb. Log.*, 75(3):868–887, 2010.

[61] Jaroslav Nešetřil and Patrice Ossona de Mendez. On nowhere dense graphs. *Eur. J. Comb.*, 32(4):600–617, 2011.

[62] Jaroslav Nešetřil and Patrice Ossona de Mendez. *Sparsity*. Springer-Verlag, 2012.

[63] Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *Symp. on Principles of Database Systems (PODS)*, 2018.

[64] Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *J. Comput. Syst. Sci.*, 58(3):407–427, 1999.

[65] Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.*, 79(6):984–1001, 2013.

[66] Detlef Seese. Linear time computable problems and first-order descriptions. *Mathematical Structures in Computer Science*, 6(6):505–526, 1996.

[67] Luc Segoufin. Enumerating with constant delay the answers to a query. In *Intl. Conf. on Database Theory (ICDT)*, 2013.

[68] Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In *Intl. Conf. on Database Theory (ICDT)*, 2017.

[69] Hong Shen and Weifa Liang. Efficient enumeration of all minimal separators in a graph. *Theor. Comput. Sci.*, 180(1-2):169–180, 1997.

[70] Yann Strozecki. *Enumeration complexity and matroid decomposition.* PhD thesis, Université Paris Diderot, Paris, France, 2010.

[71] Ken Takata. Space-optimal, backtracking algorithms to list the minimal vertex separators of a graph. *Discrete Applied Mathematics*, 158(15):1660–1667, 2010.

[72] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.

[73] Takeaki Uno. Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *Intl. Symp. Algorithms and Computation (ISAAC)*, 1997.

[74] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases (VLDB)*, 1981.

[75] Thomas Zeume. *Small dynamic complexity classes*. PhD thesis, Technical University Dortmund, Germany, 2015.