View-based query determinacy and rewritings over graph databases

Nadime Francis

NNT : 2015SACLN015

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PARIS-SACLAY,
préparée à l'École Normale Supérieure de Cachan

ÉCOLE DOCTORALE N°580
Sciences et Technologies de l'Information et de la Communication

Spécialité de doctorat : Informatique

Par

Monsieur Nadime Francis

Vues et requêtes sur les graphes de données : déterminabilité et réécritures

Thèse présentée et soutenue à Cachan, le 8 décembre 2015 :

Composition du Jury :

M. Maurizio Lenzerini	Professeur, Sapienza Università di Roma	Rapporteur
M. Luc Segoufin	Directeur de Recherche, Inria	Directeur de thèse
Mme Cristina Sirangelo	Professeur, Université Paris 7	Co-directrice de thèse
Mme Sophie Tison	Professeur, Université Lille 1	Examinatrice
M. Jan Van den Bussche	Professeur, Universiteit Hasselt	Rapporteur
M. Victor Vianu	Professeur, UC San Diego	Président

Laboratoire Spécification et Vérification
École Normale Supérieure de Cachan, UMR 8643 du CNRS
61 avenue du Président Wilson,
94235 Cachan Cedex, France

Résumé

Les graphes de données sont naturellement utilisés dans de nombreux contextes incluant par exemple les réseaux sociaux ou le Web sémantique. Dans ce modèle, les données sont structurées et représentées comme des graphes, dont les noeuds contiennent les données individuelles et où les arêtes du graphe représentent les liens entre ces données. Ainsi, dans un réseau social, chaque noeud du graphe représente un individu, et contient ses données personnelles (nom, prénom, identifiant, adresse, etc.), tandis que les arêtes du graphe spécifient la manière dont il est relié aux autres utilisateurs : ami, collègue, parent, etc. L'information contenue dans la base de données se trouve alors aussi bien dans les données mêmes que dans la topologie du graphe, c'est-à-dire dans la manière dont les données sont connectées. Cela implique donc de considérer les questions traditionnelles en théorie des bases de données pour des langages de requêtes capables de parler des chemins connectant les noeuds du graphe.

Nous nous intéressons en particulier aux problèmes de la déterminabilité et de la réécriture d'une requête à l'aide de vues. On suppose alors qu'au lieu de pouvoir accéder directement à la base de données, seules les réponses à une série de requêtes initiales (appelées *vue* de la base de données) sont connues. Il s'agit alors de décider si cette vue contient suffisamment d'information pour répondre entièrement à une requête sans consulter la base de données directement, et dans ce cas, de produire une réécriture, c'est-à-dire un moyen d'exprimer explicitement la réponse à la requête à partir de la vue. Ce cadre rencontre de nombreuses applications, notamment pour l'intégration de données, l'optimisation de requêtes ou encore en sécurité.

Nous obtenons nos résultats préliminaires en comparant ces deux questions aux autres problèmes de décision classiques dans ce contexte : calcul des réponses certaines, test de cohérence et mise à jour d'une instance de vue. Ces premiers résultats mettent en évidence les liens entre toutes ces questions et servent de base pour traiter nos deux questions initiales. Nos deux contributiosn principales améliorent ces résultats dans deux cas spécifiques.

Tout d'abord, nous considérons le cas de vues et requêtes définies par des requêtes régulières de chemin. Ces requêtes sont définies à l'aide d'une expression régulière et renvoient les paires de noeuds de la base de données connectés par un chemin dont l'étiquette satisfait l'expression régulière. Nous montrons que dans ce cas, l'existence d'une réécriture monotone coïncide avec l'existence d'une réécriture exprimable dans Datalog, et que cette dernière peut effectivement être calculée. Cela implique en particulier que la réponse à la requête peut alors être calculée en temps polynomial dans la taille de l'instance de

vue donnée.

Ensuite, nous revenons au problème de la déterminabilité dans le cadre plus simple de requêtes s'intéressant uniquement aux longueurs des chemins du graphe. Ces requêtes sont définies par un ensemble d'entiers et renvoient les paires de noeuds du graphe qui sont connectés par un chemin dont la longueur est égale à l'un de ces entiers. La question alors est de savoir, étant donnée une vue définie par de telles requêtes, quelles sont les longueurs qui peuvent être déterminées. Nous montrons alors que ce problème est décidable pourvu que les longueurs demandées soient suffisamment grandes (en fonction de la vue), ce qui conduit à définir la notion plus faible de déterminabilité asymptotique. Nous montrons également que les requêtes asymptotiquement déterminées admettent alors des réécritures du premier ordre, ce qui permet encore une fois une évaluation en temps polynomial de la requête donnée à partir de la vue.

Acknowledgements

First and foremost, I would like to thank my advisors Luc Segoufin and Cristina Sirangelo, who accepted to embark on this journey with me. I have learned much from you during this PhD, and I still feel like I would learn twice as much if I were to start again from the beginning. You have never ceased to amaze me, with your knowledge, intelligence and dedication. I am particularly thankful for the time you managed to find for me, especially when you had your own duties to attend to. Thank you.

I am very grateful to Maurizio Lenzerini and Jan Van den Bussche who kindly accepted to review this manuscript, and managed to send their reports on time despite the short notice. I also thank Sophie Tison and Victor Vianu who accepted to be part of the jury.

I would like to thank Claire David and Filip Murlak with whom I coauthored a paper during this PhD, even though it is not part of this manuscript. I also thank Leonid Libkin who has given me many useful advice and taught me how to respond to a reviewer.

I also thank all the members of LSV for making my time there as enjoyable and stimulating as possible. There are too many of you to name, but be assured that each of you is amazing in his or her own way. Thank you for everything. In particular, I want to thank the people with whom I have shared an office on the 4th floor: Wojciech Kazana, Emilien Antoine and Marie Van den Bogaard. Thank you for many useful and enjoyable, if not always professional, discussions. Thank you for making the 4th floor office more than a work place.

I would like to thank my best friend, Victor Marsault, who has worked on his own PhD at the same time as me. I thank you for listening to my struggles and telling me about yours, for commiserating with me on our failures and celebrating on our successes.

I want to give special thanks to all my family, who has always supported me and believed in me. Special thanks go to my parents, who have raised me to be who I am today. You have taught me to ask *why* and *how*, almost to a fault. I also thank my brother, Bachir. I could not have hoped for a better person to grow up with. You have always pushed me to give my best and you have always been proud of my results. I want to thank you for supporting me during this PhD, and somehow finding the mental strength to listen to my explanations even when they did not make sense. I want to thank you for all our conversations until the first hours in the morning, be they about life, philosophy or mathematics.

Finally, my biggest thanks go to my wife Emilie. You have learned to know me, accept me and love me as I am. You have supported me, even during your own PhD. I know I will always have your trust and love, and for that I have no words. Thank you.

Contents

Résumé **3**

Acknowledgements **5**

1 Introduction **9**
1.1 Databases and graphs 10
1.2 View-based query processing 10
1.3 Contribution 12
1.4 Organization 14

2 State of the art **15**
2.1 Answering queries using views 15
2.2 The many names of determinacy 16
2.3 Determinacy and rewritings 19
2.4 Conclusions 23

3 Databases and queries **25**
3.1 Databases 25
3.2 Queries 28
3.3 Algorithms 32
3.3.1 Query evaluation 32
3.3.2 Query containment 35

4 Views and operations on views **37**
4.1 Views 37
4.2 Materialized view problems 39
4.2.1 Certain answers 39
4.2.2 Inverting view images 42
4.2.3 Checking view images 45
4.2.4 View update 48
4.3 View-based query determinacy 51
4.3.1 Definition 51
4.3.2 Determinacy problem 53
4.3.3 Rewriting problem 56

5 Monotone rewritings of regular path queries **59**
5.1 Monotone determinacy . . . 59
5.2 Constraint satisfaction and certain answers . . . 64
5.2.1 Constraint satisfaction problems . . . 65
5.2.2 From certain answers to CSP . . . 65
5.3 Computing the rewriting . . . 67
5.3.1 Datalog and the local consistency game . . . 67
5.3.2 The case of simple paths . . . 69
5.3.3 From simple paths to arbitrary graph databases . . . 72
5.4 Extensions . . . 74
5.4.1 Two-way regular path queries . . . 74
5.4.2 On rewriting languages . . . 75

6 Asymptotic determinacy of single path queries **77**
6.1 Preliminaries . . . 78
6.1.1 Key properties . . . 78
6.1.2 Asymptotic determinacy . . . 82
6.2 Behavior graphs . . . 83
6.2.1 Intuitions . . . 83
6.2.2 Definitions . . . 86
6.3 Deciding asymptotic determinacy . . . 90
6.3.1 Negative direction: building counter-examples . . . 90
6.3.2 Positive direction: building a rewriting . . . 97
6.4 Extensions . . . 101
6.4.1 The case of small queries . . . 101
6.4.2 Infinite unions . . . 103
6.4.3 Multiple labels . . . 106

7 Discussions **109**
7.1 Determinacy and view-based query processing . . . 109
7.2 Datalog and the bounded width hierarchy . . . 110
7.3 Single path queries and first-order rewritings . . . 111

Bibliography **113**

Chapter 1

Introduction

Nowadays, information is everywhere. Each of us creates, shares, uses and receives huge amounts of information everyday. Each of us is exposed to information processing systems several times a day, and in many forms. This ranges from the traditional desktop or laptop computers we may use daily, at home or at work, to the cell phones we carry in our pockets and that grow smarter month after month, going through more innocuous-looking systems like GPS or the machines in which we validate our public transport cards. Each of these systems stores, transmits and queries data. The amount of data that is accessed by the day is enough to make one's head spin. A popular micro-blogging service like Twitter conveys on average 6000 messages a second, which sum up to 500 million tweets a day. A more permanent social network like Facebook registers on average 55 million status updates each day. Google, as a search engine, processes 40 thousand queries a second, summing up to 3.5 billion queries a day. This makes a traditional data storing entity like the Bibliothèque François Mitterrand look frail in comparison, with its measly 2000 new documents a day.

Of course, all this data is not only transmitted, read, and then forgotten. Every single bit of data is stored, indexed, and ready to be processed and queried again. Thus, not only are we daily exposed to information systems, but we are also daily leaving tracks in a vast number of databases. This raises an obvious question: how do we process such a huge amount of data in an efficient way? We mentioned libraries as a more traditional form of databases. Let us extend the comparison for a little while. Designing a library is not only a matter of stockpiling huge amounts of books in a huge amount of crates, so as to protect them from the passing of time. A library should also provide several services that extend its usefulness. Perhaps the books are stored by alphabetical order, or sorted by genre. Perhaps the librarian knows the whole content of the library by heart and can tell whether a book is available instantly. Perhaps the librarian runs very fast and can fetch you the book you are looking for in a matter of seconds. Perhaps the librarian even knows the topic of your research and can give you a book which contains the answers you are looking for without you even asking or knowing such a book existed. While this might look silly in a physical world, it is exactly what we are asking of our digital databases.

1.1 Databases and graphs

In this work, we will consider specifically graph databases: databases that are structured and represented as graphs. In this model, individual data is stored as nodes in the graph while the links between these data points are represented as edges of the graph. Social networks are a typical example of graph databases: each node of the network corresponds to a person, and contains individual information, such as name, phone number, date of birth or professional activity. On the other hand, relationships between members of the network will be materialized as edges linking their corresponding nodes. For instance, a node attributed to John will be linked by a *parent* edge to Jack to indicate that John is the father of Jack, while Jack could be linked to Lise by a two-way *sibling* edge, stating that Jack and Lise are brother and sister. The Web offers another immediate example: web pages are nodes of a graph whose edges are the hyperlinks relating pages to one another. The very image of this network as a web or net comes from its mental representation as a graph. Other scenarios where these graph databases naturally occur include crime detection, the representation of biological data, or the Semantic Web through the RDF format.

We are not however interested in graph databases simply because some data is naturally presented to us in the form of a graph. Indeed, any conventional database could be represented as a graph, and any graph database could be turned into any of the usual representations for databases. The main reason for using graph databases is that, in these typical scenarios where graph databases naturally occur, the information that we want to extract lies as much in the content of the graph as in its shape. Indeed, it is much more practical to look at a Paris metro map to determine how to go from Châtelet to Nation rather than going through tables of each metro line. It is in the same sense that query languages designed for conventional databases are not adapted to express the kind of questions that we want to ask on data that is naturally presented as a graph. On the contrary, graph databases provide the framework for queries to easily talk about links, paths, loops and so on, in a much more efficient way than their more conventional counterparts. Thus, the choice of a model for a database is not only a matter of representing data, but more importantly it is a matter of how we intend to use it, which translates in the kind of queries that we want to issue to it.

1.2 View-based query processing

The most natural way to use a database is to query it, that is to ask a question to the database, and receive answers from it. Here, however, we will assume that the database is not available to process our query. Instead, the only thing that we are allowed to "see" of the database is the set of answers to a set of queries, called a view, that were previously provided by the database. We are then wondering whether this view of the database contains enough information to process our original query. This is the goal of view-based query processing, which will be the main focus of our work.

At its core, the question asked here can be restated in a very natural way: can we

decide if some given knowledge implicitly contains more information, and if so, how can we make it explicit? Thus, it is perhaps not surprising that the setting considered here encompasses a large number of applications, especially in the database context. Let us discuss some of them.

Query optimization and caching. Caching is perhaps the most intuitive application that comes to mind. Assume that the database already provided the answers to a set of initial queries, and that these answers was kept in cache on our local machine. This constitutes our view of the database. Imagine now that the database is not readily available to answer a new query, for instance because our connection to the network is slow, costly or unstable. Then we would like to compute the answers to our new query without having to access the database.

Query optimization follows a similar setting. Assume again that we are keeping the answers to a set of initial queries. When a new query arises, it might be more efficient to compute it from already known answers rather than recompute it from scratch, especially if some parts of the computational work of the query were already processed by the view. In this setting, it is quite common to also include the whole database in the view, thus we already know that the information we are looking for is present in the view, and we can focus on *how* to express it. Among all the possible ways of evaluating the query, that might or might not make use of the stored answered, which one is the most efficient?

Data integration. Data integration is the problem of unifying data coming from multiple heterogeneous source databases as a single global database. The ideal result here is to allow the user to query the information available in these databases seamlessly, as if it came from a single database, and thus without having to take into consideration each source separately. A way to restate this problem as a view-based query processing task is known as the Local-as-View setting. We consider the sources as a view of the virtual global database. Then, when the user wants to query this global database, we decide whether the information she seeks is implicitly contained in the view, and if so we recompute it as an explicit query over the view. The answers to this rewritten query over the data sources thus correspond to the answers to the original query over the virtual database.

Data independence. In the data independence setting, we want to provide an interface to the user that allows her to query the database without this interface necessarily matching the physical implementation of the database. Similarly to the data integration setting, we consider the physical implementation of the database as a view of the virtual database provided to the user. When the user then queries the virtual database, we rewrite the queries over the physical implementation, and answer them there. This allows us in particular to modify the way data is stored physically, say for more efficient computing or to accomodate to new hardware or data, without impacting the virtual database, and thus the way the user interacts with the database.

Privacy. The last application we discuss here provides a nice example of a case where we want our question answered by the negative. In the privacy setting, we assume that we are managing a database that contains private, and thus sensitive, information. We allow users to query the database for public information. However, we want to make sure that by doing so, we are not disclosing more information than intended. In other words, we need to ensure that the users cannot use the information given to them to deduce private data. Consider the queries on public data as a view of the whole database, then we have to make sure that this view does not implicitly contain any private data. Conversely, if we were to attack such a system, finding an explicit way to deduce this data from the view would reveal exactly the series of queries that need to be issued in order to breach the security of the system.

We have seen through these applications that we are in particular interested in two tasks: *determinacy*, which consists in deciding whether some information is implicitly present in the view, and *rewriting*, which consists in finding a way to make it explicit. More precisely, given a view and a query, the goal of determinacy is to decide whether the information contained in the view can always be used to fully answer the query, and this regardless of the specific database that provided the view. When this is true, we are then interested in finding a rewriting: a way of actually computing the answers to the original query by only looking at the content of the view.

These two tasks are challenging on multiple levels. Both have been studied extensively over many years, but many questions still remain open. Regarding the determinacy problem, the ideal would of course be to design an algorithm that, given the definition of the view and the query, checks whether the view determines the query. However, this has only been achieved in very specific cases. For many fundamental query and view languages, it is still unknown whether such an algorithm exists.

On the topic of rewriting, the question has several different aspects. On the simplest level, it only asks to provide an explicit way, an algorithm, for computing the answer to the query from the view. However, it is often better to be able to express this algorithm as a new query, in some query language, that can then be answered over the view. While it would be ideal to express this rewriting in the same language as the query and the view, it has been shown in many cases that the rewriting actually needs *more* expressive power. In many of those cases, it is still unknown how much more expressive power is needed. This also raises the question of finding rewritings that have as low expressive power as possible in order to achieve good computational properties, but even deciding whether these rewritings exist is already a challenging task.

1.3 Contribution

The work we present in this document revolves mostly around the determinacy and rewriting problems. Our goal is to develop a better understanding of when a view determines a query, and the relationship this has with the expressive power needed to define rewritings. We would like to highlight our two main contributions.

Monotone rewritings of regular path queries. For this contribution, we have looked at regular path queries. These queries are defined by a regular expression, and select pairs of nodes in the database that are linked by a path whose sequence of labels conforms to the regular expression. We have considered the setting where a view defined by regular path queries determines a regular path query in a monotone way, meaning that we assume the existence of a monotone rewriting of the query using the view. This setting is known to be decidable from [12]. We have shown that, in this case, we can compute an explicit rewriting of the query using the view as a Datalog program. This is formalized as the following theorem:

Theorem 5.23. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query such that $\boldsymbol{V}$ determines Q in a monotone way. Then there exists a Datalog rewriting of Q using $\boldsymbol{V}$.*

This theorem has several important implications. First, it proves that the existence of a Datalog rewriting of the query using the view coincides with the existence of a monotone rewriting. Thus, the decision procedure given in [12] translates into an algorithm for deciding the existence of a Datalog rewriting of the query using the view. Second, the theorem proves the existence of a rewriting that enjoys polynomial time data complexity, whereas the previously best known bound was $\text{NP} \cap \text{coNP}$. Finally, since the rewriting can effectively be computed, it also provides an algorithm for answering the query using the view with polynomial time data complexity.

This result was published in [24], with a long version to appear at the Logical Methods in Computer Science journal.

Asymptotic determinacy. For our second contribution, we have looked at the determinacy problem for queries that select pairs of nodes linked by a path whose length falls in a given set. We have defined a weaker version of the determinacy problem that only asks to decide determinacy for queries that look for a path that is "long enough" when compared to the paths selected by the view, in a precise sense. We call it the α-asymptotic determinacy problem, where α is a function that provides the explicit definition of "long enough". We have proved that, in this case, there exists an implicit α that makes the problem decidable, and for which all determined queries can be rewritten as first-order formulas over the view.

Theorem 6.1. *There is an explicit and computable function α for which the α-asymptotic determinacy problem is decidable for single path queries and unions of single path views. Moreover, when the view determines the query, the decision procedure effectively computes a first-order rewriting of the query using the view.*

This result is a significant step towards solving the determinacy problem in this setting, a question that remains open for now. It also has two important corollaries. First, it proves that almost all single path queries that are determined by unions of single path views can be rewritten as first-order queries, meaning that first-order is almost complete for rewritings, as defined in [3]. Second, it shows that almost all single path queries that are determined can be answered with polynomial time data complexity using the view, whereas the previously best known bound was once again $\text{NP} \cap \text{coNP}$.

This result was published in [23], with a long version to appear in a special issue of the Theory of Computer Systems journal.

1.4 Organization

This document follows the outline below:

In **Chapter 2: State of the art**, we briefly summarize the relevant results that were used as a starting point for our work. We give an overview of the various settings in which the determinacy problem has been considered, and the many names under which it has been studied. We discuss the known results, and the main questions that are left open. Note that this chapter is intended for the specialist, and thus assumes some familiarity with database theory. This chapter works independently of the rest the document, and can safely be skipped.

In **Chapter 3: Databases and queries**, we set the main definitions and notations for the rest of our work. We formally define databases and queries, as well as the query languages that we will consider throughout our work. This chapter also contains lots of examples and should be a good beginning for most readers.

In **Chapter 4: Views and operations on views**, we formally define views, as well as the most common computational tasks related to view-based query processing. This chapter most importantly defines the determinacy and rewriting problems that will occupy most of our attention. It also contains several side results and observations that will serve as preliminaries for Chapter 5 and Chapter 6.

In **Chapter 5: Monotone rewritings of regular path queries**, we consider the case where a regular path view determines a regular path query in a monotone way. We define the notion of monotone determinacy and work on building the proof of Theorem 5.23 mentioned previously. This will take us on a journey through Constraint Satisfaction Problems and Local Consistency Games.

In **Chapter 6: Asymptotic determinacy of single path queries**, we define the asymptotic determinacy problem. We make several key observations that help draw an intuitive picture of what asymptotic determinacy really means. We explain how asymptotic determinacy relates to general determinacy, and how it translates to necessary conditions for the general setting. We finally move on to proving Theorem 6.1, and discuss several possible extensions.

Finally, we conclude this work in **Chapter 7: Discussions**, where we sum up the results that have been achieved in the previous chapters, and mention several possible continuations and perspectives.

Chapter 2

State of the art

In this state of the art chapter, we give a brief overview of the different techniques and results that are relevant to our work. This is by no means intended to be an exhaustive description of the field. For more references, we direct the reader to the survey [30], that also discusses algorithmic matters and practical implementations. We also mention the survey [29] by the same author, which focuses on the theoretical aspects of query determinacy.

This chapter is intended for the specialist who wants to see how our work fits in the existing picture. Hence, we assume that the reader is familiar with database theory, and we will not spend much space on the most common definitions. We direct the interested reader to [2] for an excellent reference on the general background.

The reader who is either new to the field or takes this document as a stand-alone can safely skip this chapter. The important notions discussed here will be defined formally later and every external result will be cited again when we use it.

In Section 2.1, we discuss the problem of answering a query from a view instance, which has strong ties with the rewriting problem that occupies a significant part of our work. In Section 2.2, we give an overview of the several notions that are very close (and sometimes equivalent) to what we call determinacy in this work. Finally, in Section 2.3, we describe the various settings in which the determinacy and rewriting problems have been considered and solved, and those where the question is still open.

2.1 Answering queries using views

In this section, we assume that we are given a query Q and a view $\mathbf{V}$ that is defined by a set of queries. We are also given a view instance E which we assume to be the result of computing the queries in $\mathbf{V}$ over some database D that we do not have access to. The question here is: what is the best estimation that we can have of Q(D), the answers to our query on the original database, by looking only at the information available in E?

Of course, the complexity of solving such a task will depend heavily on the query languages that are used to define both Q and the queries in $\mathbf{V}$. A less intuitive parameter is the assumption on *how* E was computed from D. In [10], the view may be considered

exact, which means that E is exactly the result of computing $\mathbf{V}$ on D, that is, $E = \mathbf{V}(D)$. Otherwise, the view is only *sound*, meaning that E contains answers to $\mathbf{V}$ on D, but not necessarily all of them, that is $E \subseteq \mathbf{V}(D)$. In [1], the same possibilities are considered, and they are respectively called the *closed world assumption*, where E contains precisely all the answers to $\mathbf{V}$ on D, and the *open world assumption*, where we only know that E is contained in $\mathbf{V}(D)$. Note that [10] also considers the case where $\mathbf{V}$ is *arbitrary*, that is, some queries in $\mathbf{V}$ might be sound and other exact, but we wont make any use of this case in our work.

In [10], the authors also consider two other assumptions, which they call *closed domain assumption* and *open domain assumption*. In the closed domain assumption, it is additionally assumed that each element of D appears in some tuple of E, whereas the open domain assumption makes no such claim. In the literature, it is generally assumed that E does not have to contain all elements of D, and therefore the open domain assumption is the norm. To avoid confusion with the open and closed world assumption, we will refrain from using these names. We always assume that E does not have the closed domain restriction. In the table below, we explicitly say that $dom(E) = dom(D)$ when it is assumed that this is the case.

The approach in both [10] and [1] relates the problem of answering a queries using views to techniques usually used for querying incomplete databases, as observed in [9]. [10, 1] define the best possible estimation of $Q(D)$ as the *certain answers* to Q based on E: the intersection of all $Q(D')$, for all D' that are consistent with E.[1] Here, D' is consistent with E if D' is a candidate databasse on which $\mathbf{V}$ can produce E: that is, $\mathbf{V}(D') = E$ under the exact view assumption, and $\mathbf{V}(D') \supseteq E$ under the sound view assumption.

The authors of [1] find the data complexity of computing certain answers under both assumptions for a variety of query and view languages, including conjunctive queries with or without inequalities, positive queries, first-order queries and Datalog. We have reported the results that are relevant to our work in Figure 2.1 and Figure 2.2. In [10], the problem is considered for regular path queries and regular path views, under various assumptions and for data complexity, expression complexity and combined complexity. The results can be seen on Figure 2.3.

As we will see in Section 2.2 and throughout all this work, the problem of answering queries using views is very closely related to the determinacy and rewriting problems.

2.2 The many names of determinacy

Most of this work is dedicated to view-based query determinacy and rewriting. Given a view $\mathbf{V}$ and a query Q, we want to know if, for all databases D, the information extracted by the view, $\mathbf{V}(D)$, is enough to completely determine the answers to the query, $Q(D)$. However, if we take some time to think about determinacy outside of the database context, we can restate it as a very natural and fundamental question: can we decide if some given knowledge implicitly contains more information, and if so, how can we make it explicit?

[1] In the incomplete database setting, certain answers are usually defined as the intersection of all $Q(D')$ for all databases that are a completion of D, the original incomplete database given as input.

view \ query	CQ	Datalog	FO
CQ	PTIME	PTIME	Undecidable
Datalog	coNP	Undecidable	Undecidable
FO	Undecidable	Undecidable	Undecidable

Figure 2.1: Data complexity of computing certain answers for various query and view languages under the sound view assumption, taken from [1].

view \ query	CQ	Datalog	FO
CQ	coNP	coNP	Undecidable
Datalog	Undecidable	Undecidable	Undecidable
FO	Undecidable	Undecidable	Undecidable

Figure 2.2: Data complexity of computing certain answers for various query and view languages under the exact view assumption, taken from [1].

dom(D) = dom(E)?	Assumption on views	Complexity		
		Data	Expression	Combined
Yes	Sound	coNP	coNP	coNP
	Exact	coNP	coNP	coNP
No	Sound	coNP	PSPACE	PSPACE
	Exact	coNP	PSPACE	PSPACE

Figure 2.3: Complexity of computing certain answers for regular path queries and views under various assumptions, taken from [10].

Thus, it is understandable that this question has been considered in many different lights, even in the database context. In our work, we adopt the information theoretic perspective of [41, 36, 37]. We say that a view $\mathbf{V}$ determines a query Q if, for all databases D and D′, $\mathbf{V}(\mathrm{D}) = \mathbf{V}(\mathrm{D}')$ implies that $\mathrm{Q}(\mathrm{D}) = \mathrm{Q}(\mathrm{D}')$. Then, a rewriting R of Q using $\mathbf{V}$ is a query that satisfies that for every database D, $\mathrm{Q}(\mathrm{D}) = \mathrm{R}(\mathbf{V}(\mathrm{D}))$. [37] also introdues the notion of *completeness* of a query language for view-to-query rewritings, stating that a language L is complete for a query language L_Q and a view language L_V if, whenever a query in L_Q is determined by a view in L_V, then there always exists a rewriting that is expressible in L. In this section, we give an overview of some of the various related notions that have been considered so far.

On the semantic level, the first formalization of determinacy can be traced back to Tarski. In [43], he introduces *implicit definability*, which defines when a theory implicitly defines a relation in terms of another set of relational symbols. He further introduces the notion of *completeness for definitions*, stating that a logical system L is complete for definition if, whenever a theory implicitly defines a relation Q in terms of a set of relations $\mathbf{V}$, then Q can be explicitly expressed as a formula of L using only the symbols in $\mathbf{V}$. As we can see, this is extremely close to saying that there exists a rewriting of a query Q using the view $\mathbf{V}$ in the language L. Thus, completeness for definitions is extremely similar to completeness of a query language for view-to-query rewritings. The main difference between the logical setting of Tarski and the database setting we consider here is that databases are usually considered to be finite structures, whereas there is usually no such restriction in the logical setting.

In the database context, the problem of deciding whether a view $\mathbf{V}$ determines a query Q is first mentioned in [34]. Here, the problem is referred to as *computing queries from derived relations*. A relational algebra expression R is said to be *algebraically derivable* from a set of expressions $R_1, \ldots, R_n$ if there exists an expression F using the symbols $R_1, \ldots, R_n$ that is equivalent to R on every database. In this case, F is called a *deriving expression*, which is exactly the analog of a rewriting in our setting. Remark that R being algebraically derivable from $R_1, \ldots, R_n$ does not say that $R_1, \ldots, R_n$ determines R, but that there exists a rewriting of R using $R_1, \ldots, R_n$, which might be stronger. However, it was observed in [37, 20] that the two notions actually coincide.

Following the same idea, [20] defines the notion of *invertibility*, saying that a view $\mathbf{V}$ is invertible relative to a query Q if there exists an inverse query Q^{-1} such that, for every database, $\mathrm{Q}(\mathrm{D}) = \mathrm{Q}^{-1}(\mathbf{V}(\mathrm{D}))$. Remark that this is exactly the same as saying that Q^{-1} is a rewriting of Q using $\mathbf{V}$. Thus, as said previously, $\mathbf{V}$ is invertible relative to Q if and only if $\mathbf{V}$ determines Q.

In [28], determinacy is viewed in terms of distinguishing power. More precisely, it is said that a set of queries $\mathbf{Q}_1$ *subsumes* a set of queries $\mathbf{Q}_2$ if all databases that are distinguished by $\mathbf{Q}_2$ are also distinguished by $\mathbf{Q}_1$. By contraposition, if $\mathbf{Q}_1(\mathrm{D}) = \mathbf{Q}_1(\mathrm{D}')$, then $\mathbf{Q}_2(\mathrm{D}) = \mathbf{Q}_2(\mathrm{D}')$, for all databases D and D′. Thus, saying that a view $\mathbf{V}$ subsumes $\{\mathrm{Q}\}$ corresponds exactly to saying that $\mathbf{V}$ determines Q.

Finally, [12] defines the notion of *losslessness*. Given a query Q, a view $\mathbf{V}$ and a view instance E, $\mathbf{V}$ is said to be lossless with respect to Q relative to E if it is the same to evaluate Q on any database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$, or to evaluate the certain answers

of Q on E. Intuitively, this means that E contains enough information to make it so that the certain answers of Q on E coincide with the possible answers of Q on E, and are thus complete. The authors go further by defining the notion of losslessness abstracted from a given view instance, by saying that $\mathbf{V}$ is lossless with respect to Q if $\mathbf{V}$ is lossless with respect to Q relative to all view instances. In other words, $\mathbf{V}$ is lossless with respect to Q if and only if the certain answers of Q are a rewriting of Q using $\mathbf{V}$.

Note that the notion of losslessness is defined under both the sound or exact view assumptions, meaning respectively that any query that computes certain answers under the sound or exact view assumptions, as defined in Section 2.1, is a rewriting of Q using $\mathbf{V}$. However, it was observed (for instance in [37]), that when Q determines $\mathbf{V}$, then any query that computes the certain answers of Q under the exact view assumption *is* a rewriting of Q using $\mathbf{V}$. Thus, the notion of losslessness under the exact view assumption coincides with the notion of determinacy.

In this work, we also consider a stronger form of determinacy, called *monotone determinacy*, defined in [37]. In addition to determinacy, it also requires the existence of a monotone rewriting of the query using the views. It turns out that losslessness under the sound view assumption, while intuitively a different notion (it refers to assumptions on the way views are materialized, and not on the form of the rewritings), coincides with monotone determinacy. Indeed, it states that any query computing certain answers under the sound view assumption is a rewriting. However, it can be checked that such a query is a monotone mapping. Moreover, it can also be checked (we do it formally in Chapter 5) that when $\mathbf{V}$ determines Q in a monotone way, then any query that computes certain answers under the sound view assumption is a rewriting, which establishes the equivalence. Note that monotone determinacy has also been considered in [39] under the name *strong determinacy*.

We conclude this section by mentioning that the reader should be careful with the notion of *rewriting*. In this work, we say that a rewriting is a query R over the schema defined by the view $\mathbf{V}$ such that, for all databases D, $Q(D) = R(\mathbf{V}(D))$. However, this name can take different meanings in the literature. For instance, in [12, 13, 11], a rewriting only has to verify that $R(\mathbf{V}(D)) \subseteq Q(D)$, and a rewriting such that $R(\mathbf{V}(D)) = Q(D)$ is called an *exact* rewriting, whereas a rewriting that coincides with certain answers is called a *perfect* rewriting. In [35], rewritings follow a different definition in that they can use relational symbols from both the original schema of the database and from the schema defined by the view. They are called *complete* rewritings when they only use symbols from the view schema.

2.3 Determinacy and rewritings

In this section, we discuss some of the most important results related to the determinacy and rewriting problems, as well as some of the most important open problems. Following the comparisons described in Section 2.2, we will restate all these results and problems in our framework, with our definitions.

Our starting point is the work presented in [37]. First, the authors carefully make

the distinction between determinacy, which is the semantic notion stating that a view **V** provides enough information to answer a query Q, and the existence of a rewriting in a specific language. This distinction will be an important guideline in all our work. As mentioned in Section 2.2, they also observe that when a view **V** determines a query Q, then any query computing the certain answers of Q using **V** under the exact view assumption is a rewriting of Q using **V**. This implies that the results from Section 2.1 immediately translate into upper bounds for the complexity of evaluating rewritings of a query Q using a view **V**. Note that we carefully say *upper* bounds even though the bounds of Section 2.1 are tight. Indeed, the existence of a hard rewriting for given query and view languages does *not* imply that all rewritings of said query should be hard. There may exists other rewritings that have a lower evaluation complexity.

Following these remarks, [37] provides a series of positive and negative results, for various query languages L_Q and view languages L_V. First they show that if either satisfiability in L_Q or validity in L_V is undecidable, then it is undecidable whether a view in L_V determines a query in L_Q. As an important corollary, this shows that determinacy is undecidable when either L_V or L_Q contains the set of all first-order queries. However, it is also proved that determinacy is also undecidable for much weaker view and query languages: indeed, the problem is already undecidable when L_V and L_Q are the set of unions of conjunctive queries. The authors then move on to discussing the expressive power that is required to express rewritings in various cases. They show that, as a consequence of Craig's Interpolation theorem, whenever a first-order view determines a first-order query, then a rewriting can be expressed as a first-order query. However, this only works in the *unrestricted* case, where databases are allowed to be infinite. Indeed, in the finite case, they prove that any language powerful enough to express the rewritings of first-order queries using first-order views must be Turing complete. However, for queries in first-order and views in existential first-order, both universal and existential second-order queries are complete rewriting languages. This is optimal in a sense: [37] proves that any complete language for rewritings in this case must be able to express all queries in $\exists\text{SO}\cap\forall\text{SO}$, and that this still holds for unions of conjunctive views.

A significant part of [37] discusses the case of monotone determinacy. It is shown that when a conjunctive view determines a conjunctive query in a monotone way, then there exists a conjunctive rewriting of the query using the view. Similarly, when both the view and the query are defined using unions of conjunctive queries, and the view determines the query in a monotone way, then a rewriting can be expressed as a union of conjunctive queries. This makes both problems decidable, as it is shown in [35] that the existence of a CQ (resp. UCQ) rewriting of a CQ (resp. UCQ) query using a CQ (resp. UCQ) view is decidable, actually NP-complete. Remark that this does not solve the non-monotone case, as [37] provides an example where a CQ view **V** determines a CQ query Q and where no monotone query can be a rewriting of Q using **V**, which means in particular that no CQ query can be a rewriting. At this point, [37] leaves two major open problems: is determinacy decidable for conjunctive views and queries ? What is a good rewriting language when a conjunctive view determines a conjunctive query ?

These two questions have received quite a bit of attention in both old and recent work. In [17], the authors identify a case where the NP-complete problem of deciding whether

there exists a CQ rewriting of a CQ query using CQ views become polynomial. They define the notion of *query width*, which is a generalization of acyclicity: a conjunctive query has query width 1 if and only if it is acyclic. From that, they prove that testing whether a CQ query has a CQ rewriting using CQ views is polynomial, provided that the query has bounded query width and does not use repeated predicates. Their proof is strongly related to the containment problem for conjunctive queries, for which it is known that acyclicity implies tractability.

In [3], the author identifies a fragment of conjunctive queries, called CQ_{chain}, for which determinacy is decidable. CQ_{chain} is the class of conjunctive queries over a binary schema whose underlying graphs is a directed simple path and whose endpoints are the only two free variables. Note that CQ_{chain} corresponds to the language that we call single path queries in Chapter 3. It is shown there that for queries and views defined in CQ_{chain}, determinacy is decidable and rewritings can always be expressed as first-order queries. Note also that CQ_{chain} queries and views retain the property of CQ queries and views that rewritings are not always monotone, so that there is no hope that CQ_{chain} can be a complete rewriting language in this case. However, if we further restrict the schema to a single binary symbol, then CQ_{chain} becomes *almost complete* for rewriting, meaning that for each CQ_{chain} view $\mathbf{V}$, there exists only a finite number of CQ_{chain} queries that are determined by $\mathbf{V}$ but do not have a CQ_{chain} rewriting. It is worth mentioning that the test for whether a CQ_{chain} view $\mathbf{V}$ determines a CQ_{chain} query Q is also very elegant: it reduces to checking whether there is an undirect path from x to y in $\mathbf{V}(\pi_{\mathrm{Q}})$, where π_{Q} is the simple directed path deduced from Q, whose endpoints are x and y.

In [38], the author builds upon the results from [3] by showing that it is decidable whether a CQ_{chain} view determines a CQ_{graph} query, where CQ_{graph} is the class of conjunctive queries on a binary schema whose underlying graph is connected and that contain exactly two free variables. In other words, [38] lifts the restriction from [3] on the shape of the query, by moving from single paths to arbitrary connected graphs. [38] further shows that when a CQ_{chain} view determines a CQ_{graph} query, a rewriting can still be expressed as a first-order query. Finally, [38] provides several decidable necessary conditions for a CQ_{graph} view to determine a CQ_{graph} query, but also prove that these conditions are unfortunately not sufficient.

The determinacy problem for conjunctive queries and views finally finds a partial answer in the very recent work [27], where it is shown that determinacy is undecidable in the *unrestricted case*. In this setting, it is said that a view $\mathbf{V}$ determines a query Q if, for all finite *and infinite* structures D and D′, $\mathbf{V}(\mathrm{D}) = \mathbf{V}(\mathrm{D}')$ implies that $\mathrm{Q}(\mathrm{D}) = \mathrm{Q}(\mathrm{D}')$. Of course, if a view determines a query in the unrestricted case, then the view also determines the query in the finite case. However, the converse does not hold. Indeed, it was observed in [37] that determinacy in the unrestricted case is recursively enumerable, whereas determinacy in the finite is co-recursively enumerable. Should both notions coincide, then both would be decidable, which [27] proved not to be the case. For now, the question remains open in the finite case.

Aside from the conjunctive case, determinacy has been studied for plenty of other view and query languages. In [5], the authors identify a fragment of first-order queries that is well-behaved for determinacy: the guarded-negation first-order queries (GNFO).

They show that GNFO is complete for rewriting of a GNFO query Q using a GNFO view $\mathbf{V}$, provided that both $\mathbf{V}$ and Q are answer-guarded, which means that the free variables of $\mathbf{V}$ and Q occur together in some relation of the schema. The proof technique goes through showing that GNFO has Craig's interpolation property. Remark also that determinacy in this case is decidable: this is a consequence of the fact that determinacy of a GNFO query using GNFO views can be expressed in GNFO, which is a decidable logic [6].

The case where either the view or the query is expressed in Datalog has also received some attention. In [20], it has been proved that determinacy is undecidable for a Datalog view and a conjuctique query, and similary for a conjunctive view and a Datalog query. In both cases, the proof relies on the fact that the containment problem is undecidable for two Datalog queries [42]. In [19], it is further shown that it is undecidable, given a conjunctive view and a Datalog query, whether there exists a Datalog rewriting of the query using the view. Furthermore, [1] proves that certain answers of a Datalog query using a conjunctive view under the sound view assumption is expressible in Datalog. This implies that in this case, monotone determinacy coincides with the existence of a Datalog rewriting (as explained in Section 2.2), thus together with [19], it proves that the monotone determinacy problem of a Datalog query using a conjunctive view is also undecidable. However, [19] provides an algorithm for computing, given a conjunctive view and a Datalog query, a Datalog program R over the schema defined by the view that is maximally contained in Q among all other Datalog program over the same schema. Thus, if the view determines the query in a monotone way, then this Datalog query R is a rewriting.

We spend the remaining part of this chapter discussing the case of regular path queries. It has been extensively studied [10, 11, 12, 13], however the problem of deciding whether a regular path view determines a regular path query remains open. This question is the starting point of a large part of this thesis, and of most of Chapter 4 and all of Chapter 5. We review now what is already known.

The main results of [10] are summarized in Figure 2.3. As already explained at the beginning of this section, these results immediately translate to upper bounds for the complexity of evaluating the rewriting of a regular path query determined by a regular path view. In particular, we can deduce from this results that when a regular path view determines a regular path query, then there exists a rewriting of the query using the view that has coNP data complexity, both for general and monotone determinacy.

In [11], the authors specifically consider the problem of answering regular path queries using regular path views under the sound view assumption. It is already known from [10] that this task is coNP-complete in data complexity. In [11], the authors provide another proof of this complexity bound, by showing that regular path query answering using views has strong connections with the constraint satisfaction problem (CSP). In particular, they show that computing the certain answers of a regular path query with respect to a regular path view reduces to the satisfiability of (the negation of) a uniform CSP. Building on this connection and on the known links between CSP and Datalog [21], they show how to compute approximations of this CSP in Datalog. More precisely, they show that, for each given integer n, regular path query Q, and regular path view $\mathbf{V}$, one

can compute a Datalog program R_n that uses only $n+2$ variables in each rule and such that for all databases D, $R_n(\mathbf{V}(D))$ is maximally contained in Q(D) among all Datalog programs that use only n variables in each rule. Of course, none of these programs is in general equivalent to certain answers, since Datalog has polynomial time data complexity, whereas computing certain answers is in general coNP-hard (unless PTime = coNP). However, the main result of Chapter 5 will build on this by showing that, when a regular path view determines a regular path query in a monotone way, then one of these Datalog programs is indeed a rewriting.

The determinacy problem is considered again in [12]. It is shown there that it is decidable, actually ExpSpace-complete, whether a regular path view determines a regular path query in a monotone way. However, [12] also provides an example where a regular path view determines a regular path query in such a way that there cannot exist any monotone rewriting. This shows that in this setting, determinacy does not coincide with monotone determinacy. The determinacy problem for regular path views and queries remains open for now.

Finally, in [13], the author consider the problem of rewriting a regular expression in terms of other regular expressions. Although most of their results have a more language theoretic flavor, some of them apply directly to the determinacy setting. In particular, they show that the problem of deciding, given a regular path query Q and a regular path view $\mathbf{V}$, whether there exists a rewriting of Q using $\mathbf{V}$ that can be expressed as a regular path query is 2ExpSpace-complete. [13] also provides a 2ExpSpace algorithm for computing this rewriting when it exists. Remark that, together with the result from [12] stating that monotone determinacy is decidable in ExpSpace, this proves that monotone determinacy does *not* coincide with the existence of a regular path rewriting. In Chapter 5, Example 5.9, we give an explicit example witnessing this fact. This is to be put in perspective with the result from [37] which proved that for conjunctive views and queries, monotone determinacy *does* coincide with the existence of a conjunctive rewriting.

2.4 Conclusions

We have seen in this chapter that the determinacy world is full of interesting and difficult problems, many of which are still open today. We want to highlight two of them:

1. The problem of deciding whether a regular path view determines a regular path query;

2. The problem of finding a low complexity rewriting of a regular path query using a regular path view, provided that the view determines the query.

In this work, we provide partial answers to both problems. In Chapter 5, we show that when a regular path view determines a regular path query *in a monotone way*, then there exists a Datalog rewriting of the query using the view, which can thus be evaluated with polynomial time data complexity. In Chapter 6, we extend the result of [3] by showing

how to add disjunctions to the language CQ_{path} used there to defined the views, which is a first step towards solving the first problem.

We conclude this chapter by highlighting again, as a take-away message, the fact that the problem of deciding whether a conjunctive view determines a conjunctive query is still open in the finite case. This is perhaps the most fundamental question of this setting, for which we unfortunately do not have any new insight.

Chapter 3

Databases and queries

This chapter sets the main definitions that will be used throughout this work. In Section 3.1, we formally define databases. We first recall the traditional relational model, and then move on to graph databases, which are the focus of our work. This also defines all the useful vocabulary related to databases and paths. In Section 3.2, we give a semantic definition of queries as mappings from databases to sets of answers. We also give unified definitions and notations for the query languages that will be relevant to our work. Finally, in Section 3.3, we discuss the query evaluation and query containment problems, two very common computational tasks related to databases and queries.

3.1 Databases

Databases. A relational schema σ is a finite set of relational symbols of finite arity. A database D is a finite structure over such a relational schema σ, and is also called a σ-structure. It consists of a finite set of elements $\mathrm{dom(D)} = \{x_1, \ldots, x_n\}$, called the domain of D, and of a set $\mathrm{I}_a(\mathrm{D})$ of k-tuples of dom(D) for each symbol a in σ of arity k, called the interpretation of a in D. The active domain of D, denoted by adom(D), is the set of elements of dom(D) that additionally appear in some tuple in $\mathrm{I}_a(\mathrm{D})$, for some a. When $(x_1, \ldots, x_k)$ belongs to $\mathrm{I}_a(\mathrm{D})$, we simply say that $a(x_1, \ldots, x_k)$ holds in D. The size of D, denoted by $|\mathrm{D}|$, refers to the number of elements in dom(D). Here, $|\mathrm{D}| = n$. Finally, if A is a subset of dom(D), we use $\mathrm{D}[A]$ to refer to the substructure of D induced by A, that is, the database whose domain is A, and such that $a(x_1, \ldots, x_k)$ holds in $\mathrm{D}[A]$ if and only if $(x_1, \ldots, x_k)$ is a tuple of elements in A and $a(x_1, \ldots, x_k)$ holds in D.

Example 3.1.

Book		
Twelfth Night	*Shakespeare*	*Play*
Nineteen Eighty-Four	*Orwell*	*Novel*
Leaves of Grass	*Whitman*	*Poetry*
A Treatise of Human Nature	*Hume*	*Essay*
The Tempest	*Shakespeare*	*Play*

Author	
Shakespeare	*British*
Orwell	*British*
Whitman	*American*
Hume	*British*
Locke	*British*

The two tables above represent the content of a database for a fictional library. Its schema is $\{Book, Author\}$*, where Book is a symbol of arity 3, and Author is a symbol of arity 2. The active domain of the database is the set that contains Twelft Night, Nineteen Eighty-Four, Leaves of Grass, A Treatise of Human Nature, The Tempest, Shakespear, Orwell, Whitman, Hume, Play, Novel, Poetry, Essay, British and American. Thus, its size is 15.*

The interpretation of Book in the database is the set of triples represented by the first table, and the interpretation of Author is the set of pairs represented by the second table. For instance, Book(The Tempest, Shakespeare, Play) holds in the database.

The substructure of this database induced by $\{$*The Tempest, Shakespeare, Play, British, American*$\}$ *is the one below. Notice that Twelfth Night does not appear in it, although it is related to both Play and Shakespeare, since it is not in the subset. Similarly, the tuple (Whitman, American) does not appear in the interpretation of Author in the substructure.*

Book		
The Tempest	*Shakespeare*	*Play*

Author	
Shakespeare	*British*

Graph databases. In this work, we focus our attention on graph databases. A graph database D is simply a database over a binary schema σ, that is a schema in which all relational symbols are of arity 2. This means that, for each symbol a in σ, $\mathrm{I}_a(\mathrm{D})$ is a set of pairs of elements of D. Hence, D can be interpreted as a directed graph with edges carrying labels from the finite alphabet σ: the set of nodes of the graph is dom(D), and for each symbol a in σ, the set of edges of label a of the graph is $\mathrm{I}_a(\mathrm{D})$. We adopt here the graph perspective that gives their name to graph databases. Thus, for two elements x and y of D, also called nodes, and a symbol a of σ, also called a label, we will interchangeably say that $a(x, y)$ holds in D, that there is an edge of label a from x to y, denoted $x \xrightarrow{a} y$.

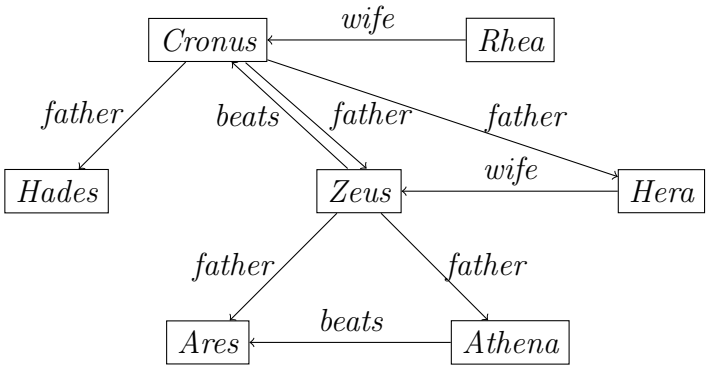
Example 3.2. *The tables below represent a database for a fictional social network. Remark that all relations are of arity 2, thus it is a graph database.*

Father	
Cronus	*Zeus*
Cronus	*Hades*
Cronus	*Hera*
Zeus	*Ares*
Zeus	*Athena*

Wife	
Rhea	*Cronus*
Hera	*Zeus*

Beats	
Zeus	*Cronus*
Athena	*Ares*

We can give a more natural representation of this database as the following graph:



Paths. In graph databases, we are typically interested in how the nodes of the graph are linked together, that is, we want to know what the paths of the graph are. A path π in a graph database D is a finite sequence $\pi = x_0 a_0 x_1 \dots x_{m-1} a_{m-1} x_m$, in which each x_i is a node of D, each a_i is a label in σ, and for all i, $a(x_i, x_{i+1})$ holds in D. x_0 and x_m are called the endpoints of π, and we say that π is a path from x_0 to x_m, which will often be denoted $x_0 \xrightarrow{\pi} x_m$. The size of π, denoted by $|\pi|$, refers to the number of (possibly repeating) edges in the sequence. Here, $|\pi| = m$. The label of π, denoted by $\lambda(\pi)$, is the sequence $a_0 \dots a_{m-1}$, seen as a word over the alphabet σ, or equivalently as an element from the free monoid σ^*. If no node occurs twice in the sequence defining π, we say that π is a simple path. If $x_0 = x_m$, then π is called a cycle, and π is a simple cycle if $x_0 = x_m$ is the only repetition. Finally, we will sometimes view π as a graph database itself, whose nodes and edges are exactly the nodes and edges that occur in the sequence. Remark that this does not coincide with the substructure of D induced by the nodes of π. In other words, we do not necessarily have $\pi = \mathrm{D}[\{x_0, \dots, x_m\}]$.

Example 3.3. *Let D be the graph database from Example 3.2. We explicit some information about its paths:*

- *Cronus* $\xrightarrow{father}$ *Hades is a simple path from Cronus to Hades.*
- *Rhea* $\xrightarrow{wife}$ *Cronus* $\xrightarrow{father}$ *Zeus* $\xrightarrow{beats}$ *Cronus is a path from Hera to Cronus. It is not simple because the node Cronus appears twice in it. The length of this path is 3 and its label is wife · father · beats.*
- *Cronus* $\xrightarrow{father}$ *Hera* $\xrightarrow{wife}$ *Zeus* $\xrightarrow{beats}$ *Cronus is a simple cycle.*
- *Consider the path* π = *Cronus* $\xrightarrow{father}$ *Zeus* $\xrightarrow{father}$ *Athena* $\xrightarrow{beats}$ *Ares. Then* π *can be seen as a graph database whose domain is* $\{Cronos, Zeus, Athena, Ares\}$ *and in which the following relations (and only these) hold: father(Cronus,Zeus), father(Zeus,Athena), beats(Athena,Ares).*

  Remark that $\pi \neq D[Cronus, Zeus, Athena, Ares]$. *Specifically, beats(Zeus,Cronus) and father(Zeus,Ares) hold in* $D[Cronus, Zeus, Athena, Ares]$ *but not in* π.
- *There is no path from Hades to Zeus.*

3.2 Queries

Queries. A query Q over a schema σ is a function that maps each database D over σ to a set of tuples of elements of D, that we simply note Q(D). Note that this is a purely semantic definition, which is not tied to any specific way of expressing or representing the query, as there can be many different representations of the same query. Moreover, a query Q defines, for each database D, what Q(D) should be, but does not state *how* to compute it. The question of computing Q(D) from D and the definition of Q is an algorithmic problem in itself, that will be discussed in Section 3.3.

In this work, we are mostly interested in binary queries, that is queries such that Q(D) is a set of pairs of elements of D. In this case, Q(D) can also be seen as a binary relation over the domain of D, which will be useful later on. In general, the size of the tuples returned by Q is called the arity of Q. We spend the rest of this section defining the relevant classes of queries that we use.

Queries defined by a formal language. Let L be a formal language over a binary schema σ, that is a (possibly infinite) set of words of σ^*. We denote by $\langle L \rangle$ the query induced by L. For any graph database D over σ, $\langle L \rangle$(D) is the set of pairs (x, y) such that there exists in D a path π from x to y with $\lambda(\pi) \in L$. Conversely, given a query Q defined by a formal language, we use L(Q) to refer to the set of words defining Q. This generic definition covers several classical query classes, depending on the properties of L:

- Atomic queries : L contains a single letter $a \in \sigma$.
- Single path queries - SPQ: L contains a single word $w \in \sigma^*$.
- Union of single path queries - UPQ: L is a finite set of words in σ^*.
- Regular path queries - RPQ: L is any regular langage over σ. By abuse of notation, we will not make the distinction between L or a regular expression e describing L. In this case, we will also use $\langle e \rangle$ to refer to $\langle L \rangle$.
- Context-free path queries - cfPQ: L is any context-free language over σ.
- (Any language) Path queries : L is any formal language, that is, any subset $L \subseteq \sigma^*$.

Example 3.4. *Once again, let D be the graph database from Example 3.2. Here are some example queries and their answers on D:*

- $\langle beats \rangle$ *is an atomic query. On D, it returns* (*Zeus,Cronus*) *and* (*Athena,Ares*).
- $\langle wife \cdot father \cdot father \rangle$ *is a single path query. On D, it returns the pairs* (*Rhea,Ares*) *and* (*Rhea,Athena*)*, telling us that Rhea is the grandmother of both Ares and Athena.*
- $\langle father^+ \rangle$ *is a regular path query. It realizes the transitive closure of the father relation, linking each (arbitrary grand) father with his (arbitrary grand) children. On D, it returns the pairs* (*Cronus,Hades*)*,* (*Cronus,Zeus*)*,* (*Cronus,Hera*)*,* (*Cronus,Ares*)*,* (*Cronus,Athena*)*,* (*Zeus,Ares*) *and* (*Zeus,Athena*).

A significant part of this work deals with the case where σ only contains a single symbol a, that is $\sigma = \{a\}$. In this case, we can remark that any language L over σ is a (possibly infinite) set of the form $\{a^{k_1}, \ldots, a^{k_n}, \ldots\}$, where the only relevant information about L is the length of the words it contains. By abuse of notation and when this is clear from the context, we will simply write $L = \{k_1, \ldots, k_n, \ldots\}$. Similarly, we will simply write $\langle k_1, \ldots, k_n, \ldots \rangle$ for the query $\mathrm{Q} = \langle \{a^{k_1}, \ldots, a^{k_n}, \ldots\} \rangle$.

Remark 3.5. *In an attempt to give a unified presentation for multiple query languages that occur frequently in related work, we have slightly deviated from the usual vocabulary. Our terminology relies on the idea that a path query is any query that selects pairs of nodes based on the existence of a path linking the two nodes and whose label satisfies some condition. The adjectives then attached to the name of the query languages are used to restrict the kind of conditions that queries of the languages are allowed to express. Thus, a* regular *path query is a query whose associated language has to be* regular*, instead of any language. This has the following consequences:*

- *We call single path queries the class known as chain queries in [3];*
- *We call unions of single path queries the class known as unions of path queries in [23];*
- *The class known as path queries in [3] should not be confused with what our definition of path queries. In our work, path queries correspond to the most general form of queries induced by a formal language, whereas in [3], path queries are single path queries on a single letter schema, that is, queries that select pairs of nodes if they are linked by a path of a given length.*

Queries defined by a logic formula. Let $\varphi(x_1, \ldots, x_m)$ be a logic formula in some logic class, with free variables $x_1, \ldots, x_m$. We use $\langle \varphi \rangle$ to denote the query induced by φ. Given a database D, $\langle \varphi \rangle(\mathrm{D})$ is the set of tuples $(x_1, \ldots, x_m)$ of elements of the database such that $\varphi(x_1, \ldots, x_m)$ holds in D. This covers several query languages, depending on the logic class to which φ belongs, namely depending on which connectives and atoms are allowed to occur in φ. When a query is used as an atom, it should be understood as a relational symbol whose interpretation is the set of answers to the query over the given database.

- Conjunctive queries - CQ: φ is an existential positive conjunctive formula using atomic queries as atoms. Allowed connectives are existential quantification ($\exists$) and conjunction ($\wedge$).
- Union of conjunctive queries - UCQ: φ is an existential positive formula using atomic queries as atoms. Allowed connectives are existential quantification ($\exists$), conjunction ($\wedge$) and disjunction ($\vee$).
- First-order queries - FO: φ is a first order formula using atomic queries as atoms. Allowed connectives are all first-order connectives.

- Conjunctive regular path queries - CRPQ: φ is an existential positive conjunctive formula using RPQ queries as atoms.

Example 3.6. *Once again, let D be the graph database in Example 3.2. Here are some examples of the newly introduced query languages:*

- *Let* $\varphi_1(x, y) = \exists z, \mathit{father}(z, x) \wedge \mathit{father}(z, y) \wedge \mathit{wife}(y, z)$. *Then* $Q_1 = \langle \varphi_1 \rangle$ *is a conjunctive query. It returns the pairs that correspond to married couples that are also siblings. On D, it returns the pair (Zeus,Hera).*

- *Let* $\varphi_2(x) = \forall y, \neg(\mathit{father}(x, y) \wedge \mathit{beats}(y, x)) \wedge \exists y, \mathit{father}(x, y)$. *Then* $Q_2 = \langle \varphi_2 \rangle$ *is a first-order query. It returns all nodes that correspond to fathers that have not been beaten by one of their children. On D, it returns only Zeus.*

- *Finally, let* $\varphi_3(x) = \exists y, \langle \mathit{father}^+ \rangle(x, y) \wedge \langle \mathit{beats} \rangle(y, z)$. *Then* $Q_3 = \langle \varphi_3 \rangle$ *is a conjunctive regular path query. It returns all nodes that correspond to people that have been beaten by one of their descendants. On D, it returns Cronus.*

Datalog. A Datalog query is a set of rules of the form $p(\bar{x}) :- p_1(\bar{x}_1) \wedge \ldots \wedge p_n(\bar{x}_n)$, where $\bar{x}$ and the $\bar{x}_i$'s are tuple of variables, p is a predicate symbol that is internal to the query, sometimes called an *intensional predicate*, and the p_i's can either be symbols from σ, sometimes called *extensional predicates* in this context, or other intensional predicates of the query. $p(\bar{x})$ is called the *head* of the rule, and $p_1(\bar{x}_1) \wedge \ldots \wedge p_n(\bar{x}_n)$ is called the *body* of the rule. All variables that appear in the head of the rule must also occur in the body of the rules, and the other variables of the body should be understood as being existentially quantified. One designated intensional symbol is called the *goal* of the query. The *arity* of the query is defined as the arity of its goal.

Intuitively, a Datalog query can be seen as a set of mutually defined conjunctive queries. On a given database D, a Datalog query incrementally populates the relations associated with each of these queries by applying the rules. The answer of the query on D is the relation associated with its goal, when no more tuple can be added to any relation. Note that this is well defined, as each rule is monotone, which makes it so that a fixpoint is always reached in a finite number of steps.

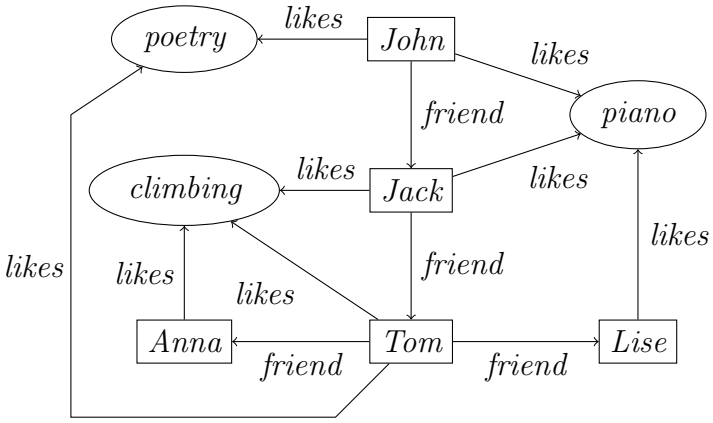
In this work, we will also consider a restricted class of Datalog queries. For two integers ℓ and k, $\mathrm{Datalog}_{\ell,k}$ is the set of Datalog queries of arity r whose rules contain at most $k + r$ variables, and whose intensional predicates are of arity at most $\ell + r$.

Example 3.7. *The typical example of a Datalog query is the "friends of friends" that implements the transitive closure of a binary predicate, in this example the transitive closure of the friend relation in an hypothetical social network.*

However, we have seen previously that this can easily be expressed as a regular path query. Here, we give an interesting twist to this example, making it so that it cannot be expressed in any of the other query languages presented here. The following Datalog query links all nodes x to potential friends y that are connected through a chain of friends and additionally ensures that, at each step, each of the potential friends share a common interest with x:

- $pf(x, y) :- friend(x, y)$
- $pf(x, y) :- pf(x, z) \wedge friend(z, y) \wedge likes(x, t) \wedge likes(y, t)$

Consider the following fictional social network:



On this database, the Datalog query deduces the following facts at the following steps:

1. *pf(John,Jack), pf(Jack,Tom), pf(Tom,Anna) and pf(Tom,Lise) by using the first rule.*
2. *pf(John,Tom) by using the second rule with $z = Jack$ and $t = poetry$.*
   pf(Jack,Lise) by using the second rule with $z = Tom$ and $t = piano$.
   pf(Jack,Anna) by using the second rule with $z = Tom$ and $t = climbing$.
3. *pf(John,Lise) by using the second rule with $z = Tom$ and $t = piano$.*

After step 3, no new fact can be deduced, so the answer to the query is fully defined. In particular, we do not have pf(John,Anna) since they are not friends and do not share a common interest. Remark also that we do have pf(Tom,Lise) even though they do not have a common interest, since they are already friends.

Monotonicity. A query Q is said to be monotone if it defines a monotone mapping from databases to query answers. In other words, Q is monotone if and only if, for all databases D and D′, if $D \subseteq D'$, then $Q(D) \subseteq Q(D')$. By extension, a query language is said to be monotone if all queries that belong to the language are monotone.

Remark that, except for first-order queries, all query languages presented here select tuples based on the existence of some object in the database: a path, a specific tuple or relation, a combination of such conditions, and so on. Therefore, all these query languages are monotone. We can also see that first-order queries are *not* always monotone. Indeed, remember the first-order query Q_2 given in Example 3.6. On the considered database D, it returns the answer Zeus. Consider a new database D′ that is a copy of D with the exception that *beats(Athena,Zeus)* holds in D′. Then $D \subseteq D'$ but $Q_2(D') = \emptyset$, which proves that Q_2 is not monotone.

This property of query languages will play a key role in Chapter 5.

Notations. For two binary relations R and S, we write $R \cdot S$ for the following relation:

$$R \cdot S = \{(x, y) \mid \exists z, R(x, z) \text{ and } S(z, y)\}$$

Let n be a positive integer, we also use $R^1 = R$, and $R^n = R^{n-1} \cdot R$ if $n \geq 2$. For two binary queries Q and P, we define $\mathrm{Q} \cdot \mathrm{P}$ as the query that maps a database D to the set of pairs (x, y) such that there exists z with $(x, z) \in \mathrm{Q(D)}$ and $(z, y) \in \mathrm{P(D)}$. This notation is consistent with the fact that $\mathrm{Q} \cdot \mathrm{P(D)} = \mathrm{Q(D)} \cdot \mathrm{P(D)}$, that is, it is the same to look at the answer of the composed query, or two take each set of answers separately and then compose them. We similarly define Q^n.

Remark that, for all query languages $\mathcal{L}$ defined here, with the exception of atomic queries, it is the case that if Q and P are in $\mathcal{L}$, then $\mathrm{Q} \cdot \mathrm{P}$ can also be expressed as a query in $\mathcal{L}$. When this is the case, we say that the query language $\mathcal{L}$ is closed under concatenation.

For instance, if Q and P are unions of single path queries, defined by $\mathrm{Q} = \langle\{u_1, \ldots, u_n\}\rangle$ and $\mathrm{P} = \langle\{v_1, \ldots, v_m\}\rangle$, then $\mathrm{Q} \cdot \mathrm{P}$ is equivalent to the union of single path queries defined as $\langle\{u_i \cdot v_j \mid i \leq n, j \leq m\}\rangle$.

3.3 Algorithms

3.3.1 Query evaluation

Query evaluation is perhaps the most natural computational task to consider when dealing with databases and queries. Given a database D and a query Q, it consists in computing the set of answers to Q in D, that is Q(D). Its decision counterpart is expressed as follows: given a database D, a query Q and a tuple of elements $(x_1, \ldots, x_m)$ of D, decide whether $(x_1, \ldots, x_m) \in \mathrm{Q(D)}$.

Problem :	Query evaluation for query language $\mathcal{L}$
Input :	A database D, a tuple $(x_1, \ldots, x_m)$ of elements of D A query Q in language $\mathcal{L}$
Question :	Does $(x_1, \ldots, x_m)$ belong to Q(D)?

This problem shows a natural trade-off: for practical purposes, we want to chose a query language $\mathcal{L}$ that has both high expressive power and low evaluation complexity. Of course, one comes at the cost of the other, so that simple query languages, like atomic queries, will have linear time query evaluation but very low expressive power, whereas a powerful language such as conjunctive queries has NP-complete query evaluation. In this work, we are only interested in the so-called *data complexity* for query evaluation: we assume that the query to be evaluated is fixed, and only the database D and the tuple of elements of D are part of the input. This is in line with the idea that, in practical applications, the queries are most likely small (for instance because they are manually typed) whereas databases can be huge (for instance, the whole Web).

All query languages considered in Section 3.2 have PTIME data complexity. We prove it here by showing that Datalog contains all the query languages considered here, except for First-order queries, and then giving a PTIME algorithm for evaluating a Datalog query. For a finer complexity picture, refer to Figure 3.1 and to the literature [40, 7, 32, 2].

Lemma 3.8. *Let Q be a cfPQ or a CRPQ. Then Q can be expressed as a Datalog query.*

Proof. Let Q be a cfPQ. Let $G = \langle V, P, S\rangle$ be a context-free grammar that recognizes $L(\mathrm{Q})$. Assume without loss of generality that G is in Chomsky Normal Form. That is, every rule in G is either of the form $A \to a$, with $A \in V$ and $a \in \sigma$ or of the form $A \to BC$ with $A, B, C \in V$. For a variable $A \in V$, we denote by $L_G(A)$ the language produced by G when A is taken as the initial symbol. Thus, $L(G) = L_G(S)$.

We define a Datalog program Q' which contains the following rules:

- $A(x, y) :- a(x, y)$ for all rule $X \to a$ of G;
- $A(x, y) :- B(x, z) \wedge C(z, y)$ for all rule $A \to BC$ of G;
- S is the goal of Q'.

Let us prove that Q is equivalent to Q'. This is an immediate consequence of the following claim:

Claim 3.9. *Let D be a database. Let $A \in V$. Let x, y be two nodes of D. Then, there exists a path π from x to y in D such that $\lambda(\pi) \in L_G(A)$ if and only if Q' produces $A(x, y)$ on D.*

We prove the direct direction by induction over the depth of the derivation in G that produces $\lambda(\pi)$ from A.

- Assume that the derivation is of depth 1. Then $\lambda(\pi) = a$, for some $a \in \sigma$, and $A \to a$ is a rule of G. Thus $A(x, y) :- a(x, y)$ is a rule of Q'. Since $a(x, y)$ holds in D, then Q' produces $A(x, y)$ on D by applying this rule.
- Assume that the derivation is of depth more than 1. Then the first rule used by G is necessarily of the form $A \to BC$. Thus, there exist two words w_b and w_c such that $w_b \in L_G(B)$, $w_c \in L_G(C)$ and $\lambda(\pi) = w_b \cdot w_c$.

  Let z be a node such that $x \xrightarrow{w_b} z \xrightarrow{w_c} y$ is a path in D. By the induction hypothesis, Q' produces $B(x, z)$ and $C(z, y)$ on D. Additionally, since $A \to BC$ is a rule of G, then $A(x, y) :- B(x, z) \wedge C(z, y)$ is a rule of Q'. Thus Q' produces $A(x, y)$ on D.

We prove the converse direction in a very similar way, by reasoning by induction on the length of the deduction sequence in Q' that produces $A(x, y)$. This concludes the proof of the claim.

Now, let D be a database, and let $(x, y) \in \mathrm{Q}(\mathrm{D})$. Then, there exists a path π from x to y in D such that $\lambda(\pi) \in L_G(S)$. Then the claim gives us that Q' produces $S(x, y)$ on D. Since S is the goal of Q', then $(x, y) \in \mathrm{Q}'(\mathrm{D})$.

Conversely, assume that $(x, y) \in \mathrm{Q}'(\mathrm{D})$. Then, Q' produces $S(x, y)$ on D. It follows from the claim that there exists a path π from x to y such that $\lambda(\pi) \in L_G(S)$. Since S is the axiom of G, this proves that $\lambda(\pi) \in L(\mathrm{Q})$, and then that $(x, y) \in \mathrm{Q}(\mathrm{D})$.

This concludes the proof that Q is equivalent to Q'.

Let now Q be a CRPQ. Then there exists an existential conjunctive positive formula $\varphi(x, y)$ such that $\mathrm{Q} = \mathrm{Q}(\varphi)$. Then φ is of the form $\varphi(x, y) = \exists z_1, \ldots, z_n, L_1(s_1, t_1) \wedge \ldots \wedge L_m(s_m, t_m)$, where the s_i's and t_i's are variables among $x, y, z_1, \ldots, z_n$, and the L_i's are atoms of the formula that represent RPQs $\mathrm{Q}_1, \ldots, \mathrm{Q}_n$.

Let $\mathrm{Q}'_1, \ldots, \mathrm{Q}'_n$ be Datalog queries that are respectively equivalent to $\mathrm{Q}_1, \ldots, \mathrm{Q}_n$, whose existence follows from the first part of the proof. Assume without loss of generality that the intensional predicates of the Q'_i's are pairwise distinct. Then it is straightforward to see that Q is equivalent to the Datalog query Q' defined as follows:

- Q' contains all the rules of the Q'_i's;
- Q' contains the rule $S(x, y) :- S_1(s_1, t_1) \wedge \ldots \wedge S_m(s_m, t_m)$, where S is a fresh intensional predicate that does not appear in the Q'_i's, and the S_i's are the respective goals of the Q'_i's;
- the goal of Q' is S.

This concludes the proof of the lemma. □

We now prove that Datalog has PTime data complexity. The proof we give here is quite naive and elementary. More efficient techniques can be found in the literature, but they are not the focus of this work. See [2] for reference.

Lemma 3.10. *The query evaluation problem for Datalog queries has* PTime *data complexity.*

Proof. Let Q be a fixed Datalog query, and D be a database. We evaluate Q over D in a bottom-up way as follows:

1. For each intensional predicate A of Q of arity k, we define a set $\mathrm{I}_A(\mathrm{D})$ of k-tuples of elements of D. We initialize $\mathrm{I}_A(\mathrm{D})$ as the empty set. Remark that the size of $\mathrm{I}_A(\mathrm{D})$ is at most $|\mathrm{D}|^k$, and is thus polynomial in $|\mathrm{D}|$.
2. For each rule of the form $A(\bar{x}) :- \bigwedge_i A_i(\bar{y}_i) \bigwedge_j a_j(\bar{z}_j)$, where A and the A_i's are intensional predicates of Q and the a_j's are extensional predicates, for all $\bar{y}_i \in \mathrm{I}_{A_i}(\mathrm{D})$ and all $\bar{z}_j \in \mathrm{I}_{a_j}(\mathrm{D})$, we add the corresponding $\bar{x}$ to $\mathrm{I}_A(\mathrm{D})$. This step takes at most $|\mathrm{D}|^{dk}$ where d is the maximum size of a rule in Q and k is the maximum arity of an intensional or extensional predicate in Q. Thus this step takes a polynomial time in $|\mathrm{D}|$.
3. Repeat step 2 until no more new tuple can be added to one of the $\mathrm{I}_A(\mathrm{D})$. This step is used at most $O(|\mathrm{D}|^k)$, where k is the maximum arity of an intensional predicate in Q.

	Data complexity	Combined complexity
Atomic queries	Linear time	Linear time
Single Path queries	Linear time	PTime
Regular path queries	NLogSpace	PTime
Context-free path queries	PTime	PTime
Conjunctive queries	LogSpace	NP
First-order queries	LogSpace	PSpace
Conjunctive regular path queries	NLogSpace	NP
Datalog	PTime	ExpTime

Figure 3.1: Complexity of the query evaluation problem for some query languages.

4. Return $\mathrm{I}_G(\mathrm{D})$, where G is the goal of Q.

Altogether, this gives a correct algorithm for evaluating a Datalog query in time polynomial in $|\mathrm{D}|$. □

In view of Lemma 3.8 and Lemma 3.10, we can immediately conclude that all query languages presented in this chapter have PTime data complexity.

3.3.2 Query containment

Query containment is a common static analysis problem that asks the following question: given two queries Q_1 and Q_2 in some query language $\mathcal{L}$, is it true that the answers to Q_1 are always contained in the answers to Q_2. In other words, can we deduce, for any database D, that if a tuple $(x_1, \ldots, x_m)$ is in $\mathrm{Q}_1(\mathrm{D})$, then it necessarily also belong to $\mathrm{Q}_2(\mathrm{D})$?

Problem :	Query containment for query language $\mathcal{L}$
Input :	Two queries Q_1 and Q_2 in language $\mathcal{L}$
Question :	Is it true that $\mathrm{Q}_1(\mathrm{D}) \subseteq \mathrm{Q}_2(\mathrm{D})$ for any database D?

The query containment problem is described as a static analysis problem because it only depends on the definition of Q_1 and Q_2, the two queries given as input. In particular, it does not depend on any given database D, and thus can be solved offline. Remark that, for two path queries Q_1 and Q_2, the query containment problem coincides with the containment problem of the two languages $L(\mathrm{Q}_1)$ and $L(\mathrm{Q}_2)$.

Lemma 3.11. *Let Q_1 and Q_2 be two path queries. Then, $L(Q_1) \subseteq L(Q_2)$ if and only if for all database D, $Q_1(D) \subseteq Q_2(D)$.*

Proof. Assume that $L(\mathrm{Q}_1) \subseteq L(\mathrm{Q}_2)$. Let D be a database, and $(x, y) \in \mathrm{Q}_1(\mathrm{D})$. Then there exists a path π from x to y in D such that $\lambda(\pi) \in L(\mathrm{Q}_1)$. Hence, $\lambda(\pi) \in L(\mathrm{Q}_2)$, which implies that $(x, y) \in \mathrm{Q}_2(\mathrm{D})$.

	Complexity
Atomic queries	Constant time
Single Path queries	Linear time
Regular path queries (DFA)	PTime
Regular path queries (NFA)	PSpace
Context-free path queries	Undecidable
Conjunctive queries	NP
Conjunctive regular path queries	ExpSpace
First-order queries	Undecidable
Datalog	Undecidable

Figure 3.2: Complexity of the query containment problem for some query languages.

Conversely, assume that for all databases D, $Q_1(D) \subseteq Q_2(D)$. Let $w \in L(Q_1)$. Consider the database D that consists of the simple path $x_0 w_0 x_1 \dots x_{n-1} w_{n-1} x_n$, where n is the length of w. Then $(x_0, x_n) \in Q_1(D)$, which implies that $(x_0, x_n) \in Q_2(D)$. Since π is the only path from x_0 to x_n, we can deduce that $\lambda(\pi) \in L(Q_2)$, and conclude that $w \in L(Q_2)$. □

Corollary 3.12. *Let $\mathcal{C}$ be a class of formal languages over σ and $\mathcal{L}$ be the corresponding class of path queries. Then the query containment problem for query language $\mathcal{L}$ is equivalent to the containment problem for $\mathcal{C}$.*

This corollary allows us to easily fill a big part of Figure 3.2, by translating known results from formal language theory. Note that this depends on how the specific language is represented. This makes a difference in the case of RPQs, The complexity results for query languages that are not path queries require specific proofs from the literature, see [16] for conjunctive queries, [22] for conjunctive regular path queries and [40] for first-order queries. Note that the result for Datalog comes easily from the fact that Datalog is more expressive that Context-free Path Queries.

Chapter 4

Views and operations on views

This chapter introduces the notion of views and discusses several computational problems related to views. In Section 4.1, we give a formal definition of views, view instances and view images. Section 4.2 discusses the computational tasks related to the processing of a given view instance along with the view definition. We show how these problems relate to each other, which results in most of them being intractable for the query and view languages that are relevant to us. More importantly, in Section 4.3, we introduce view-based query determinacy, seen as a static analysis variant of the problems of Section 4.2. This last section is of crucial importance for the rest of our work, and should be considered as preliminaries for Chapter 5 and Chapter 6.

4.1 Views

Let σ and τ be two relational schemas. Then a *view* $\mathbf{V}$ from σ to τ is a set of queries over σ, one for each symbol of τ. Additionally, for each $b \in \tau$, the associated query Q_b in $\mathbf{V}$ has the same arity as b.

Given a database D over σ, the *view image* of D, $\mathbf{V}(\mathrm{D})$, is defined as the database over τ such that:

- its domain is $\{x \mid x \in \bar{x} \text{ and } \bar{x} \in \mathrm{Q}_b(\mathrm{D}), \text{ for some } b \in \tau\}$;
- For each $b \in \tau$, $\mathrm{I}_b(\mathbf{V}(\mathrm{D})) = \mathrm{Q}_b(\mathrm{D})$.

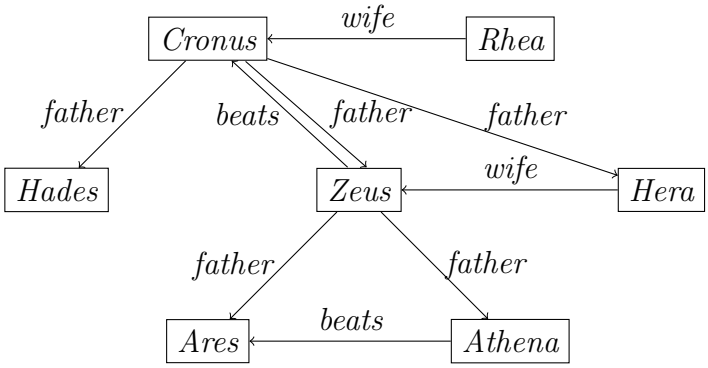
By abuse of notation, we will use the same symbol (typically V) to denote both a symbol $b \in \tau$ and its associated query $\mathrm{Q}_b \in \mathbf{V}$. Thus, it is the same to say that $V(x, y)$ holds in $\mathbf{V}(\mathrm{D})$, or that $(x, y) \in V(\mathrm{D})$. In other words, the interpretation of (the symbol) V in $\mathbf{V}(\mathrm{D})$, and the answers to (the query) V on D represent the same relation.

Remark that τ is a relational schema, so we can define databases over τ. In this context, a database over τ will be called a *view instance* and will be typically denoted by E. It is important to note that not all view instances are view images. In other words, for a given database E over τ, there might exist a database D over σ such that $\mathrm{E} = \mathbf{V}(\mathrm{D})$, or there might not. It turns out that, depending on the query language used to define

V, deciding whether a view instance is a view image can be a hard task, as we will see in Section 4.2.

When all the queries used to define the a view belong to a certain query language, we will informally extend the name of the query language to the view. For instance, a view defined as a set of conjunctive queries will be called a conjunctive view, and a view defined as a set of first-order queries will be called a first-order view.

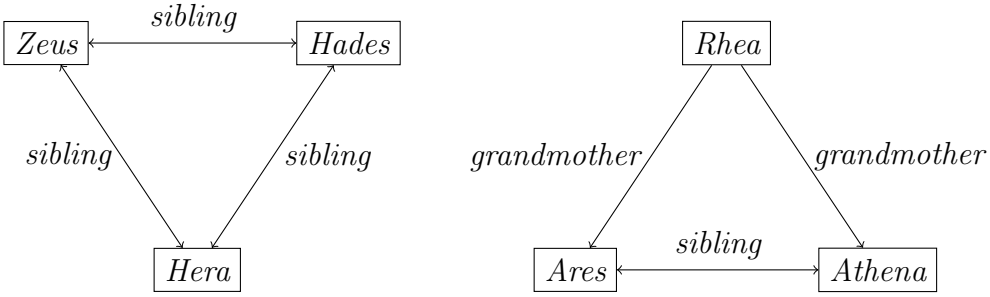
Example 4.1. *Let D be the database of Example 3.2 that we recall here:*



And consider the first-order view $\boldsymbol{V}$ which consists of the two following queries:

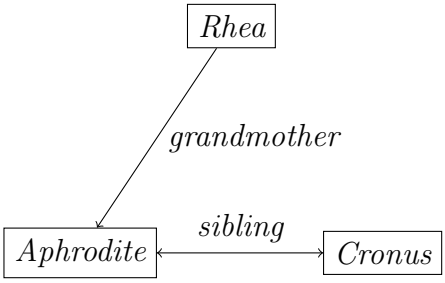
- $sibling(x, y) = \exists z, father(z, x) \wedge \big(\forall z, father(z, x) \Leftrightarrow father(z, y)\big) \wedge x \neq y$
- $grandmother(x, y) = \exists z, wife(x, z) \wedge \exists z', father(z, z') \wedge father(z', y)$

Then the view image of D is the following database $\boldsymbol{V}(D)$:



Notice that the node Cronus is missing from the view image, despite being used to satisfy the queries. This kind of missing information will be one of the main challenges when trying to reason about the view images, as will be seen throughout this work.

Consider now the following view instance E:



Then one can see that E is not actually a view image. Indeed, assume that there exists a database D' such that $E = \boldsymbol{V}(D')$. Then we have $(Rhea, Aphrodite) \in grandmother(D')$. Thus, there exists z and z' in D', such that the following path is in D':

$$Rhea \xrightarrow{wife} z \xrightarrow{father} z' \xrightarrow{father} Aphrodite$$

Since $(Aphrodite, Cronus) \in sibling(D')$, then we also know that $father(z', Cronus)$ holds in D'. Thus, the following path is also in D':

$$Rhea \xrightarrow{wife} z \xrightarrow{father} z' \xrightarrow{father} Cronus$$

Hence, $(Rhea, Cronus) \in grandmother(D')$, but $grandmother(Rhea, Cronus)$ does not hold in E, which is a contradiction. Thus, E cannot be a view image.

While this might look like a toy example, it is actually telling of a phenomemon that can happen while trying to integrate data as though it was the view of a virtual global database. This correponds to the local-as-view paradigm, as briefly explained in Chapter 1. In this case, we can imagine that E was obtained as the composition of two databases, and it so happens that the first one follows the myth in which Aphrodite is the daughter of Ouranos, whereas the second one places Aphrodite as the daughter of Zeus. E not being a view image specifically tells us that these two databases are incompatible.

4.2 Materialized view problems

Assume that we have a view instance E, for a given view $\mathbf{V}$ from σ to τ. One natural question is: can we use E to deduce more information than what is already available in E? This section discusses some of the most common tasks that are related to reasoning about view instances. These tasks are called *materialized view problems*, because they all work around a *given* view instance.

4.2.1 Certain answers

The first task we consider is perhaps the most natural: can we get a query Q over σ answered as if it was asked on a database D from which our view instance E is taken? There are several ways through which this question has been formalized in the literature. Here we consider the *certain answers* perspective, which relates the question to answering the following queries over τ:

$$\mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E}) = \bigcap_{\mathrm{D} \mid \mathrm{E}=\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D})$$

$$\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E}) = \bigcap_{\mathrm{D} \mid \mathrm{E}\subseteq\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D})$$

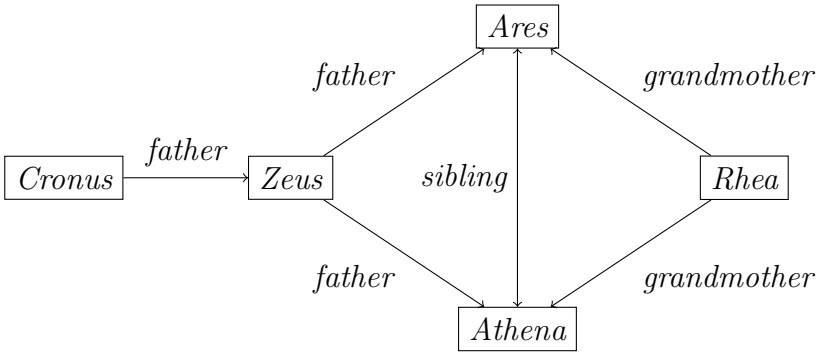
The first query is usually referred to as certain answers under the *exact view assumption* or *closed world assumption* in the literature [1, 10, 12]. Similarly, the second query is referred to as certain answers under the *sound view assumption* or *open world assumption*.

Let $\bar{x}$ be a tuple of elements of E such that $\bar{x} \in \text{cert}^{\text{exact}}_{\text{Q},\mathbf{V}}(\text{E})$. Then, for all database D such that $\mathbf{V}(\text{D}) = \text{E}$, we have $\bar{x} \in \text{Q}(\text{D})$. This means that if we want to answer a query Q on a database D while we only have access to $\text{E} = \mathbf{V}(\text{D})$, the only guaranteed tuples are those that appear in $\text{cert}^{\text{exact}}_{\text{Q},\mathbf{V}}(\text{E})$. Indeed, if $\bar{x} \notin \text{cert}^{\text{exact}}_{\text{Q},\mathbf{V}}(\text{E})$, then there exists a database D', such that $\bar{x} \notin \text{Q}(\text{D}')$ and $\mathbf{V}(\text{D}') = \text{E}$. This D' might as well be our initial D, so that we cannot ensure that $\bar{x} \in \text{Q}(\text{D})$.

The situation is the same for $\text{cert}^{\text{sound}}_{\text{Q},\mathbf{V}}$, except that we only assume that we have access to a view instance E such that $\text{E} \subseteq \mathbf{V}(\text{D})$. Thus, a tuple $\bar{x}$ is certain if and only if it belongs to $\text{Q}(\text{D}')$ for all D' such that $\text{E} \subseteq \mathbf{V}(\text{D}')$.

Remark that if there exists no database D such that $\text{E} = \mathbf{V}(\text{D})$ (respectively, $\text{E} \subseteq \mathbf{V}(\text{D})$), then this means that the view instance E is inconsistent for $\text{cert}^{\text{exact}}_{\text{Q},\mathbf{V}}$ (respectively, $\text{cert}^{\text{sound}}_{\text{Q},\mathbf{V}}$). This corresponds to a degenerate case in which all tuples are certain.

Example 4.2. *Let $\boldsymbol{V}$ be the view consisting of the queries father, sibling and grandmother as defined in Example 4.1. Consider the following view instance E:*



Let Q be the query $\langle wife \rangle$. Then, we can see that $(Rhea, Cronus) \in cert^{exact}_{Q,\boldsymbol{V}}(E)$. However, $(Rhea, Cronus) \notin cert^{sound}_{Q,\boldsymbol{V}}(E)$. Indeed, Rhea is a grandmother, so by definition, she has to be the wife of a grandfather. In the exact view setting, we know that the only available grandfather is Cronus, so Rhea must be the wife of Cronus. In the sound view setting, it could however be the case that Rhea is the wife of some other grandfather of Ares and Athena, whose node is currently missing from the view instance.

We give a decision version of the problem of answering certain answers as follows:

PROBLEM :	CERTAIN ANSWERS FOR QUERY AND VIEW LANGUAGES $\mathcal{L}$ AND $\mathcal{L}'$ UNDER THE EXACT (RESP. SOUND) VIEW ASSUMPTION
INPUT :	A query Q in $\mathcal{L}$, a view $\mathbf{V}$ in $\mathcal{L}'$, a view instance E, A tuple $(x_1, \ldots, x_m)$ of elements of E
QUESTION :	Does $(x_1, \ldots, x_m)$ belong to $\text{cert}^{\text{exact}}_{\text{Q},\mathbf{V}}(\text{E})$? (resp. $\text{cert}^{\text{sound}}_{\text{Q},\mathbf{V}}(\text{E})$?)

The complexity of answering this problem naturally depends on languages $\mathcal{L}$ and $\mathcal{L}'$ used for defining views and queries. In this work, we will use the following result:

Theorem 4.3 ([10])**.** *Answering certain answers under both sound and exact view assumptions has* coNP*-complete data complexity for regular path views and queries.*

Remark 4.4. *It actually comes from the proof of Theorem 4.3 that computing certain answers under both assumptions has* coNP*-complete data complexity for regular path views and queries even when the view instances given as input are assumed to be view images.*

As a side result, we also show that computing certain answers under the sound view assumption has coNP data complexity for regular path queries and any class of path views $\mathcal{L}$. In order to be constructive, the proof of Proposition 4.5 below only requires the emptiness problem of the intersection of a language in $\mathcal{L}$ with a regular language to be decidable.

Proposition 4.5. *Let* $\mathbf{V}$ *be any path view and Q be a regular path query. Then* $cert^{sound}_{Q,\mathbf{V}}$ *can be evaluated with* coNP *data complexity.*

Proof. Let $\mathbf{V}$ be a any path view, and Q be a regular path query over some alphabet σ. We prove that $\text{cert}^{\text{sound}}_{\text{Q},\mathbf{V}}$ can be evaluated with coNP data complexity by reducing it to the case of regular path views. Let $A = \langle S, \delta, q_0, F\rangle$ be a deterministic minimal automaton for $L(\text{Q})$. For all $V \in \mathbf{V}$, we define $\tilde{V}$ with

$$L(\tilde{V}) = \{w \in \sigma^* \mid \exists w' \in L(V), \delta(\cdot, w) = \delta(\cdot, w')\}$$

and we simply define $\tilde{\mathbf{V}}$ as

$$\tilde{\mathbf{V}} = \{\tilde{V} \mid V \in \mathbf{V}\}$$

Remark now that $\tilde{\mathbf{V}}$ is a regular path view. Let E be a view instance for $\mathbf{V}$. We define $\tilde{\text{E}}$ as a copy of E where each V relation is replaced by $\tilde{V}$. Hence, $\tilde{\text{E}}$ is a view instance for $\tilde{\mathbf{V}}$. We now show that:

$$\text{cert}^{\text{sound}}_{\text{Q},\mathbf{V}}(\text{E}) = \text{cert}^{\text{sound}}_{\text{Q},\tilde{\mathbf{V}}}(\tilde{\text{E}})$$

and thus that it can be evaluated in coNP in the size of $\tilde{\text{E}}$, which is also the size of E.

- Assume that $(u, v) \in \text{cert}^{\text{sound}}_{\text{Q},\tilde{\mathbf{V}}}(\tilde{\text{E}})$. Hence, for all D such that $\tilde{\mathbf{V}}(\text{D}) \supseteq \tilde{\text{E}}$, there exists a path π from u to v such that $\lambda(\pi) \in L(\text{Q})$. Let D be a database such that $\mathbf{V}(\text{D}) \supseteq \text{E}$. Remark that, for all $V \in \mathbf{V}$, $L(V) \subseteq L(\tilde{V})$. Hence, $\tilde{\mathbf{V}}(\text{D}) \supseteq \tilde{\text{E}}$. Hence, there exists a path π in D from u to v such that $\lambda(\pi) \in L(\text{Q})$, which means that $(u, v) \in \text{cert}^{\text{sound}}_{\text{Q},\mathbf{V}}(\text{E})$.

- Conversely, assume that $(u, v) \notin \text{cert}^{\text{sound}}_{\text{Q},\tilde{\mathbf{V}}}(\tilde{\text{E}})$. Hence, there exists a database D such that $\tilde{\mathbf{V}}(\text{D}) \supseteq \tilde{\text{E}}$, but no path from u to v in D satisfies Q. From D, we build a database D' as follows:
  - Start with D' as a copy of D.
  - For all $V \in \mathbf{V}$, for all $(x, y) \in \text{E}$, if $(x, y) \in V$, then $(x, y) \in \tilde{V}$ in $\tilde{\text{E}}$. We pick a path π in D' from x to y of label w' such that $w' \in L(\tilde{V})$. Hence, there exists $w \in L(V)$ such that $\delta(\cdot, w') = \delta(\cdot, w)$. Then, we add in D' a simple path from x to y using only fresh nodes of label w. Hence $(x, y) \in V(\text{D}')$.

Remark then that $\mathbf{V}(\mathrm{D}') \supseteq \mathrm{E}$. Let π' be a path from u to v in D'. Then π' is of the form $\pi' = \pi_1\mu_1\pi_2 \ldots \pi_{n-1}\mu_{n-1}\pi_n$, where each π_i is a path that was originally in D and each μ_i is a new path using only fresh nodes. Then, for each μ_i, there exists a path ρ_i in D with the same starting and ending nodes and such that $\delta(\cdot, \lambda(\mu_i)) = \delta(\cdot, \lambda(\rho_i))$. Hence, we can define a path π of D as $\pi = \pi_1\rho_1\pi_2 \ldots \pi_{n-1}\rho_{n-1}\pi_n$. Hence, $\delta(\cdot, \lambda(\pi')) = \delta(\cdot, \lambda(\pi))$.

Since $(u, v) \notin \mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\tilde{\mathbf{V}}}(\tilde{\mathrm{E}})$, then $\delta(q_0, \lambda(\pi)) \notin F$. Hence, $\delta(q_0, \lambda(\pi')) \notin F$, which proves that $(u, v) \notin \mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E})$.

□

4.2.2 Inverting view images

In this section, we assume furthermore that the view instance E that we are given is a view image. This means that there exists a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$. The question here is: can we find such a database D?

The following lemma proves that, when $\mathbf{V}$ is a regular path view, then for any view image E, there exists an antecedent database D whose size is polynomial in $|\mathrm{E}|$.

Lemma 4.6. *Let $\boldsymbol{V}$ be an RPQ view from σ to τ. Let E be a view instance. If $E = \boldsymbol{V}(D)$ for some D then $E = \boldsymbol{V}(D')$, for some D' of size quadratic in $|E|$.*

Proof. Let $\mathbf{V}$ and E be as in the statement of the lemma. We show that if there exists D such that $\mathrm{E} = \mathbf{V}(\mathrm{D})$ then there exists a new database D' of size $O(|\mathrm{E}|^2)$ such that $\mathbf{V}(\mathrm{D}') = \mathbf{V}(\mathrm{D})$. D' is obtained from D in several steps. First D is "normalized", without altering its view, so that nodes not occurring in E appear in only one path linking two nodes of E. The normalized D turns out to consist of a constant number of disjoint paths between each pair of nodes of E (where the constant only depends on the size of the view automaton). Then a Ramsey argument is used to show that these paths can be "cut" without changing the view. The resulting database D' thus consists of a constant number of paths of constant length between each pair of nodes of E. The size of D' is therefore $O(|\mathrm{E}|^2)$. We now formalize this argument.

Assume that there exists a database D such that $\mathrm{E} = \mathbf{V}(\mathrm{D})$. We prove the lemma by constructing a new database D' such that $\mathbf{V}(\mathrm{D}') = \mathbf{V}(\mathrm{D})$, with $|\mathrm{D}'| = O(|\mathrm{E}|^2)$.

Let $A = \langle S_{\mathbf{V}}, \delta_{\mathbf{V}}, q^0_{\mathbf{V}}, F_{\mathbf{V}} \rangle$ be the product automaton of all the deterministic minimal automata of all the regular expressions of the RPQs in $\mathbf{V}$. Let $N(\mathbf{V})$ be the number of states of A, i.e $|S_{\mathbf{V}}|$.

In what follows, for $w \in \sigma^*$, $\delta_{\mathbf{V}}(\cdot, w)$ denotes the function from $S_{\mathbf{V}}$ to $S_{\mathbf{V}}$ sending q to p such that there is a run of A on w starting in state q and arriving in state p.

We say that a path π from u to v in a database D' is $\mathbf{V}$-minimal if u, v are elements of $\mathbf{V}(\mathrm{D}')$ and no other nodes of π are in the domain of $\mathbf{V}(\mathrm{D}')$.

We first build a database D_1 such that :

- $\mathbf{V}(\mathrm{D}_1) = \mathbf{V}(\mathrm{D})$;

- each node of D_1 is in a $\mathbf{V}$-minimal path and no two $\mathbf{V}$-minimal paths in D_1 intersect;
- the number of $\mathbf{V}$-minimal paths in D_1 is bounded by $|\mathbf{V}(D)|^2 \cdot N(\mathbf{V})^{N(\mathbf{V})}$.

D_1 is constructed as follows: All elements of $\mathbf{V}(D)$ are elements of D_1. Moreover, for each function $f : S_{\mathbf{V}} \to S_{\mathbf{V}}$ and each pair (x, y) of elements of $\mathbf{V}(D)$, if there exists a $\mathbf{V}$-minimal path π from x to y in D and such that $f = \delta_{\mathbf{V}}(\cdot, \lambda(\pi))$, then we add to D_1 a copy of π that uses only fresh, non-repeating nodes, except for x and y. Figure 4.1 illustrates the main idea of this construction.

It is now easy to check that D_1 has the desired properties. The second bullet holds by construction. Clearly the number of $f : S_{\mathbf{V}} \to S_{\mathbf{V}}$ is bounded by $N(\mathbf{V})^{N(\mathbf{V})}$ hence the third bullet holds. It remains to check that $\mathbf{V}(D_1) = \mathbf{V}(D)$. There is an obvious canonical homomorphism sending D_1 to D. Hence $\mathbf{V}(D_1) \subseteq \mathbf{V}(D)$. For the converse direction, consider a path π witnessing the fact that $(u, v) \in \mathbf{V}(D)$. Decompose π into $\mathbf{V}$-minimal paths. By construction, each of these $\mathbf{V}$-minimal paths can be simulated in D_1. Hence $(u, v) \in \mathbf{V}(D_1)$.

From D_1 we construct the desired D' by replacing each $\mathbf{V}$-minimal path of D_1 by another one whose length is bounded by a constant r and without affecting the view image. Altogether D' will have a size bounded by $r \cdot |\mathbf{V}(D)|^2 \cdot N(\mathbf{V})^{N(\mathbf{V})}$, hence polynomial in $|\mathbf{V}(D)|$ as desired.

Let r be the Ramsey's number that guarantees the existence of a monochromatic 3-clique in an r-clique using $N(\mathbf{V})^{N(\mathbf{V})} \cdot 2^{N(V)^{N(V)}}$ colors.

Consider a $\mathbf{V}$-minimal path $\pi = xa_0x_1a_1 \dots x_m a_m y$ in D_1 such that $m > r$. For $1 \leq s < t \leq m$ we denote by $\pi_{s \to t}$ the subpath of π that starts at position s and ends at position t, that is $\pi_{s \to t} = x_s a_s x_{s+1} a_{s+1} \dots a_{t-1} x_t$.

To each pair of nodes (x_i, x_j) in π with $i < j$, we attribute the color (f_{ij}, Δ_{ij}) where:

$$\begin{aligned} f_{ij} &= \delta_{\mathbf{V}}(\cdot, \lambda(\pi_{i \to j})) \\ \Delta_{ij} &= \{f : S_{\mathbf{V}} \to S_{\mathbf{V}} \mid \exists \alpha,\ i < \alpha < j \text{ and} \\ &\qquad\qquad f = \delta_{\mathbf{V}}(\cdot, \lambda(\pi_{i \to \alpha}))\} \end{aligned}$$

Then, by our choice of r, we know that there exist $i < j < k$ such that $f_{ij} = f_{jk} = f_{ik}$ and $\Delta_{ij} = \Delta_{jk} = \Delta_{ik}$. Let π' be the path constructed from π by replacing the subpath $\pi_{i \to k}$ by $\pi_{j \to k}$.

Let D_2 be the database constructed from D_1 by replacing π by π'. We now prove that $\mathbf{V}(D_2) = \mathbf{V}(D_1)$. As D_2 still has all the properties of D_1 listed above, by repeating this operation until all $\mathbf{V}$-minimal paths have length less than r we eventually get the desired database D'.

Let $(u, v) \in \mathbf{V}(D_1)$ as witnessed by a path μ in D_1. Then μ neither starts nor ends in an internal node of π as internal nodes do not appear in $\mathbf{V}(D_1)$. Hence either μ does not use π or it uses all of it. In the former case, μ witnesses the fact that $(u, v) \in \mathbf{V}(D_2)$. In the latter, notice that $f_{ik} = f_{jk}$ implies that $\lambda_{\mathbf{V}}(\cdot, \lambda(\pi)) = \lambda_{\mathbf{V}}(\cdot, \lambda(\pi'))$, hence replacing π by π' in μ witnesses the fact that $(u, v) \in \mathbf{V}(D_2)$. Altogether we have shown that $\mathbf{V}(D_1) \subseteq \mathbf{V}(D_2)$.

Suppose now that $(u, v) \in \mathbf{V}(\mathrm{D}_2)$ as witnessed by a path μ in D_2. If μ does not go through x_j (i.e. x_j is not an internal node of μ), it is also a path in D_1 and $(u, v) \in \mathbf{V}(\mathrm{D}_1)$. If μ goes through x_j but does not end between x_j and x_k we can also conclude that $(u, v) \in \mathbf{V}(\mathrm{D}_1)$ using the fact that $f_{ik} = f_{jk}$. It remains to consider the case when μ ends with $x_j a_j \dots a_{\beta-1} x_\beta$ for some β with $j < \beta < k$ (in particular $v = x_\beta$). As $\Delta_{ij} = \Delta_{jk}$ there exists α with $i < \alpha < j$ such that $\delta_{\mathbf{V}}(\cdot, \lambda(\pi_{i\to\alpha})) = \delta_{\mathbf{V}}(\cdot, \lambda(\pi_{j\to\beta}))$. From this we can construct a path μ' in D_1 replacing in μ the segment $x_j a_j \dots a_{\beta-1} x_\beta$ by $x_i a_i \dots a_{\alpha-1} x_\alpha$, witnessing the fact that $(u, x_\alpha) \in \mathbf{V}(\mathrm{D}_1)$, a contradiction as x_α is not an element of $\mathbf{V}(\mathrm{D}_1)$. Altogether we have proved that $\mathbf{V}(\mathrm{D}_2) \subseteq \mathbf{V}(\mathrm{D}_1)$. Hence, $\mathbf{V}(\mathrm{D}_2) = \mathbf{V}(\mathrm{D}_1) = \mathbf{V}(\mathrm{D})$.

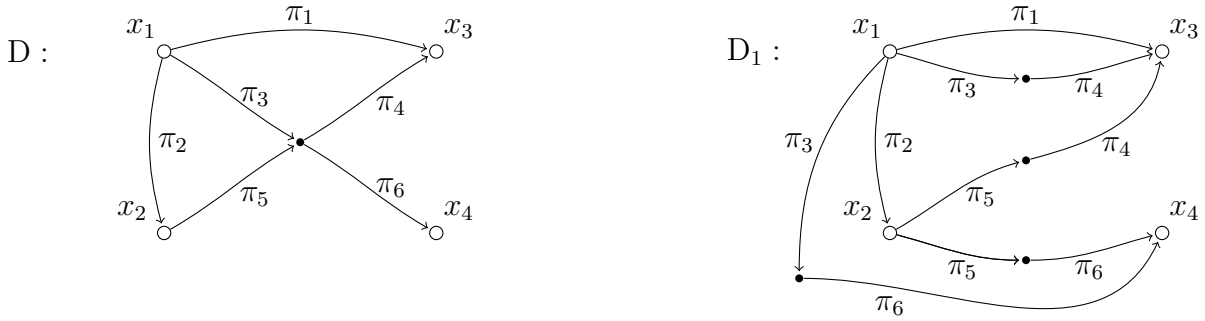


Figure 4.1: Illustration of the transformation from D to D_1 in Lemma 4.6. Nodes are colored white or black depending on whether they appear in $\mathbf{V}(\mathrm{D})$ or not.

□

This reduces the problem of searching for a view inverse to a polynomial search, and thus provides an NP algorithm. We will see in the next section that it is unlikely that we can do better, as Lemma 4.7 proves that deciding if a view instance is a view image is NP hard. Assuming that we can find a view inverse in polynomial time would however provide the following polynomial time decision procedure for this problem: for a given view instance E, we look for a view inverse with a timeout corresponding to the polynomial bound. If the algorithm produces a candidate view inverse D, then it remains to check that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$, which successfully proves that E is a view image and can be done in polynomial time in the size of D, and thus in the size of E. Otherwise, if the algorithm fails or times out, we can conclude that E is not a view image.

With a more intricate pumping argument, Lemma 4.6 actually extends to conjunctive regular path views. Thus, for any conjunctive regular path view $\mathbf{V}$ and view image E, there exists a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$ and D is of polynomial size in $|\mathrm{E}|$. However, we will see again in the next section that deciding if a view instance of a context-free view is a view image is undecidable. Thus, by applying the same reasoning, we can conclude that there exists no computable bound on the size of a view inverse for a context-free view image.

4.2.3 Checking view images

As we have discussed previously, not all view instances are view images. In this section, we consider the problem of checking whether a view instance is indeed a view image. In other words, given a view $\mathbf{V}$ and a view instance E, can we decide whether there exists a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$? This question is closely related to that of Section 4.2.2. The main difference is that the input is not a view image but a view instance, and thus can be inconsistent, and that we do not need to produce a view inverse, but only checks that it exists.

PROBLEM :	VIEW CHECKING FOR VIEW LANGUAGE $\mathcal{L}$
INPUT :	A view $\mathbf{V}$ in $\mathcal{L}$, a view instance E
QUESTION :	Does there exist D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$?

In this section, we will mainly consider data complexity, by assuming that $\mathbf{V}$ is fixed. We start by showing that the problem has NP-complete data complexity for regular path views.

Lemma 4.7. *The complement of certain answers for regular path queries and views under the exact view assumption reduces to view checking for regular path views.*

Proof. Let $\mathbf{V}$ be a regular path view and Q be a regular path query over a schema σ. From $\mathbf{V}$ and Q, we build a view $\mathbf{V}'$ over $\sigma' = \sigma \cup \{s\} \cup \{e\}$ as follows:

$$\mathbf{V}' = \mathbf{V} \cup \{\langle s\rangle \cdot \mathrm{Q} \cdot \langle e\rangle\} \cup \{\langle s\rangle\} \cup \{\langle e\rangle\}$$

Let E be a view instance for $\mathbf{V}$ and (x, y) be a pair of elements of E. We now build a view instance E' for $\mathbf{V}'$ as follows:

- E' contains all the nodes and edges of E;
- E' contains two fresh nodes *start* and *end*;
- $\langle s\rangle(\mathit{start}, x)$ holds in E';
- $\langle e\rangle(y, \mathit{end})$ holds in E';
- the interpretation of $\langle s\rangle \cdot \mathrm{Q} \cdot \langle e\rangle$ is empty in E'.

We now prove that $(x, y) \notin \mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E})$ if and only if E' is a view image for $\mathbf{V}'$, which will conclude the proof.

Assume that $(x, y) \notin \mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E})$. Then there exists a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$ and $(x, y) \notin \mathrm{Q}(\mathrm{D})$. Consider the database D' that is a copy of D except that it contains two additional nodes *start* and *end* such that $s(\mathit{start}, x)$ and $e(y, \mathit{end})$ hold in D'. Then it is immediate that $\mathbf{V}'(\mathrm{D}') = \mathrm{E}'$, which implies that E' is a view image for $\mathbf{V}'$.

Assume that E' is a view image for $\mathbf{V}'$. Then there exists a database D' such that $\mathbf{V}'(\mathrm{D}') = \mathrm{E}'$. Remark that $(x, y) \notin \mathrm{Q}(\mathrm{D}')$, otherwise $\langle s\rangle \cdot \mathrm{Q} \cdot \langle e\rangle(\mathit{start}, \mathit{end})$ would hold in E'. We define D as the projection of D' over schema σ (that is, we remove the s and e edges from D). Then, it is immediate to check that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$, which proves that $(x, y) \notin \mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E})$. □

We know from Theorem 4.3 that the certain answers have coNP-complete data complexity. From this, Lemma 4.6 and Lemma 4.7, we easily deduce that view checking for regular path views is NP-complete.

Corollary 4.8. *View checking for regular path views has* NP*-complete data complexity.*

Remark 4.9. *The proof of Lemma 4.7 is actually quite modular. It does not use specific properties of regular path views. For the proof to apply, we only need the considered view language $\mathcal{L}$ to have the following properties:*

- *$\mathcal{L}$ contains atomic queries;*
- *$\mathcal{L}$ is closed under concatenation;*
- *Queries in $\mathcal{L}$ are preserved by extension of the schema, and addition of edges using the new symbols.*

While the first two properties are usually given for any interesting class of path queries, the last property is not that natural. It is true of all classes considered in Chapter 3. However a query that cares about distances between nodes, while disregarding their labels would not, for instance, have this property.

We deduce from this remark that the bound for regular path views also holds for conjunctive regular path views:

Corollary 4.10. *View checking for conjunctive regular path views has* NP*-complete data complexity.*

We conclude this section by showing that the problem becomes undecidable for context-free path queries. A simple argument shows that it has undecidable combined complexity:

Lemma 4.11. *View checking for context-free path views has undecidable combined complexity.*

Proof. We prove this by reduction from the universality problem for context-free languages. Let L be a context-free language over a schema σ. Let \$ be a fresh symbol that does not appear in σ. Let $\mathbf{V} = \{V_1, V_2\}$, where $V_1 = \langle\$\rangle \cdot \langle L\rangle \cdot \langle\$\rangle$ and $V_2 = \langle\$\rangle \cdot \langle\sigma^*\rangle \cdot \langle\$\rangle$. Finally, let E be the view instance that contains a single pair (x, y) in V_2 and no pair in V_1. Then there exists D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$ if and only if L is not universal over σ.

- Assume that there exists a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$. Then there exists a path π from x to y such that $\lambda(\pi) \in L(V_2)$. Hence there exists $w \in \sigma^*$ such that $\lambda(\pi) = \$ \cdot w \cdot \$$. However, $\lambda(\pi) \notin L(V_1)$. Hence $w \notin L$ and L is not universal.
- Conversely, assume that L is not universal. Then there exists $w \in \sigma^*$ such that $w \notin L$. Then it is easy to check that the database D consisting of a simple path labeled by $\$ \cdot w \cdot \$$ satisfies $\mathbf{V}(\mathrm{D}) = \mathrm{E}$.

□

A more intricate argument shows that undecidability already holds for a *fixed* view definition $\mathbf{V}$.

Lemma 4.12. *View checking for context-free path views has undecidable data complexity.*

Proof. Let $\sigma = \{(, ;,), a, b, \$, 1\}$. Let $\boldsymbol{\sigma}$ be a copy of σ with fresh symbols. For $\alpha \in \sigma$, we denote by $\boldsymbol{\alpha}$ the corresponding symbol in $\boldsymbol{\sigma}$. For w a word, $\tilde{w}$ denote the word corresponding to w read from right to left. $\mathbf{V}$ consists of queries that reveal each symbol in σ, that is, for all $\alpha \in \sigma$, $\mathbf{V}$ contains a query $V_\alpha = \langle \alpha \rangle$. Additionally, $\mathbf{V}$ contains the queries V_u, V_v, V'_u, V'_v, V_g and V_c defined by the following equations:

$$L(V_u) = \left\{ \begin{array}{c} \$ \cdot w \cdot \$ \cdot (i_1;v_1;u_1) \dots (i_n;v_n;u_n) \cdot \$ \mid \\ w, u_k, v_k \in \{\boldsymbol{a}, \boldsymbol{b}\}^*, i_k \in \mathbf{1}^*, u_1 \cdot \ldots \cdot u_n = \tilde{w} \end{array} \right\}$$

$$L(V_v) = \left\{ \begin{array}{c} \$ \cdot w \cdot \$ \cdot (i_1;v_1;u_1) \dots (i_n;v_n;u_n) \cdot \$ \mid \\ w, u_k, v_k \in \{\boldsymbol{a}, \boldsymbol{b}\}^*, i_k \in \mathbf{1}^*, v_1 \cdot \ldots \cdot v_n = \tilde{w} \end{array} \right\}$$

$$L(V'_u) = \left\{ \begin{array}{c} \$ \cdot w \cdot \$ \cdot (i_1;v_1;u_1) \dots (i_n;v_n;u_n) \cdot \$ \mid \\ w, u_k, v_k \in \{\boldsymbol{a}, \boldsymbol{b}\}^*, i_k \in \mathbf{1}^*, u_1 \cdot \ldots \cdot u_n \neq \tilde{w} \end{array} \right\}$$

$$L(V'_u) = \left\{ \begin{array}{c} \$ \cdot w \cdot \$ \cdot (i_1;v_1;u_1) \dots (i_n;v_n;u_n) \cdot \$ \mid \\ w, u_k, v_k \in \{\boldsymbol{a}, \boldsymbol{b}\}^*, i_k \in \mathbf{1}^*, v_1 \cdot \ldots \cdot v_n \neq \tilde{w} \end{array} \right\}$$

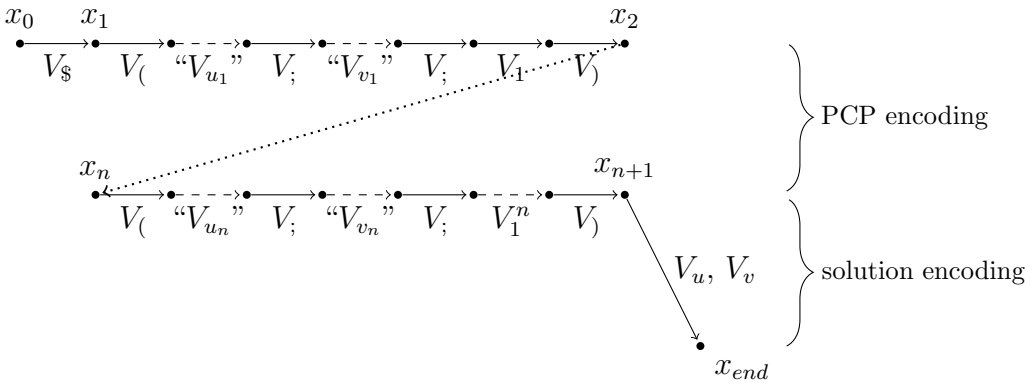
$$L(V_g) = \left\{ \begin{array}{c} \$ \cdot (u_1; v_1; i_1) \cdot \ldots \cdot (u_n; v_n; i_n) \cdot \$ \cdot \boldsymbol{\sigma}^* \cdot \$ \cdot \boldsymbol{\sigma}^* \cdot (i';v';u') \mid \\ u_k, v_k \in \{a, b\}^*, i_k \in 1^*, u', v' \in \{\boldsymbol{a}, \boldsymbol{b}\}^*, i' \in \mathbf{1}^*, i' > i_n \end{array} \right\}$$

$$L(V_c) = \left\{ \begin{array}{c} \$ \cdot (u_1; v_1; i_1) \cdot \ldots \cdot (u_n; v_n; i_n) \cdot \$ \cdot \boldsymbol{\sigma}^* \cdot \$ \cdot \boldsymbol{\sigma}^* \cdot (i';v';u') \mid \\ u_k, v_k \in \{a, b\}^*, i_k \in 1^*, u', v' \in \{\boldsymbol{a}, \boldsymbol{b}\}^*, i' \in \mathbf{1}^*, \\ \exists k, i_k = \varphi(i'), u_k \neq \varphi(\tilde{u'}) \text{ or } v_k \neq \varphi(\tilde{v'}) \end{array} \right\}$$

where φ is the function that maps each symbol in $\boldsymbol{\sigma}$ to the corresponding symbol in σ.

One can check that all these languages are actually context-free languages.

We now prove that, given a view instance E for this specific view $\mathbf{V}$, it is undecidable whether there exists a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$. We prove this by reduction from the Post Correspondence Problem (PCP). Let $(u_i, v_i, i)_{0<i\leq n}$ be an instance of PCP over $\{a, b\}$, where the third argument explicitly gives the index of each pair. We build the following view instance E:



We now show that there exists D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$ if and only if the PCP instance is satisfiable. Intuitively, E consists of two parts. The first part, from x_0 to x_{n+1} is the encoding of the PCP instance. It uses letters from σ that are all revealed by the view. All tuples are simply enumerated in the natural order, where the ith tuple is encoded between x_i and x_{i+1}. The dashed arrows V_{u_i} and V_{v_i} represent the correct succession of V_a and V_b that naturally encode u_i and v_i, whereas the V_1^i part is the unary encoding of i, the index of the tuple. The second part of the instance states the existence of a solution for this instance, and uses "hidden" letters from $\boldsymbol{\sigma}$. V_u and V_v states that there exists a solution, and the fact that all other views are empty checks that this solution is correct.

- Assume that there exists a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$. Then there exists a path π from x_{n+1} to x_{end} such that $\lambda(\pi) \in L(V_u)$. Hence, this path is of the form $\$ \cdot w \cdot \$ \cdot (i_1;v'_1;u'_1) \dots (i_m;v'_m;u'_m) \cdot \$$, where w is a word in $\boldsymbol{\sigma}^*$ and $u'_1 \dots u'_m = \tilde{w}$. Remark that is also holds that $v'_1 \dots v'_m = \tilde{w}$, otherwise $\lambda(\pi) \in V'_v$, which would imply that $(x_{n+1}, x_{end}) \in V'_v(\mathrm{D})$, and lead to a contradiction.

  Hence, $u'_1 \dots u'_m = v'_1 \dots v'_m$. It remains to show that each $(i_i;v'_i;u'_i)$ is an encoding of the mirror of some tuple in the PCP instance, which would imply a solution as $\tilde{u}'_m \dots \tilde{u}'_1 = \tilde{v}'_m \dots \tilde{v}'_1$. In other words, $u_{|i_m|} \dots u_{|i_1|} = v_{|i_m|} \dots v_{|i_1|}$.

  Assume that one of the $(i_i;v'_i;u'_i)$ is not the mirror of some tuple encoded in the first half of the instance. Remark that $|i_i| \leq n$. Otherwise, there exists a path whose label is in $L(V_g)$, which leads to a contradiction. Hence, either $u'_i \neq \tilde{u}_{|i_i|}$ or $v'_i \neq \tilde{v}_{|i_i|}$. Both cases lead to the existence of a path whose label is in $L(V_c)$, and thus to a contradiction.

- Assume that there exists a solution $i_1 \dots i_m$ to the PCP instance. Then the database D that consists of the following simple path is such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$:

$$\$(u_1;v_1;1) \dots (u_n;v_n;1^n)\mathbf{\$u_{i_1} \dots u_{i_m}\$(1^{i_m};\tilde{v}_{i_m};\tilde{u}_{i_m}) \dots (1^{i_1};\tilde{v}_{i_1};\tilde{u}_{i_1})\$}$$

  where $\mathbf{u_i}$ and $\mathbf{v_i}$ simply represent the corresponding u_i and v_i written using $\boldsymbol{a}$ and $\boldsymbol{b}$ instead of a and b.

□

4.2.4 View update

We have seen in Section 4.2.3 that checking whether a view instance is actually a view image is a hard problem. In this section, we consider a related question, formalized as the view update problem, which has been for instance considered in [26, 4, 18, 25]. In the view update setting, we start with a view image E. We are then asked if E is still a view image after some update, namely after the addition or deletion of an edge in E. It is clear that this problem is at most as hard as the view checking problem, as one could just ignore the additional information provided by the update process and just try and check the view instance E that results from it. Unfortunately, we prove in this section that it is actually just as hard for the relevant query languages.

For a view $\mathbf{V}$ and $V \in \mathbf{V}$ we define add_V as the function that maps a view instance E and a tuple of elements $\bar{x}$ of E of same arity as V to a copy of E in which the interpretation of V also contains the tuple $\bar{x}$. Similarly, we define $\text{del}_V(\text{E}, \bar{x})$ as the copy of E in which the tuple $\bar{x}$ is not in the interpretation of V.

Lemma 4.13. *There exists a regular path view $\boldsymbol{V}$ such that, given a view image E, $V \in \boldsymbol{V}$ and two nodes x, y in E, it is* NP*-complete to decide whether $\text{add}_V(E, x, y)$ is a view image. It is also* NP*-complete to decide whether $\text{del}_V(E, x, y)$ is a view image.*

Proof. Both upper bounds come directly from the results for the view checking problem. We prove the lower bounds by reduction from the complement of certain answers for regular path queries and views under the exact view assumption. This proof is very similar to the proof of Lemma 4.7, but additionally uses Remark 4.4.

Let $\mathbf{V}$ be a regular path view and Q be a regular path query over some schema σ, for which $\text{cert}^{\text{exact}}_{\text{Q},\mathbf{V}}$ has coNP-hard data complexity, even when restricted to view images.

Case 1: add_V. We define $\sigma' = \sigma \cup \{s\} \cup \{e\} \cup \{i\}$, and a view $\mathbf{V}'$ over σ' as follows:

$$\mathbf{V}' = \mathbf{V} \cup \{\langle s\rangle \cdot \text{Q} \cdot \langle e\rangle\} \cup \{\langle s\rangle\} \cup \{\langle e\rangle\} \cup \{\langle i\rangle\}$$

Let E be a view image for $\mathbf{V}$ and (x, y) be a pair of elements of E. We build a view instance E′ as follows:

- E′ contains all the nodes and edges of E;
- E′ contains two fresh nodes *start* and *end*;
- $\langle s\rangle(\mathit{start}, x)$ holds in E′;
- $\langle i\rangle(\mathit{end}, \mathit{end})$ holds in E′;
- the interpretation of $\langle e\rangle$ is empty in E′;
- the interpretation of $\langle s\rangle \cdot \text{Q} \cdot \langle e\rangle$ is empty in E′.

It is straightforward to check that E′ is actually a view image. Indeed, let D be a database such that $\mathbf{V}(\text{D}) = \text{E}$. Then, by adding two nodes *start* and *end* to D as well as the edges $\mathit{start} \xrightarrow{s} x$ and $\mathit{end} \xrightarrow{i} \mathit{end}$, we easily get a database D′ such that $\mathbf{V}'(\text{D}') = \text{E}'$.

We now claim that $(x, y) \notin \text{cert}^{\text{exact}}_{\text{Q},\mathbf{V}}(\text{E})$ if and only if $\text{add}_{\langle e\rangle}(\text{E}, y, \mathit{end})$ is a view image for $\mathbf{V}'$. The proof of this claim goes exactly as the proof for Lemma 4.7.

Case 2: del_V.

We define $\sigma' = \sigma \cup \{s\} \cup \{e\} \cup \{q\}$, and a view $\mathbf{V}'$ over σ' as follows:

$$\mathbf{V}' = \mathbf{V} \cup \{\langle s\rangle \cdot (\text{Q} \cup \langle q\rangle) \cdot \langle e\rangle\} \cup \{\langle s\rangle\} \cup \{\langle e\rangle\}$$

Let E be a view image for $\mathbf{V}$ and (x, y) be a pair of elements of E. Once again, we build a view instance E′ as follows:

- E′ contains all the nodes and edges of E;

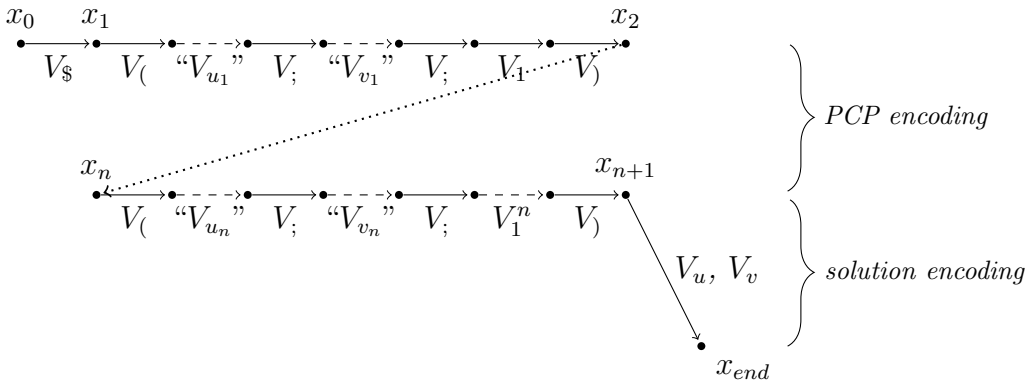
- E′ contains two fresh nodes *start* and *end*;
- $\langle s\rangle(start, x)$ holds in E′;
- $\langle e\rangle(y, end)$ holds in E′;
- $(\langle s\rangle \cdot (\mathrm{Q} \cup \langle q\rangle) \cdot \langle e\rangle)(start, end)$ holds in E′.

Once again, we can check that E′ is a view image for $\mathbf{V}'$. Indeed, let D be a database such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$. Then, by adding two nodes *start* and *end* to D as well as the edges $start \xrightarrow{s} x$, $y \xrightarrow{e} end$ and $x \xrightarrow{q} y$, we easily get a database D′ such that $\mathbf{V}'(\mathrm{D}') = \mathrm{E}'$.

We now claim that $(x, y) \notin \mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E})$ if and only if $\mathrm{del}_{(\langle s\rangle\cdot(\mathrm{Q}\cup\langle q\rangle)\cdot\langle e\rangle)}(start, end)$ is a view image for $\mathbf{V}'$, with the proof of this claim being once again the same as in Lemma 4.7. □

Corollary 4.14. *There exists a conjunctive regular path view* $\boldsymbol{V}$ *such that, given a view image E,* $V \in \boldsymbol{V}$ *and two nodes* x, y *in E, it is* NP*-complete to decide whether* $add_V(E, x, y)$ *is a view image. It is also* NP*-complete to decide whether* $del_V(E, x, y)$ *is a view image.*

Remark 4.15. *The proof of Lemma 4.12 can be adapted to show that both view update questions are undecidable for context-free views. Indeed, by keeping all the notations from that proof, recall that the hard instance had the following form:*



Consider a copy E′ of this view instance without the V_v *edge in the end. It is immediate to check that E′ is a view image as it now only requires the solution encoding to conform to the left part of the problem. Hence, the hard instance can be expressed as* $add_{V_v}(E', x_{n+1}, x_{end})$*, where E′ is a view image, which proves that view update under addition is undecidable.*

Similarly, consider a copy E″ which contains a V_g *edge. This edge allows the solution encoding to contain arbitrary subwords that were not given in the problem description, and thus E″ is a view image. Once again, removing this edge yields the hard instance, which proves that view update under deletion is undecidable.*

4.3 View-based query determinacy

4.3.1 Definition

In Section 4.2, we have worked with materialized view problems, that is, problems where a view instance or a view image was given as input, along with its view definition. On the contrary, the problem that we discuss in this section and which is at the heart of our work is a *static analysis problem*. This means that we are not given a specific database D or a specific view instance E to work with. Instead, we are only given a view $\mathbf{V}$ and a query Q and we are asked to deduce facts and properties about their behavior *offline*, that is, independently of a specific database.

Determinacy. Query determinacy is a notion that specifies when a view $\mathbf{V}$ always provide enough information to answer a query Q. When that is the case, we say that $\mathbf{V}$ determines Q, and write $\mathbf{V} \twoheadrightarrow \mathrm{Q}$. This means that, for any database D, Q(D) can always be computed by looking only at $\mathrm{E} = \mathbf{V}(\mathrm{D})$. This naturally implies that Q(D) only depends on $\mathbf{V}(\mathrm{D})$ and not on the particular D that yields $\mathbf{V}(\mathrm{D})$. In other words, for any database D' such that $\mathbf{V}(\mathrm{D}') = \mathbf{V}(\mathrm{D})$, we also have $\mathrm{Q}(\mathrm{D}) = \mathrm{Q}(\mathrm{D}')$. Thus, we can formally define determinacy as follows:

Definition 4.16 (Determinacy)**.** *We say that a view $\boldsymbol{V}$ determines a query Q if :*

$$\forall D, D', \quad \boldsymbol{V}(D) = \boldsymbol{V}(D') \;\Rightarrow\; Q(D) = Q(D')$$

Remark that determinacy implies a functional dependency from view images to query answers: there exists a function f from view images to sets of answers such that, for any database D, $f(\mathbf{V}(\mathrm{D})) = \mathrm{Q}(\mathrm{D})$. We call f the *function induced by Q using $\boldsymbol{V}$*. Provided that Q and $\mathbf{V}$ are computable functions, then f is also computable. Indeed, for a given view image E, a very naive algorithm for computing f consists in enumerating all possible databases until we find a database D such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$, and then evaluating Q on D. This process is guaranteed to terminate, as E is assumed to be a view image, and thus has a view inverse.

Rewritings. Let $\mathbf{V}$ be a view from a schema σ to a schema τ, and Q be a query over σ such that $\mathbf{V}$ determines Q. Let f be the function induced by Q using $\mathbf{V}$. A *rewriting* of Q using $\mathbf{V}$ is a query R over τ that coincides with f on view images: for all view images E, $\mathrm{R}(\mathrm{E}) = f(\mathrm{E})$, or alternatively, for all databases D, $\mathrm{R}(\mathbf{V}(\mathrm{D})) = f(\mathbf{V}(\mathrm{D})) = \mathrm{Q}(\mathrm{D})$. The intuition behind this definition is that, in order to evaluate Q on a database D, it is enough to evaluate R on $\mathbf{V}(\mathrm{D})$. Thus we have rewritten Q, a query over σ, as a new query R over τ.

Remark that the only requirement on R is its evaluation on view images. As we have seen previously, not all databases over τ (that is, not all view instances) are view images. This means that two queries R_1 and R_2 might take different values on view instances that are not view images, and still be rewritings of Q using $\mathbf{V}$, provided that they agree with f on view images. Thus, there are possibly many rewritings of Q using $\mathbf{V}$. This is

to put in perspective with the fact that f is unique. This is illustrated on the following example.

Example 4.17. *Consider* $\boldsymbol{V} = \{V_1, V_2\}$ *with* $V_1 = \langle \sigma^3 \rangle$ *and* $V_2 = \langle \sigma^4 \rangle$ *testing for the existence of a path of length 3 and 4, respectively. Let* $Q = \langle \sigma^5 \rangle$ *be the query testing for the existence of a path of length 5.*

It turns out that $\boldsymbol{V}$ *determines* Q *[3]. This is not immediate to see but the first-order query* $R = \langle \varphi \rangle$ *is a rewriting of* Q *using* $\boldsymbol{V}$*, with* φ *defined as follows:*

$$\varphi(x, y) = \exists z \ (V_2(x, z) \wedge \forall z' \ (V_1(z', z) \Rightarrow V_2(z', y)))$$

Let us prove that, for all database D, $Q(D) = R(\boldsymbol{V}(D))$*:*

- *Assume that* $(x, y) \in Q(D)$*. Then there exists a path* π *from* x *to* y *in* D *such that* $|\pi| = 5$*. Let* z *be the fourth successor of* x *along this path. Then* $(x, z) \in V_2(D)$*, and there is an edge from* z *to* y*. Let* z' *be any node such that* $(z', z) \in V_1(D)$*. Then there is a path of length 3 from* z' *to* z*, which implies that there is a path of length 4 from* z' *to* y*. Thus* $(z', y) \in V_2(D)$*, which proves that* $\varphi(x, y)$ *holds in* $\boldsymbol{V}(D)$*.*

- *Conversely, assume that* $(x, y) \in R(\boldsymbol{V}(D))$*. There exists* z *in* $\boldsymbol{V}(D)$ *such that* $V_2(x, z)$ *holds in* $\boldsymbol{V}(D)$ *and for all* z' *in* $\boldsymbol{V}(D)$*,* $V_1(z', z)$ *implies* $V_2(z', y)$*. Since* $V_2(x, z)$ *holds in* $\boldsymbol{V}(D)$*, then there exists a path of length 4 from* x *to* z *in* D*. Let* z' *be the successor of* x *along this path. Thus* $(z', z) \in V_1(D)$*, which implies that* z' *is a node of* $\boldsymbol{V}(D)$ *and that* $V_1(z', z)$ *holds in* $\boldsymbol{V}(D)$*. Thus,* $V_2(z', y)$ *holds in* $\boldsymbol{V}(D)$*. Finally, there is a path of length 1 from* x *to* z' *and a path of length 4 from* z' *to* y *in* D*. Altogether, this implies that there is a path of length 5 from* x *to* y *in* D*, which proves that* $(x, y) \in Q(D)$*.*

Now, consider the first-order query $R' = \langle \varphi' \rangle$*, with* φ' *defined as follows:*

$$\varphi'(x, y) = \varphi(x, y) \wedge \exists t \ V_1(x, t)$$

It's easy to adapt the previous proof to show that R' *is also a rewriting of* Q *using* $\boldsymbol{V}$*. Indeed, if* E *is a view image, as soon as* $V_2(x, z)$ *holds for two nodes* x *and* z *of* E*, then there exists a node* t *such that* $V_1(x, t)$ *also holds, which satisfies the additional condition in* φ'*. This proves that* R *and* R' *agree on view images, which should be expected, as they are both rewritings of* Q *using* $\boldsymbol{V}$*. However,* R *and* R' *can disagree on view instances that are not view images, and they actually do. Consider the instance* E *that consists of a single edge* $V_2(x, y)$*. Then* $(x, y) \in R(E)$ *but* $R'(E) = \emptyset$*.*

These definitions give rise to the *determinacy* and *rewriting problems*. The determinacy problem is the problem of deciding, given a view $\mathbf{V}$ and a query Q, whether $\mathbf{V}$ determines Q. Its complexity depends of course on the class of query languages from which $\mathbf{V}$ and Q are taken:

PROBLEM :	DETERMINACY FOR QUERY AND VIEW LANGUAGES $\mathcal{L}$ AND $\mathcal{L}'$
INPUT :	A query Q in $\mathcal{L}$, a view $\mathbf{V}$ in $\mathcal{L}'$
QUESTION :	Does $\mathbf{V}$ determine Q?

The rewriting problem is the problem of finding a rewriting of Q using $\mathbf{V}$, for a query Q and a view $\mathbf{V}$ such that $\mathbf{V}$ determines Q. This problem has several different aspects. At its core, it is just asking to design an algorithm that maps view instances E to sets of tuples of E and implements the function f induced by Q using $\mathbf{V}$ on view images: if E is a view image, then the algorithm should return $f(\mathrm{E})$. Then, provided that we know how to construct such an algorithm, we are looking for one that has the lowest possible (data) complexity. Finally, we ask the question of whether this algorithm can be expressed in a good query languages, namely one that enjoys low evaluation complexity.

All these questions, about both the determinacy and rewriting problems, will occupy the rest of this work. In the remainder of this chapter, we give some side results about both problems before moving on, in Chapter 5 and Chapter 6, to our main contributions.

4.3.2 Determinacy problem

As we have seen in Chapter 2, the determinacy problem is only solved in very specific cases. More importantly, its decidability status is still open for regular path queries and views. In this section, we show that it is undecidable for conjunctive regular path views and for context-free path views, and that this is already the case even when the query is assumed to be a regular path query. These two undecidability results emphasize that regular path queries and views are likely to be the crucial question for determining the decidability frontier.

Proposition 4.18. *Given a context-free path view $\boldsymbol{V}$ and a regular path query Q, it is undecidable whether $\boldsymbol{V}$ determines Q.*

Proof. We prove this by reduction from the universality problem for context-free languages. Let L be a context-free language over some alphabet σ. Let \$ be a fresh symbol that does not appear in σ. Let $\mathbf{V} = \{V\}$ where V is defined by $L(V) = \$ \cdot L \cdot \$$. Let Q be defined by $L(\mathrm{Q}) = \$ \cdot \sigma^* \cdot \$$. Then $\mathbf{V}$ determines Q if and only if L is universal over σ.

- Assume that L is universal. Then $\mathrm{Q} = V$ and it is easy to check that $R = V$ is a rewriting of Q using $\mathbf{V}$.
- Conversely, assume that L is not universal. Then there exists $w \in \sigma^*$ such that $w \notin L$. Consider the database D consisting of a simple path labeled by $\$ \cdot w \cdot \$$, and the empty database D′. Then $\mathbf{V}(\mathrm{D}) = \emptyset = \mathbf{V}(\mathrm{D}')$, but Q(D) contains the first and last node of the path, whereas Q(D′) is empty. Hence, $\mathbf{V}$ does not determine Q.

□

Proposition 4.19. *Given a conjunctive regular path view $\boldsymbol{V}$ and a regular path query Q, it is undecidable whether $\boldsymbol{V}$ determines Q.*

Proof. We prove this by reduction from the word problem for graph databases.

PROBLEM :	WORD PROBLEM FOR GRAPH DATABASES
INPUT :	A list of pairs $(u_i, v_i)_{0<i\leq n}$, a pair (u, v), where u, v and u_i, v_i, for every i, are words over σ, viewed as RPQs
QUESTION :	Is the following statement true? For every graph database D, if $\forall i, u_i(\mathrm{D}) = v_i(\mathrm{D})$, then $u(\mathrm{D}) = v(\mathrm{D})$

A straightforward reduction from the word problem for finite semigroups shows:

Lemma 4.20. *The word problem for graph databases is undecidable.*

Proof. We prove this by reduction from the word problem for finite semigroups. This problem has the same input as the word problem for graph databases but asks whether for all semigroup S and all homomorphism h from σ^* to S such that $h(u_i) = h(v_i)$ for all i, it is the case that $h(u) = h(v)$.

We now prove that any input is accepting for the word problem for finite semigroups if and only if it is accepting for the word problem for graph databases.

1. Assume that the input is accepting for the word problem for finite semigroups. Let D be a graph database such that for all i, $u_i(\mathrm{D}) = v_i(\mathrm{D})$. From D, we compute the semigroup S_D and the homomorphism $h : \sigma^* \to S_\mathrm{D}$ as follows:

   - The elements of S_D are the set of pairs $w(\mathrm{D})$ for all $w \in \sigma^*$. As D is finite S_D is finite.
   - Let x and y be two elements of S_D. Let $u, v \in \sigma^*$ such that $x = u(\mathrm{D})$ and $y = v(\mathrm{D})$. Then $x \cdot y$ is defined as $u \cdot v(\mathrm{D})$. It is easy to check that this operation is associative and well defined (i.e. does not depend on the specific choice of u and v).
   - For all $\alpha \in \sigma$ we set $h(\alpha) = \alpha(\mathrm{D})$. Hence for all $u \in \sigma^*$ we have $h(u) = u(D)$.

   By construction we therefore have for all i, $h(u_i) = h(v_i)$. Hence, $h(u) = h(v)$, which implies that $u(\mathrm{D}) = v(\mathrm{D})$.

2. Assume that the input is accepting for the word problem for graph databases. Let S be a finite semigroup, and h an homomorphism from σ^* to S, such that, for all i, $h(u_i) = h(v_i)$. From S and h, we define the graph database D_h as follows:

   - The sets of nodes of D_h is $h(\sigma^+) \cup \{\varepsilon\}$. This set is finite since $h(\sigma^+)$ is a subset of S.
   - Let x and y be two nodes of D_h. Then there is an edge α from x to y if either $x = \varepsilon$ and $y = h(\alpha)$ or $x \neq \varepsilon$ and $x \cdot h(\alpha) = y$.

   Assume that $(x, y) \in u_i(\mathrm{D}_h)$. Then either $x = \varepsilon$, hence $y = h(u_i) = h(v_i)$ and $(x, y) \in v_i(\mathrm{D}_h)$, or $x \cdot h(u_i) = y$, which implies that $x \cdot h(v_i) = y$ and $(x, y) \in v_i(\mathrm{D}_h)$. Hence, $u_i(\mathrm{D}_h) = v_i(\mathrm{D}_h)$ for all i and therefore $u(\mathrm{D}_h) = v(\mathrm{D}_h)$. Hence, $(\varepsilon, h(u)) \in v(\mathrm{D}_h)$, which implies that there is a path v from ε to $h(u)$ and thus that $h(u) = h(v)$.

□

Let $(u_i, v_i)_{0<i\leq n}$ and (u, v) be an input for the word problem. Let σ' be a copy of σ using only fresh symbols. For each $\alpha \in \sigma$, we use α' to denote the corresponding symbol in σ'. We define the following query and view:

- Q is the RPQ defined by $L(\mathrm{Q}) = \{u, v'\}$ where v' is a copy of v using symbols of σ'.
- For all $\alpha \in \sigma$, V_α is a query of the view defined by the RPQ $L_\alpha = \{\alpha, \alpha'\}$.
- For all i, V_i is also a query of the view defined by the RPQ $L_i = \{u_i, v_i'\}$, where v_i' is a copy of v_i using symbols of σ'.
- For all $\alpha, \beta \in \sigma$, $T_{\alpha,\beta}$ is a query of the view defined by the CRPQ: $\alpha(x, y) \wedge \exists z, t\ \beta'(z, t)$.
- For all $\alpha, \beta \in \sigma$, $T'_{\alpha,\beta}$ is a query of the view defined by the CRPQ: $\alpha'(x, y) \wedge \exists z, t\ \beta(z, t)$.

We now prove that $\mathbf{V} = \{V_\alpha, V_i, T_{\alpha,\beta}, T'_{\alpha,\beta} \mid \alpha, \beta \in \sigma, 0 < i \leq n\}$ determines Q if and only if the input is accepting for the word problem for graph databases.

1. Assume that the input is accepting for the word problem for graph databases. Let D and D′ be two graph databases such that $\mathbf{V}(\mathrm{D}) = \mathbf{V}(\mathrm{D}')$. Consider first the case where D uses symbols from both σ and σ', then $T_{\alpha,\beta}$ and $T'_{\alpha,\beta}$ reveal D entirely, which implies that $\mathrm{D} = \mathrm{D}'$, and thus $\mathrm{Q}(\mathrm{D}) = \mathrm{Q}(\mathrm{D}')$. Similarly, if both D and D′ use only symbols from σ, then V_α reveals D entirely ensuring that $\mathrm{D} = \mathrm{D}'$. It remains to consider the case where D only uses symbols from σ and D′ only uses symbols from σ'. Notice that, since $V_\alpha(\mathrm{D}) = V_\alpha(\mathrm{D}')$, then D and D′ are isomorphic (by renaming each α to α').

   Let $(x, y) \in u_i(\mathrm{D})$. Hence, $(x, y) \in V_i(\mathrm{D})$, which implies that $(x, y) \in V_i(\mathrm{D}')$, and finally that $(x, y) \in v_i'(\mathrm{D}')$. By isomorphism $(x, y) \in v_i(\mathrm{D})$. Similarly, we can show that $(x, y) \in v_i(\mathrm{D})$ implies $(x, y) \in u_i(\mathrm{D})$. Hence, $u(\mathrm{D}) = v(\mathrm{D})$. Let $(x, y) \in \mathrm{Q}(\mathrm{D})$. Then, $(x, y) \in u(\mathrm{D})$, which implies that $(x, y) \in v'(\mathrm{D}')$, and thus that $(x, y) \in \mathrm{Q}(\mathrm{D}')$. A similar reasoning also gives the converse, and we can conclude that $\mathbf{V}$ determines Q.

2. Assume that $\mathbf{V}$ determines Q. Let D be a graph database over σ that satisfies the condition for the word problem. Let D′ be the copy of D given by renaming the symbols in σ by the corresponding symbols in σ'. Remark now that $\mathbf{V}(\mathrm{D}) = \mathbf{V}(\mathrm{D}')$. Indeed, $V_\alpha(\mathrm{D}) = V_\alpha(\mathrm{D}')$ is given by the fact that D′ is a copy of D over σ'. $V_i(\mathrm{D}) = V_i(\mathrm{D}')$ is given by the fact that D satisfies the condition for the word problem. Finally, $T_{\alpha,\beta}(\mathrm{D}) = T_{\alpha,\beta}(\mathrm{D}') = T'_{\alpha,\beta}(\mathrm{D}) = T'_{\alpha,\beta}(\mathrm{D}') = \emptyset$ comes from the fact that D (resp. D′) uses only symbols from σ (resp. σ').

   Since $\mathbf{V}$ determines Q, this implies that $\mathrm{Q}(\mathrm{D}) = \mathrm{Q}(\mathrm{D}')$. Let $(x, y) \in u(\mathrm{D})$. Then $(x, y) \in \mathrm{Q}(\mathrm{D})$, which implies that $(x, y) \in \mathrm{Q}(\mathrm{D}')$. Hence, $(x, y) \in v'(\mathrm{D}')$, and since

D′ is a copy of D, this yields $(x, y) \in v(\mathrm{D})$. A similar reasoning also gives the converse, and we can conclude that the input is accepting for the word problem for graph databases.

□

4.3.3 Rewriting problem

In this section, we consider the rewriting problem in the following way: we assume given a view $\mathbf{V}$ and a query Q such that $\mathbf{V}$ determines Q, and we are interested in the data complexity of computing $\mathrm{Q}(\mathrm{D})$ from a view image $\mathbf{V}(\mathrm{D})$. A first, immediate, result is that computing $\mathrm{Q}(\mathrm{D})$ from $\mathbf{V}(\mathrm{D})$ reduces to computing the certain answers to Q on $\mathbf{V}(\mathrm{D})$ under the exact view assumption. Therefore, any algorithm for computing certain answers actually defines a rewriting of Q using $\mathbf{V}$.

Lemma 4.21. *Let Q be a query and $\boldsymbol{V}$ be a view such that $\boldsymbol{V}$ determines Q. Then, for all database D, $Q(D) = cert^{exact}_{Q,\boldsymbol{V}}(\boldsymbol{V}(D))$.*

Proof. Let D be any database. Recall that:

$$\mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathbf{V}(\mathrm{D})) = \bigcap_{\mathrm{D}' \mid \mathbf{V}(\mathrm{D}')=\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D}')$$

Since $\mathbf{V} \twoheadrightarrow \mathrm{Q}$, then for all D′ such that $\mathbf{V}(\mathrm{D}') = \mathbf{V}(\mathrm{D})$, $\mathrm{Q}(\mathrm{D}') = \mathrm{Q}(\mathrm{D})$. Thus, we have:

$$\begin{aligned}\mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathbf{V}(\mathrm{D})) &= \bigcap_{\mathrm{D}' \mid \mathbf{V}(\mathrm{D}')=\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D}') \\ &= \bigcap_{\mathrm{D}' \mid \mathbf{V}(\mathrm{D}')=\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D}) \\ &= \mathrm{Q}(\mathrm{D})\end{aligned}$$

□

This has two important corollaries for any query language $\mathcal{L}$ and view language $\mathcal{L}'$: first, any query language capable of expressing certain answers of queries in $\mathcal{L}$ for views in $\mathcal{L}'$ is a suitable language for expressing rewritings of queries in $\mathcal{L}$ using views in $\mathcal{L}'$. Second, the evaluating such a rewriting reduces to the problem of evaluating certain answers for $\mathcal{L}$ and $\mathcal{L}'$, which yields our first complexity upper bounds.

In the case of regular path queries and views, Theorem 4.3 thus proves that there exist rewritings that have coNP data complexity. However, thanks to Lemma 4.6, we know that for a given view $\mathbf{V}$ and view image E, we can find a database D of size polynomial in E such that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$. This yiels another algorithm for evaluating rewritings which has NP data complexity: for a given view image E, we guess a database D of polynomial size and check that $\mathbf{V}(\mathrm{D}) = \mathrm{E}$, which can be done in polynomial time since regular path queries have polynomial time evaluation. Finally, we evaluate the query Q on D. This returns the desired answers, since we assumed that $\mathbf{V} \twoheadrightarrow \mathrm{Q}$.

Corollary 4.22. *Let Q be a regular path query and $\boldsymbol{V}$ be a regular path view such that $\boldsymbol{V}$ determines Q. Then there exists a rewriting of Q using $\boldsymbol{V}$ that can be evaluated with* NP *data complexity, and a rewriting of Q using $\boldsymbol{V}$ that can be evaluated with* coNP *data complexity.*

Preliminary conclusions. At this point, these are the best known bounds. However, they come from reductions from view manipulation problems. As we have seen in Section 4.2, operations on view instances all seem to have high complexity. The task we are trying to tackle here is likely to be much simpler: we do not need specific semantic properties when our rewriting is evaluated on view instances that are not view images (unlike the certain answers algorithm which provides the coNP bound) nor do we need to actually provide a view inverse along with the answers to the query (as does the algorithm that provides the NP bound).

These remarks are the starting point of our work. Our goal now is to find more suitable algorithms both for the determinacy and rewriting problems, from which to derive better complexity bounds at least in restricted cases. In Chapter 5, we come back to the rewriting problem for regular path queries and views, with the added assumption that the rewriting is monotone. We show that, in this case, the rewriting can always be expressed as a Datalog query, and thus enjoys polynomial time evaluation. In Chapter 6, we restrict our attention to single path queries and union of single path views. We show that, for a given union of single path view, we can decide determinacy for almost all single path queries, and provide first-order rewritings for those determined queries. These rewriting, once again, enjoy polynomial time evaluation.

Chapter 5

Monotone rewritings of regular path queries

In this chapter, we will mainly work with regular path queries and views. The goal here is to push the known complexity bounds for rewritings from Section 4.3.3, albeit in a more restricted case. Indeed, we will work here with *monotone determinacy*, a stronger and decidable form of determinacy that we define in Section 5.1. In Section 5.2, we recall the link between the well-known constraint satisfaction problem and the monotone rewritings for regular path queries and views. In the final section of this chapter, we show how this link can be used to compute monotone rewritings that have polynomial time data complexity.

5.1 Monotone determinacy

We have seen in Chapter 3 that many crucial query languages were monotone. Since regular path queries are also monotone, it is natural to wonder whether these languages can be used to express the rewritings of a regular path query using a regular path view, assuming the view determines the query. However, Example 4.17 from Chapter 4 can be used to show that this is not possible in general. Indeed, the function induced by the query using the views in this example is not monotone, therefore no monotone query language can express it. This is illustrated on the example below:

Example 5.1. *Consider again the view* $\boldsymbol{V} = \{V_1, V_2\}$ *with* $V_1 = \langle \sigma^3 \rangle$ *and* $V_2 = \langle \sigma^4 \rangle$*, and the query* $Q = \langle \sigma^5 \rangle$*. We already know from Example 4.17 that* $\boldsymbol{V} \twoheadrightarrow Q$*. However, Figure 5.1 shows that the function induced by* Q *using* $\boldsymbol{V}$ *is not monotone. Indeed, it is a simple matter to verify that* D *and* D' *are such that* $\boldsymbol{V}(D) \subseteq \boldsymbol{V}(D')$*, but* $(x_0, x_5) \in Q(D)$*, whereas* $(x_0, x_5) \notin Q(D')$*.*

Following Example 5.1, we are thus interested in defining a stronger notion of determinacy which additionally ensures that there always exist a monotone rewriting of the query using the view:

Definition 5.2 (Monotone determinacy)**.** *We say that a view* $\boldsymbol{V}$ *determines a query* Q *in a monotone way if* $\boldsymbol{V}$ *determines* Q *and the function induced by* Q *using* $\boldsymbol{V}$ *is monotone.*

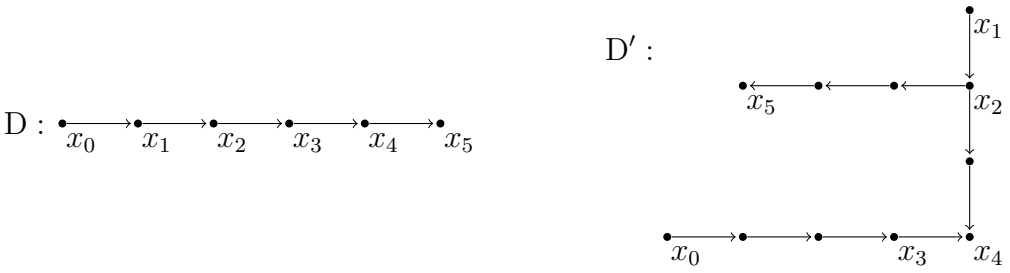


Figure 5.1: Illustration for Example 5.1

This definition can actually be rewritten in a way that closely resembles Definition 4.16.

Lemma 5.3 (Monotone determinacy)**.** *Let $\boldsymbol{V}$ be a view and Q be a query. Then $\boldsymbol{V}$ determines Q in a monotone way if and only if:*

$$\forall D, D', \quad \boldsymbol{V}(D) \subseteq \boldsymbol{V}(D') \;\Rightarrow\; Q(D) \subseteq Q(D')$$

Proof. Assume that $\mathbf{V}$ determines Q and that f, the function induced by Q using $\mathbf{V}$ is monotone. Let D and D′ be two databases such that $\mathbf{V}(\mathrm{D}) \subseteq \mathbf{V}(\mathrm{D}')$. Then $f(\mathbf{V}(\mathrm{D})) \subseteq f(\mathbf{V}(\mathrm{D}'))$, thus $\mathrm{Q}(\mathrm{D}) \subseteq \mathrm{Q}(\mathrm{D}')$.

Conversely, assume that for all databases D and D′ such that $\mathbf{V}(\mathrm{D}) \subseteq \mathbf{V}(\mathrm{D}')$, we have $\mathrm{Q}(\mathrm{D}) \subseteq \mathrm{Q}(\mathrm{D}')$. In particular, if $\mathrm{D} = \mathrm{D}'$, we have $\mathrm{Q}(\mathrm{D}) \subseteq \mathrm{Q}(\mathrm{D}')$ and $\mathrm{Q}(\mathrm{D}') \subseteq \mathrm{Q}(\mathrm{D})$. Hence, $\mathrm{Q}(\mathrm{D}) = \mathrm{Q}(\mathrm{D}')$, and $\mathbf{V}$ determines Q. Let f be the function induced by Q using $\mathbf{V}$. Then $f(\mathbf{V}(\mathrm{D}) = \mathrm{Q}(\mathrm{D}) \subseteq \mathrm{Q}(\mathrm{D}') = f(\mathbf{V}(\mathrm{D}'))$, from which we conclude that f is monotone. □

It appears that monotone determinacy coincides with the notion of *losslessness under the sound view assumption* defined in [12].

Definition 5.4 ([12])**.** *A view $\boldsymbol{V}$ is said to be lossless with respect to a query Q under the sound view assumption if, for all databases D, $Q(D) = cert^{sound}_{Q,\boldsymbol{V}}(\boldsymbol{V}(D))$.*

Transposed in our setting, this definition exactly says that a view is lossless with respect to a query under the sound view assumption if and only if the certain answers query under the sound view assumption is a rewriting of the query using the view. Since this rewriting is also monotone, it remains to prove that, whenever a view determines a query in a monotone way, then certain answers under the sound view assumption are a rewriting. This is to be put in perspective with Lemma 4.21 in the non-monotone case.

Lemma 5.5. *Let $\boldsymbol{V}$ be a view and Q be a query such that $\boldsymbol{V}$ determines Q in a monotone way. Then, for all database D, $Q(D) = cert^{sound}_{Q,\boldsymbol{V}}(\boldsymbol{V}(D))$.*

Proof. Since $\mathbf{V}$ determines Q, we already know from Lemma 4.21 that $\mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathbf{V}(\mathrm{D}))$ is a rewriting of Q using $\mathbf{V}$. Now, remark that, for all database D:

$$\begin{aligned}\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}(\mathbf{V}(\mathrm{D})) &= \bigcap_{\mathrm{D}' \mid \mathbf{V}(\mathrm{D}')\subseteq\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D}') \\ &= \bigcap_{\mathrm{D}' \mid \mathbf{V}(\mathrm{D}')=\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D}') \bigcap \bigcap_{\mathrm{D}' \mid \mathbf{V}(\mathrm{D}')\subsetneq\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D}')\end{aligned}$$

Since $\mathbf{V}$ determines Q in a monotone way, we deduce that the right term contains the left term, thus:

$$\begin{aligned}\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}(\mathbf{V}(\mathrm{D})) &= \bigcap_{\mathrm{D}' \mid \mathbf{V}(\mathrm{D}')\subseteq\mathbf{V}(\mathrm{D})} \mathrm{Q}(\mathrm{D}') \\ &= \mathrm{cert}^{\mathrm{exact}}_{\mathrm{Q},\mathbf{V}}(\mathbf{V}(\mathrm{D}))\end{aligned}$$

from which we deduce that $\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}$ is a rewriting of Q using $\mathbf{V}$. □

Thus, a view $\mathbf{V}$ determines a query Q in a monotone way if and only if $\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}$ is a rewriting of Q using $\mathbf{V}$, that is, if and only if $\mathbf{V}$ is lossless with respect to Q under the sound view assumption. Moreover, it was shown in [12] that losslessness is decidable for regular path queries and views, from which we immediately deduce that monotone determinacy is decidable for regular path queries and views.

Theorem 5.6 ([12])**.** *Checking whether a regular path view $\boldsymbol{V}$ is lossless with respect to a regular path query Q under the sound view assumption is* ExpSpace*-complete.*

Corollary 5.7. *The monotone determinacy problem for regular path queries and views is* ExpSpace*-complete.*

Remark also that in the proof of Proposition 4.18, when the view determines the query, it is also the case that the rewriting is monotone. From this, we deduce that monotone determinacy is also undecidable for context-free path views and regular path queries.

Corollary 5.8. *The monotone determinacy problem for regular path queries and context-free path views is undecidable.*

Example 5.1 shows that, given a regular path query and a regular path view such that the view determines the query, it is not always the case that a monotone rewriting can be found. A similar phenomenon happens with conjunctive views and queries, as shown in [37], where it can happen that a conjunctive view determines a conjunctive query with no rewriting being monotone. However, it was also shown there that when the view determines the query in a monotone way, a rewriting can always be expressed as a conjunctive query. Thus, it is natural to wonder if the same holds true for regular path queries and views. In other words, is the non-monotonicity of the function induced

by the query using the views the only property preventing the existence of a rewriting expressible as a regular path query?

It turns out that this is not the case. Actually, it was shown in [13] that, given a regular path query and a regular path view, it is 2ExpSpace-complete to decide whether there exists a rewriting of the query using the view that can be expressed as a regular path query. Hence, a simple complexity argument shows that the existence of an RPQ rewriting cannot coincide with the monotone determinacy of the query using the view, which can be decided in ExpSpace. We give here a concrete example witnessing this fact:

Example 5.9. *Let $\sigma = \{a, b, c\}$. Let Q and $\boldsymbol{V}$ be defined as follows:*

- $Q = \langle ab^*a \mid ac^*a \rangle$
- $\boldsymbol{V} = \{V_1, V_2, V_3\}$ *with*
  - $V_1 = \langle ab^* \rangle$
  - $V_2 = \langle ac^* \rangle$
  - $V_3 = \langle b^*a \mid c^*a \rangle$

One can verify that $\boldsymbol{V}$ determines Q as witnessed by the following rewriting $R = \langle \varphi \rangle$:

$$\varphi(x, y) = \exists z \; V_1(x, z) \wedge V_2(x, z) \wedge V_3(z, y)$$

That R is a rewriting is illustrated in Figure 5.2. Consider the database D of Figure 5.2 which is a typical database such that $(x, y) \in Q(D)$. The choice of z witnessing $(x, y) \in R(\boldsymbol{V}(D))$ is then immediate. Conversely, consider the database D' of Figure 5.2. It is a typical database such that $(x, y) \in R(\boldsymbol{V}(D))$. The top path shows that $(x, y) \in Q(D)$.

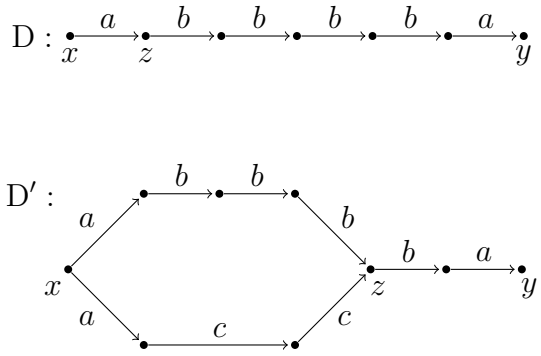


Figure 5.2: Databases D and D′ for Example 5.9.

Since R is monotone, $\boldsymbol{V}$ determines Q in a monotone way. It can also be shown that no RPQ rewriting exists.

In the previous example we have exhibited a rewriting expressible as a conjunctive regular path query. However the following example suggests that conjunctive regular path queries are not expressive enough to cover all monotone rewritings of regular path queries using regular path views.

Example 5.10. *Let $\sigma = \{a\}$. Let $\boldsymbol{V}$ and Q be defined as follows:*

- $Q = \langle a(a^6)^* \mid aa(a^6)^* \rangle$ *(words of length 1 or 2 modulo 6)*
- $\boldsymbol{V} = \{V_1, V_2\}$ *with*
  - $V_1 = \langle a \mid aa \rangle$ *(words of length 1 or 2)*
  - $V_2 = \langle aa \mid aaa \rangle$ *(words of length 2 or 3)*

It can be verified that $\boldsymbol{V}$ determines Q in a monotone way as witnessed by the rewriting $R = \langle \varphi \rangle$, with:

$$\varphi(x, y) = \exists z \; V_1(x, z) \wedge T^*(z, y)$$

where $T(x, y)$ is defined as:

$$\exists z_1, z_2 \; V_1(x, z_1) \wedge V_2(x, z_1) \wedge V_1(z_1, z_2) \wedge V_2(z_1, z_2) \wedge V_1(z_2, y) \wedge V_2(z_2, y)$$

The query T is such that if $T(x, y)$ holds in $\boldsymbol{V}(D)$, then in D the nodes x and y are either linked by a path of length 6 or by both a path of length 5 and a path of length 7. This fact can be checked by a simple case analysis. One such case is illustrated in Figure 5.3. In this case there is no path of length 6 in D, but the top path has length 5, and the path starting with the bottom segment and then the last two top segments has length 7.

From this, a simple induction shows that if $T^*(x, y)$ holds in $\boldsymbol{V}(D)$, then in D the nodes x and y are either linked by a path of length 0 modulo 6, or by both a path of length 1 modulo 6 and a path of length 5 modulo 6.

Assume now that $R(x, y)$ holds in $\boldsymbol{V}(D)$. Then in D there exists a z such that x is at distance 1 or 2 from z, and such that $T^*(z, y)$ holds in $\boldsymbol{V}(D)$. Assume first that z and y are at distance 0 modulo 6 in D. In this case, regardless of the distance between x and z, $Q(x, y)$ holds in D. Otherwise, in D there exist both a path of length 1 modulo 6 and a path of length 5 modulo 6 from z to y. Therefore, if x and z are at distance 1, the first path from z to y yields a path of length 2 modulo 6 and, if x and z are at distance 2, the second path from z to y yields a path of length 1 modulo 6, see Figure 5.4.

Conversely, it is easy to check that $R(x, y)$ holds in $\boldsymbol{V}(D)$ whenever $Q(x, y)$ holds in D. This follows from the fact that $T(x, y)$ holds in $\boldsymbol{V}(D)$ for all x and y that are at distance 6 in D.

Notice that R is monotone. A tedious combinatorial argument can show that R cannot be expressed as a conjunctive regular path query.

As a side result before moving on to the core of this chapter, we remark that Proposition 4.5 together with Lemma 5.5 prove that, whenever a path view $\mathbf{V}$ determines a regular path query Q in a monotone way, then there exists a rewriting of Q using $\mathbf{V}$ that can be evaluated with coNP data complexity.

Corollary 5.11. *Let Q be a regular path query and $\boldsymbol{V}$ be any path view such that $\boldsymbol{V}$ determines Q in a monotone way. Then there exists a rewriting of Q using $\boldsymbol{V}$ that can be evaluated with* coNP *data complexity.*

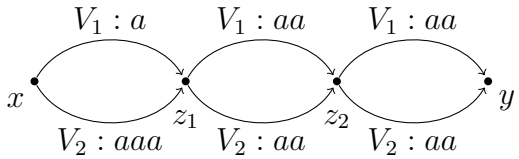


Figure 5.3: Example 5.10: An arbitrary database D whose view satisfies $T(x, y)$. Each arrow of the form $V_i : w$ from a node u to a node v should be understood as a path from u to v whose label is w which witnesses $(u, v) \in V_i(D)$.

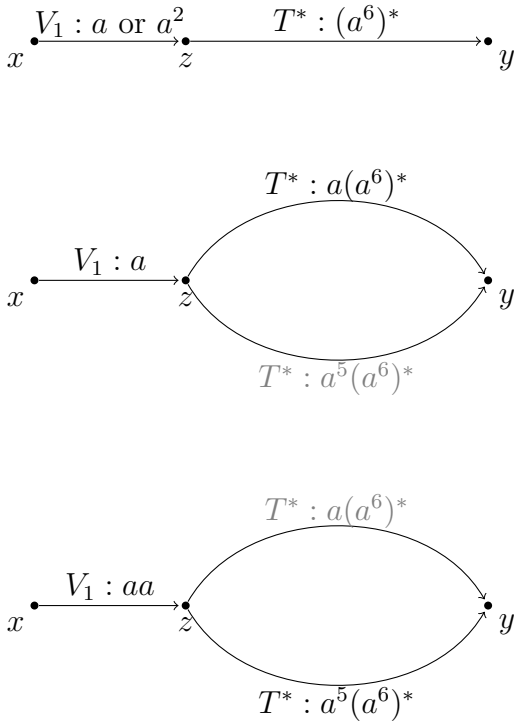


Figure 5.4: The three cases of Example 5.10. The parts that are not used for Q are shaded out.

5.2 Constraint satisfaction and certain answers

We already know from Lemma 5.5 that when a view $\mathbf{V}$ determines a query Q, then certain answers under the sound view assumption are a rewriting of the query using the view. It turns out that, when both $\mathbf{V}$ and Q are defined by regular path queries, then $\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}$ can be expressed as the complement of a constraint satisfaction problem, which gives us another way of expressing rewritings. We start this section by recalling the definition of constraint satisfaction problems, and showing how to adapt it to fit our setting.

5.2.1 Constraint satisfaction problems

There are several ways to define constraint satisfaction problems (CSP). In this work, we adopt the homomorphism point of view, and refer the reader to [44, 31] for more details about how to build links between the different presentations. A constraint satisfaction problem $\mathrm{CSP}(\mathcal{A}, \mathcal{B})$ is defined by two classes of structures $\mathcal{A}$ and $\mathcal{B}$, and asks, for a given $A \in \mathcal{A}$ and $B \in \mathcal{B}$ whether there exists a homomorphism from A to B. Here, we will work mostly with so-called non-uniform CSPs: CSPs for which $\mathcal{B}$ is reduced to a single structure T. Additionally, we will consider the case where $\mathcal{A}$ covers all the structures on the schema of T, and thus we will simply write $\mathrm{CSP}(T)$ for $\mathrm{CSP}(\mathcal{A}, \{T\})$. T is called the *template* of the CSP.

PROBLEM :	CONSTRAINT SATISFACTION PROBLEM ON TEMPLATE T
INPUT :	A structure A
QUESTION :	Is there a homomorphism that maps A to T?

As was done in [11], we slightly tweak the definition of CSPs in a way that turns them into a binary query language. First, we extend the signature of the template T with two additional unary predicates called *source* and *target*. Then, we define $\langle \mathrm{CSP}(T) \rangle$ as the query that maps a database D to the set of pairs (x, y) of nodes of D such that there exists a homomorphism from D to T that sends x to a source node of T and y to a target node of T:

PROBLEM :	QUERY EVALUATION FOR CSPS
INPUT :	A database D, a pair (x, y) of elements of D,
	A template T with source and target relations
QUESTION :	Is there a homomorphism h that maps D to T
	such that source$(h(x))$ and target$(h(y))$ hold in T?

As mentioned in the introduction of this section, we will actually work with the complement of the constraint satisfaction problem. For a given template T, we denote by $\neg\mathrm{CSP}(T)$ the complement problem of $\mathrm{CSP}(T)$. Thus, a structure A is accepting for $\neg\mathrm{CSP}(T)$ if it is not accepting for $\mathrm{CSP}(T)$, in other words, if there exists no homomorphism that maps A to T. Likewise, $\langle \neg\mathrm{CSP}(T) \rangle$ is the query that selects all pairs of nodes (x, y) of a given database D that are not selected by $\langle \mathrm{CSP}(T) \rangle$: $(x, y) \in \langle \neg\mathrm{CSP}(T) \rangle$ if and only if $(x, y) \notin \langle \mathrm{CSP}(T) \rangle$.

5.2.2 From certain answers to CSP

It was shown in [11] that, for a regular path view $\mathbf{V}$ and a regular path query Q, $\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}$ can actually be expressed as the complement of a constraint satisfaction problem. This is formalized in the proposition below:

Proposition 5.12 ([11]). *Let $\boldsymbol{V}$ be a regular path view from σ to τ and Q be a regular path query over σ. Then there exists a template $T_{Q,\boldsymbol{V}}$ over $\tau \cup \{source, target\}$ such that, for all view instance E and all pairs (x, y) of elements of E, $(x, y) \in cert^{sound}_{Q,\boldsymbol{V}}(E)$ if and only if $(x, y) \in \langle \neg CSP(T_{Q,\boldsymbol{V}}) \rangle(E)$.*

Together with Lemma 5.5, Proposition 5.12 gives us the following corollary:

Corollary 5.13. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query such that $\boldsymbol{V}$ determines Q in a monotone way. Then there exists a template $T_{Q,\boldsymbol{V}}$ such that $\langle \neg CSP(T_{Q,\boldsymbol{V}})\rangle$ is a rewriting of Q using $\boldsymbol{V}$.*

It actually comes from the proof of Proposition 5.12 that the template $T_{\mathrm{Q},\mathbf{V}}$ can be effectively computed from Q and $\mathbf{V}$. It is well known that $\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})$ reduces to the evaluation of a formula of the existental monadic second-order logic. Thus, Corollary 5.13 implies that if a regular path view determines a regular path query in a monotone way, then there always exists a rewriting that can be expressed in the universal monadic second-order logic. Moreover, this rewriting can effectively be computed from the query and the view.

However, we know from Theorem 4.3 that certain answers under the sound view assumption, and therefore $\langle \neg \mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$, have coNP-complete data complexity. We now remark that this remains true even when we assume that the view $\mathbf{V}$ determines the query Q in a monotone way:

Proposition 5.14. *There exist a regular path view $\boldsymbol{V}$ and a regular path query Q such that $\boldsymbol{V}$ determines Q in a monotone way and it is* coNP*-complete to decide, given a view instance E and nodes (u, v) in E, whether $(u, v) \in cert^{sound}_{Q,\boldsymbol{V}}(E)$.*

Proof. The upper bound immediately comes from Theorem 4.3.

We now prove the lower bound with a simple reduction from the case where we do not assume that $\mathbf{V}$ determines Q in a monotone way.

Let Q be a regular path query and $\mathbf{V}$ be a regular path view. We define

$$\mathbf{V}' = \mathbf{V} \cup \{\mathrm{Q}\}$$

Remark now that $\mathbf{V}'$ is a regular path view and that $\mathbf{V}'$ determines Q in a monotone way, as witnessed by the rewriting R = Q.

Let E be a view instance for $\mathbf{V}$. Then the structure E′, defined as E extended with the empty relation for Q, is a view instance for $\mathbf{V}'$. It just remains to remark that $\mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}}(\mathrm{E}) = \mathrm{cert}^{\mathrm{sound}}_{\mathrm{Q},\mathbf{V}'}(\mathrm{E}')$. Indeed, for all databases D, $\mathbf{V}(\mathrm{D}) \supseteq \mathrm{E}$ if and only if $\mathbf{V}'(\mathrm{D}) \supseteq \mathrm{E}'$, which concludes the proof. □

Besides its technical simplicity, the proof of Proposition 5.14 highlights the idea that knowing that a view determines a query actually conveys no information if we are working with view instances and not view images. Indeed, the view can go as far as plainly containing the query, it does not make computing certain answers easier if the view instances given as input omit this information. Note however that Remark 4.4 no longer applies in this case, as the view instance E cannot be assumed to be a view image for the view $\mathbf{V}'$ built in the proof.

From the result of Proposition 5.14, we know that the rewriting that we considered in this section has coNP-complete data complexity. Thus, none of them are suitable for our purpose of finding a rewriting with polynomial data complexity. However, this is

most likely due to their behavior on view instances that are not view images, as we know from Corollary 4.22 that rewritings are likely not to be coNP-hard in general (unless coNP is included in NP), and by definition have to agree on view images.

In the next section, we show how to approximate the CSP rewriting in a way that allows for polynomial time evaluation, while still retaining its behavior on view images.

5.3 Computing the rewriting

We know from Corollary 5.13 that, when a regular path view $\mathbf{V}$ determines a regular path query Q, then we can compute a template $T_{\mathrm{Q},\mathbf{V}}$ such that $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$ is a rewriting of Q using $\mathbf{V}$. However, as explained in Section 5.2, there exist some queries and views for which this particular rewriting has coNP-complete data complexity. In this section, we show how to move from this rewriting to another rewriting which enjoys polynomial time data complexity. This is done in three steps: first, we present the *local consistency game*, which provides a tool to approximate $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$ and express these approximations as Datalog queries. Then, we compute one such approximation that turns out to coincide with $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$ when evaluated on the view image of simple path databases. Finally, we show that it is enough for this approximation to be exact on the view of simple paths to actually cover all view images. Thus, we get a rewriting of Q using $\mathbf{V}$ with polynomial time data complexity that is moreover expressible as a Datalog query.

5.3.1 Datalog and the local consistency game

The local consistency game is based on the homomorphism problem. Given two structures A and B, two players dispute with limited tools the existence (or lack thereof) of a homomorphism from A to B. More precisely, Player 1 tries to prove that there is no homomorphism from A to B. To this end, she selects on each turn a set of nodes of A, and asks Player 2 to provide a homomorphism from the substructure of A induced by this set to B, with the added constraint that this homomorphism must agree with the homomorphism from previous turn on common nodes. If Player 2 cannot provide such a homomorphism, she loses the game. Otherwise, the game continues, and Player 2 wins if she can play forever. The game takes its *local* name from the two parameters l and k that rule Player 1 moves: on any turn, she should not select more than k nodes, and should not keep more than ℓ node from last turn.

In this work, we use the local consistency game to approximate $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$. The source structure A will be the view instance E on which we are evaluating $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$, and the target structure B will be fixed as $T_{\mathrm{Q},\mathbf{V}}$. We also distinguish two nodes x and y in E, which represent the nodes for which we want to evaluate whether they belong to $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathrm{E})$. We now give a formal definition of the game adaptated to our setting:

Definition 5.15 ((ℓ,k) local consistency game)**.** *Let ℓ, k be two integers, with $\ell \leq k$, let E be a view instance and x, y be two nodes of E. The (ℓ,k)-game on $(E, T_{Q,\mathbf{V}}, x, y)$ is played by two players as follows:*

- *The game begins with $A_0 = \emptyset$ and h_0 being the empty function over A_0.*

  For $i \geq 0$, round $i+1$ is defined as follows:

- *Player 1 selects a set A_{i+1} of nodes of E, with $|A_{i+1}| \leq k$ and $|A_i \cap A_{i+1}| \leq \ell$.*

- *Player 2 responds by giving a homomorphism $h_{i+1} : E[A_{i+1}] \to T_{Q,\boldsymbol{V}}$ that coincides with h_i on $A_i \cap A_{i+1}$ and such that $h_{i+1}(x)$ is a source node and $h_{i+1}(y)$ is a target node whenever x or y are in A_{i+1}.*

Player 1 wins if at any point Player 2 has no possible move. Player 2 wins if she can play forever.

Observe that if there is a homomorphism from a view instance E to $T_{\mathrm{Q},\mathbf{V}}$ sending x to a source node and y to a target node, then Player 2 has a winning strategy for the (ℓ, k)-two-player game on $(\mathrm{E}, T_{\mathrm{Q},\mathbf{V}}, x, y)$. This strategy consists in always playing the restriction of the homomorphism on the set selected by Player 1. In this sense, the set of pairs of nodes of E for which Player 1 has a winning strategy is an under-approximation of $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathrm{E})$: if Player 1 has a winning strategy for $(\mathrm{E}, T_{\mathrm{Q},\mathbf{V}}, x, y)$ then $(x, y) \in \langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathrm{E})$.

The converse inclusion does not necessarily hold. If Player 2 has a winning strategy for $(\mathrm{E}, T_{\mathrm{Q},\mathbf{V}}, x, y)$, this only means that she can always exhibit partial homomorphisms from E to $T_{\mathrm{Q},\mathbf{V}}$; this is in general not sufficient to guarantee the existence of a suitable global homomorphism. In that sense, the set of pairs for which Player 2 has a winning strategy is an over-approximation of $\langle\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathrm{E})$.

In order to approximate $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathrm{E})$, we are now interested in computing the set of pairs of nodes of E for which Player 1 has a winning strategy. It turns out that this is expressible in Datalog, as formalized by the following lemma:

Lemma 5.16 ([21, 11])**.** *Let ℓ, k be two integers, with $\ell \leq k$. Let Q be a regular path query and $\boldsymbol{V}$ be a regular path view. Then there exists a program $Q_{\ell,k}(x, y)$ in $Datalog_{\ell,k}$ such that for every view instance E, $Q_{\ell,k}(E)$ is the set of pairs (x, y) such that Player 1 has a winning strategy for the (ℓ, k)-game on $(E, T_{Q,\boldsymbol{V}}, x, y)$.*

Moreover the program in the above lemma can be effectively constructed from $T_{\mathrm{Q},\mathbf{V}}$, and therefore from Q and $\mathbf{V}$. It will be simply denoted by $Q_{\ell,k}$ when Q and $\mathbf{V}$ are clear from the context.

At this point, we know that for each pair (ℓ, k), $Q_{\ell,k}$ is an under-approximation of $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$. The contribution of the two following sections is to show that we can actually compute from Q and $\mathbf{V}$ a specific pair (ℓ, k) for which $Q_{\ell,k}$ is exact on view images: for a given view image E, $Q_{\ell,k}(\mathrm{E})$ is precisely $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathrm{E})$. Of course, $Q_{\ell,k}$ and $\langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$ still differ in general on view instances that are not view images, otherwise $Q_{\ell,k}$ would inherit CSP's coNP-hardness which is a contradiction, unless PTime = coNP.

5.3.2 The case of simple paths

Our first step is to prove that there exists suitable values of ℓ and k such that $Q_{\ell,k}$ coincides with a rewriting of Q using $\mathbf{V}$ on the view of simple path databases. This is formalized in the proposition below, whose proof is the focus of this section.

Proposition 5.17. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query. There exists ℓ such that for every simple path database D from x to y,*

$$(x, y) \in Q_{\ell,\ell+1}(\boldsymbol{V}(D)) \text{ iff } (x, y) \in \langle \neg CSP(T_{Q,\boldsymbol{V}})\rangle(\boldsymbol{V}(D)).$$

In particular if $\boldsymbol{V}$ determines Q in a monotone way,

$$(x, y) \in Q_{\ell,\ell+1}(\boldsymbol{V}(D)) \text{ iff } (x, y) \in Q(D).$$

Let $\mathbf{V}$ and Q be an RPQ view and an RPQ query, and let D be a graph database consisting of a simple path from node x to node y, that is, D is the path $\pi = x_0 a_0 x_1 \dots x_{m-1} a_{m-1} x_m$, with $x_0 = x$ and $x_m = y$. Assume $x, y \in \mathbf{V}(\mathrm{D})$.

We will show, in Lemma 5.20 below, that for large enough ℓ, if Player 2 has a winning strategy on the game on $(\mathbf{V}(\mathrm{D}), T_{\mathrm{Q},\mathbf{V}}, x, y)$ then we can exhibit a homomorphism witnessing the fact that $(x, y) \in \langle \mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathbf{V}(\mathrm{D}))$. Before that we prove crucial properties of $\mathbf{V}(\mathrm{D})$ which will be exploited in the sequel. For that we need the following simple definitions and claims.

Let $\mathrm{E} = \mathbf{V}(\mathrm{D})$ and let $A = \langle S_{\mathbf{V}}, \delta_{\mathbf{V}}, q^0_{\mathbf{V}}, F_{\mathbf{V}}\rangle$ be the product automaton of all the deterministic minimal automata of all the regular expressions of the RPQs in $\mathbf{V}$. Let $N(\mathbf{V})$ be the number of states of A, i.e. $|S_{\mathbf{V}}|$.

In what follows, for $q \in S_{\mathbf{V}}$ and $w \in \sigma^*$, $\delta_{\mathbf{V}}(q, w)$ denotes the state $p \in S_{\mathbf{V}}$ such that there is a run of A on w starting in state q and arriving in state p.

For every $k \leq m + 1$, and every $i, j \leq k$, we say that $x_i \sim_k x_j$ in $\mathbf{V}(\mathrm{D})$ if, for all $V \in \mathbf{V}$, for all $r \geq k$,

$$(x_i, x_r) \in V(\mathrm{D}) \quad \Leftrightarrow \quad (x_j, x_r) \in V(\mathrm{D})$$

For all k, the relation $\sim_k$ is an equivalence relation over $\{x_i \mid i \leq k\}$. We now prove the main property of $\mathbf{V}(\mathrm{D})$, namely that the index of all $\sim_k$ is bounded by the size of $\mathbf{V}$.

Claim 5.18. *For all $k \leq m + 1$:*

$$\left| \{x_i \mid i \leq k\} / \sim_k \right| \leq N(\boldsymbol{V})$$

Proof. To each node x_i in π with $i \leq k$, we associate a state $\varphi(x_i) \in S_{\mathbf{V}}$ defined as :

$$\varphi(x_i) = \delta_{\mathbf{V}}(q^0_{\mathbf{V}}, \lambda(\pi_{i\to k}))$$

where $\pi_{s\to t}$ is defined as the subpath of π that starts at position s and ends at position t, that is $\pi_{s\to t} = x_s a_s x_{s+1} \dots x_{t-1} a_{t-1} x_t$.

Assume that there exist two nodes x_i and x_j, with $i, j \leq k$, that have the same image in φ. It follows that:

$$\delta_{\mathbf{V}}(q^0_{\mathbf{V}}, \lambda(\pi_{i\to k})) = \delta_{\mathbf{V}}(q^0_{\mathbf{V}}, \lambda(\pi_{j\to k}))$$

Let us prove that $x_i \sim_k x_j$. Assume that there exist $r \geq k$ and $V \in \mathbf{V}$ such that $(x_i, x_r) \in V(\mathrm{D})$. Then $\delta_{\mathbf{V}}(q^0_{\mathbf{V}}, \lambda(\pi_{i\to r}))$ is final for V. Remark that $\lambda(\pi_{i\to r}) = \lambda(\pi_{i\to k})\lambda(\pi_{k\to r})$, from which we can deduce that :

$$\delta_{\mathbf{V}}(q^0_{\mathbf{V}}, \lambda(\pi_{i\to r})) = \delta_{\mathbf{V}}(\varphi(x_i), \lambda(\pi_{k\to r}))$$

Hence,

$$\delta_{\mathbf{V}}(q^0_{\mathbf{V}}, \lambda(\pi_{i\to r})) = \delta_{\mathbf{V}}(\varphi(x_j), \lambda(\pi_{k\to r}))$$

We can now conclude that $\delta_{\mathbf{V}}(q^0_{\mathbf{V}}, \lambda(\pi_{j\to r}))$ is final for V, which means that $(x_j, x_r) \in V(\mathrm{D})$. A symmetric argument easily proves the other direction of the equivalence. Hence, $x_i \sim_k x_j$, and we can finally conclude that there cannot be more that $N(\mathbf{V})$ distinct equivalence classes of $\sim_k$ over the nodes $\{x_i \mid i \leq k\}$ of π. □

The following easily verified property of the equivalence relations $\sim_k$ will also be useful:

Claim 5.19. *Let $k_1, k_2 \leq m + 1$, with $k_1 \leq k_2$. Let x and y be two elements of π that occur before x_{k_1}. Then $x \sim_{k_1} y$ implies $x \sim_{k_2} y$.*

We are now ready to prove the statement of Proposition 5.17.

Let $\ell = |T_{\mathrm{Q},\mathbf{V}}| \cdot N(\mathbf{V})$. We prove that $(x, y) \in Q_{\ell,\ell+1}(\mathrm{E})$ iff $(x, y) \in \langle\neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle(\mathrm{E})$. In view of the fact that $Q_{\ell,\ell+1}$ encodes the $(\ell, \ell+1)$-game in the sense of Lemma 5.16, it is enough to prove the following:

Lemma 5.20. *Player 2 has a winning strategy for the $(\ell, \ell+1)$-two-player game on $(E, T_{Q,\mathbf{V}}, x, y)$ iff there is a homomorphism from E to $T_{Q,\mathbf{V}}$ sending x to a source node and y to a target node.*

Proof. The right-left direction is obvious. If there is a suitable homomorphism h from E to $T_{\mathrm{Q},\mathbf{V}}$, then Player 2 has a winning strategy which consists in playing according to h.

Conversely, assume that Player 2 has a winning strategy for the $(\ell, \ell+1)$-game on $(\mathrm{E}, T_{\mathrm{Q},\mathbf{V}}, x, y)$. Let $\{s_1, s_2, \ldots, s_r\}$ be an ordering of the elements of E, according to the order on π, that is, in such a way that $\forall j \leq k$, s_j occurs before s_k in π. Clearly $s_1 = x$ and $s_r = y$. If $r \leq \ell + 1$, Player 1 can select all elements of E in a single round, and then Player 2 has to provide a full homomorphism from E to $T_{\mathrm{Q},\mathbf{V}}$, which concludes the proof.

Assume $r > \ell + 1$. For ease of notations, we will number rounds starting from $\ell + 1$. This can be seen just as a technicality, or equivalently as Player 1 selecting the empty set for the first ℓ rounds. Since Player 2 has a winning strategy, she has, in particular, a winning response against the following play of Player 1:

- On round $\ell + 1$, Player 1 plays $A_{\ell+1} = \{s_1, \ldots, s_{\ell+1}\}$. Player 2 has to respond with a partial homomorphism $h_{\ell+1}$, which she can do, since she has a winning strategy.

- Assume that, on round i, A_i is of size $\ell+1$ and its element of biggest index is s_i (as it is the case on round $\ell+1$). Given the choice of ℓ, the set A_i is sufficiently "big", that is by Claim 5.18, there exist two elements $s_j, s_k \in A_i$ such that $s_j \sim_i s_k$, and $h_i(s_j) = h_i(s_k)$. On round $i+1$, Player 1 picks $A_{i+1} = (A_i - \{s_j\}) \cup \{s_{i+1}\}$. This choice maintains that A_{i+1} is of size $\ell+1$ and that its element of biggest index is s_{i+1}. Once again, Player 2 has to respond with a partial homomorphism h_{i+1}, which she can do.

- Following this play, on round r, A_r contains s_r, the element of biggest index in E. From now on, we no longer care about Player 1's move, that is, we arbitrarily set $A_i = \emptyset$ for all $i > r$.

We can now define h as follows:

$$h(s_i) = \begin{cases} h_{\ell+1}(s_i) & \text{if } i \leq \ell+1 \\ h_i(s_i) & \text{if } \ell+1 < i \leq r \end{cases}$$

Observe that, by definition, the mapping h sends x to a source node and y to a target node (since so do all the h_i's used in the game). It remains to prove that h is a homomorphism from E to $T_{\mathrm{Q},\mathbf{V}}$. We prove by induction on $i \geq \ell+1$ that :

(H_1) h is a homomorphism from $\mathrm{E}[\{s_1, \ldots, s_i\}]$ to $T_{\mathrm{Q},\mathbf{V}}$.

(H_2) h coincides with h_i on A_i.

(H_3) for all $j \leq i$, there exists $s \in A_i$ such that $s_j \sim_i s$ and $h(s_j) = h(s)$.

Base case : For $i = \ell+1$, the mapping h coincides by definition with $h_{\ell+1}$ on $\{s_1, \ldots, s_{\ell+1}\}$. Hence, ($H_1$) and ($H_3$) follow easily.

Inductive case : Assume that there exists i with $\ell+1 \leq i < r$ such that (H_1),(H_2) and (H_3) holds for i; we prove them for $i+1$.

(H_2) Let $s \in A_{i+1}$. If $s = s_{i+1}$, then, by definition, $h(s_{i+1}) = h_{i+1}(s_{i+1})$. Otherwise, $s \in A_i \cap A_{i+1}$. (H_2) for i implies that $h(s) = h_i(s)$, and the definition of h_{i+1} thus yields $h_{i+1}(s) = h_i(s) = h(s)$. Hence, ($H_2$) holds for $i+1$.

(H_3) Let $j \leq i+1$. If $j = i+1$, then $s_j \in A_{i+1}$, and the result is obvious. Otherwise, (H_3) for i implies that there exists $s \in A_i$ such that $s_j \sim_i s$ and $h(s_j) = h(s)$. From Claim 5.19, we deduce that $s_j \sim_{i+1} s$. If $s \in A_{i+1}$, there is nothing more to prove. Otherwise, it means that s is exactly the element that was removed from A_i on round $i+1$, which means that there exists another element $s' \in A_i \cap A_{i+1}$ such that $s \sim_i s'$ and $h_i(s) = h_i(s')$. Then Claim 5.19 and (H_2) imply that $s_j \sim_{i+1} s'$ and $h(s_j) = h(s')$. Hence (H_3) holds for $i+1$.

(H_1) By definition, h already preserves any self-loop. Moreover, (H_1) for i implies that h is a homomorphism from $\mathrm{E}[\{s_1, \ldots, s_i\}]$ to $T_{\mathrm{Q},\mathbf{V}}$. Hence, any edge between two elements of $\{s_1, \ldots, s_i\}$ in § is already preserved by h. Let $s_j \in \{s_1, \ldots, s_i\}$. Remark

that, since π is a simple path, there are no edges from s_{i+1} to s_j in E. Thus, we just have to prove that all edges from s_j to s_{i+1} are preserved by h.

(H_3) for $i+1$ implies that there exists an element $s \in A_{i+1}$ such that $s_j \sim_{i+1} s$ and $h(s_j) = h(s)$. Since h_{i+1} is a homomorphism on $\mathrm{E}[A_{i+1}]$, it preserves all edges from s to s_{i+1}. Moreover, (H_2) for $i+1$ implies that h and h_{i+1} coincide on A_{i+1}, which means that h preserves all edges from s to s_{i+1}. Finally, the definition of $\sim_{i+1}$ implies that s_j and s have the same edges to s_{i+1}. Hence, h preserves all edges from s_j to s_{i+1}.

Finally, (H_1) applied for r proves that h is indeed a homomorphism from E to $T_{\mathrm{Q},\mathbf{V}}$. This completes the proof of Lemma 5.20. □

From Lemma 5.20, we deduce that $Q_{\ell,\ell+1}$ coincides with $\langle \neg\mathrm{CSP}(T_{\mathrm{Q},\mathbf{V}})\rangle$ on the view images of simple path databases. Finally, in the case that $\mathbf{V}$ determines Q in a monotone way, Corollary 5.13 tells us that $(x,y) \in Q_{\ell,\ell+1}(\mathbf{V}(\mathrm{D}))$ if and only if $(x,y) \in \mathrm{Q}(\mathrm{D})$. This completes the proof of Proposition 5.17.

5.3.3 From simple paths to arbitrary graph databases

In Proposition 5.17, we proved that if a regular path view $\mathbf{V}$ determines a regular path query Q, then there exists ℓ such that $Q_{\ell,\ell+1}$ coincides with rewritings of Q using $\mathbf{V}$ on the view images of simple path databases. In this section, we prove that this property actually implies that $Q_{\ell,\ell+1}$ is a rewriting of Q using $\mathbf{V}$.

This result is not actually specific to $Q_{\ell,\ell+1}$ but holds true in a more general sense, which comes from the following remark. Assume that a regular path view $\mathbf{V}$ determines a query Q in a monotone way. Then the databases for which a pair of nodes (x,y) belong to the query result are exactly those databases whose view contains the image of a path from x to y which satisfies the query. Indeed, when a pair satisfies the query, there must exist in the database a path whose label satisfies the query, and its image thus belongs to the image of the database. The converse is immediately given by the monotone determinacy property: if the view of a path is included in the view of the database, then the query result on the path must be contained in the query result of the database. This makes it so that it is enough for an under-approximation of a rewriting to be correct on the view images of simple paths and to be closed under homomorphism to actually be a rewriting. This intuition is formalized in Proposition 5.21 below.

Proposition 5.21. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query such that $\boldsymbol{V}$ determines Q in a monotone way. Assume P is a query of schema τ such that:*

1. *P is closed under homomorphisms: for all databases E, E', and all pair of elements (x,y) of E, if $(x,y) \in P(E)$ and there exists a homomorphism $h : E \to E'$ then $(h(x), h(y)) \in P(E')$.*

2. *P is a rewriting on view images of simple path databases: for all simple path databases D from x to y such that x and y are in the domain of $\boldsymbol{V}(D)$, we have $(x,y) \in P(\boldsymbol{V}(D))$ iff $(x,y) \in Q(D)$.*

3. *P is an under-approximation of a rewriting: for all graph databases D and elements x and y of $\boldsymbol{V}(D)$, if $(x, y) \in P(\boldsymbol{V}(D))$ then $(x, y) \in Q(D)$.*

Then P is a rewriting of Q using $\boldsymbol{V}$.

Proof. Let D be a database, and (x, y) be a pair of elements of $\mathbf{V}(\mathrm{D})$, such that $(x, y) \in \mathrm{Q}(\mathrm{D})$. Then there exists in D a path π_0 from x to y, such that $\lambda(\pi_0) \in L(\mathrm{Q})$.

Consider the simple path $\pi = x_0 a_0 x_1 \dots x_{m-1} a_{m-1} x_m$ defined such that $\lambda(\pi) = \lambda(\pi_0)$. Since $\mathbf{V}$ determines Q in a monotone way and $\lambda(\pi) \in L(\mathrm{Q})$, then x_0 and x_m are in the domain of $\mathbf{V}(\pi)$, and $(x_0, x_m) \in \mathrm{Q}(\pi)$. Hence, (2) implies that $(x_0, x_m) \in P(\mathbf{V}(\pi))$.

Additionally, it is clear that there exists a homomorphism h from π to D with $h(x_0) = x$ and $h(x_m) = y$. Observe that h extends to the views of π and D, that is h is a homomorphism from $\mathbf{V}(\pi)$ to $\mathbf{V}(\mathrm{D})$, and (1) thus implies that $(x, y) \in P(\mathbf{V}(\mathrm{D}))$.

The other direction is immediately given by (3). □

We now have all the elements to prove the following proposition:

Proposition 5.22. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query such that $\boldsymbol{V}$ determines Q in a monotone way. There exists ℓ such that $Q_{\ell,\ell+1}$ is a rewriting of Q using $\boldsymbol{V}$.*

Indeed, let $\mathbf{V}$ be a regular path view and Q be a regular path query such that $\mathbf{V}$ determines Q in a monotone way. Then, we already know from Proposition 5.17 that there exists ℓ such that $Q_{\ell,\ell+1}$ is a rewriting of Q using $\mathbf{V}$ when restricted to the view images of simple path databases. Additionally, we know that Datalog queries are preserved under homomorphism. Finally, we have already observed in Section 5.3.2 that $Q_{\ell,\ell+1}$ is an under-approximation of a rewriting[1]. Thus we can apply Proposition 5.21, which proves that $Q_{\ell,\ell+1}$ is a rewriting of Q using $\mathbf{V}$ and concludes the proof of Proposition 5.22. This immediately leads to the main result of this chapter:

Theorem 5.23. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query such that $\boldsymbol{V}$ determines Q in a monotone way. Then there exists a Datalog rewriting of Q using $\boldsymbol{V}$.*

We conclude this chapter by mentioning two important corollaries of Theorem 5.23. First, we remark that the existence of a Datalog rewriting coincides with the existence of a monotone rewriting, since all Datalog queries are monotone. Thus, the existence of a Datalog rewriting is decidable, by Corollary 5.7.

Corollary 5.24. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query. It is decidable,* ExpSpace*-complete, whether there exists a Datalog rewriting of Q using $\boldsymbol{V}$.*

Second, and most important, we now know that answering a query using views when the view determines the query in a monotone way can be done with polynomial time data complexity.

Corollary 5.25. *Let $\boldsymbol{V}$ be a regular path view and Q be a regular path query such that $\boldsymbol{V}$ determines Q in a monotone way. Then there exists a rewriting of Q using $\boldsymbol{V}$ that can be evaluated with* PTime *data complexity.*

[1]This is actually true of all $Q_{\ell,k}$ regardless of the specific values of ℓ and k.

5.4 Extensions

This final section discusses possible extensions of the work presented in this chapter in an informal way. In Section 5.4.1, we consider two-way regular path queries, an extension of path queries with the ability to navigate the edges of the database in both directions. In Section 5.4.2, we discuss a possible continuation of our work, that aims at finding rewritings in query languages that are simpler than Datalog.

5.4.1 Two-way regular path queries

Two-way regular path queries (2RPQ) are defined using regular expressions over an extended schema $\sigma_2 = \sigma \cup \bar{\sigma}$, where $\bar{\sigma}$ is a schema that contains a fresh copy $\bar{a}$ of each symbol $a \in \sigma$. The intuitive idea here is that a 2RPQ can navigate the a edges of the database in the forward direction by using symbol a, or in the backward direction by using symbol $\bar{a}$. This extends the expressive power of regular path queries by allowing two-way navigation in the database. Since 2RPQ not covered by the framework described in Chapter 3 nor are they the focus of our work, the discussion in this section will remain on a very informal level.

We can summarize the main ingredients of the results presented in this chapter as follows:

- Corollary 5.7, showing that the monotone determinacy problem for regular path queries and views is ExpSpace-complete,
- Corollary 5.13, showing that when a regular path view determines a regular path query in a monotone way, then a rewriting can be expressed as the negation of a CSP,
- Lemma 5.16, showing that any CSP can be approximated as a Datalog program,
- Proposition 5.17, showing that one of the Datalog approximations is exact on the view images of simple paths,
- Proposition 5.21, showing that the case of simple paths can be lifted to arbitrary databases and thus that the Datalog approximation is a rewriting.

Remark now that Lemma 5.16 and Proposition 5.21 do not depend on the query or view languages. Additionally, [15] provides a counterpart to Corollary 5.7 by showing that monotone determinacy for 2RPQ queries and views is ExpSpace-complete. Moreover, it was proved in [14] that Corollay 5.13 also extends to the case of 2RPQ queries and views.

Finally, in order to extend Theorem 5.23, it remains only to prove that Proposition 5.17 extends to the case of 2RPQ queries and views. This turns out to be the case, by following a very similar proof to Proposition 5.17, that revolves around turning a winning strategy for Player 2 in the local consistency game into a global homomorphism for the CSP on view images of simple paths.

5.4.2 On rewriting languages

In this chapter, we have shown that whenever a regular path view $\mathbf{V}$ determines a regular path query Q in a monotone way, then there exists a Datalog rewriting of Q using $\mathbf{V}$. The main consequence of this result is the existence of a rewriting that enjoys polynomial time data complexity. However, Datalog is not a very simple or user-friendly query language. Thus, Theorem 5.23 leaves an interesting question open: does there exist a rewriting of Q using $\mathbf{V}$ that can be expressed in a simpler query language?

We have seen in Example 5.9 that there exist a regular path view $\mathbf{V}$ and a regular path query Q such that $\mathbf{V}$ determines Q in a monotone way and such that no regular path query can be a rewriting of Q using $\mathbf{V}$. Similarly, Example 5.10 provides a concrete case where no conjunctive regular path query can be a rewriting.

In Example 5.10, we expressed a rewriting in an extension of binary conjunctive regular path queries that is closed under transitive closure. It turns out that this is the most difficult concrete example that we are aware of: in all examples that we know of, whenever a regular path view determines a regular path query in a monotone way, a rewriting can be expressed in this query language. Thus it is natural to wonder whether this holds in general. Failing that, another good candidate would be the linear fragment of binary Datalog, in which all internal predicates are of arity 2 and at most one of them may occur in the body of each rule. Indeed, this language contains the transitive closures of binary conjunctive regular path queries, and is closer to the general Datalog programs that we used in this chapter.

Chapter 6

Asymptotic determinacy of single path queries

In this chapter, we move away from the monotone determinacy problem of Chapter 5 and come back to general determinacy. The goal here is to push the known decidability for the determinacy problem. We know from [3] that determinacy is decidable for single path queries and single path views. In this chapter, we show that we can decide a more restricted form of determinacy, that we call *asymptotic determinacy*, for single path queries and *union* of single path views, provided that the schema σ with which we are working contains a single relational symbol.

In all this chapter, we will work with single path queries (SPQ) and union of single paths (UPQ) views over a schema $\sigma = \{a\}$ which contain a single relational symbol. There are two ways to apply the results of this chapter to a database D defined over a schema σ with $|\sigma| \geq 2$. First and most immediate, it directly applies to views and queries that only make use of one symbol $a \in \sigma$, in which case they would be evaluated over a projection D_a of the database that removes all other symbols. Another more natural perspective is to consider that the work presented here applies to *distance queries*, that is queries that select pairs of nodes that are linked by a path of a given length, while disregarding the labels along the path. Thus, these queries correspond to single path queries that are not directly evaluated on D but on the underlying structure of D, which is an unlabeled graph.

Finally, as explained in Section 3.2, since the queries we consider are defined over a schema with only one label $\sigma = \{a\}$, we will simply write $\mathrm{Q} = \langle k_1, \ldots, k_n, \ldots \rangle$ instead of $\mathrm{Q} = \langle \{a^{k_1}, \ldots, a^{k_n}, \ldots\} \rangle$.

In Section 6.1, we give a first formal definition of asymptotic determinacy. Then, after a serie of small results, we restate this first definition in a more intuitive and workable way. In Section 6.2, we introduce *behavior graphs*, which will be the main tool for deciding asymptotic determinacy in Section 6.3. Finally, in Section 6.4, we discuss several extensions of this work, as well as some related questions that are left open.

6.1 Preliminaries

Formally, the *asymptotic determinacy problem* is defined as a variant of the determinacy problem, that asks whether a view $\mathbf{V}$ determines a query Q, but with the added information that Q is "big" compared to $\mathbf{V}$: for each view $\mathbf{V}$, there are a finite number of "small" queries Q for which the question is not asked. The problem is characterized by a parameter α: a function that maps each view $\mathbf{V}$ to a natural number, such that the queries Q for which the problem is defined are those that ask for paths longer than $\alpha(\mathbf{V})$.

PROBLEM :	α-ASYMPTOTIC DETERMINACY
INPUT :	A union of single paths view $\mathbf{V}$,
	A single path query $Q = \{n\}$ with $n > \alpha(\mathbf{V})$
QUESTION :	Does $\mathbf{V}$ determine Q?

While this definition might seem very formal and unintuitive, we will see several key properties in Section 6.1.1 that allow us in Section 6.1.2 to restate the problem in a more practical and intuitive way. Note however that providing an answer to the determinacy problem or a rewriting in "almost all" cases is something that has already been considered in the same context. For instance, in [3], it is shown that for a given single path view $\mathbf{V}$, almost all single path queries Q that are determined by $\mathbf{V}$ have a conjunctive rewriting. However, for a finite number of such queries, no conjunctive rewritings can be found, as in Example 4.17 and Example 5.1.

The main purpose of this chapter is to prove the following theorem:

Theorem 6.1. *There is an explicit and computable function α for which the α-asymptotic determinacy problem is decidable. Moreover, when the view determines the query, the decision procedure effectively computes a first-order rewriting of the query using the view.*

Arithmetic Notations. Some of the proofs in this chapter involve a lot of arithmetic reasonings. We present here the notations that we use. Given two integers n and d, $n[d]$ represents the remainder in the division of n by d. We say that two integers n_1 and n_2 are equivalent modulo d, and we write $n_1 \equiv n_2[d]$ if they have the same remainder modulo d. We denote by $\gcd(A)$ the greatest common divisor of a set of integers A, and we use $n_1 \wedge n_2$ for $\gcd(\{n_1, n_2\})$.

6.1.1 Key properties

In this section, we consider a UPQ view $\mathbf{V}$ and an SPQ query Q, such that $\mathbf{V}$ determines Q. It turns out that this implies several simple properties for both $\mathbf{V}$ and Q: necessary conditions without which a UPQ view cannot possibly determine an SPQ query. These are our key results, as taking these conditions into account will allow us to gain a better understanding of the determinacy problem, and lead to a new perspective on the asymptotic determinacy in Section 6.1.2.

Our first key result states that $\mathbf{V}$ cannot possibly determine Q if $\mathbf{V}$ does not at least contain a single path query. In other words, even though $\mathbf{V}$ is a UPQ view, at least one of the queries that appear in $\mathbf{V}$ must be an SPQ and thus cannot make use of the union.

Lemma 6.2. *Assume that a UPQ view $\boldsymbol{V}$ and an SPQ query Q are such that $\boldsymbol{V} \twoheadrightarrow Q$. Then there exists $C \in \boldsymbol{V}$ such that $C = \langle k \rangle$, for some k.*

Proof. Assume by contraposition that, for all $V \in \mathbf{V}$, $|L(V)| > 1$. Let $\mathrm{Q} = \langle n \rangle$. We build a database D as follows:

- D contains $n+1$ distinct nodes $x_0, \ldots, x_n$.
- For all $i < n$, $a(x_i, x_{i+1})$ holds in D.
- For all $i \leq n$, for all $V \in \mathbf{V}$ such that $i \in V$, we add to D a simple path $\pi_{i,V}$ from x_0 to x_i, such that $|\pi_{i,V}| \in L(V) - \{i\}$. Such a path exists because $|L(V)| > 1$.

We then construct another database D′ which is a copy of D except that $a(x_0, x_1)$ does not hold in D′. It is then easy to check that $\mathbf{V}(\mathrm{D}) = \mathbf{V}(\mathrm{D}')$ and that $\mathrm{Q}(\mathrm{D}) \neq \mathrm{Q}(\mathrm{D}')$. In particular, $(x_0, x_n) \in \mathrm{Q}(\mathrm{D})$ and $(x_0, x_n) \notin \mathrm{Q}(\mathrm{D}')$. Hence $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$, which concludes the proof. This construction is illustrated on Figure 6.1. □

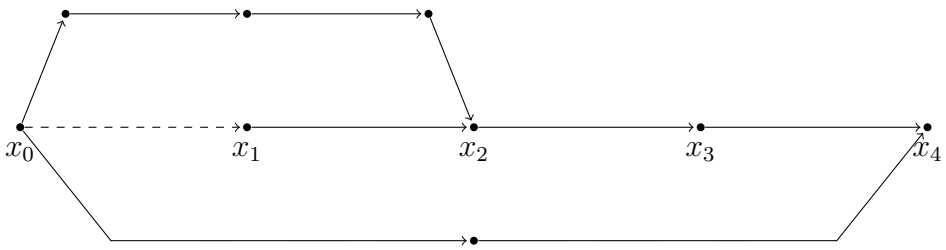


Figure 6.1: Illustration for the proof of Lemma 6.2, showing here that $V = \langle 2, 4 \rangle \not\twoheadrightarrow \langle 4 \rangle$. Following the notations in the proof, the top path is $\pi_{2,V}$ and the bottom path is $\pi_{4,V}$. Remark then that adding or removing the dashed edge does not change the view, but changes the query result.

Our next necessary condition is similar to the condition found in [3]. Let $\mathrm{Q} = \langle n \rangle$ be an SPQ and $\mathbf{V}$ be an SPQ view, without unions. Consider a database D which consists of a simple path of length n: $x_0 \ldots x_n$. Let $\mathrm{E} = \mathbf{V}(\mathrm{D})$, and let G be the underlying undirected graph of E, that is a graph whose set of nodes is exactly the set of nodes of E, and such that there is an undirected and unlabeled edge between two nodes x and y in G if and only if there is any edge going from x to y or from y to x in E. Then it was shown in [3] that, if $\mathbf{V} \twoheadrightarrow \mathrm{Q}$, then x_0 and x_n belong to the same connected component in G. We remark that this condition still holds in our case, where $\mathbf{V}$ is a UPQ view.

Lemma 6.3. *Let $\boldsymbol{V}$ be a UPQ view and Q be an SPQ with $Q = \{n\}$. Let $\pi = x_0 \ldots x_n$. If $\boldsymbol{V} \twoheadrightarrow Q$ then there is an undirected path from x_0 to x_n in $\boldsymbol{V}(\pi)$.*

Proof. Let $\mathbf{V}$, Q and π be as in the statement of the lemma. Assume by contraposition that x_0 and x_n are in different connected components of $\mathbf{V}(\pi)$.

Let D be a database that consists of two disjoints copies of π, that we denote by $\mu = y_0, \ldots, y_n$ and $\nu = z_0, \ldots, z_n$. Then $\mathbf{V}(\mathrm{D})$ has at least four distinct connected components, that respectively contain y_0, z_0, y_n and z_n. This comes from the fact that there are no paths from y_i to z_j, for any i and j. Since $\mathbf{V}$ is a path view, it cannot link two nodes that are not connected with each other. Remark that these connected components are pairwise identical, up to renaming y_i to z_i and z_i to y_i.

Consider now the database D′ that is a copy of D except that for all i, y_i is renamed to z_i and z_i is renamed to y_i if y_i is in the connected component of y_n in $\mathbf{V}(\mathrm{D})$.

Remark now that $\mathbf{V}(\mathrm{D}) = \mathbf{V}(\mathrm{D}')$. Indeed, the connected components of $\mathbf{V}(\mathrm{D})$ that do not contain y_n or z_n are left untouched in $\mathbf{V}(\mathrm{D}')$, whereas the connected components that contain y_n or z_n are simply renamed one to the other. However, $\mathrm{Q}(\mathrm{D}) = \{(y_0, y_n), (z_0, z_n)\}$, whereas $\mathrm{Q}(\mathrm{D}') = \{(y_0, z_n), (z_0, y_n)\}$, which concludes the proof. □

In view of Lemma 6.2, we know that we can restrict our attention to views that contain a single path query C, otherwise we immediately conclude that the view does not determine any SPQ. This allows us to state the following definition for such views:

Definition 6.4 (Complete)**.** *Let $\boldsymbol{V}$ be a UPQ view and $C \in \boldsymbol{V}$ such that $C = \langle c \rangle$. We say that $\boldsymbol{V}$ is* C-complete *if, for all $i \in \{0, \ldots, c-1\}$, there exists $V \in \boldsymbol{V}$ and $k \in L(V)$ such that $k \equiv i[c]$.*

It turns out that Lemma 6.2 together with Lemma 6.3 allows us to restrict our attention further to cases where $\mathbf{V}$ is C-complete for some single path query $C \in \mathbf{V}$. This is formalized in the lemma below:

Lemma 6.5. *Let $\boldsymbol{V}$ be a UPQ view and $C \in \boldsymbol{V}$ such that $C = \langle c \rangle$. Let $Q = \langle n \rangle$, and $\pi = x_0 \ldots x_n$. Assume that there is an undirected path from x_0 to x_n in $\boldsymbol{V}(\pi)$. Then we can effectively compute a view $\boldsymbol{V}'$ with $C' \in \boldsymbol{V}'$ such that $C' = \langle c' \rangle$ and a query Q' such that $\boldsymbol{V}'$ is C'-complete and $\boldsymbol{V} \twoheadrightarrow Q$ if and only if $\boldsymbol{V}' \twoheadrightarrow Q'$.*

Proof. Let $\mathbf{V}$ and Q be defined as in the statement of the lemma. Let U be the set of all numbers that appear in $\mathbf{V}$, that is:

$$U = \bigcup_{\substack{V \in \mathbf{V} \\ m \in L(V)}} \{u\}$$

and $d = \gcd(U)$.

Assume $d = 1$. Then there exists $m_1, \ldots, m_k \in U$ such that $m_1 + \ldots + m_k \equiv 1[c]$. This means that there exist $V_1, \ldots, V_k \in \mathbf{V}$ and $m \in L(V) = L(V_1) \cdot \ldots \cdot L(V_k)$ such that $m \equiv 1[c]$. We define $\mathbf{V}'$ as

$$\mathbf{V}' = \mathbf{V} \cup \bigcup_{i=1}^{c} \{V^i\}$$

and $\mathrm{Q}' = \mathrm{Q}$. It follows that $\mathbf{V}'$ is C-complete. Indeed, for all $k \in \{0, \ldots, c-1\}$, $L(V^k)$ contains $kv \equiv k[c]$. Additionally, $\mathbf{V}' \twoheadrightarrow \mathrm{Q}'$ if and only if $\mathbf{V} \twoheadrightarrow \mathrm{Q}$, as all queries in $\mathbf{V}$ are also in $\mathbf{V}'$, and all queries in $\mathbf{V}'$ can be written as compositions of queries in $\mathbf{V}$.

Assume now that $d \neq 1$. Then d divides all the numbers in $\mathbf{V}$. Additionally, since there exists a path from x_0 to x_n in $\mathbf{V}(x_0 \ldots x_n)$, it implies that d divides n as well. For each $V \in \mathbf{V}$, we define $V' = \langle \frac{m}{d} \mid m \in V \rangle$. We then define $\mathbf{V}' = \{V' \mid V \in \mathbf{V}\}$ and $\mathrm{Q}' = \{\frac{n}{d}\}$.

Claim 6.6. *$\boldsymbol{V} \twoheadrightarrow Q$ if and only if $\boldsymbol{V}' \twoheadrightarrow Q'$.*

- Assume that $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$. Then there exists two databases D_1 and D_2 such that D_1 and D_2 agree on $\mathbf{V}$ but not on Q. We build two new databases D'_1 and D'_2 that are copies of D_1 and D_2 except that there is an edge between x and y in D'_i if and only if there is a path of length d from x to y in D_i.

  Let x and y be two nodes of D'_1 such that $(x, y) \in V'(\mathrm{D}'_1)$ for some $V' \in \mathbf{V}'$. Then there exists $m \in V'$ such that x and y are at distance m in D'_1. By construction, this means that x and y are at distance dm in D_1. Hence, $(x, y) \in V(\mathrm{D}_1)$. Then $(x, y) \in V(\mathrm{D}_2)$. Thus, there exists $r \in L(V)$ such that x and y are at distance r in D_2. It follows that x and y are at distance $\frac{r}{d}$ in D'_2, and finally that $(x, y) \in V'(\mathrm{D}'_2)$. Hence D'_1 and D'_2 agree on $\mathbf{V}'$. A similar reasoning shows that D'_1 and D'_2 don't agree on Q', so that we can conclude that $\mathbf{V}' \not\twoheadrightarrow \mathrm{Q}'$.

- Assume that $\mathbf{V}' \not\twoheadrightarrow \mathrm{Q}'$. Then there exists two dabatases D'_1 and D'_2 such that D'_1 and D'_2 agree on $\mathbf{V}'$ but not on Q'. We build two new databases D_1 and D_2 as follows:

  - For each node x of D'_i and each $\alpha \in \{0, \ldots, d-1\}$, (x, α) is a node of D_i.
  - For each x and each $\alpha < d-1$, there is an edge from (x, α) to $(x, \alpha+1)$.
  - For each x, y such that there is an edge from x to y in D'_i, there is an edge from $(x, d-1)$ to $(y, 0)$ in D_i.

  Let (x, α) and (y, β) be two nodes of D_1 such that $\big((x, \alpha), (y, \beta)\big) \in V(\mathrm{D}_1)$ for some $V \in \mathbf{V}$. Then there exists $m \in L(V)$ such that (x, α) and (y, β) are at distance m in D_1. Since $m \in L(V)$, we have $m \equiv 0[d]$, which implies that $\alpha = \beta$. By construction, this implies that x and y are at distance $\frac{m}{d}$ in D'_1. Hence, $(x, y) \in V'(\mathrm{D}'_1)$. Then $(x, y) \in V'(\mathrm{D}'_2)$. Thus there exists $r \in V'$ such that x and y are at distance r in D'_2. By construction, this implies that (x, α) and (y, α) are at distance dr in D_2, and thus $\big((x, \alpha), (y, \alpha)\big) \in V(\mathrm{D}_2)$. Hence D_1 and D_2 agree on $\mathbf{V}$. A similar reasoning shows that D_1 and D_2 don't agree on Q, so that we can conclude that $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$.

Finally, we get a new set of views $\mathbf{V}'$ for which we can apply the first case of the proof and compute a new $\mathbf{V}'$ that is C'-complete. □

The last lemma of this section shows that the set of queries determined by a UPQ view $\mathbf{V}$ which contains a path query $C = \langle c \rangle$ is closed under adding c. While perhaps obvious, this result is key to defining asymptotic determinacy, as will be explained in Section 6.1.2.

Lemma 6.7. *Let $\boldsymbol{V}$ be a UPQ view and $C \in \boldsymbol{V}$ such that $C = \langle c \rangle$. Let $Q = \langle n \rangle$, and assume that $\boldsymbol{V} \twoheadrightarrow Q$. Then, for all positive integers k, $\boldsymbol{V} \twoheadrightarrow \langle n + kc \rangle$.*

Proof. Let R be a rewriting of Q using $\mathbf{V}$. Let k be a positive integer. Then it is easy to check that $R \cdot C^k$ is a rewriting of $\mathrm{Q}' = \langle n + kc \rangle$ using $\mathbf{V}$. □

6.1.2 Asymptotic determinacy

In this section, we show how to use the results of Section 6.1.1 to assess the situation from a slighty different perspective. Let V be a view defined by unions of single path queries. Assume that, instead of being given a specific single path query Q for which we want to decide whether $\mathbf{V} \twoheadrightarrow \mathrm{Q}$, we want to compute the complete determinacy picture of $\mathbf{V}$. In other words, we want to know all single path queries Q such that $\mathbf{V} \twoheadrightarrow \mathrm{Q}$.

We start by using Lemma 6.2. This allows us to say that, if $\mathbf{V}$ does not contain a single path query $C = \langle c \rangle$, then there is no single path query Q such that $\mathbf{V} \twoheadrightarrow \mathrm{Q}$, which answers our question, as well as the asymptotic determinacy problem. Let us now assume that $\mathbf{V}$ does indeed contain a path query $C = \langle c \rangle$. Consider a natural number $o \in \{0, \ldots, c-1\}$. Then, there are two cases for a query $\mathrm{Q} = \langle m \rangle$ such that $m \equiv o[c]$:

- Case 1: $\mathbf{V}$ does not determine any query $\mathrm{Q}' = \langle n \rangle$ with $n \equiv o[c]$. In particular, this means that $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$, which answers our question for such queries.
- Case 2: There exists some query $\mathrm{Q}' = \langle n \rangle$ with $n \equiv o[c]$ such that $\mathbf{V} \twoheadrightarrow \mathrm{Q}$. Let us assume, without loss of generality, that Q' is actually the smallest such query. Then either $m < n$, in which case we can easily conclude that $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$, or $m \geq n$, in which case Lemma 6.7 immediately proves that $\mathbf{V} \twoheadrightarrow \mathrm{Q}$. Thus the determinacy status for such queries is entirely determined by this specific n.

What this means is that, if we restrict our attention to *big enough* queries $\mathrm{Q} = \langle m \rangle$ with $m \equiv o[c]$, there are only two possibilities. Either *none* of them are determined by Q (case 1), or *all* of them are (case 2). Thus, deciding the determinacy status of big enough queries becomes much easier: it simply amounts to deciding, for each $o \in \{0, \ldots, c-1\}$, if it behaves as in case 1 or case 2. This is what we call the *asymptotic determinacy picture* of $\mathbf{V}$.

This gives a new perspective on the asymptotic determinacy problem: given a view $\mathbf{V}$, we can first compute the asymptotic determinacy picture of $\mathbf{V}$, and then compute a safety threshold $\alpha(\mathbf{V})$ that ensures that all queries Q that ask for paths longer than $\alpha(\mathbf{V})$ comply to this asymptotic determinacy picture. Then, given a query $\mathrm{Q} = \langle n \rangle$ with $n > \alpha(\mathbf{V})$, it simply remains to check if $n[c]$ is in case 1 or 2, which determines whether $\mathbf{V} \twoheadrightarrow \mathrm{Q}$. Finally, by using Lemma 6.5, we can restrict our attention to C-complete views. Altogether, this discussion shows that Theorem 6.1 is a consequence of the following proposition:

Proposition 6.8. *Given a C-complete view $\boldsymbol{V}$ defined by unions of single path queries, such that $C \in \boldsymbol{V}$ with $C = \langle c \rangle$ for some $c \in \mathbb{N}$ and a natural number $o \in \{0, \ldots, c-1\}$, it is decidable whether there exists a query $Q = \langle n \rangle$ such that $n \equiv o[c]$ and $\boldsymbol{V} \twoheadrightarrow Q$. If this is the case, such a query Q and a first-order rewriting of Q with regards to $\boldsymbol{V}$ can be effectively computed.*

Indeed, if $\mathbf{V}$ is C-complete, the specific α required by Theorem 6.1 can be defined as the function that maps $\mathbf{V}$ to the maximal n given by Proposition 6.8, and is thus computable. Moreover, the first-order rewritings of queries Q' that ask for paths that are longer that n are easily deduced from the rewriting provided by Proposition 6.8 through the use of Lemma 6.7. If $\mathbf{V}$ is not C-complete, then Lemma 6.5 allows us to compute a C-complete view $\mathbf{V}'$ for which the previous argument provides a suitable α'. It then remains to define $\alpha(\mathbf{V}) = k\alpha'(\mathbf{V}')$, where k is the gcd of all numbers that appear in $\mathbf{V}$, as in the proof of Lemma 6.5. Then, given a query $\mathrm{Q} = \langle n \rangle$, with $n \geq \alpha(\mathbf{V})$, we define $\mathrm{Q}' = \langle \frac{n}{k} \rangle$ if k divides n, and apply Proposition 6.8 with $\mathbf{V}'$ and Q', or conclude that $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$ otherwise.

Remark that, in order to produce the complete determinacy picture of $\mathbf{V}$, and thus to solve the general determinacy problem, we would need to compute the smallest query that is determined by $\mathbf{V}$ for each o. We do not know how to solve this challenging task yet, and discuss it further in Section 6.4. Our main tool for proving Proposition 6.8 is introduced in Section 6.2, while the proof itself is the goal of Section 6.3.

6.2 Behavior graphs

This section introduces the main tool of this chapter: *behavior graphs*. To each UPQ view $\mathbf{V}$, we will attach a finite set of behavior graphs $\mathcal{G}_{\mathbf{V}}$. This set $\mathcal{G}_{\mathbf{V}}$ only depends on $\mathbf{V}$, and thus is not attached to any query. We will show in Section 6.3 that it contains enough information to determine both the asymptotic determinacy picture of $\mathbf{V}$ and the safety threshold after which the asymptotic determinacy picture can be used to solve the determinacy problem, as explained in Section 6.1.2. We start in Section 6.2.1 by giving some intuitions as to how $\mathcal{G}_{\mathbf{V}}$ is built, before providing a formal definition in Section 6.2.2.

6.2.1 Intuitions

Assume that we are given a UPQ view $\mathbf{V}$ and an SPQ query $\mathrm{Q} = \langle n \rangle$, and we are asked whether $\mathbf{V}$ determines Q. One very naive way to solve the problem is as follows:

- Enumerate all databases D;
- For each database D, compute $\mathrm{E} = \mathbf{V}(\mathrm{D})$;
- Enumerate all view inverses D' such that $\mathbf{V}(\mathrm{D}') = \mathrm{E}$;
- Test whether $\mathrm{Q}(\mathrm{D}') = \mathrm{Q}(\mathrm{D})$ and:
  - If so: start from the top with the next database D.

– If not: stop here and conclude that $\mathbf{V}$ does not determine Q.

This gives a coRE algorithm for deciding whether $\mathbf{V}$ determines Q. Indeed, if $\mathbf{V}$ does not determine Q, then there exist a pair of databases that gives a counter-example, and the algorithm is guaranteed to find it eventually. Otherwise, the algorithm will of course run forever, as there are infinitely many databases.

However, in the setting of this chapter, we have two major assets to help us refine this very naive algorithm into a workable decision procedure. Namely, we know that σ is reduced to a single symbol and that Lemma 6.2 implies that $\mathbf{V}$ has to contain an SPQ $C = \langle c \rangle$. Together, these two properties make it so that databases and their views have some kind of uniformity and periodicity. Informally, this means that we wont have to enumerate all databases, but only small fragments of them. Moreover, we wont have to care about *where* those fragments occur in the database, as long as we know that they do occur. This will allow us to reduce our problem of enumerating all databases to only enumerating those small fragments that occur in them. This will lead to a finite representation of the infinite set of databases that will still contain all required information to draw the asymptotic determinacy picture.

For now, let us forget about the problem of having a finite representation, and let us focus on the properties that we want to extract from the databases. More precisely, assume that we are given a view image $\mathrm{E} = \mathbf{V}(\mathrm{D})$, and we want to prove that D contains a path π of length n by looking only at E. If D does indeed contain such a path, then the following properties must necessarily hold in E:

C1. E contains the $n+1$ (not necessarily distinct) nodes of π, $x_0, \ldots, x_n$.

C2. For each $V \in \mathbf{V}$ and $u \in L(V)$, $V(x_i, x_{i+u})$ holds in E for all i. In particular, $C(x_i, x_{i+c})$ holds for all i.

C3. For each x in E such that $V(x, x_i)$ holds in E, there exists an appropriate value of k and j such that $C^k(x, x_j)$ holds in E. The values of k and j depend on the witness path that proves $V(x, x_i)$, as shown in Figure 6.2.

If $\mathrm{E} = \mathbf{V}(\mathrm{D})$ does not satisfy these properties, then we can safely conclude that D does not contain a path of length n going from x_0 to x_n. Let us fix $x_0, \ldots, x_n$ and assume that E satisfies (C2) and (C3) for these fixed nodes. One possibility is that these nodes are the consecutive nodes of a path of length n going from x_0 to x_n. Unfortunately, there are many other ways for E to satisfy (C2) and (C3) without D actually having a path of length n from x_0 to x_n, let alone one that goes through all the x_i's in the right order.

We want to quantify how much D differs from the case where the nodes $x_0, \ldots, x_n$ are the consecutive nodes of a simple path. For instance, if there exists some $V \in \mathbf{V}$ such that $u \in L(V)$, our intention is for x_i and x_{i+u} to be linked by a path of length u. However, by looking at E, we only see that $V(x_i, x_{i+u})$ holds, which could mean that x_i and x_{i+u} are linked by a path of length v, for some other $v \in L(V)$. In this case, we say that this path incurs a $(v-u)$ delay.

More precisely, let μ be a path in D from some x_i to some x_j. We define the *delay* of this path as $\delta(\mu) = |\mu| - (j-i)$. $\delta(\mu)$ caracterizes the difference between μ and the

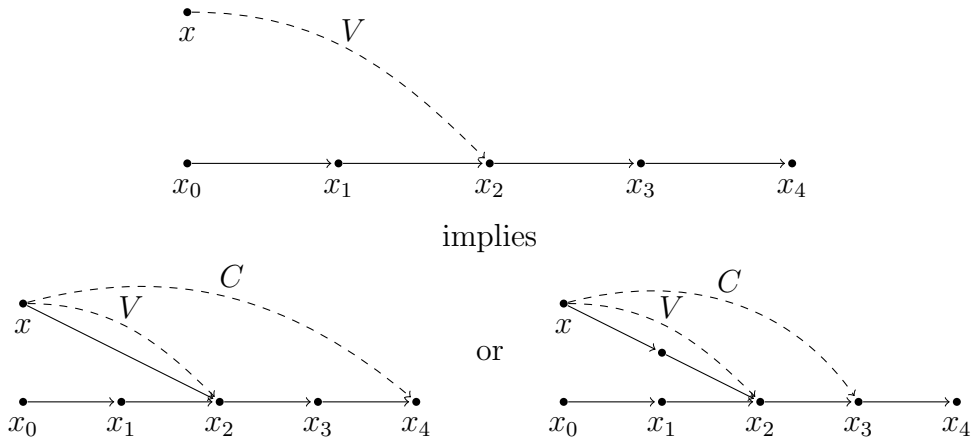


Figure 6.2: Example of possible behaviors for a database (full) and its view (dashed), with $C = \langle 3 \rangle$ and $V = \langle 1, 2 \rangle$. Assume we know the information represented in the top figure. Then, one of the two bottom pictures must hold. More generally, if $C = \langle c \rangle$ and $V(x, x_i)$ holds in E, then $C^k(x, x_j)$ must also hold, with $j = i + (c - v[c])$ and $kc = v + (c - v[c])$ for some $v \in V$.

section of the path of length n going through the x_i's, that we expected to find in D. If μ is of the intended $(j - i)$ length, then its delay will be zero. Otherwise $\delta(\mu)$ can be positive, if μ is longer than intended, or negative, if μ is shorter than intended.

These delays are exactly the information that we want to extract from D. Of course, the number of paths can vary wildly from one database to another, so we will only focus on paths that are implied by the conditions (C2) and (C3). We represent this in a graph H_{D} as explained below. Note that H_{D} is not unique: it depends on the choice of $x_0, \ldots, x_n$, as there can be many multiple quantifications that satisfy conditions (C1), (C2) and (C3).

- H_{D} has $n + 1$ nodes that represent $x_0, \ldots, x_n$, as in (C1). We simply note them $0, \ldots, n$.
- For all $V \in \mathbf{V}$ and $u \in L(V)$, (C2) implies that $V(x_i, x_{i+u})$ holds in E. Hence, there exists a path μ in D going from x_i to x_{i+u} of length v, for some $v \in L(V)$. For each such μ:
  - We represent it as an edge in H_{D} going from i to $i + u$ of label $\delta(\mu) = (v - u)$.
  - For all $u' < v$ such that $u' \in L(V')$ for some $V' \in \mathbf{V}$, we know that $V'(x, x_{i+u})$ holds in E, where x is the u'th predecessor of x_{i+u} along μ. We apply (C3) as shown in Figure 6.3. This leads to a path μ' in D from x_i to $x_{i+u+(c-v'[c])}$ such that $\delta(\mu') = (v - u) + (v' - u')$, for some $v' \in L(V')$. We similarly represent each such μ' in H_{D}.

Assume that there is a path from node 0 to node n in H_{D} whose sum of labels is 0. By composing all the paths in D that led to this path in H_{D}, we prove that there exists

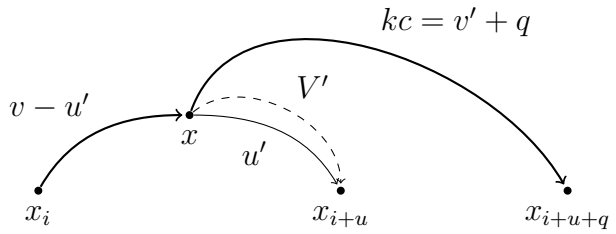


Figure 6.3: Illustration for the existence of the path μ' of delay $(v-u)+(v'-u')$ in the construction of H_{D}. Full arrows represent paths in D and are labeled by their length. Dashed arrows represent edges in E. μ' is the thick path, and $q = c - v'[c]$.

in D a path π from x_0 to x_n such that $\delta(\pi) = 0$. Hence, π is of length n, and we have actually found a path of length n from x_0 to x_n in D.

Consider the case where this is true for all databases D, that is, for all databases D such that $\mathbf{V}(\mathrm{D})$ satisfies the necessary conditions, H_{D} contains such a path. Then all these databases contain a path of length n from x_0 to x_n. This means that the necessary conditions for the existence of a path of length n in D are also sufficient. Since these conditions can be checked by looking only at the view instance, it implies that $\mathbf{V}$ determines $\langle n \rangle$.

At this point, we know that the information for which we are looking lies in the set of all H_{D} for all dabatases D that satisfy the necessary conditions (C1), (C2) and (C3). It remains now to find a finite representation of this set. To do so, we identify in H_{D} all nodes i and j such that $i \equiv j[c]$. Note that this is consistent with the fact that such nodes were already linked by paths of delay 0 thanks to $C \in \mathbf{V}$. These merged graphs are the fragments of databases that were hinted at, at the beginning of this section. In a sense, they describe how a small set of nodes of D are linked with each other. When the query $\mathrm{Q} = \langle n \rangle$ asks for a big enough[1] n, in each database that satisfy the necessary conditions, one of these small sets this will be repeated many times over. This is what will allow us to recompute the global behavior of the database. Thus, when all the small sets have a path whose sum of labels from 0 to $n[c]$ is 0, we will be able to conclude that all databases that satisfy the necessary conditions have a path of length n from x_0 to x_n, which will prove as before that $\mathbf{V}$ determines $\mathrm{Q} = \langle n \rangle$.

In the next section, we make the intuitions given here more precise as we provide a formal definition of these small sets, that we call *behavior graphs*.

6.2.2 Definitions

We start by defining the set $\mathcal{H}_{\mathbf{V}}$ of all *choice graphs* for a given view $\mathbf{V}$, with $C = \langle c \rangle \in \mathbf{V}$. Each of these graphs represent the delay of paths that connect a small set of nodes of a database D whose view image satisfy the conditions (C1), (C2) and (C3) from

[1]Those "big enough" queries are precisely the queries that will lie past the safety threshold of asymptotic determinacy.

Section 6.2.1. Note that we only consider the delays for paths that were implied by the conditions. The reader should refer to the construction of H_{D} from Section 6.2.1 to understand how the labels along the edges of these choice graphs are chosen. Note also that the definition of $\mathcal{H}_{\boldsymbol{V}}$ is made independent from any given database D, as we consider all possibilites.

Definition 6.9 (Choice graph)**.** *Given a C-complete view $\boldsymbol{V}$ such that $C \in \boldsymbol{V}$ with $C = \langle c \rangle$, we define $\mathcal{H}_{\boldsymbol{V}}$ as the set of all directed, edge-labeled graphs H such that:*

1. *H has c nodes, which we will simply note $0, 1, \ldots, c-1$.*
2. *The edges of H carry labels in $\{-2(m-1), \ldots, 2(m-1)\}$, where m is the biggest element that appears in the views, that is $m = \max_{V \in \boldsymbol{V}} \max_{u \in L(V)} u$.*
3. *For each $i, j \in \{0, \ldots, c-1\}$, for each $V \in \boldsymbol{V}$, for each $u \in L(V)$ such that $u \equiv (j-i)[c]$, there exists $v \in L(V)$ such that:*
   - *there is an edge in H from i to j labeled by $v - u$.*
   - *for each $V' \in \boldsymbol{V}$, for each $u' \in L(V')$, there exist $v' \in V'$ and an edge in H from i to $(j - v')[c]$ labeled by $(v - u) + (v' - u')$.*

Remark 6.10. *For a given $\boldsymbol{V}$, the number of nodes and edges of a graph $H \in \mathcal{H}_{\boldsymbol{V}}$ is bounded, thus $\mathcal{H}_{\boldsymbol{V}}$ is finite. Moreover all $H \in \mathcal{H}_{\boldsymbol{V}}$ are complete graphs, because $\boldsymbol{V}$ is C-complete.*

As in Section 6.2.1, the relevant information contained in a choice graph consists of the sums of the labels along the paths of the graph. We call these sums the *weights* of the paths.

Definition 6.11 (Weight)**.** *The weight of a path in a graph H is the sum of all labels along edges of the path. A path with no edge is of weight 0.*

Behavior graphs are similar to choice graphs, with yet another necessary condition added. Assume that a database D is such that $\mathbf{V}(\mathrm{D})$ satisfies the conditions (C1), (C2) and (C3) from Section 6.2.1. Assume then that there exists in D a path μ from x_i to x_j such that $\delta(\mu) \equiv (i-j)[c]$. Then, we deduce that $|\mu| \equiv 0[c]$. This implies that μ appears as a sequence of C edges in $\mathbf{V}(\mathrm{D})$. It turns out that this allows to deduce the existence of more paths of D with the same delay. This is illustrated on Figure 6.4 and is the last requirement for defining behavior graphs.

Definition 6.12 (Behavior graph)**.** *Given a C-complete view $\boldsymbol{V}$ such that $C \in \boldsymbol{V}$ with $C = \langle c \rangle$, we define $\mathcal{G}_{\boldsymbol{V}}$ as the set of all directed, edge-labeled graphs G constructed as follows:*

1. *Pick $H \in \mathcal{H}_{\boldsymbol{V}}$, and start with $G = H$.*
2. *Pick $i, j \in \{0, \ldots, c-1\}$ such that:*

- *There exists in G a path from i to j of weight $(i-j)[c]$. Let a be the weight of a path of minimal length satisfying this property.*
- *For all $a' \equiv a[c]$, there exists i', j' such that $(j'-i') \equiv (j-i)[c]$, and there is no edge from i' to j' of label a'.*

Then, for all i', j' such that $(j'-i') \equiv (j-i)[c]$, add an edge a from i' to j'.

3. *Repeat step 2 until no more edges can be added.*

Remark 6.13.

- *Step 2 of the construction of $\mathcal{G}_{\boldsymbol{V}}$ can only be applied a finite number of times for each G, since it can be done at most once for each (i,j). Moreover, there is a finite amount of choice at each step. Hence, $\mathcal{G}_{\boldsymbol{V}}$ is finite.*
- *As soon as there is a path from some i to some j of weight $(i-j)[c]$, then there is a weight $a \equiv (i-j)[c]$ such that all i', j' that are at the same distance than i is from j are linked by an edge of this particular weight.*

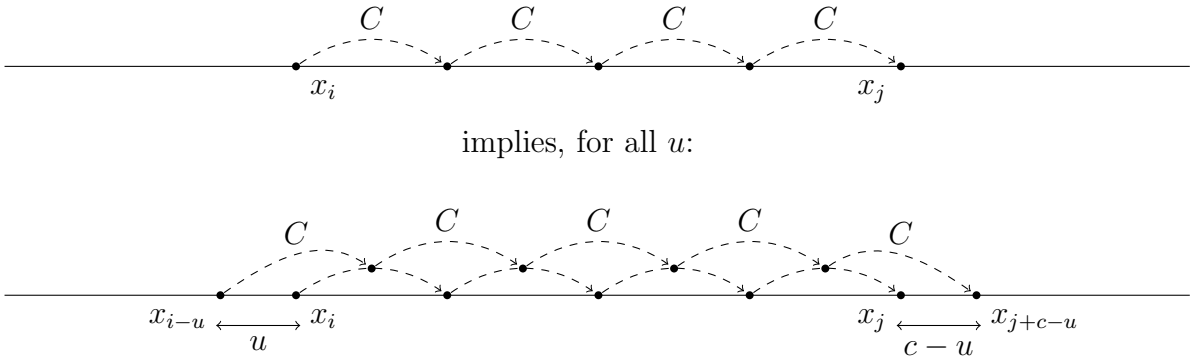


Figure 6.4: Illustration of the intuition for the construction of a behavior graph. Assume that the nodes from x_0 to x_n form a path of length n. If x_i and x_j are connected via a sequence of C's, represented by the dashed edges, then for all $u < c$, there exists some intermediate nodes such that x_{i-u} and x_{j+c-u} are connected as shown in the picture.

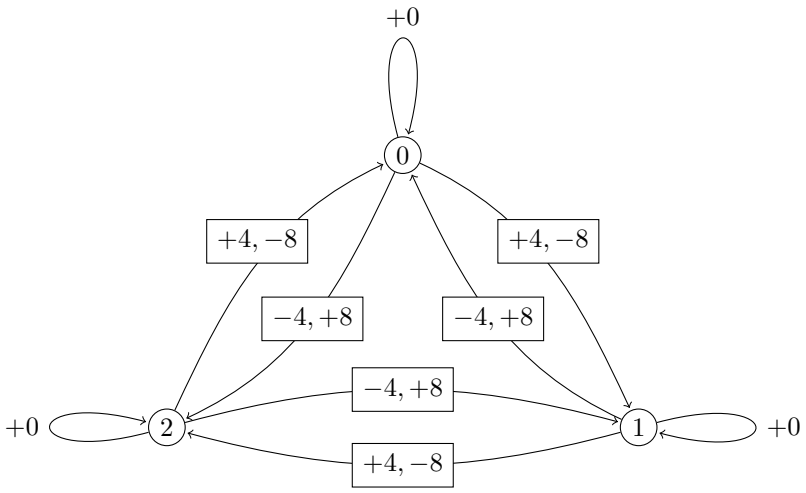
Example 6.14. *Consider the view $\boldsymbol{V} = \{C, V\}$, with $C = \langle 3 \rangle$ and $V = \langle 1, 5 \rangle$. We represent here one of the graphs in $\mathcal{G}_{\boldsymbol{V}}$. Remark that it satisfies the properties of Definition 6.9. Here are some of its features:*

- *It has 3 nodes: 0, 1 and 2;*
- *The edges from 0 to 1 are of weight 4 and -8;*
- *Remark that $(1-0) = 1 \in L(V)$, thus there should exist an edge from 0 to 1 of weight $(1-v)$ for some $v \in L(V)$. This is the case with $v = 5$. In this example, we have a similar situation for the other pairs: (1,2) and (2,0).*

- *Similarly,* $(0-1) \equiv 5[3]$*, thus there should exist an edge from* 1 *to* 0 *of weight* $(5-v)$ *for some* $v \in L(V)$*. Here, it is the edge of weight 4. It is the same for pairs (2,1) and (0,2).*

- *The last condition from Definition 6.9 is harder to see. We just give an example here, using the notations of the definition. For* $i = 0$*,* $j = 1$*,* $u = 1$*,* $v = 5$*,* $u' = 1$ *and* $v' = 5$*, we indeed find an edge from* i *to* $(j - v')[3]$ *of weight* $(v - u) + (v' - u')$*. That is, there is an edge from* 0 *to* 2 *of weight* 8*.*

- *Finally, the additional condition from Definition 6.12 is trivial here. The only paths that satisfy the condition are the cycles, which are all of weight 0 modulo c, so no additional edge needs to be added.*



We now have all the tools to give an overview of the rest of the proof. In Section 6.3, we will show how the intuitions given in Section 6.2.1 formally apply: we will show how the information contained in $\mathcal{G}_{\mathbf{V}}$ for a given UPQ view $\mathbf{V}$ fully determine the asymptotic determinacy picture of $\mathbf{V}$.

More precisely, in Section 6.3.1, we prove that, if there exists a behavior graph $G \in \mathcal{G}_{\mathbf{V}}$ that contains no path of weight 0 from 0 to o, then we can build two databases that have the same view image but disagree on paths of length n, for all $n \equiv 0[c]$. In other words, we can build a database whose view satisfies all the necessary conditions for the existence of a path of length n, while still maintaining a non-zero delay between the relevant nodes. For instance, in the case of Example 6.14, we will use the behavior graph provided to show that $\mathbf{V}$ cannot determine any query $\mathrm{Q} = \langle n \rangle$ where $n \equiv 1[3]$. This is seen on the behavior graph from the fact that it contains no path of weight 0 from 0 to 1.

Conversely, in Section 6.3.2, we show that if all behavior graphs in $\mathcal{G}_{\mathbf{V}}$ contain a path of weight 0 from 0 to o, then there exists some natural number $n \equiv o[c]$ such that $\mathbf{V}$ determines $\mathrm{Q} = \langle n \rangle$. In other words, we will prove that all databases that satisfy the necessary conditions for some nodes $x_0, \ldots, x_n$ have to behave like one of the behavior graphs in $\mathcal{G}_{\mathbf{V}}$. Since all these graphs have a path of weight 0 from 0 to o, we will show

that this implies a path of delay 0 from x_0 to some $x_{o'}$ for some $o' \equiv o[c]$. This in turn will immediately imply a path of delay 0 from x_0 to x_n, which satisfies the query.

Our decision algorithm uses these properties of behavior graphs as follows. For a given C-complete view $\mathbf{V}$ and a given natural number $o \in \{0, \ldots, c-1\}$, we are simply looking for the occurrence of a specific graph $G \in \mathcal{G}_{\mathbf{V}}$, namely one that does not contain a path of weight 0 from node 0 to node o. If we do find one such G, then, for all $n \equiv o[c]$, $\mathbf{V} \not\twoheadrightarrow \langle n \rangle$. Otherwise, $\mathbf{V} \twoheadrightarrow \langle n \rangle$ for some $n \equiv o[c]$. We do not actually need to compute $\mathcal{G}_{\mathbf{V}}$: we simply guess the appropriate graph G and check that it does contain the critical path. Since G is of size polynomial in $\mathbf{V}$ and the considered path, if it exists, can be assumed to be polynomial in the size of G (thanks to Bezout's Identity), our decision procedure is in PSPACE, more precisely in Π_2^{P}.

6.3 Deciding asymptotic determinacy

The goal of this section is to prove Proposition 6.8. Indeed, Proposition 6.8 is an immediate consequence of the two propositions below, that formalize the intuitions of Section 6.2:

Proposition 6.15. *Let $\boldsymbol{V}$ be a C-complete UPQ view such that $C \in \boldsymbol{V}$ with $C = \langle c \rangle$. Let $o \in \{0, \ldots, c-1\}$ be a natural number. Assume that there exists a graph $G \in \mathcal{G}_{\boldsymbol{V}}$ that contains no path of weight 0 from 0 to o. Then, for all $n \equiv o[c]$, $\boldsymbol{V} \not\twoheadrightarrow \langle n \rangle$.*

Proposition 6.16. *Let $\boldsymbol{V}$ be a C-complete UPQ view such that $C \in \boldsymbol{V}$ with $C = \langle c \rangle$. Let $o \in \{0, \ldots, c-1\}$ be a natural number. Assume that all $G \in \mathcal{G}_{\boldsymbol{V}}$ contain a path of weight 0 from 0 to o. Then there exists $n \equiv o[c]$ such that $\boldsymbol{V} \twoheadrightarrow \langle n \rangle$ and we can effectively compute a first-order rewriting that witnesses it.*

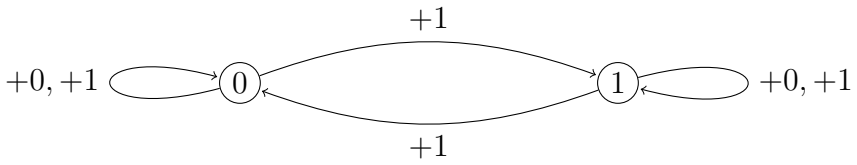
The proofs of these two propositions are the respective goals of Section 6.3.1 and Section 6.3.2.

6.3.1 Negative direction: building counter-examples

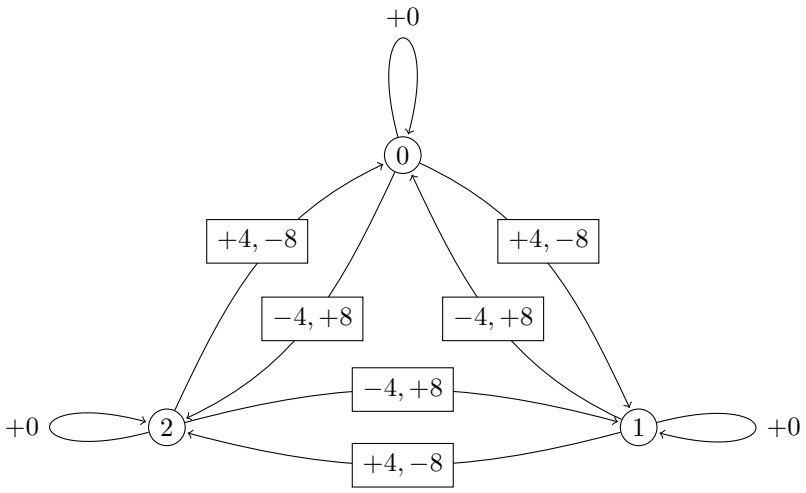
In this section, we prove Proposition 6.15 by showing how to turn a behavior graph with no path of weight 0 from 0 to o into a pair of databases that proves that $\mathbf{V}$ does not determine any query $\mathrm{Q} = \langle n \rangle$ with $n \equiv o[c]$.

For all this section, we fix a C-complete UPQ $\mathbf{V}$ such that $C \in \mathbf{V}$ with $C = \langle c \rangle$, and we assume that there exists a behavior graph in $G \in \mathcal{G}_{\mathbf{V}}$ that contains no path of weight 0 from 0 to o. It turns out that the canonical counter-examples that we build in order to prove that $\mathbf{V} \not\twoheadrightarrow \langle n \rangle$, for any $n \equiv o[c]$, depend dramatically on whether G only contains cycles of positive or negative weights, or if it actually has both. We start by giving examples of both situations.

Example 6.17. *Consider the case where $\boldsymbol{V} = \{C, V\}$, $C = \langle 2 \rangle$ and $V = \langle 1, 2 \rangle$. Then the behavior graph below is one of the graphs in $\mathcal{G}_{\boldsymbol{V}}$. Note that it has no path of weight 0 from 0 to 1, and that all its cycles are of non-negative weights.*



Example 6.18. *Consider again Example 6.14, with* $\boldsymbol{V} = \{C, V\}$, $C = \{3\}$ *and* $V = \{1, 5\}$. *The behavior graph from that example also contains no path of weight* 0 *from* 0 *to* 1, *and contains cycles of both positive and negative weights.*



The two cases are split across Lemma 6.19 and Lemma 6.25. We start with Lemma 6.19 which solves the case where all cycles of G are of the same sign.

Lemma 6.19. *Assume that there exists* $G \in \mathcal{G}_{\boldsymbol{V}}$ *such that* 0 *does not have both a cycle of positive weight and a cycle of negative weight and that there is no path of weight* 0 *from* 0 *to* o. *Then, for all* $n \equiv o[c]$, $\boldsymbol{V} \not\twoheadrightarrow \langle n \rangle$.

Proof. Assume that all cycles of 0 in G have positive or zero weight. An example of such a case is given in Example 6.17.

Let M be the set of all maximum element of each query in $\mathbf{V}$, that is:

$$M = \{\max(L(V)) \mid V \in \mathbf{V}\}$$

Let $d = \gcd(M)$. Remark that d divides c, as c is the maximum (and only) element of $L(C)$.

Claim 6.20. *d does not divide o.*

Assume d divides o. Then, Bezout's Identity provides $u_1, \ldots, u_k \in M$ such that $u_1 + \ldots + u_k \equiv o[c]$. Then, by construction of G, there exists a path from 0 to o whose weight is of the form $(v_1 - u_1) + (v_2 - u_2) + \ldots + (v_k - u_k)$ where each u_i is the maximum element of some $V_i \in \mathbf{V}$, and v_i is an element of the same V_i. Hence, all terms of the sum are negative or zero. If all are zero, then there is a path of weight zero from 0 to

o. Otherwise, there is a path of negative weight from 0 to o, and by applying the same reasoning from o to 0, we get a cycle of negative weight from 0 to 0. Both these cases are false by assumption, which proves the claim.

For each $V_i \in V$, let u_i be the maximum element of $L(V_i)$. By construction of d, we know that d divides u_i. Let n be any natural number such that $n \equiv o[c]$. We can now construct a database D as follows:

- D contains two simple paths of length n, whose respective nodes are $x_0, \ldots, x_n$ and $x'_0, \ldots, x'_n$.
- For each $s, t \leq n$ such that $t - s \in L(V_i)$ for some i, and d does not divide $t - s$, we add to D a new path $\pi^i_{s,t}$ of length $u_i - 2$, and we connect x_s and x'_s to its initial node, and we connect its final node to x_t and x'_t.

We then construct D′, a copy of D, in which x_j and x'_j switch roles for each $j \equiv o[d]$. Note that, since d does not divide o, this means that x_0 and x'_0 are not switched, but x_n and x'_n are. See Figure 6.5 for an example of the construction.

Claim 6.21. *D and D′ agree on $\mathbf{V}$, but each path from x_0 to x_n in D′ is strictly longer than n. Hence D and D′ disagree on Q.*

Let $(x, y) \in V_i(\mathrm{D})$ for some $V_i \in \mathbf{V}$. If either x or y belongs to one of the new paths of the form $\pi^j_{s,t}$, then the symmetry of the construction between the two original simple paths shows that $(x, y) \in V_i(\mathrm{D}')$. Otherwise $x = x_s$ or $x = x'_s$ and $y = x_t$ or $y = x'_t$, for some s and t. Then either $t - s \equiv 0[d]$, in which case either both x and y are switched with their copy in D′, or none are. Then, once again the symmetry of the construction concludes that $(x, y) \in V_i(\mathrm{D}')$. Otherwise, d does not divide $t - s$, which implies that (x, y) are linked by $\pi^i_{s,t}$, and thus that $(x, y) \in V_i(\mathrm{D}')$. Hence, $\mathbf{V}(\mathrm{D}) = \mathbf{V}(\mathrm{D}')$. Remark now that each path from x_0 to x_n has to cross one of the $\pi^j_{s,t}$, which is longer than $t - s$. It follows that each path from x_0 to x_n is longer than n, which proves the claim.

It easily follows from this claim that $\mathbf{V} \not\twoheadrightarrow \langle n \rangle$. The case where 0 only has cycles of negative or zero weight is dealt with in a very similar way, which concludes the proof of the lemma. □

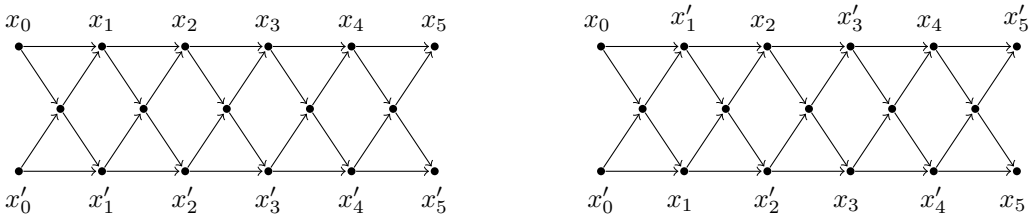


Figure 6.5: Example of the construction in Lemma 6.19 for the view defined in Example 6.17, that is $\mathbf{V} = \{C, V\}$ with $C = \{2\}$ and $V = \{1, 2\}$.

We now move on to the case where G has both positive and negative cycles. It turns out that this proof is a little more involved than the proof of Lemma 6.19, and relies

mainly on the arithmetic properties of behavior graphs. We explore these properties in Lemma 6.22 below.

Lemma 6.22. *Let $G \in \mathcal{G}_{\mathbf{V}}$ such that 0 has both cycles of positive and negative weight. Let W be the set of all weights of cycles of 0 in G, and let $d = \gcd(W)$. Then G has the following properties:*

1. *For all $i, j \in \{0, \dots, c-1\}$ all paths from i to j have the same weight modulo d. We denote this value by $w(i,j)$. Moreover, $w(i,j)$ is compatible with composition. Namely :*
   - $w(i,i) \equiv 0[d]$
   - *For all k,* $w(i,k) + w(k,j) \equiv w(i,j)[d]$
   - $w(i,j) \equiv -w(j,i)[d]$
2. *For all $0 \leq i < j < c \wedge d$, $w(i,j) \not\equiv i - j[c \wedge d]$.*
3. *For all $i \in \{0, \dots, c-1\}$, $w(i, i + c \wedge d) \equiv 0[d]$.*

Proof.

1. By construction of d, we already know that all cycles of 0 have weight $0[d]$. Let $i, j \in \{0, \dots, c-1\}$. Let π_1 and π_1' be two paths from i to j of respective weights w_1 and w_1'. Let π_0 be a path from 0 to i of weight w_0 and π_2 be a path from j to 0 of weight w_2. Then both $\pi_0 \cdot \pi_1 \cdot \pi_2$ and $\pi_0 \cdot \pi_1' \cdot \pi_2$ are cycles of 0. Hence, we have $w_0 + w_1 + w_2 \equiv 0[d]$ and $w_0 + w_1' + w_2 \equiv 0[d]$, which implies $w_1 \equiv w_1'[d]$, so that $w(i,j)$ is correctly defined, as in the statement of the lemma.

   The other properties are easy consequences of this fact.

2. To prove this property, we make use of the following claim:

   Claim 6.23. *For all $i \in \{0, \dots, c-1\}$, for all natural numbers k, there exists $j \in \{0, \dots, c-1\}$ such that $w(i,j) \equiv -j + k[c \wedge d]$.*

   Proof of claim. Let $i \in \{0, \dots, c-1\}$, let k be any natural number. Since $\mathbf{V}$ is complete, there exists $V, V' \in \mathbf{V}$ such that there exist $u \in L(V)$ and $u' \in L(V')$ with $u \equiv 1[c]$ and $u' \equiv i - k[c]$. Then, Property 3 of Definition 6.9 with $i-1$, i, u and u' gives $v \in V$ and $v' \in V'$ such that:

   - $w(i-1, i) \equiv (v - u)[d]$
   - $w(i-1, i-v') \equiv (v-u) + (v' - u')[d]$

   Then it follows that:

$$\begin{aligned}
w(i-1, i-v') &\equiv (v-u) + (v'-u')[c \wedge d] \\
w(i-1, i-v') &\equiv (v-u) + (v'-i+k)[c \wedge d] \\
w(i-1,i) + w(i, i-v') &\equiv (v-u) + (v'-i+k)[c \wedge d] \\
(v-u) + w(i, i-v') &\equiv (v-u) + (v'-i+k)[c \wedge d] \\
w(i, i-v') &\equiv (v'-i) + k[c \wedge d]
\end{aligned}$$

and we conclude the proof of the claim by renaming $v' - i$ to j. □

We can now move on to the proof of the property. Assume by contradiction that there exists i, j such that $w(i, j) \equiv i - j[c \wedge d]$. Then $w(i, j) \equiv i - j + kd[c]$ for some k. Since $\gcd(W) = d$ and W contains both positive and negative elements, by using Bezout's Identity with positive coefficients, we show that there exists a cycle of i of weight $-kd$. Hence, there exists a path from i to j of weight $i - j[c]$. Since this path satisfies the requirement in Definition 6.12, we can show that for all $r \in \{0, \ldots, c-1\}$, $w(r, r + (j - i)) \equiv i - j[c \wedge d]$.

This is a contradiction with the claim. Indeed, the claim implies that for all k, there exists l such that $w(0, l) \equiv -l + k[c \wedge d]$. Since, k can take $c \wedge d$ different values, but $j - i < c \wedge d$, this means that we can find two values $k \neq k'$ for which the claim produces l and l' that are in the same class modulo $j - i$. More precisely, we have $w(0, l) \equiv -l + k[c \wedge d]$, $w(0, l') \equiv -l' + k'[c \wedge d]$, and $l \equiv l'[j - i]$. Hence, there exists some α such that $l' = l + \alpha(j - i)$. By using what we just proved above α times, we get $w(l, l') \equiv \alpha(i - j)[c \wedge d]$. Hence, $w(l, l') \equiv l - l'[c \wedge d]$. Thus, we have:

$$\begin{aligned} w(0, l) &\equiv -l + k[c \wedge d] \\ w(0, l) + w(l, l') &\equiv w(l, l') - l + k[c \wedge d] \\ w(0, l') &\equiv l - l' - l + k[c \wedge d] \\ w(0, l') &\equiv -l' + k[c \wedge d] \end{aligned}$$

This final equality implies that $k = k'$, which is a contradiction.

3. We first prove that $w(0, 0 + c \wedge d) \equiv 0[c \wedge d]$. This is a purely arithmetic consequence of Property 2, that is detailed in Lemma 6.24.

   We rewrite this equality as $w(0, 0 + c \wedge d) \equiv -c \wedge d + kd[c]$. Then, by following the same reasoning as in the proof for Property 2, we prove that there exists a path from 0 to $c \wedge d$ of weight $-c \wedge d[c]$. Since this path satisfies the requirement in Definition 6.12, then there must exist some weight $w \equiv -c \wedge d[c]$ such that, for all $i \in \{0, \ldots, c-1\}$, there is an edge from i to $i + c \wedge d$ of weight w, which we denote by $\pi_{i, i+c\wedge d}$.

   Let $c' = \frac{c}{c \wedge d}$. Then $\pi_{0, c\wedge d} \cdot \pi_{c\wedge d, 2(c\wedge d)} \cdot \ldots \cdot \pi_{(c'-1)(c\wedge d), c'(c\wedge d)}$ is a cycle of 0 of weight $c'w$. Thus, d divides $c'w$. By construction, $c' \wedge d = 1$, hence d divides w, that is, $w \equiv 0[d]$. This implies that for all $i \in \{0, \ldots, c-1\}$, $w(i, i + c \wedge d) \equiv 0[d]$.

□

The proof of Lemma 6.22 uses the following arithmetical result:

Lemma 6.24. *Let $d \in \mathbb{N}$, and $a_1, \ldots, a_k \in \{0, \ldots, d-1\}$. Assume that for all i, j, $a_i + a_{i+1} + \ldots + a_j \not\equiv i - j - 1[d]$. Then there are at most $d - k - 1$ possible values for a_{k+1} such that the sequence $a_1, \ldots, a_{k+1}$ also satisfies this property. In particular, if $k = d - 1$, then there are no possible continuation.*

Proof. Assume everything defined as in the statement of the lemma. Let $i \in \{1, \ldots, k\}$. Then $a_i + \ldots + a_k$ forbids a value for a_{k+1}, that is, a_{k+1} must be chosen so that $a_i + \ldots + a_k + a_{k+1} \not\equiv i - k - 2[d]$.

Assume that $a_i + \ldots + a_k$ forbids $-1[d]$. Then $a_i + \ldots + a_k - 1 \equiv i - k - 2[d]$. Hence, $a_i + \ldots + a_k \equiv i - k - 1[d]$, which is a contradiction.

Assume that there exists $j > i$ such that $a_j + \ldots + a_k$ forbids the same value a. Then

$$a_j + \ldots + a_k + a \equiv j - k - 2[d]$$

Then

$$a \equiv j - k - 2 - a_j - \ldots - a_k[d]$$

But we also have

$$a_i + \ldots + a_k + a \equiv i - k - 2[d]$$

Hence

$$a_i + \ldots + a_{j-1} + j - k - 2 \equiv i - k - 2[d]$$

Finally

$$a_i + \ldots + a_{j-1} \equiv i - j[d]$$

which is a contradiction.

This proves that each $i \in \{1, \ldots, k\}$ forbids a distinct value, which is not -1. Hence, there are $k+1$ forbidden values, and $d-k-1$ remaining possibilites. This concludes the proof. □

We are now ready to give the full proof of Lemma 6.25:

Lemma 6.25. *Assume that there exists $G \in \mathcal{G}_{\boldsymbol{V}}$ such that 0 has both cycles of positive and negative weight, and that there is no path of weight 0 from 0 to o. Then, for all $n \equiv o[c]$, $\boldsymbol{V} \not\twoheadrightarrow \langle n \rangle$.*

Proof. Let $G \in \mathcal{G}_{\mathbf{V}}$ be defined as in the statement of the lemma. An example of such a case is given in Example 6.18. We also define d as in Lemma 6.22.

Let f be the function defined as follows:

$$\forall i \in \{0, \ldots, c \wedge d - 1\},\ f(i) = i + w(0, i)[d]$$

Remark that f is one-to-one. Indeed, assume that i, j are such that $f(i) = f(j)$. Then :

$$\begin{aligned} i + w(0,i) &\equiv j + w(0,j)[d] \\ w(0,i) - w(0,j) &\equiv j - i[d] \end{aligned}$$

And by using Property 1 of Lemma 6.22 we get:

$$w(j, i) \equiv j - i[d]$$

This final equation implies that $i = j$. Otherwise, this would be a contradiction with Property 2 of Lemma 6.22. We can then define g as:

$$\forall i \in \{0, \ldots, d-1\},\ g(i) = i + w(0, i)[d]$$

Property 3 of Lemma 6.22 then gives us:

$$\forall i \in \{0, \ldots, d-1\},\ g(i) = i + w(0, i[c \wedge d])[d]$$

Remark now that g can also be written:

$$\forall i \in \{0, \ldots, d-1\},\ g(i) = f(i[c \wedge d]) + (i - (i[c \wedge d]))[d]$$

from which we deduce that g is one-to-one, because f is. Thus g is a permutation of $\{0, \ldots, d-1\}$.

We can now define two databases D and D$'$ such that D is a cycle of length d whose nodes are $x_0, \ldots, x_{d-1}$, and D$'$ is a copy of D in which x_i is replaced by $x_{g(i)}$ for all i. This is well defined and also a cycle of length d because g is a permutation. See Figure 6.6 for an example of this construction.

Claim 6.26. *D and D' agree on $\boldsymbol{V}$.*

Let x_i and x_j be two nodes of D such that $(x_i, x_j) \in V(\mathrm{D})$ for some $V \in \mathbf{V}$. Then there exists u such that $u \in L(V)$ and $j - i \equiv u[d]$. Let γ be the length of any path from x_i to x_j in D$'$. Then we have:

$$\begin{aligned}
&\gamma \equiv g^{-1}(j) - g^{-1}(i)[d] \\
&\gamma \equiv (j - w(0, j)) - (i - w(0, i))[d] \\
&\gamma \equiv (j - i) - (w(0, j) - w(0, i))[d] \\
&\gamma \equiv u - w(i, j)[d]
\end{aligned}$$

By definition of G, there exists $v \in L(V)$ such that $v - u \equiv w(i, j)[d]$. Hence, we have $\gamma \equiv v[d]$. This means that there is a path from x_i to x_j in D$'$ of length v, and thus $(x_i, x_j) \in V(\mathrm{D}')$. A similar reasoning proves the other direction, and concludes the proof of the claim.

Claim 6.27. *For all $n \equiv o[c]$, $g(n) \not\equiv n[d]$.*

Assume that $g(n) \equiv n[d]$. Then $w(0, n) \equiv 0[d]$. Hence, Property 3 of Lemma 6.22 implies that $w(0, o) \equiv 0[d]$. Hence there exists a path in G from 0 to o of weight kd for some k. Since d is the gcd of all cycles of 0, Bezout's Identity implies that there exists a path from 0 to 0 of weight $-kd$. Hence there exists a path from 0 to o of weight 0, which is a contradiction.

It easily follows from the claim that, for all $n \equiv 0[c]$, $\mathbf{V} \not\twoheadrightarrow \langle n \rangle$. Indeed, the only path of length n starting from x_0 ends in x_n in D, whereas it ends in $x_{g(n)}$ in D$'$. This concludes the proof of the lemma. □

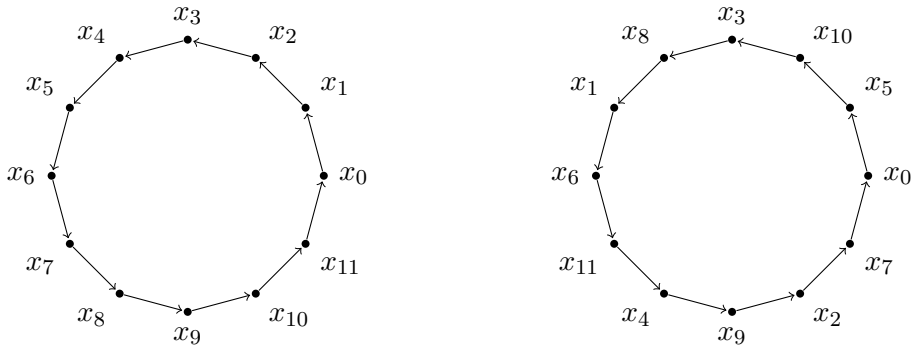


Figure 6.6: Example of the construction in Lemma 6.25 for the view defined in Example 6.18, that is $\mathbf{V} = \{C, V\}$ with $C = \{3\}$ and $V = \{1, 5\}$.

6.3.2 Positive direction: building a rewriting

In this section, we solve the positive case of Proposition 6.8. We start by giving a simple example that shows some of the features of the rewritings that will be used to prove Proposition 6.16.

Example 6.28. *In this example, we work with:*

- $\boldsymbol{V} = \{C, V_1, V_2\}$
- $C = \{2\}$
- $V_1 = \{1, 2\}$
- $V_2 = \{2, 3\}$
- $Q = \{5\}$

We show that $\boldsymbol{V} \twoheadrightarrow Q$. *Indeed,* $R = \langle\varphi\rangle$ *is a rewriting of* Q *using* $\boldsymbol{V}$, *with:*

$$\varphi(x, y) = \exists x_0, \ldots, x_5,\ x_0 = x \wedge x_5 = y \wedge CQ_{\pi_5} \wedge \Big(\forall z, V_1(z, x_3) \Rightarrow (C(z, x_3) \vee C(z, x_4))\Big)$$

where π_5 *is a simple path whose nodes are* $x_0, \ldots, x_5$ *and* CQ_{π_5} *is the conjunctive query that states all the atoms that hold in* $\boldsymbol{V}(\pi_5)$. *First, remark that* R *only states necessary conditions for the existence of a path of length* 5 *from* x *to* y, *as explained in Section 6.2, hence, for all* D, $Q(D) \subseteq R(\boldsymbol{V}(D))$.

Assume now that $(x, y) \in R(\boldsymbol{V}(D))$. *Let* $x_0, \ldots, x_5$ *be a quantification for which* $\varphi(x, y)$ *is satisfied. We can prove the following:*

- $C(x_0, x_2)$, $C(x_1, x_3)$ *and* $C(x_2, x_4)$ *hold in* $\boldsymbol{V}(D)$. *Hence, these pairs of nodes are at distance* 2 *in* D.
- $V_1(x_4, x_5)$ *holds in* $\boldsymbol{V}(D)$. *Hence,* x_4 *and* x_5 *are either at distance* 1 *or* 2. *If this distance is* 1, *then we immediately get a path of length* 5 *from* x_0 *to* x_5 *by using the previous point, as* $x_0 \xrightarrow{2} x_2 \xrightarrow{2} x_4 \xrightarrow{1} x_5$.
- *Similarly,* $V_2(x_0, x_3)$ *holds in* $\boldsymbol{V}(D)$. *If the distance from* x_0 *to* x_3 *is* 3, *we immediately get* $x_0 \xrightarrow{3} x_3 \xrightarrow{2} x_5$. *Otherwise, there exists* z *such that* $x_0 \to z \to x_3$. *This implies* $V_1(z, x_3)$.

- *The remaining case is represented in Figure 6.7, with the two possible implications of $V_1(z, x_3)$ given by φ. Both possibilities also imply a path of length 5 from x_0 to x_5.*

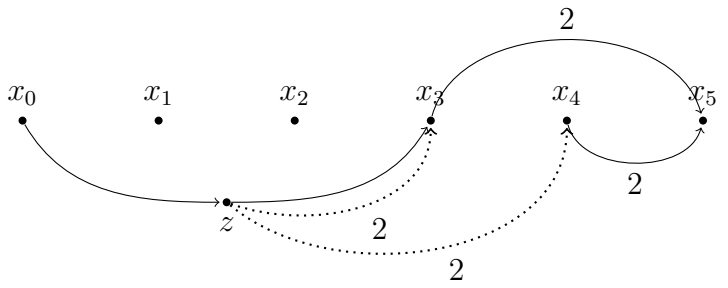


Figure 6.7: Illustration for the last case of Example 6.28. The full edges represent paths in the database, along with their length when it is more than 1. The dotted edges represent the two possible implications of $V_1(z, x_3)$ given by R.

We are now ready to give the proof of Proposition 6.16, that we recall here:

Proposition 6.16. *Let $\boldsymbol{V}$ be a C-complete UPQ view such that $C \in \boldsymbol{V}$ with $C = \langle c \rangle$. Let $o \in \{0, \dots, c-1\}$ be a natural number. Assume that all $G \in \mathcal{G}_{\boldsymbol{V}}$ contain a path of weight 0 from 0 to o. Then there exists $n \equiv o[c]$ such that $\boldsymbol{V} \twoheadrightarrow \langle n \rangle$ and we can effectively compute a first-order rewriting that witnesses it.*

Proof. For each $G \in \mathcal{G}_{\mathbf{V}}$, let π_G be the shortest path in G of weight 0 from 0 to o. Let ρ_G be the longest path that is used to build G from some $H \in \mathcal{H}_{\mathbf{V}}$ and k_G be the number of iterations of step 2 of the definition used to build G from H. Let $k = \max_{G \in \mathcal{G}_{\mathbf{V}}} k_G$. Let $\rho = \max_{G \in \mathcal{G}_{\mathbf{V}}} |\rho_G|$. Let $K = \rho^k$. Let $L = \max_{G \in \mathcal{G}_{\mathbf{V}}} |\pi_G|$. Let $M = 2ck + 3cm$ where m is the biggest number that occurs in one of the views. Let $N = |\mathcal{H}_{\mathbf{V}}|$. Let $n' = K \cdot L \cdot M \cdot N$. Let n be the smallest number such that $n \equiv o[c]$ and $n \geq n'$.

Claim 6.29. *$\boldsymbol{V}$ determines $Q = \{n\}$.*

Let φ_1 be the $n+1$-ary conjunctive formula deduced from $\mathbf{V}(x_0 \dots x_n)$, with $x_0, \dots, x_n$ as free variables, that is, the formula that states all the atoms that hold in $\mathbf{V}(x_0 \dots x_n)$. We also define:

$$\varphi_2(x_0, \dots, x_n) = \forall z, \bigwedge_{i=0}^{n} \bigwedge_{V \in \mathbf{V}} V(z, x_i) \Rightarrow \bigvee_{j=0}^{c} \bigvee_{\substack{u \in V \\ u \equiv j[c]}} C^{\frac{u-j}{c}+1}(z, x_{i+c-j})$$

and:

$$\varphi_3(x_0, \dots, x_n) = \bigwedge_{i,j=0}^{n} \bigwedge_{k=1}^{\frac{n+A}{c}} C^k(x_i, x_j) \Rightarrow \bigwedge_{l=0}^{c-1} C^{k+1}(x_{i-l}, x_{j+c-l})$$

where A is the biggest weight that occurs in a graph in $\mathcal{G}_{\mathbf{V}}$.

Finally, we define $R = \langle\varphi\rangle$ with:

$$\varphi(x,y) = \exists x_0,\dots,x_n,\ x_0 = x \wedge x_n = y \wedge \bigwedge_{i=1}^{3} \varphi_i(x_0,\dots,x_n)$$

Then we can rephrase the previous claim as:

Claim 6.30. *R is a rewriting of $Q = \{n\}$ with regards to $\boldsymbol{V}$.*

Let D be a database, and x and y be two distinguished nodes of D. Assume that $(x,y) \in \mathrm{Q}(\mathrm{D})$. Then there exists a path of length n from x to y. Let $x_0,\dots,x_n$ be the $n+1$, possibly repeating, nodes of this path. Then it is easy to check that $x_0,\dots,x_n$ satisfy φ_1, φ_2 and φ_3 in $\mathbf{V}(\mathrm{D})$. Hence, $(x,y) \in \mathrm{R}(\mathbf{V}(\mathrm{D}))$.

Conversely, assume that $(x,y) \in \mathrm{R}(\mathbf{V}(\mathrm{D}))$. There exists $x_0,\dots,x_n$ such that $x = x_0$, $y = x_n$, and $\varphi_1(x_0,\dots,x_n)$, $\varphi_2(x_0,\dots,x_n)$ and $\varphi_3(x_0,\dots,x_n)$ all hold in $\mathbf{V}(\mathrm{D})$. We define, for all $r < K \cdot L \cdot N$:

$$p_r = \{x_{r\cdot M+ck}, x_{r\cdot M+ck+1}, \dots, x_{r\cdot M+ck+3cm} = x_{(r+1)\cdot M-ck}\}$$

Each p_r is a set of $3cm+1$ consecutive nodes among the x_i's, and all p_r's are disjoint. Additionally, for all path π in D from some x_i to some x_j, we define $\delta(\pi) = |\pi| - (j-i)$. To each p_r we associate[2] a directed edge-labeled graph H_r defined as follows:

- H_r has c nodes which we will simply note $0,1,\dots,c-1$.
- For all $i \in \{0,\dots,c-1\}$, let $\alpha_i = r\cdot M + ck + i + 2cm$. Remark that $\alpha_i \equiv i[c]$. Then, for all $V \in \mathbf{V}$, for all $u \in V$:
  - We pick a path π in D from x_{α_i} to x_{α_i+u} that satisfies V. There exists one, because $\varphi_1(x_0,\dots,x_n)$ holds. Let $v = |\pi|$. We add to H_r an edge from i to $i+u[c]$ labeled $\delta(\pi) = v-u$.
  - For all $V' \in \mathbf{V}$, for all $u' \in V'$, let l be the smallest number such that $l+v \geq u'$ and $l \equiv 0[c]$. Let π_l be a path of D of length l from x_{α_i-l} to x_{α_i}. There exists one because $\varphi_1(x_0,\dots,x_n)$ holds. Let $\pi_0 = \pi_l \cdot \pi$. Let z be the u'th predecessor of x_{α_i+u} along this path. Then $V'(z, x_{\alpha_i+u})$ holds. Then φ_2 implies that there exists $v' \in V'$ and a path π_1 in D from z to $x_{\alpha_i+u+c-(v'[c])}$ of length $v' - (v'[c]) + c$. Hence, there is a path π' from x_{α_i-l} to $x_{\alpha_i+u+c-(v'[c])}$ of length $((l+v)-u'+(v'-(v'[c])+c))$. We then add to H_r an edge from i to $i+u-v'[c]$ labeled $\delta(\pi') = (v-u)+(v'-u')$.

See Figure 6.8 for a visual representation of the various notations.

Claim 6.31.

- *For all r, $H_r \in \mathcal{H}_{\boldsymbol{V}}$.*

[2] arbitrarily, when there are more than one possibility.

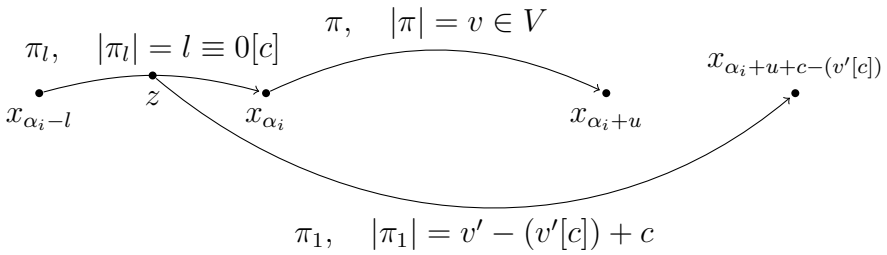


Figure 6.8: Illustration for the various notations in the definition of H_r. An additional, undrawn, information is that the path from z to x_{α_i+u} along the drawn edges is of length u'.

- *For all r, for all edge of label a from i to j in H_r, there exists $x_{i'}$ and $x_{j'}$ in p_r such that there exists a path π in D from $x_{i'}$ and $x_{j'}$ with $\delta(\pi) = a$, $i' \equiv i[c]$ and $j' \equiv j[c]$.*

Since there are $K \cdot L \cdot N$ different p_r's, then there are at least $K \cdot L$ of them that are attributed the same graph $H \in \mathcal{H}_{\mathbf{V}}$. This means that there exists an increasing function f such that, for all $r < K \cdot L$, $p_{f(r)}$ is attributed H. For ease of notations, we rename $p_{f(r)}$ to q_r^0 and H to G_0. Let $G_0, \ldots, G_k$ be k successive iterations as described in Definition 6.12 on G_0. Then $G_k \in \mathcal{G}_{\mathbf{V}}$ and for each s, G_{s+1} is deduced from G_s by doing one iteration of step 2 in Definition 6.12.

Claim 6.32. *For all $s \in \{0, \ldots, k\}$, there exist $\frac{K \cdot L}{\rho^s}$ disjoint sets q_r^s that consist of consecutive nodes among the x_i's such that:*

1. *The distance between q_r^s's last index and q_{r+1}^s's first index is at least $2c(k-s)$. Additionally, the first index of q_0^s is at least $c(k-s)$ and the last index of $q_{\frac{K \cdot L}{\rho^s}}^s$ is at most $n' - c(k-s)$.*

2. *For all r, for all edge of label a from i to j in G_s, there exists $x_{i'}$ and $x_{j'}$ in q_r^s such that there exists a path π in D from $x_{i'}$ and $x_{j'}$ with $\delta(\pi) = a$, $i' \equiv i[c]$ and $j' \equiv j[c]$.*

We prove this claim by induction on s. For $s = 0$, the correctly named q_r^0's already satisfy the required properties.

Assume that, at step s, the properties are true for some q_r^s's. For each $l < \frac{K \cdot L}{\rho^{s+1}}$, let α_l be the first index of $q_{l\rho}^s$ and β_l be the last index of $q_{(l+1)\rho-1}^s$. Then we define q_l^{s+1} as $q_l^{s+1} = \{x_{\alpha_l - c}, \ldots, x_{\beta_l + c}\}$.

It is easy to see that the q_l^{s+1}'s defined as such satisfy Property 1 and also Property 2 for the edges of G_{s+1} that are already in G_s. Let μ be the path in G_s that is used to build G_{s+1} by applying step 2. Then μ is a path of label a and of length $\eta \leq \rho$ and we have $\mu = i_0 a_1 i_1 \ldots i_{\eta-1} a_\eta i_\eta$, where, for all t, there is an edge of label a_t from i_t to i_{t+1} in G_s. Then Property 2 applied at step s implies that for all t there exists a two nodes $x_{i'_t}$ and $x_{i''_{t+1}}$ in $q_{l\rho+t}^s$ such that there exists a path π_t from $x_{i'_t}$ to $x_{i''_{t+1}}$ with $\delta(\pi_t) = a_t$, $i'_t \equiv i_t[c]$ and $i''_{t+1} \equiv i_{t+1}[c]$.

Since $\varphi_1(x_0, \ldots, x_n)$ holds, and for all t $i''_t \equiv i'_t[c]$, then there exists a path π'_t from i''_t to i'_t with $\delta(\pi'_t) = 0$. Hence, we can define π as $\pi = \pi_0 \cdot \pi'_0 \cdot \pi_1 \ldots \pi_{\eta-2} \cdot \pi'_{\eta-2} \pi_{\eta-1}$. π is a

path from $x_{i'_0}$ to $x_{i''_\eta}$ with $\delta(\pi) = \sum a_t = a$, $i'_0 \equiv i_0[c]$ and $i''_\eta \equiv i_\eta[c]$. Then Property 2 is true for all edges of G_{s+1} from i_0 to i_η.

Since $\delta(\pi) = a$, then $\delta(\pi) \leq A$. Hence $|\pi| \leq n+A$. Hence, we can apply $\varphi_3(x_0, \ldots, x_n)$ and get the other required paths. This ends the proof of the claim.

Finally, the claim applied for $s = k$ proves that there exists L sets q_r^k of consecutive x_i's that satisfy property 1 and 2 for some $G_k \in \mathcal{G}_{\mathbf{V}}$. By hypothesis, there exists a path in G_k from 0 to o of length at most L and of weight 0. We conclude by applying once more the reasoning in the proof of the claim. We deduce that there exists two nodes x_i and x_j such that there exists a path π in D with $\delta(\pi) = 0$, $i \equiv 0[c]$ and $j \equiv o[c]$. We complete π with a path π_1 from x_0 to x_i and a path π_2 from x_j to x_n with $\delta(\pi_1) = \delta(\pi_2) = 0$ that are provided by $\varphi_1(x_0, \ldots, x_n)$. Hence $\pi_1 \cdot \pi \cdot \pi_2$ is a path from x_0 to x_n with $\delta(\pi_1 \cdot \pi \cdot \pi_2) = 0$. Thus, the length of $\pi_1 \cdot \pi \cdot \pi_2$ is n, and $(x, y) \in \mathrm{Q}(\mathrm{D})$. This ends the proof of the proposition. □

6.4 Extensions

In the final section of this chapter, we discuss several possible extensions of the work presented here. More precisely, Section 6.4.1 shows the issue that remains to be solved in order to go from asymptotic determinacy to general determinacy. Section 6.4.2 and Section 6.4.3 consider extensions to stronger view and query languages.

6.4.1 The case of small queries

This section is devoted to producing the full determinacy picture for the view below. As the asymptotic determinacy picture of this view is already known thanks to Theorem 6.1, this should highlight what remains to be done in order to decide general determinacy. In all this section, we consider the following view:

- $\mathbf{V} = \{C, V_1, V_2\}$
- $V_1 = \langle 1, 2 \rangle$
- $C = \langle 2 \rangle$
- $V_2 = \langle 2, 5 \rangle$

Claim 6.33. *For all even n, $\boldsymbol{V} \twoheadrightarrow Q = \langle n \rangle$. This easily comes from $C = \langle 2 \rangle$.*

By applying Theorem 6.8 we can show that there exists some odd n such that $\mathbf{V} \twoheadrightarrow \mathrm{Q} = \langle n \rangle$, hence $\mathbf{V}$ also determines all bigger queries. In order to get the full picture, we need to find the smallest odd n that is determined by $\mathbf{V}$. Our work so far actually gives us:

Claim 6.34. *For all odd $n \leq 7$, $\boldsymbol{V} \not\twoheadrightarrow Q = \langle n \rangle$.*

To prove this claim, we use a technique that is very similar to Lemma 6.19. More precisely, the two databases in Figure 6.9 agree on $\mathbf{V}$, but disagree on all $\mathrm{Q} = \langle n \rangle$ when n is odd and not greater than 7.

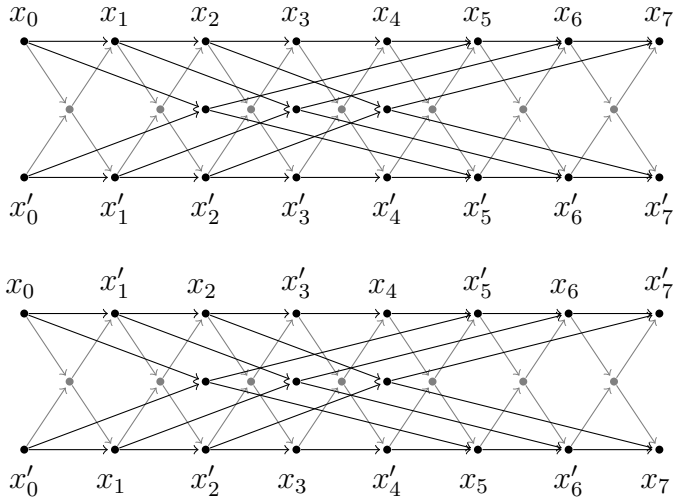


Figure 6.9: The two databases above are a proof that $\mathbf{V} \not\twoheadrightarrow \mathrm{Q} = \langle n \rangle$ for any odd n that is not greater than 7. Indeed, we can check that both databases agree on $\mathbf{V}$. However, there is no path of length 1 (respectively 3, 5 and 7) from x_0 to x_1 (respectively x_3, x_5 and x_7) in the bottom database.

Note that this technique does not work for n greater than 7. Indeed, in the case shown above, any path that goes from x_0 to x_7 in the bottom database has to cross from the top section to the bottom section. By doing so, it suffers a delay of either $+1$ or -3 compared to the expected value. It works here because 7 is "too small" and does not provide enough space to catch-up on this delay. Assume now that $n = 9$, then a delay of -3 can be mitigated by following a $+1$ path three times, and thus does not provide a counter-example.

Claim 6.35. *For all $n \geq 11$, $\boldsymbol{V} \twoheadrightarrow Q = \langle n \rangle$.*

We show this by arguing that $\mathbf{V} \twoheadrightarrow \mathrm{Q} = \langle 11 \rangle$. This is done by actually proving that the canonical rewriting R given in Section 6.3.2 works in this case. Although the proof given in Section 6.3.2 does not apply (because 11 is not "big enough" for all the combinatorial arguments to go through), a careful enumeration of all the possibilities for a database satisfying R actually shows that $\mathrm{R}(x, y)$ implies a path of length 11 from x to y, as was done in Example 6.28.

It is then straightforward to prove that $\mathbf{V}$ determines every odd query bigger than 11. Let $n = 11 + 2k$ be such a query. Then a rewriting for n is simply $\mathrm{R}_{11} \cdot C^k$, as in Lemma 6.7. As we already know that $\mathbf{V}$ determines every even query, this end the proof of the claim.

The case of $n = 9$. There remains only a single unsolved case, which is $n = 9$. This qualifies as a "small query" for the view $\mathbf{V}$: a query for which we are unable to either build a generic counter-example, as in Section 6.3.1, or provide a generic rewriting, as in Section 6.3.2. We actually proved that $\mathbf{V} \not\twoheadrightarrow \mathrm{Q} = \langle 9 \rangle$. However, the smallest counter-example that we know is a pair of databases of 154 nodes each, that were built by hand

through a very tedious trial and error process and checked by a computer program. At this time, we are unfortunately unable to provide any technique to generate such a counter-example for other views and queries. We conjecture that the combinatorial complexity of these "small queries" might be way higher than what we have dealt with so far.

A graphical representation of this counter-example can be seen on Figure 6.10 and Figure 6.11. It was checked by a computer program that these two databases agree on $\mathbf{V}$ but not on $\langle 9 \rangle$.

6.4.2 Infinite unions

In this section, we consider arbitrary path queries, that is path queries that are defined as arbitrary unions of single path queries. Remark that arbitrary path queries are strictly more expressive than regular path queries on a single letter alphabet. For instance, $\mathrm{Q} = \{p \mid p \text{ is prime}\}$ is an arbitrary path queries that is not regular. While considering infinite unions may seem rather strange, this should be understood on a conceptual level, as a way to ease comparisons and extensions to existing work.

Remark also that we do not have any theoretical requirement on the way in which the infinite sets associated with these queries should be represented. Indeed, we will shortly see that when an arbitrary path view determines a single path query, then its finite component already determines the same query. However, for the following construction to be effective, we do require:

- the ability to decide, given a query, whether its associated set is infinite.
- the ability to effectively list all the elements in the associated set, when it is finite.

The main result of this section gives a formal statement to the intuition that infinite unions cannot be used to determine a single path query. This result immediately extends our work to a lot of other view languages on a single-letter alphabet, such as RPQ views, context-free views, and so on, by making their additional expressive power, in comparison to finite unions of path queries, actually be irrelevant.

Lemma 6.36. *Let Q be a single path query and $\boldsymbol{V}$ be a arbitrary path view. Let $\boldsymbol{V} = \boldsymbol{V}_f \uplus \boldsymbol{V}_\infty$, such that $\boldsymbol{V}_f$ only contains queries defined by finite sets, and $\boldsymbol{V}_\infty$ only contains queries defined by infinite sets. Then $\boldsymbol{V} \twoheadrightarrow Q$ if and only if $\boldsymbol{V}_f \twoheadrightarrow Q$.*

Proof. It is easy to see that if $\mathbf{V}_f \twoheadrightarrow \mathrm{Q}$, then $\mathbf{V} \twoheadrightarrow \mathrm{Q}$. Conversely, assume that $\mathbf{V}_f$ does not determine Q. Then there exists two databases D_1 and D_2 such that D_1 and D_2 agree on $\mathbf{V}_f$ but not on Q. Let k be the biggest number that appears in $L(\mathrm{Q}) \cup L(\mathbf{V}_f)$. We transform D_1 into a new database D'_1 as follows:

- We add to D'_1 $k+1$ new nodes $x_0,\ldots,x_k$, as well as the following edges:
  - For all i, $a(x_i, x_{i+1})$ holds in D'_1.
  - $a(x_0, x_0)$ and $a(x_k, x_k)$ hold in D'_1.
- For each original node x of D_1, we add $a(x, x_0)$ and $a(x_k, x)$ to D'_1.

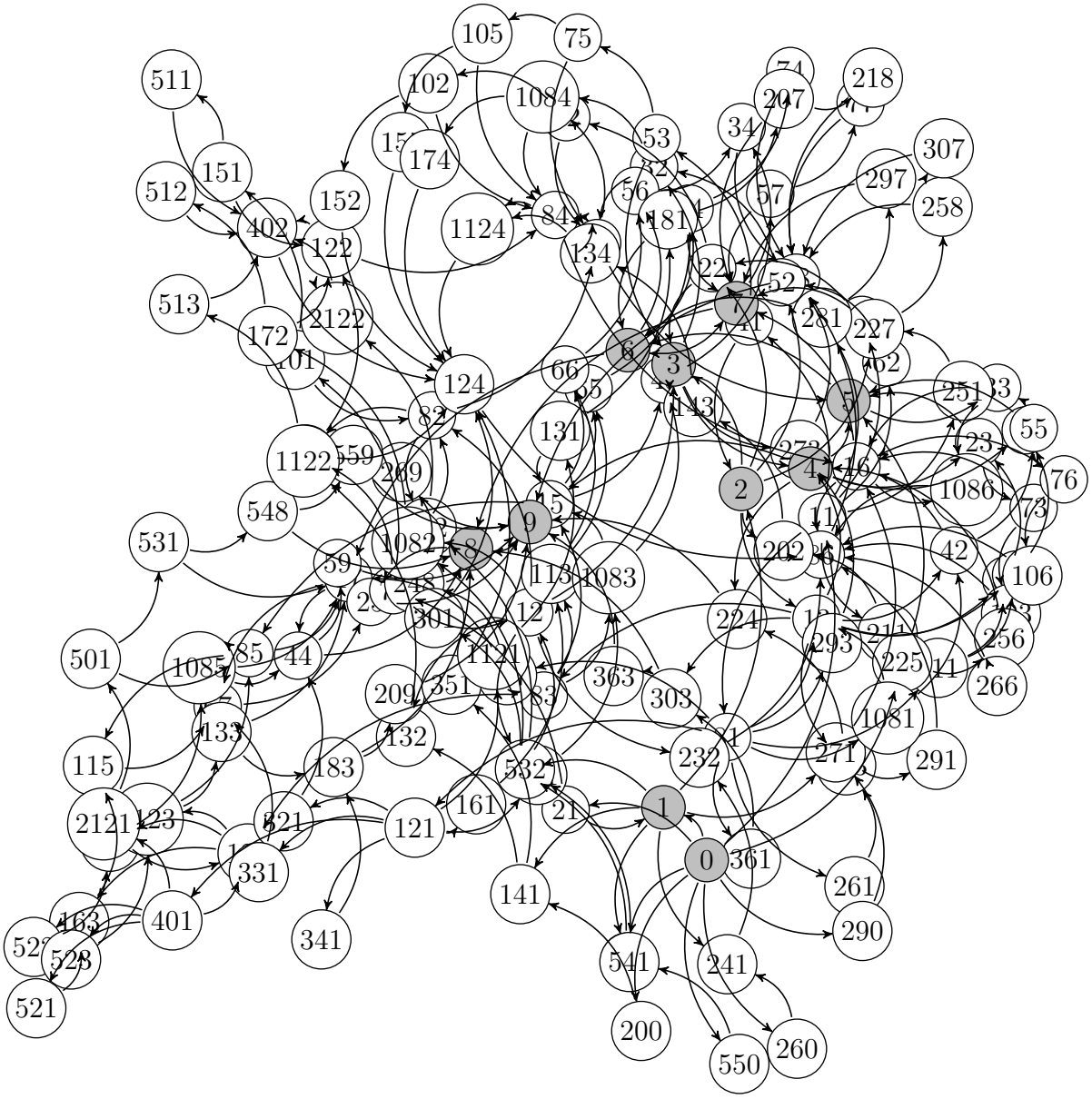


Figure 6.10: One of the two databases that are a proof of $\mathbf{V} \not\twoheadrightarrow \mathrm{Q} = \langle 9 \rangle$. It agrees with the database from Figure 6.11 on $\mathbf{V}$ but not on $\langle 9 \rangle$. This database contains a path of length 9 from node 0 to node 9.

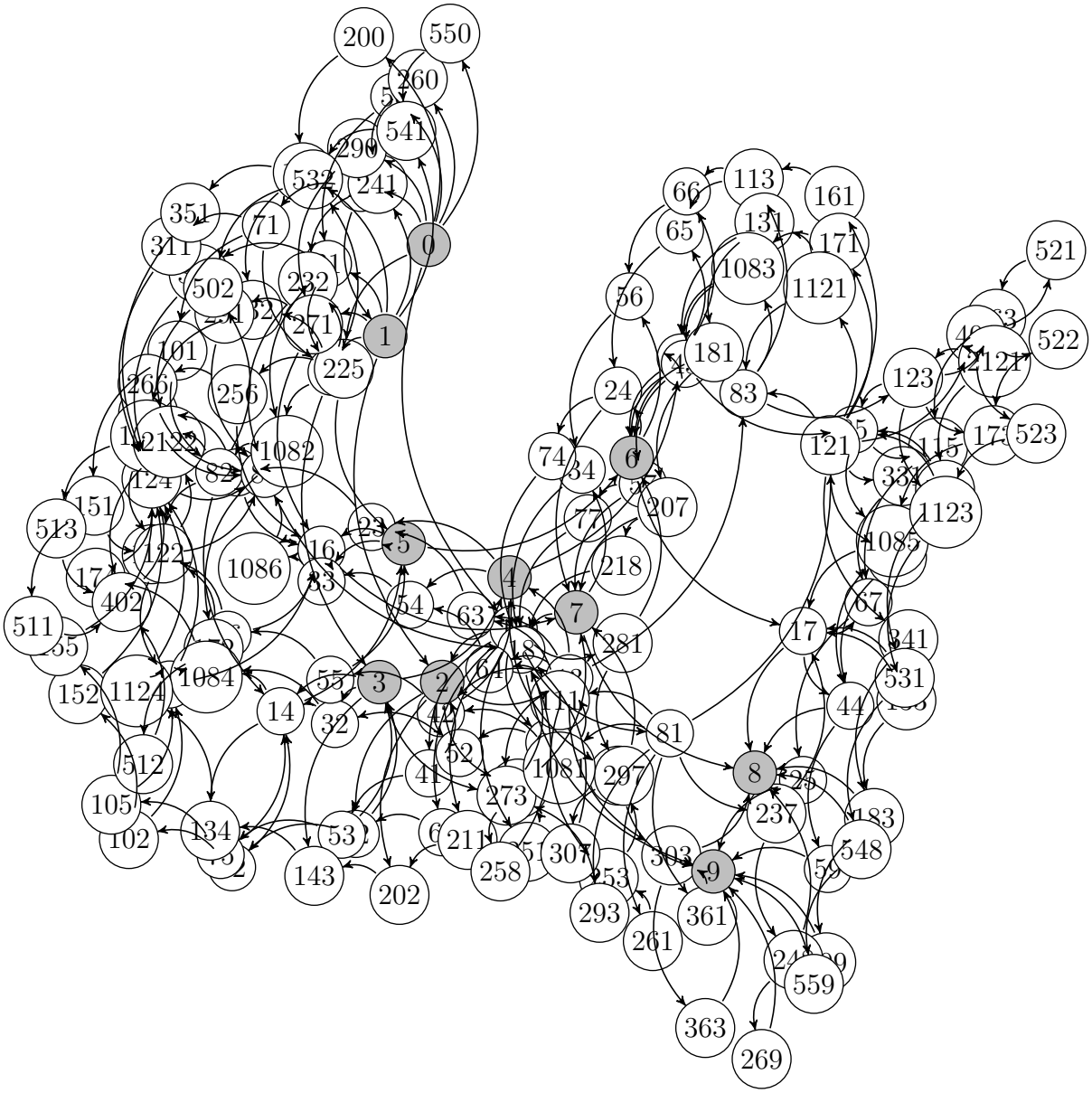


Figure 6.11: One of the two databases that are a proof of $\mathbf{V} \not\twoheadrightarrow \mathrm{Q} = \langle 9 \rangle$. It agrees with the database from Figure 6.10 on $\mathbf{V}$ but not on $\langle 9 \rangle$. This database does *not* contain a path of length 9 from node 0 to node 9.

We then apply the same steps to D_2 and get a new database D'_2. This construction has no effect on Q or $\mathbf{V}_f$ for the original nodes of D_1 and D_2. However, it makes it so that for each $(x, y) \in \mathrm{D}_1$ (respectively D_2), for each $V \in \mathbf{V}_\infty$, $V(x, y)$ holds in $\mathbf{V}_\infty(\mathrm{D}'_1)$ (respectively $\mathbf{V}_\infty(\mathrm{D}'_2)$). Thus, we can check that D'_1 and D'_2 agree on $\mathbf{V}$ but not on Q. Hence $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$, which concludes the proof. □

6.4.3 Multiple labels

In this section, we discuss how some of our results can be extended to cover queries that work over a schema σ with multiples labels. We show how to translate the necessary conditions from Section 6.1.1 when $|\sigma| \geq 2$. We start by giving an analogue to Lemma 6.2, by showing that if $\mathbf{V} \twoheadrightarrow \mathrm{Q}$, then $\mathbf{V}$ contains a single path query. We can say even more: each edge in the graph representation of Q must actually belong to a path that satisfies some single path query in $\mathbf{V}$. While this was the case in the previous setting, it was also trivial then.

Lemma 6.37. *Let $Q = \langle w \rangle$ be a single path query, and $\boldsymbol{V}$ be a view defined by unions of single path queries. Let D consist of the simple path $\pi = x_0 a_0 x_1 \dots x_{n-1} a_{n-1} x_n$ such that $\lambda(\pi) = w$. Assume that $\boldsymbol{V} \twoheadrightarrow Q$. Then for each $0 \leq i < n$, there exists $j \leq i \leq k$ such that $C = \langle a_j \cdot \ldots \cdot a_k \rangle \in \boldsymbol{V}$.*

Proof. This proof is an extension of the proof of Lemma 6.2. Assume by contradiction that there exists $0 \leq i < n$ such that the edge $x_i \xrightarrow{a_i} x_{i+1}$ of D is not used to satisfy any single path query of $\mathbf{V}$. In other words, there is no single path query $C \in \mathbf{V}$ such that $(x_j, x_k) \in C(\mathrm{D})$ for any $j \leq i$ and $k > i$.

We now build a database D_1 as follows:

- D_1 contains the simple path π.
- For each $j \leq i$ and $k > i$ such that there exists $V \in \mathbf{V}$ with $a_j \dots a_k \in L(V)$, we add to D_1 a simple path $\pi_{j,k,V}$ from x_j to x_k such that $\lambda(\pi_{j,k,V}) \in L(V) - \{a_j \dots a_k\}$. Such a path exists because V is not a single path query, by our hypothesis.

We then build a database D_2 that is a copy of D_1 except that $a_i(x_i, x_{i+1})$ does not hold in D_2. Then, it remains to check that $\mathbf{V}(\mathrm{D}_1) = \mathbf{V}(\mathrm{D}_2)$ as was the case in the proof of Lemma 6.2, and that $\mathrm{Q}(\mathrm{D}_1) \neq \mathrm{Q}(\mathrm{D}_2)$. Thus $\mathbf{V} \not\twoheadrightarrow \mathrm{Q}$, which concludes the proof. □

This lemma greatly restricts the form of single path queries Q that can possibly be determined by a given view $\mathbf{V}$. Indeed, a single path query $\mathrm{Q} = \langle w \rangle$ can only be determined by a view $\mathbf{V}$ if w consists of (possibly overlapping) words taken from the single path queries in $\mathbf{V}$ and stitched together. Remark that what makes the problem non-trivial is precisely this overlapping, in the same way that in the single letter case, a view $\mathbf{V}$ defined by unions of single path queries can determine a single path query Q that is not a multiple of the necessary single path query in $\mathbf{V}$.

Next, we remark that Lemma 6.7 immediately translates this setting. Indeed, if $\mathbf{V} \twoheadrightarrow \mathrm{Q}$, then $\mathbf{V}$ necessarily contains a single path query C. Then, we can deduce that $\mathbf{V} \twoheadrightarrow \mathrm{Q} \cdot C^k$, for any k.

There are two main challenges left here. First, it is not clear how asymptotic determinacy should be defined. While the main definition of Section 6.1 is generic enough and does not depend on the size of the alphabet, we do not have the same picture as in Section 6.1.2. Although Lemma 6.7 still states that once a query Q is determined by a view $\mathbf{V}$, then $\mathrm{Q} \cdot C$ is also determined by $\mathbf{V}$ for some single path query $C \in \mathbf{V}$, there also exist arbitrarily big queries Q' that are not of the form $\mathrm{Q} \cdot C$ for any smaller Q and any C, and are thus not covered by this argument.

Second, an implicit argument that is crucial for the arithmetic work done in Section 6.3 is that, when the alphabet is reduced to a single letter, then the resulting monoid is commutative. This makes it so that the order and the exact position of the delays that appear in the behavior graphs of the view are not needed in order to exhibit a path that satisfies the query, as the gaps can always be filled with the path query C, regardless of their position. The setting we consider here behaves differently, and will most likely require building a more complex machinery. We leave this question open for now.

We conclude this section by remarking that this setting is compatible with the result from Section 6.4.2, and thus extends seamlessly to arbitrary unions. Thus, the setting presented here actually covers all classes of path queries, such as regular path queries, context-free path queries, and so on.

Chapter 7

Discussions

In this final chapter, we conclude by discussing the results that were presented throughout this document, how they fit in the general picture and the questions that are left open. In Section 7.1, we come back to the determinacy problem, and sum up its links to other tasks related to view-based query processing. In Section 7.2, we mention interesting results around the CSP problem, and explain their relevance to our work. Finally, in Section 7.3, we take some time to discuss the consequences of the asymptotic determinacy results from Chapter 6 and the questions they open.

7.1 Determinacy and view-based query processing

In Chapter 4, we have presented various view-based computational tasks and we have explained how they relate to each other and to the determinacy problem. In particular, we have shown how the hardness results for computing certain answers translate to hardness results for the view checking and view update problems. In the case of regular path queries and views, which has been of particular importance for our work, this unfortunately implies intractability, as all these problems have either coNP-complete or NP-complete data complexity. The bottom line of this overview is that reasoning about view instances seems to be a particularly difficult task, even for simple view and query languages.

Nonetheless, these complexity results only translate to upper bound for the problem of evaluating the rewriting of a query determined by views. Indeed, for the rewriting problem, we work with the added hypotheses that (1) our inputs are view images instead of general view instances and (2) that the view determines the query. Computing certain answers remains hard even if we assume (1) or if we assume (2). However, when both hypotheses are present, the problem changes from computing certain answers to evaluating the rewriting of a query determined by views, and its data complexity changes significantly: from coNP-complete to NP$\cap$coNP. In Chapter 4, we have explained how the two hypotheses combined together to achieve this new upper bound. This discussion has been the starting point of Chapter 5, which finally led to the PTime data complexity of answering a query using a view image, assuming that the view determines the query in a monotone way, as proved in Theorem 5.23 and the subsequent corollary. We can

also consider an alternative interpretation of hypotheses (1) and (2), by pointing out that they actually mean that we are only interested in computing the set of certain answers when it coincides with the set of *possible answers*, which gives another perspective on the NP $\cap$ coNP upper bound.

Before closing this section, we want to highlight two central determinacy questions that are still open:

Q1: Can we decide determinacy for conjunctive views and queries?

Q2: Can we decide determinacy for regular path views and queries?

7.2 Datalog and the bounded width hierarchy

In Chapter 5, we have made use of the connections between certain answers, local consistency games, Datalog and the constraint satisfaction problem. These connections have been crucial in proving the existence of a Datalog rewriting of a regular path query using a regular path view, assuming monotone determinacy of the query using the view. That being said, these connections also play a crucial role in the study of constraint satisfaction problems themselves.

Indeed, [21] identifies a special class of non-uniform CSPs that enjoy polynomial time evaluation: the problems of bounded width. A CSP is said to be of width (ℓ, k) if it can be solved by the (ℓ, k) local consistency game (as defined in Chapter 5) in the following sense: the accepting instances for the CSP are exactly the instances on which Player 2 has a winning strategy. In other words, for these problems, it is enough to check local consistency in order to ensure global consistency: if there exist partial homomorphisms on sets of k nodes that are consistent on sets of ℓ nodes, then there exists a global homomorphism from the input to the template.

There are several equivalent characterizations of the CSPs of bounded width, see [33] for reference. The characterization that is most relevant to us is in terms of Datalog programs: the CSPs of width (ℓ, k) are exactly the CSPs whose complement can be solved by a Datalog-(ℓ, k) program. When a query is defined using CSPs, as was done in Chapter 5, it is of particular interest to us to know whether the CSP is of bounded width, thus allowing us to express it in Datalog.

Remark that when a CSP is of width (ℓ, k), we can immediately conclude that it is of width (ℓ', k'), for all $\ell' \geq \ell$ and $k' \geq k$. This is what is known as the *bounded width hierarchy*, a classification of the CSPs according to the computational power required to solve them, expressed in terms of the size of the local consistency checks. This hierarchy culminates with the problems of *unbounded width*, for which we know no general polynomial time algorithm. The question whether this hierarchy is strict has received a lot of attention until [8] recently showed that it actually collaspes:

Theorem 7.1 ([8]). *For every relational structure T, precisely one of the following statements is true:*

1. *CSP(T) has width $(1,1)$;*
2. *CSP(T) has width $(2,3)$;*
3. *CSP(T) does not have bounded width.*

In other words, this theorem implies that all CSPs of bounded width actually have width $(2,3)$. In particular, this means that all CSPs that can be solved by a Datalog program can also be solved by a Datalog-$(2,3)$ program.

Let us consider again the proof technique of Theorem 5.23. We have proved that when a regular path view determines a regular path query in a monotone way, then a rewriting can be expressed as the negation of a CSP. Then we have shown that there exists a number ℓ for which the approximation of this CSP in Datalog-$(\ell, \ell+1)$ is actually exact over the set of view images. In light of Theorem 7.1, we might wonder whether there exists a simpler rewriting, in Datalog-$(2,3)$.

Q3: Does the monotone determinacy of a regular path query using a regular path view coincide with the existence of a Datalog-$(2,3)$ rewriting of the query using the view? With the existence of a Datalog-$(2,k)$ rewriting, for some k?

Note that Theorem 7.1 does not immediately apply here. Indeed, the CSP that was built in Chapter 5 has NP-complete data complexity in general, and this remains true even when the view determines the query in a monotone way. However, we are not trying to solve this CSP for all input structures (that is, all view instances), but only for view images. This asks the question whether the results from [8] can be adapted for CSPs of unbounded width that can still be solved by local consistency methods on specific subsets of their inputs. It is worth noting that in all concrete examples that we are aware of, the rewritings can always be expressed as a binary conjunctive regular path query with transitive closure, as in Example 5.10. These rewritings can then easily be expressed as Datalog-$(2,3)$ programs.

7.3 Single path queries and first-order rewritings

Finally, in Chapter 6, we have considered the determinacy problem for path queries and unions of single path views on a single letter alphabet. We have shown that it is decidable whether a UPQ view determines a SPQ query, provided that the query is long enough compared to the view. When this is the case, we have also shown how to compute first-order rewritings of the query using the view. In Section 6.4, we have discussed some natural extensions of our work, as well as the issues that remain to be solved in order for our work to cover the general determinacy problem.

In this section, we want to discuss the case of SPQ queries and UPQ views from the perspective of the rewriting problem. In [3], the author defines the notion of languages

that are *almost complete* for rewritings. A language L is said to be almost complete for rewritings for query language L_Q using view language L_V if, given a view $\mathbf{V}$ in L_V, all but a finite number of queries in L_Q that are determined by $\mathbf{V}$ can be rewritten in L. It is then shown that, on a one letter alphabet, SPQ is almost complete for rewritings of SPQ queries using SPQ views: for each SPQ view $\mathbf{V}$, only a finite number of SPQ queries that are determined by $\mathbf{V}$ require a non-monotone rewriting, while all the other determined queries can be rewritten as SPQ queries over the view. We remark that this no longer holds in the case of UPQ views considered in Chapter 6, where we can easily extend Example 6.28 to show that there exists a UPQ view $\mathbf{V}$ for which arbitrarily long queries that are determined by $\mathbf{V}$ require a non-monotone rewriting.

Let us consider again the statement of Theorem 6.1, from the perspective of the rewriting problem. Remark that it immediately implies that first-order queries are almost complete for rewritings of SPQ queries using UPQ views. This is a new result in the following sense: for now, the best known bound (from [37]) to express rewritings of conjunctive queries using unions of conjunctive views is $\exists\text{SO}\cap\forall\text{SO}$. As of yet, this is the smallest known complete rewriting language that covers UPQ views and SPQ queries. This result makes us wonder whether first-order queries might actually be a complete language for rewritings of SPQ queries using UPQ views.

Q4: Does the determinacy of an SPQ query using a UPQ view coincide with the existence of a first-order rewriting of the query using the view?

A lead towards solving this question is the canonical query used in the positive case of Theorem 6.1. Although in Chapter 6 we were only able to show that this first-order query is a rewriting in the asymptotic cases, it actually turns out that in all concrete examples that we are aware of (even the small ones), when the view determines the query, then this canonical query is a rewriting. Such examples are provided in Example 6.28 and in Section 6.4.1. Remark however that even if this canonical query fails to be a rewriting in all cases, the question whether first-order is a complete rewriting language in this setting remains relevant. Finally, we conclude this section and the whole document by highlighting again the question that has kept us wondering during our work and that remains unsolved as of yet.

Q5: Can we decide determinacy for SPQ queries using UPQ views?

Bibliography

[1] Serge Abiteboul and Oliver M Duschka. Complexity of answering queries using materialized views. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 254–263. ACM, 1998.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*, volume 8. Addison-Wesley Reading, 1995.

[3] Foto N Afrati. Determinacy and query rewriting for conjunctive queries and views. *Theoretical Computer Science*, 412(11):1005–1021, 2011.

[4] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.

[5] Vince Bárány, Michael Benedikt, and Balder Ten Cate. Rewriting guarded negation queries. In *Mathematical Foundations of Computer Science 2013*, pages 98–110. Springer, 2013.

[6] Vince Bárány, Balder Ten Cate, and Luc Segoufin. Guarded negation. In *Automata, Languages and Programming*, pages 356–367. Springer, 2011.

[7] Pablo Barcelo, Leonid Libkin, Anthony W Lin, and Peter T Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems (TODS)*, 37(4):31, 2012.

[8] Libor Barto. The collapse of the bounded width hierarchy. *Journal of Logic and Computation*, 2014.

[9] Catriel Beeri, Alon Y Levy, and Marie-Christine Rousset. Rewriting queries using views in description logics. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 99–108. ACM, 1997.

[10] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Answering regular path queries using views. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 389–398. IEEE, 2000.

[11] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. View-based query processing and constraint satisfaction. In *Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on*, pages 361–371. IEEE, 2000.

[12] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Lossless regular views. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 247–258. ACM, 2002.

[13] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. Rewriting of regular expressions and regular path queries. *Journal of Computer and System Sciences*, 64:443–465, 2002.

[14] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. View-based query containment. In *PODS*, volume 2003, pages 56–67, 2003.

[15] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. View-based query processing: On the relationship between rewriting, answering and losslessness. *Theoretical Computer Science*, 371(3):169–182, 2007.

[16] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[17] Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. In *Database Theory ICDT'97*, pages 56–70. Springer, 1997.

[18] Stavros S Cosmadakis and Christos H Papadimitriou. Updates of relational views. *Journal of the ACM (JACM)*, 31(4):742–760, 1984.

[19] Oliver M Duschka and Michael R Genesereth. Answering recursive queries using views. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 109–116. ACM, 1997.

[20] Wenfei Fan, Floris Geerts, and Lixiao Zheng. View determinacy for preserving selected information in data transformations. *Information Systems*, 37(1):1–12, 2012.

[21] Tomás Feder and Moshe Y Vardi. The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998.

[22] Daniela Florescu, Alon Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 139–148. ACM, 1998.

[23] Nadime Francis. Asymptotic determinacy of path queries using union-of-paths views. In *ICDT*, 2015.

[24] Nadime Francis, Luc Segoufin, and Cristina Sirangelo. Datalog rewritings of regular path queries using views. In *ICDT*, pages 107–118, 2014.

[25] Enrico Franconi and Paolo Guagliardo. On the translatability of view updates. In *AMW*, pages 154–167, 2012.

[26] Enrico Franconi and Paolo Guagliardo. The view update problem revisited. *arXiv preprint arXiv:1211.3016*, 2012.

[27] Tomasz Gogacz and Jerzy Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *Logic in Computer Science (LICS), 2015 30th Annual ACM/IEEE Symposium on*, pages 281–292, July 2015.

[28] Stéphane Grumbach and Leonardo Tininini. On the content of materialized aggregate views. *Journal of Computer and System Sciences*, 66(1):133–168, 2003.

[29] Alon Y Halevy. Theory of answering queries using views. *ACM SIGMOD Record*, 29(4):40–47, 2000.

[30] Alon Y Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[31] Phokion G Kolaitis and Moshe Y Vardi. A logical approach to constraint satisfaction. In *Complexity of Constraints*, pages 125–155. Springer, 2008.

[32] Martin Lange. Model checking propositional dynamic logic with all extras. *Journal of Applied Logic*, 4(1):39–49, 2006.

[33] Benoit Larose and László Zádori. Bounded width problems and algebras. *Algebra universalis*, 56(3-4):439–466, 2007.

[34] Per-Ake Larson and H. Z. Yang. Computing queries from derived relations. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 259–269. VLDB Endowment, 1985.

[35] Alon Y Levy, Alberto O Mendelzon, and Yehoshua Sagiv. Answering queries using views. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM, 1995.

[36] Alan Nash, Luc Segoufin, and Victor Vianu. Determinacy and rewriting of conjunctive queries using views: A progress report. In *Database Theory–ICDT 2007*, pages 59–73. Springer, 2006.

[37] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Transactions on Database Systems (TODS)*, 35(3):21, 2010.

[38] Daniel Pasaila. Conjunctive queries determinacy and rewriting. In *Proceedings of the 14th International Conference on Database Theory*, pages 220–231. ACM, 2011.

[39] Jorge Pérez. *Schema Mapping Management in Data Exchange Systems*. PhD thesis, Pontifica Universidad Católica de Chile, 2011.

[40] Nicole Schweikardt, Thomas Schwentick, and Luc Segoufin. Database theory: Query languages. In *Algorithms and theory of computation handbook*, pages 19–19. Chapman & Hall/CRC, 2010.

[41] Luc Segoufin and Victor Vianu. Views and queries: determinacy and rewriting. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 49–60. ACM, 2005.

[42] Oded Shmueli. Equivalence of datalog queries is undecidable. *The Journal of Logic Programming*, 15(3):231–241, 1993.

[43] Alfred Tarski. Einige methodologifche unterfuchungen über die definierbarkeit der begriffe. *Erkenntnis*, 5(1):80–100, 1935.

[44] Moshe Y Vardi. Constraint satisfaction and database theory: a tutorial. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 76–85. ACM, 2000.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Vues et requêtes sur les graphes de données: déterminabilité et réécritures

Mots-clefs : graphes de données, vues, déterminabilité, réécritures, requêtes de chemin

Résumé : Les graphes de données sont naturellement utilisés dans de nombreux contextes incluant par exemple les réseaux sociaux ou le Web sémantique. L'information contenue dans la base de données se trouve alors aussi bien dans les données mêmes que dans la topologie du graphe, c'est-à-dire dans la manière dont les données sont connectées. Cela implique donc de considérer les questions traditionnelles en théorie des bases de données pour des langages de requêtes capables de parler des chemins connectant les noeuds du graphe.

Nous nous intéressons en particulier aux problèmes de la déterminabilité et de la réécriture d'une requête à l'aide de vues. Il s'agit alors de décider si une vue de la base de données contient suffisamment d'informations pour répondre entièrement à une requête sans consulter la base de données directement, et dans ce cas, d'exprimer explicitement la réponse à la requête à partir de la vue. Ce cadre rencontre de nombreuses applications, notamment pour l'intégration de données et l'optimisation de requêtes.

Nous commençons par comparer ces deux questions aux autres problèmes de décision classiques dans ce contexte : calcul des réponses certaines, test de cohérence et mise à jour d'une instance de vue. Nous améliorons ensuite ces résultats dans deux cas spécifiques. Tout d'abord, nous montrons que pour les requêtes régulières de chemin, l'existence d'une réécriture monotone coïncide avec l'existence d'une réécriture dans Datalog. Puis, nous montrons que pour des vues s'intéressant uniquement aux longueurs des chemins du graphe, une notion plus faible de déterminabilité, appelée déterminabilité asymptotique, est décidable et résulte en des réécritures du premier ordre.

Title : View-based query determinacy and rewritings over graph databases

Keywords : graph databases, views, determinacy, rewritings, path queries

Abstract : Graph databases appear naturally in various scenarios, such as social networks and the semantic Web. In these cases, the information contained in the database lies as much in the data itself as in the topology of the graph, that is, in how the data points are linked together. This leads to considering traditional database theory questions for query languages that return data nodes based on the paths of the graph connecting them.

We focus our attention on the view-based query determinacy and rewriting problems. They ask the question whether a view of the database contains enough information to fully answer a query without accessing the database directly. If so, we then want to express the answer to the query directly with regards to the view. This setting occurs in many applications, such as data integration and query optimization.

We start by comparing these two tasks to other common tasks in this setting: computing certain answers, checking consistency of a view instance and updating it. We then build on these results in two specific cases. First, we show that for regular path queries, the existence of a monotone rewriting coincides with the existence of a rewriting expressible in Datalog. Then, we show that for views that only consider the lengths of the path in the graph, we can decide a weaker form of determinacy, called asymptotic determinacy, and produce first-order rewritings for the queries that are asymptotically determined.