Evaluation and Enumeration of Regular Simple Path and Trail Queries

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von

Tina Popp

aus Bayreuth

1. Gutachter:	Prof. Dr. Wim Martens
2. Gutachter:	Prof. Dr. Peter Wood
3. Gutachter:	Prof. Dr. Pablo Barceló

Tag der Einreichung:	27. Januar 2022
Tag des Kolloquiums:	23. Juni 2022

Abstract

Regular path queries (RPQs) are an essential component of graph query languages. Such queries consider a regular expression r and a directed edge-labeled graph G and search for paths in G for which the sequence of labels is in the language of r. In order to avoid having to consider infinitely many paths, some database engines restrict such paths to paths without repeated nodes or edges which are called *simple paths* or *trails*, respectively. Whereas arbitrary paths can be dealt with efficiently, simple paths and trails become computationally difficult already for very small RPQs.

In this dissertation we investigate decision and enumeration problems concerning simple path and trail semantics.

Evaluation Problem on Directed Graphs: Bagan, Bonifati, and Groz [20] gave a trichotomy for the evaluation problem for simple paths when the RPQ is fixed. We complement their work by giving a similar trichotomy for the evaluation problem for trails and studying various characteristics of this class.

We also study RPQs used in query logs and define a class of simple transitive expressions that is prominent in practice and for which we can prove dichotomies for the evaluation problem when the input language is not fixed, but used as a parameter. We observe that, even though simple path and trail semantics are intractable for RPQs in general, they are feasible for the vast majority of RPQs that are used in practice. At the heart of this study is a result of independent interest: the two disjoint paths problem in directed graphs is W[1]-hard if parameterized by the length of one of the two paths.

Evaluation Problem on Undirected Graphs: While graph databases focus on directed graphs, there are edges which are naturally bidirectional, such as “sibling” or “married”. Furthermore, database systems often allow to navigate an edge in its inverse direction (2RPQ), thus the study of the undirected setting gives us a better idea of what is possible. We are able to identify several tractable and intractable subclasses of regular languages when the input language is fixed. In particular, we establish that trail evaluation for simple chain regular expressions, which are common in practice, is tractable, whereas simple path evaluation is tractable for a large subclass. The problem of fully classifying all regular languages on undirected graphs is quite non-trivial, since it subsumes an intriguing problem that has been open for 30 years. Interestingly, the class of languages that are tractable under simple path semantics on undirected graphs is larger than on directed graphs, while under trail semantics the tractable classes are incomparable (assuming P $\neq$ NP).

We again complement our work using the input language as a parameter. We can show that the tractable subclass of simple transitive expressions on directed graphs is also tractable on undirected graphs, both under simple path and trail semantics. Under trail semantics, the tractable subclass of simple transitive expressions on undirected graphs is a strict superset of the one on directed graphs (under standard complexity assumptions, namely FPT $\neq$ W[1]).

Enumeration: We conclude our work by studying the enumeration setting. In this setting, the goal is to not only decide if a path with certain properties exists, but to output all such paths. Based on Yen's algorithm [209] for enumerating simple paths in directed and undirected graphs, we show that polynomial time algorithms for RPQ evaluation problems give rise to enumeration algorithms with polynomial delay between consecutive answers.

Zusammenfassung

Reguläre Pfadabfragen (RPQs) sind ein wesentlicher Bestandteil von Graphabfragesprachen. Solche Abfragen betrachten einen regulären Ausdruck r und einen gerichteten kantenbeschrifteten Graphen G und suchen nach Pfaden in G, deren Abfolge von Kantenbeschriftungen ein Wort in der Sprache von r ergibt. Um zu vermeiden, dass unendlich viele Pfade berücksichtigt werden müssen, beschränken sich einige Datenbank-Systeme auf Pfade ohne Wiederholungen von Knoten oder ohne Wiederholungen von Kanten, sogenannte *einfache Pfade* oder *Trails*. Während beliebige Pfade effizient behandelt werden können, werden einfache Pfade und Trails schon bei sehr kleinen RPQs rechnerisch schwierig.

In dieser Dissertation untersuchen wir Entscheidungs- und Aufzählungsprobleme bezüglich der Semantik von einfachen Pfaden und Trails.

Evaluationsproblem auf Gerichteten Graphen: Bagan, Bonifati und Groz [20] fanden eine Trichotomie für das Evaluationsproblem für einfache Pfade, wenn der RPQ fest ist. Wir ergänzen ihre Arbeit, indem wir eine ähnliche Trichotomie für das Evaluierungsproblem für Trails angeben und verschiedene Eigenschaften dieser Klasse untersuchen.

Desweiteren untersuchen wir RPQs, die in Abfrageprotokollen (englisch: query logs) vorkommen, und definieren eine Klasse einfacher transitiver Ausdrücke, die in der Praxis häufig vorkommt und für die wir Dichotomien für das Evaluationsproblem beweisen können, wenn die Eingabesprache nicht fest ist, sondern als Parameter verwendet wird. Wir stellen fest, dass, obwohl einfache Pfad- und Trailsemantiken für RPQs im Allgemeinen schwer sind, die zugehörigen Evaluationsprobleme für die große Mehrheit der in der Praxis verwendeten RPQs effizient lösbar sind. Im Mittelpunkt dieser Studie steht ein Ergebnis von unabhängigem Interesse: Das Problem zwei disjunkte Pfade in einem gerichteten Graphen zu finden ist W[1]-schwer, wenn es durch die Länge eines der beiden Pfade parametrisiert wird.

Evaluationsproblem auf Ungerichteten Graphen: Während sich Graphendatenbanken auf gerichtete Graphen konzentrieren, gibt es Kanten, die von Natur aus bidirektional sind, wie “Geschwister” oder “verheiratet” Relationen. Außerdem erlauben Datenbanksysteme oft eine Kante in ihrer umgekehrten Richtung zu navigieren (2RPQ), so dass die Untersuchung der ungerichteten Umgebung uns eine bessere Vorstellung davon vermittelt, was möglich ist. Wir sind in der Lage, mehrere effizient und nicht effizient lösbare Unterklassen der regulären Sprachen zu identifizieren, wenn der RPQ fest ist. Insbesondere stellen wir fest, dass das Evaluationsproblem für Trails für eine Teilklasse regulärer Ausdrücke, die in der Praxis häufig vorkommen, effizient lösbar ist, während das

Evaluationsproblem einfacher Pfade für eine große Unterklasse davon effizient lösbar ist. Das Problem der vollständigen Klassifizierung aller regulären Sprachen auf ungerichteten Graphen ist nicht trivial, da es ein faszinierendes Problem umfasst, das seit 30 Jahren offen ist. Interessanterweise ist die Klasse der Sprachen, die unter einfacher Pfadsemantik auf ungerichteten Graphen effizient lösbar sind, größer als auf gerichteten Graphen, während unter Trailsemantik die effizient lösbaren Klassen (unter der Annahme $\mathrm{P} \neq \mathrm{NP}$) nicht vergleichbar sind.

Wir ergänzen unsere Arbeit erneut, indem wir die Eingabesprache als Parameter verwenden. Wir können zeigen, dass die effizient lösbare Unterklasse der einfachen transitiven Ausdrücke auf gerichteten Graphen auch auf ungerichteten Graphen effizient lösbar ist, sowohl unter einfacher Pfad- als auch unter Trailsemantik. Desweiteren können wir zeigen, dass unter Trailsemantik die effizient lösbare Unterklasse der STEs auf ungerichteten Graphen eine strikte Obermenge der entsprechenden Klasse auf gerichteten Graphen ist (unter der komplexitätstheoretischen Annahme $\mathrm{FPT} \neq \mathrm{W}[1]$).

Aufzählung: Wir schließen unsere Arbeit ab, indem wir das Aufzählungsproblem untersuchen. Hier geht es nicht nur darum zu entscheiden, ob ein Pfad mit bestimmten Eigenschaften existiert, sondern auch darum, alle solchen Pfade auszugeben. Basierend auf Yens Algorithmus [209] zur Aufzählung einfacher Pfade in gerichteten und ungerichteten Graphen zeigen wir, dass Polynomialzeitalgorithmen für die vorher betrachteten Evaluationsprobleme zu Aufzählungsalgorithmen mit polynomieller Zeit zwischen aufeinanderfolgenden Ausgaben führen.

Acknowledgements

First and foremost I am extremely grateful to my supervisor Prof. Dr. Wim Martens for his invaluable advice, continuous support, and patience during my PhD study. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. Thanks to him and the University of Bayreuth I was able to gain valuable experience in teaching, attend and participate in several international conference meetings, and immerse myself in research.

I am grateful to Prof. Dr. Peter Wood, Prof. Dr. Pablo Barceló, Prof. Dr. Jörg Müller and Prof. Dr. Christian Knauer for taking part in the examination commission reviewing my thesis. I would also like to acknowledge Susanne Süss for patiently answering all my questions regarding the formalities required for this thesis, and Dr. Johannes Doleschal and Gregor Sönnichsen for helpful comments on an earlier version of this thesis.

I would like to extend my sincere thanks to my coauthors, colleagues, and all the other researchers and people I met on the way. Your comments, suggestions, and discussions were instructive and inspiring. Among them, I especially want to mention my colleague and coauthor Dr. Matthias Niewerth, who would always have some space on his white-board to discuss proof ideas.

On a personal note, I wish to thank my family and friends for their support and encouragement throughout the years. Thank you for checking in so often and brightening my days. I also want to thank my husband Matthias Popp for always being there for me, both in good and sad moments.

Contents

Introduction

Chapter 1

Graph Databases and Regular Path Queries

Graph databases are a popular tool to model, store, and analyze data [81, 167, 173, 201, 204] and are rapidly gaining importance [184]. They are engineered to make the *connectedness of data* easier to analyze. This is indeed a desirable feature, since some of today's largest companies have become so successful because they understood how to use the connectedness of the data in their specific domain (for example, Web search and social media). One aspect of graph databases is to bring tools for analyzing connectedness to the masses.

Regular path queries (RPQs) are a crucial component of graph databases, because they allow reasoning about arbitrarily long paths in the graph and, in particular, paths that are longer than the size of the query. We give an example. Consider the toy graph database in Figure 1.1, which is loosely inspired on a part of the Wikidata graph. If we want to find *artists*, we can easily do so using a regular path query. For instance, we can retrieve the persons who are artists with a Cypher-like query of the form

```
CONSTRUCT (x)
MATCH   (x:Person)-[:occupation]->()-[:subclassof*]->(y:Profession)
WHERE    y.name = 'artist'
```

This query asks for persons whose occupation is a profession that is connected with a `subclassof`-path to "artist". Here, we used the regular expression `subclassof`* to allow arbitrarily long directed paths in which every edge is labeled with `subclassof`. Since we may not know in advance how many `subclassof`-edges we have to consider, it is very comfortable to be able to use the regular path query `subclassof`*. The example also illustrates the robustness of regular path queries. Even when the graph database changes (for example, by introducing an additional profession such as "string instrumentalist"), the query still returns the correct results.

RPQs started as an academic idea in Cruz et al.'s seminal paper [74] and are nowadays part of SPARQL, Cypher and Oracle's PGQL. Although the main idea behind RPQs is always to match regular expressions against paths in a graph database, academic research and real-world systems do not always agree on how this should be done. Already the return type of the query varies: whereas most academic research on RPQs [16, 26, 29, 146, 160] and SPARQL [116] focuses on the first and last node of matching paths, Cypher [172]

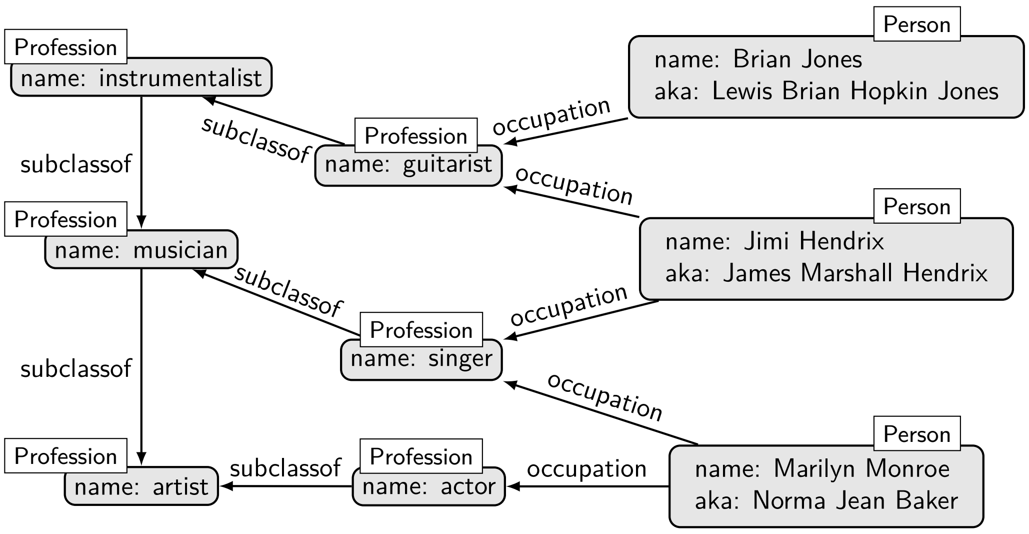


Figure 1.1: A graph database (as a *property graph*), inspired by a fragment of WikiData. A similar graph was used in [77].

returns the entire paths. Another difference lies in which paths should be considered for matching, and the most considered candidates are *all paths* or *paths without repeated nodes or edges*. Whereas academic research most commonly allows all paths (which allow polynomial time algorithms to test if a matching path exists between two given nodes), graph database systems usually revert to paths without repeated nodes or edges. There seem to be different reasons for this. First of all, this restriction always ensures that the number of paths that can match is finite, so one does not have to deal with infinity. Second, paths without repeated nodes or edges give the semantics that some users seem to prefer [Lindaaker, personal communication].

We illustrate this with an example. If we want to know which other occupation singers have, we can execute the following two-way navigational Cypher-like query on the graph in Figure 1.1.

```
CONSTRUCT (x)
MATCH (x:Profession)<-[:occupation]-(:Person)-[:occupation]->(z:Profession)
WHERE  z.name = 'singer'
```

This query asks for a profession that is connected with a reverse `occupation`-edge to a person who has the occupation “singer”. If we only allow simple paths or trails, then this query returns guitarist and actor. On the other hand, if we allow arbitrary paths, then the query additionally returns singer, which might be counter-intuitive. Therefore, it seems natural to allow different modes and leave the choice to the user.

Indeed, industry and academia [109] collaborated to create the graph query language GQL [84, 108] which is currently going through ISO standardization. While the GQL project is directly influenced[1] by academic work on regular queries [180] and GXPath [145], it builds heavily on GCore [10], which is a light-weight query language for property graphs, developed by partners in academia and industry. Indeed, in the most recent publicly available documents on the development of GQL [84], they use restrictions to simple paths or trails as means to ensure only finitely many matches and therefore termination.

Whereas unrestricted and shortest paths are fairly well understood, this is much less so for trails and simple paths. In fact, this situation is a discrepancy between research and industry, since research has a plethora of results on unrestricted (conjunctions of) RPQs [26, 28, 30, 62, 64, 85, 94, 96, 100, 160, 183], even with two-way navigation [27, 45, 46, 60, 61, 93, 181] and other extensions [28, 101, 145, 191], whereas current industrial graph database engines like Neo4j default to trails or simple paths [167].

Although simple path evaluation has already been studied in the late 80's and mid 90's [74, 160], these early results showed that even relatively simple expressions like $(aa)^*$ or a^*ba^* are NP-hard to evaluate. After these results, research on RPQ evaluation mainly focused on unrestricted paths. A renewed interest in trails and simple paths [16, 20, 146] was pushed by graph query language standards and systems, but is still lagging behind. Since trail and simple path evaluation are planned to be included in the upcoming GQL standard relatively soon, there is an urgent need for fundamentally understanding these modes.

1.1 Structure of this Thesis

In Part I we focus on directed graphs. In Chapter 2, we formally define the basic notions that are used throughout the first part of the thesis. In Chapter 3, we study the *data complexity* of RPQ evaluation under trail semantics. That is, we study variants of RPQ evaluation in which the RPQ r is considered to be fixed. As such, the input of the problem only consists of an edge-labeled (multi-)graph G and a pair (s, t) of nodes, and the task is to determine if there exists a trail from s to t on which the sequence of labels matches r. One of our main results is a trichotomy on the RPQs for which this problem is in AC^0, NL-complete, or NP-complete, respectively. By $\mathsf{T}_{\mathsf{tract}}$, we refer to the class of tractable languages (assuming NP $\neq$ NL).

In order to increase our understanding of $\mathsf{T}_{\mathsf{tract}}$, we study several important aspects of this class of languages. A first set of results is on characterizations of $\mathsf{T}_{\mathsf{tract}}$ in terms of closure properties and syntactic and semantic conditions on their finite automata. In a second set of results, we therefore compare the expressiveness of $\mathsf{T}_{\mathsf{tract}}$ with yardstick languages such as $\mathbf{FO}^2[<]$, $\mathbf{FO}^2[<,+]$, $\mathbf{FO}[<]$ (or *aperiodic languages*), and $\mathsf{SP}_{\mathsf{tract}}$. The latter class, $\mathsf{SP}_{\mathsf{tract}}$, is the closely related class of languages for which the data complexity of RPQ evaluation under *simple path* semantics is tractable.[2] Interestingly, $\mathsf{T}_{\mathsf{tract}}$ is strictly larger than $\mathsf{SP}_{\mathsf{tract}}$ and includes languages outside $\mathsf{SP}_{\mathsf{tract}}$ such as a^*bc^* and $(ab)^*$

[1]This can be seen in the GQL influence graph [109].

[2]Bagan et al. [20] called the class $\mathsf{C}_{\mathsf{tract}}$, which stands for "tractable class". We distinguish between $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$ here to avoid confusion between simple paths and trails.

that are relevant in application scenarios in network problems, genomic datasets, and tracking provenance information of food products [175] and were recently discovered to appear in public query logs [53, 55]. Furthermore, every *single-occurrence regular expression* [41] is in $\mathsf{T}_{\mathsf{tract}}$, which can be a convenient guideline for users of graph databases, since *single-occurrence* (every alphabet symbol occurs at most once) is a very simple syntactical property. It is also popular in practice: we analyzed the 50 million RPQs found in the logs of [54] and discovered that over 99.8% of the RPQs are single-occurrence regular expressions.

We then study the recognition problem for $\mathsf{T}_{\mathsf{tract}}$, that is: given an automaton, does its language belong to $\mathsf{T}_{\mathsf{tract}}$? This problem is NL-complete (respectively, PSPACE-complete) if the input automaton is a DFA (respectively, NFA). We also treat closure under common operations such as union, intersection, reversal, quotients and morphisms.

In Chapter 4 we take into account the types of expressions occurring in the query logs of the study by Bonifati et al. [53], and define the class of *simple transitive expressions (STEs)*, which capture over 99.99% of the expressions in the logs. The remainder of the expressions are *unions of STEs*, except for one single expression. In Chapter 5 we want to take a closer look at the evaluation problem of STEs. For arbitrary and shortest paths it is again known to be tractable, thus we first consider simple path semantics. This problem is challenging because it contains special cases that are quite non-trivial. One such case is testing if there exists a directed simple path of length exactly $\log n$ between two given nodes in a directed graph with n nodes, which was shown to be in P by Alon et al., using their color coding technique [7]. The question if it can be decided in P if there is a simple path of length $\log^2 n$ has been open since 1995 [7]. Notice that these two problems are special cases of RPQ evaluation under simple path semantics (that is, evaluate the RPQs $a^{\log n}$ and $a^{\log^2 n}$ in a directed graph where every edge has label a).

We therefore investigate RPQ evaluation from the angle of parameterized complexity where we use the size of the RPQ as parameter in Chapter 5. We identify a property of simple transitive expressions that we call *cuttability* and prove a dichotomy, showing that the parameterized complexity for evaluating a class $\mathcal{R}$ of STEs is in FPT if $\mathcal{R}$ is cuttable and W[1]-hard otherwise. Examples of cuttable classes of expressions are $\{a^k a^* \mid k \in \mathbb{N}\}$ and $\{(a+b)^k a^* \mid k \in \mathbb{N}\}$. Examples of non-cuttable classes are $\{a^k b^* \mid k \in \mathbb{N}\}$, $\{a^k b a^* \mid k \in \mathbb{N}\}$, and $\{a^k (a+b)^* \mid k \in \mathbb{N}\}$.

We then turn to trail semantics and prove a dichotomy similar to the one for simple path semantics. Here we show that, if a class $\mathcal{R}$ of STEs is *almost conflict-free*, the parameterized complexity of evaluation for $\mathcal{R}$ is in FPT and W[1]-hard otherwise. It should be noted that every cuttable class of expressions is also almost conflict-free, which makes evaluation under trail semantics slightly "easier" than under simple path semantics.

At the core of the dichotomies are two results of independent interest (Sections 5.3 and 5.4). The first is by the authors of [97], who showed that it can be decided in FPT if there is a simple path of length *at least* k between two nodes in a graph (Theorem 5.3.5). The second is proved in this thesis and states that the Two Disjoint Paths problem in directed graphs is W[1]-hard when parameterized by the length of one of the two paths (Theorem 5.4.7).

In Part II we turn to evaluation of RPQs on undirected graphs. Studying simple path and trail evaluation on undirected graphs is important since today's leading graph database engines and query languages allow undirected (aka bidirectional) edges and two-way navigation.[3] In Chapter 6 we obtain a number of closure and non-closure properties of the tractable classes of languages. Furthermore, we can show that the tractable classes of languages on undirected graphs both subsume $\mathsf{SP}_{\mathsf{tract}}$, while they are both incomparable to $\mathsf{T}_{\mathsf{tract}}$. We present a dichotomy on a generalization of the undirected two disjoint path problem with edge labels. We use this dichotomy to fully classify the complexity of trail and simple path evaluation of languages of the form A^*wB^* on undirected graphs, where A and B are sets of symbols and w is a word. We fully classify the complexity of languages of the form w^*, except in the case where this problem degenerates to testing the length of paths modulo some $k > 2$, which is an open problem since 1991 [17]. We study *simple chain regular expressions (SCREs)*, which is a class of practically common languages and show that their trail evaluation is always tractable. This, however, is not the case for simple path evaluation, but we are able to identify large tractable subclasses. We generalize recent results on group-labeled graphs to obtain that the data complexity of *parity languages* is tractable for both trail and simple path evaluation.

In Chapter 7 we again study the RPQ evaluation from the angle of parameterized complexity. We can prove that the tractability results from Chapter 5 also hold on undirected graphs. Furthermore, we identify a class of STEs with arbitrary number of conflict positions, for which trail evaluation on undirected graphs is tractable.

In Part III we show that all tractability results imply that enumeration of the output with polynomial time delay between answers is possible. Technically, this means that we have to prove that we cannot only solve a decision variant of the RPQ evaluation problem, but we also need to find witnessing paths. We prove that the algorithms for the decision problems can be extended to return paths which can then be combined with a variant of Yen's Algorithm [209] to give a polynomial delay enumeration algorithm. In terms of parameterized complexity, we prove that tractability on STEs also carries over to the enumeration setting, that is, FPT delay.

1.2 Connection to Published Works

The work on a trichotomy for trail semantics, Chapter 3, is based on joint work with Wim Martens and Matthias Niewerth. A preliminary version of this work was published in the International Symposium on Theoretical Aspects of Computer Science 2020 [154]. An extended version is currently under review and can be found on arXiv [153].

The work on query logs and parameterized complexity, presented in Chapters 4 and 5, is joint work with Wim Martens. A preliminary version of this work was presented

[3]Tigergraph allows undirected edges [196, Defining a Graph Schema] and Neo4j Cypher allows specifying match patterns direction-agnostically. Two-way navigation is possible in either Tigergraph's GSQL, Neo4j's Cypher, and SPARQL 1.1. Also the newest documents regarding GQL [84] allow undirected and bidirectional edges.

at International Conference on Database Theory 2018 [156] and received the best paper award. An extended version was invited and published in ACM Transaction on Database Systems 2019 [158]. The work also received an ACM SIGMOD Research Highlight Award 2018. To this end, a shorter version was published in Sigmod Record [157].

Chapter 6 contains joint work with Wim Martens. A prelimary version of this work was accepted to the Symposium on Principles of Database Systems 2022 [155].

Part III contains joint work with Wim Martens and Matthias Niewerth. Some of the results on directed graphs have been published in International Conference on Database Theory 2018 [156], ACM Transactions on Database Systems 2019 [158], SIGMOD Record [157], and International Symposium on Theoretical Aspects of Computer Science 2020 [154], while results on undirected graphs were accepted to the Symposium on Principles of Database Systems 2022 [155].

1.3 Contributions by Other Authors

The starting research questions were given by my PhD advisor, Wim Martens, and then jointly developed depending on the results. Furthermore, the papers on which this thesis is based have been jointly written with the co-authors which significantly helped to put the work in a broader research perspective and improve their presentation. The technical results in this thesis, however, were mainly obtained by the author of this thesis. We thank Jean-Éric Pin and Luc Segoufin for helping with the proof of Proposition 3.3.4(b).

1.4 Related Work

RPQs on graph databases have been studied since the end of the 80's [69, 74, 208]. A central component of querying a graph database consists of answering the following question: Given a graph database G, an RPQ r, and two nodes s and t, is there a path from s to t (possibly with loops) in G that matches r? When no loops are allowed, then we name this problem SimPath.

Simple Path Semantics Cruz, Mendelzon and Wood [74] designed one of the earliest navigational languages for graph databases. Motivated by early applications of graph databases, their language uses simple paths semantics, that is, they do not allow loops in the path. Mendelzon and Wood [160] observed that under simple path semantics querying a graph database is already NP-complete for relatively simple expressions like a^*ba^* and $(aa)^*$. These two results heavily rely on the work of Fortune et al. [98], who showed NP-completeness of the two disjoint paths problem on directed graphs, and LaPaugh and Papadimitriou [140], who showed that the even length simple path problem on directed graphs is NP-complete.

While Mendelzon and Wood [160] showed that the problem can be decided in polynomial time for downward closed languages, the overall complexity of simple path semantics was considered as too high and database systems therefore preferred arbitrary path semantics.

New interest in simple path semantics was sparked in 2010 when the W3C added regular expressions to SPARQL 1.1 queries in the form of SPARQL property paths. These property path were evaluated under a semantics based on simple paths. Because of the studies about the complexity of SPARQL 1.1 property paths [16, 146], SPARQL switched their semantics away from counting simple paths. Shortly afterwards, Bagan et al. [20] provided a dichotomy for the *data complexity* of SimPath.They defined a class $\mathsf{SP}_{\mathsf{tract}}$ such that the problem is in P for each language in $\mathsf{SP}_{\mathsf{tract}}$ and NP-complete otherwise. We introduce this class in more detail in Section 2.5.2.

There is also work that takes the shape of the database into account. For example, Mendelzon and Wood [160] proved that the complexity of SimPath becomes polynomial if both the graph and language are restricted with regard to their cycles. Other work showed the polynomiality for the class of outerplanar graphs [166] or on graphs of bounded treewidth [34]. In this setting, the polynomial time even holds if the language is not fixed. On the other hand, Barrett et al. [34] show that the problem SimPath is NP-complete for the class of grid graphs, even when the language is fixed.

Arbitrary Path Semantics In the meantime, most work focused on arbitrary and shortest paths semantics, for which the evaluation problem is well-known to be tractable using standard product automata techniques [160]. Recently, Casel and Schmid [64] showed that this approach is essentially optimal. There is a wide literature on RPQs, which led to several surveys [11, 14, 26, 63, 205].

Indeed, RPQs have been extended in various ways. For example, to two-way regular path queries [28, 33, 60] that allow navigation in the reverse direction of an edge, or nested regular path queries [32] that extend two-way regular path queries with additional node-tests. The favorable complexity of RPQs also carries over to these classes, see [26] for an overview.

Graph Pattern Matching Modern graph query languages are often based on graph pattern matching. Examples are Cypher from Neo4j [99], GSQL from TigerGraph [196], and PGQL from Oracle [176], as well as industry/academia prototypes such as G-CORE [10] and GPML [84].

Abstractly, the graph pattern matching problem is to find a homomorphic, isomorphic, or "similar" image of a given graph pattern in the data graph. Graph patterns can use regular expressions or transitive closure to allow an edge-to-path matching.

In this sense, conjunctions of regular path queries are a graph pattern based on homomorphism semantics. Conjunctions of RPQs (CRPQs) have been introduced in [74] and been studied in several follow-up works, for example [45, 63, 62, 183]. Also the static analysis of CRPQs [60, 85, 93, 94, 96] has been extensively studied. Furthermore, CRPQs were extended with the ability to output and/or compare paths in [28, 30, 31, 101] and with data value comparisons [145]. Reutter et al. [181] study regular queries, which are nonrecursive Datalog programs extended with transitive closure, and show elementary tight bounds for the containment problem for regular queries.

There is also a plethora of work that focuses on tree-shaped patterns [3, 15, 50, 77, 78, 95, 137, 162, 189, 206]. This line of work has especially focused on querying XML documents, which can be seen as node-labeled, tree-shaped graph databases, but the patterns can also be used to query arbitrary node-labeled graph databases. In several important cases [77, 162], static analysis problems on these queries are even exactly the same, independent from whether we use them to query tree-structured or graph-structured data.

Pattern matching in terms of subgraph isomorphism has been introduced by Ullmann [202]. Here, one is given two graphs G and H as input and has to determine whether G contains a subgraph which is isomorphic to H. Although this problem is NP-complete [73], it has been extensively studied on node-labeled graphs, see for example [56, 70, 102] for surveys. Wang et al. [203] extended this notion of pattern matching with a subclass of regular expressions and studied algorithms which return the best K matchings. In their work the nodes of the pattern graph H must be mapped bijectively into G, while edges can be mapped to arbitrary paths as long as the regular expressions match.

Another line of work defines pattern matching in terms of graph simulation [2, 57, 117]. This notion of matching relaxes subgraph isomorphism. More precisely, graph simulation is a relation $R \subseteq V_Q \times V$ between the nodes V_Q of the graph pattern and the nodes V_G of the directed graph such that $(u, v) \in R$ if u and v have the same label and for each u' with $(u, u') \in E_Q$ there is a v' with $(v, v') \in E_G$ and $(u', v') \in R$. For example, an a-b-cycle can be mapped into an a-b-a-b-cycle. This relaxation leads to a polynomial time evaluation problem, as opposed to the NP-completeness of subgraph isomorphism and conjunctive query evaluation [65, 73]. Fan et al. [91] show that graph pattern matching under graph simulation semantics can be extended with a subclass of regular expressions while maintaining the polynomial time complexity for evaluation, minimization, equivalence and containment problems.

Broader Scientific Context There are several ways of representing graph databases, see for example [13, 112]. Two of the most popular ways of representing graphs are the Resource Description Framework (RDF)[76], and property graphs [11]. In RDF, the data is modeled as triples (s, p, o) representing the subject s, the predicate p, and the object o of a statement about these resources. Each triple can be viewed as two nodes, s and o, and an edge labeled p from s to o. Yet, unlike in edge-labeled directed graphs, RDF triples allow an "edge-label" to be the subject of another triple, for example (s, p, o) and (p, q, t). On the other hand, property graph models the data as a mixed multigraph, that is, a multigraph with directed and/or undirected edges. Both nodes and edges can be labeled. Unlike RDF with its query language SPARQL [116], which is a W3C standard, there is no query language standard for property graphs yet. In 2019 the Joint Technical Committee 1 of ISO/IEC, which defines information technology standards for the International Organization for Standardization, and International Electrotechnical

Commission, approved a project to create a standard property graph query language, GQL. This standardization process is governed by WG3, who work together with the Linked Data Benchmark Council (LDBC).

Recently, the members of WG3 and LDBC [84] published a summary of their view on the new graph pattern matching languages. In order to allow paths to be returned, they want to ensure that there is only a finite number of results. To enforce this for paths, they either restrict results to being simple paths, trails, shortest paths, or select only k many paths.

The LDBC and the WG3 do not only collaborate on query language design. The LDBC also has a number of working groups focusing on the design of schema languages for property graphs. These groups have generated an initial proposal for key constraints for property graphs [12] and their generalization to cardinality constraints, which evolved into a deeper study on threshold queries [52]. Cardinality constraints are similar to key constraints in the sense that they do not impose uniqueness of an element in the database, but allow a given number to exist. Similarly, threshold queries return the answers of a given query until a given number of outputs is reached.

Interestingly, restricting the number of returned paths seems again important when considering keys for graphs. More precisely, Angles et al. [12] give as example that a constraint that requires a path to be unique implies that the path cannot have a cycle because these can be traversed multiple times. Thus simple path or trail semantics might be preferable over arbitrary path semantics.

The graph pattern matching language presented in [84] extends CRPQs syntactically and semantically. Syntactically, it introduces group variables and variables binding to entire paths, which allows to treat paths as first-class citizens, as advocated in the G-CORE proposal [10]. Semantically, they support a finer-grained notion of match, which permits binding variables to paths, using multiplicity-sensitive aggregation such as sum, count, and average, and restricting the number of possible returned paths.

Methodology In terms of methodology, some of this work is heavily inspired on a line of work initiated by Frank Neven [40, 41, 151]. A practical study on the shapes of regular expressions [40] motivated the study of *simple regular expressions* and *k-occurrence regular expressions or $RE^{\leq k}$* [151] and later work on schema inference, for example, [41].[4] Similarly, a practical study on the use of complex types in schemas for XML data [39] motivated inference algorithms for learning XML Schema [42] and the design of the BonXai schema language [150].

Graph Theory Edge-labeled graphs are a generalization of unlabeled graphs, which have been studied in depth by graph theoreticians. We recall some results in this field.

There is a seminal line of work accomplished by Robertson and Seymour [182] that resulted in the Robertson–Seymour theorem, also called graph minor theorem. An important consequence is that, for each fixed undirected unlabeled graph H, there is a

[4]Later work used the term *(extended) chain regular expressions* to refer to the simple regular expressions from [151].

polynomial time algorithm for testing whether an undirected unlabeled graph G has H as a minor. Their line of work has been followed by a plethora of works on simplifying the proof, extending the results, and obtaining better running time [111, 131, 133, 134]. Huynh [119] extended this minor theorem to group-labeled graphs. Although group-labeled graphs are quite different from undirected graphs, they coincide modulo 2 (but not for other modulos). Thus, in particular, Huynh's work extends the graph minor theorem with parity conditions. Independently and with different methods, Kawarabayashi et al. [132] also proved this result and with an improved running time. The problem of finding a simple path of length 0 modulo 3 between two given nodes on undirected graphs has also been studied extensively, but its complexity is not yet known. Arkin et al. [17] give a linear-time algorithm to decide whether all paths between two specified nodes are of length P mod Q, for fixed integers P and Q. Many works also look for simple paths whose length is a certain modulo in very restricted kinds of graphs. For example, Deng and Papadimitriou [83] show that between any two nodes of a cubic, planar, three-connected undirected graph there are three paths whose lengths are 0, 1, and 2 modulo 3, respectively. Amar and Manoussakis [8] give several sufficient conditions on the half-degrees of a bipartite digraph for the existence of cycles and paths of various lengths. On the other hand, on directed graphs, already the problem of deciding whether there is a simple path of even length between two given nodes is NP-complete [140].

Another important problem is the kDisjointPaths problem, which asks for k node- or edge-disjoint paths between k pairs of nodes. The first polynomial time algorithms for the kDisjointPaths with $k = 2$ on undirected graphs where given by Ohtsuki [171], Seymour [186], Shiloach [187], and Thomassen [199]. A consequence of the minor theorem [182] is that the kDisjointPaths problem on undirected graph is in polynomial time for every fixed k. If k is part of the input, the kDisjointPaths problem is NP-complete [90, 128] on both directed and undirected graphs. On directed graph the problem is already NP-complete for $k = 2$ [98]. For $k = 2$, Perl and Shiloach [174] gave a polynomial time algorithm for the kDisjointPaths problem on directed acyclic graphs. Fortune et al. [98] generalized their algorithm to fixed k. There has also been a lot of work with length constraints on path(s), for example [7, 36, 43, 47, 59, 89, 97, 192].

Some of the problems we study in this work are an extension of determining whether a compatible path in forbidden-transition graphs exists. In these undirected graphs, certain transitions are forbidden and one asks for a path whose consecutive edges respect these constraints. Determining the existence of such a path that additionally does not repeat nodes is already NP-complete [194], as are other problems which are solvable in polynomial time on standard graphs, for example [5, 35, 88, 106, 107, 126, 127, 194].

A specific case of compatible paths in forbidden-transition graphs are so called *properly edge-colored paths* in *edge-colored (undirected) graphs*. A path or cycle is properly edge-colored (PEC) if no two consecutive edges have the same color. Bang-Jensen and Gutin [24, 25] showed that one can test in linear time if there is a PEC path in an undirected edge-colored multigraph with two colors, while Szeider [194] extended their result to arbitrarily many colors. Many results and concepts from directed graphs extend to undirected edge-colored graphs [25, 67, 149].[5] We refer the reader to [23, 25, 114] for surveys on PEC paths and cycles. For example, the problem of properly colored Hamiltonian cycles and its variants have been studied extensively, recent developments can be found in [22, 71, 72, 113, 143, 144]. PEC simple paths and trails and their variants have also been studied on directed and undirected edge-colored graphs [4, 5, 106, 107]. For instance, a characterization of c-edge-colored graphs containing PEC cycles was first presented by Yeo [211] and generalized in [5] to properly-edge-colored closed trails. In [105], the authors consider a number of path and trail problems restricted to edge-colored graphs with no PEC cycles or PEC closed trails. They studied these classes since they can be seen as an edge-colored counterpart of directed acyclic graphs.

Enumeration In the last decades we have seen increased interest in enumerating answers to queries on *data which is subject to updates*, sometimes also called *incremental validation* or *incremental evaluation* [9, 21, 37, 38, 49, 51, 80, 120, 121, 136, 147, 169, 170]. While we do study results on enumerating answers to graph database queries, the data in our setting is static, that is, we do not consider scenarios in which the data is changed.

Ackerman and Shallit [6] proved that one can enumerate the words accepted by a given NFA in polynomial delay. This is easily extended to RPQ evaluation on directed and undirected multigraphs with respect to arbitrary paths and shortest paths semantics, as we observe in Section 8.2. Simple paths can be dealt with using Yen's algorithm [209], which takes an directed or undirected (unlabeled) graph, nodes s and t, and a parameter K as input and returns the K shortest simple paths from s to t. In the case of undirected graphs, Katoh, Ibaraki, and Mine [129] improved the running time of Yen's algorithm from $O(K(|V| + |V|^3))$ to $O(K(|E| + |V| \log |V|))$. As we observe in Theorem 8.3.1, Yen's algorithm can also enumerate all simple paths between two given nodes in polynomial delay. Yen's algorithm was generalized by Lawler [142] and Murty [165] to a tool for designing general algorithms for enumeration problems. Lawler-Murty's procedure has been used for solving enumeration problems in databases in various contexts, for example [104, 125, 138].

Casel and Schmid [64] take a different approach: instead of enumerating the paths between two given nodes, they enumerate pairs of nodes (u, v) such that there is a path from u to v that matches r. They give an enumeration algorithm with delay linear in the size of G.

[5]The idea is to split each directed edge e into two undirected edges with two colors according to the direction of e. To ensure that the paths start with the correct color, one can replace every start node s_i with a new one s_i^* that has only an undirected edge of color 2 to the respective s_i.

Chapter 2

Preliminaries

We use $[n]$ to denote the set of integers $\{1, \ldots, n\}$. Let Γ be an infinite set of *symbols*. We denote by Σ a finite subset of Γ. A (Σ-)*symbol* is an element of Σ. We always denote symbols by a, b, c, d and their variants, like a', a_1, b_1, etc. We denote sets of symbols by uppercase letters like A, B, and their variants, like A', A_1, A'_1 etc. The size of a set of symbols A, denoted $|A|$, is the number of elements in the set. A *word* (over Σ) is a finite sequence $w = a_1 \cdots a_n$ of Σ-symbols. The *length* of w, denoted by $|w|$, is its number of symbols n. We denote the empty word by ε. The *reverse* of w is $w^{\mathsf{rev}} = a_n \cdots a_1$. For $0 \leq i \leq j \leq n$, we denote by $w[i, j]$ the substring $a_i \cdots a_j$ of w.

2.1 Regular Expressions and RPQs

Regular expressions over an alphabet Σ are defined as follows: $\emptyset$, ε, and every Σ-symbol is a regular expression; and if r and s are regular expressions, then $(r \cdot s)$, $(r + s)$, and (r^*) are regular expressions. To improve readability, we use associativity and the standard priority rules to omit braces in regular expressions. We usually also omit the outermost braces. The *size* $|r|$ of a regular expression is the number of occurrences of Σ-symbols in r. For example, $|((a \cdot b) \cdot a)^*| = 3$. We define the *language* $L(r)$ of r as usual.

We use the following standard abbreviations and alternative notations: (rs) abbreviates $(r \cdot s)$, $(r?)$ abbreviates $(r + \varepsilon)$, and (r^+) abbreviates (rr^*). Furthermore, if $S = \{a_1, \ldots, a_n\} \subseteq \Sigma$, then we identify S with the expression $(a_1 + \cdots + a_n)$. We allow $S = \emptyset$, in which case $L(S) = \emptyset$. As such, $L(\Sigma^*)$ contains every word and $L(\emptyset^*) = \{\varepsilon\}$. For $n \in \mathbb{N}$, we use r^n to abbreviate the n-fold concatenation $r \cdots r$ of r. We abbreviate $(r?)^n$ by $r^{\leq n}$. In the context of graph databases, *regular path queries (RPQs)* are regular expressions that can be evaluated on graphs and return an output. In this thesis, we will blur the distinction between them (language acceptors vs. queries) and use "regular expression" and RPQ as synonyms.

The *reversal of a language* L is $L^{\mathsf{rev}} = \{w^{\mathsf{rev}} \mid w \in L\}$. Given a language L and a word w, the *derivative*[1] *of* L *with respect to* w is defined as

$$w^{-1}L := \{v \mid wv \in L\}.$$

[1]Also known as Brzozowski derivative [58].

2.2 Automata

A *nondeterministic finite automaton (NFA)* N over Σ is a tuple $(Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ is the set of accepting states. The *size* of an NFA N, denoted $|N|$, is its number of states $|Q|$. We define the *language* $L(N)$ of N as usual.

Strongly connected components of (the graph of) N are simply called *components*. Unless noted otherwise, components will be non-trivial, that is, containing at least one edge. We write $C(q)$ to denote the strongly connected component of state q.

By $\delta(q, w)$ we denote the states reachable from state q by reading w. Given a path p which is labeled w, we also slightly abuse notation and write $\delta(q, p)$ instead $\delta(q, w)$. We denote by $q_1 \rightsquigarrow q_2$ that state q_2 is reachable from q_1. Finally, L_q denotes the set of all words accepted from q and $L(A) = \bigcup_{q \in I} L_q$ is the set of words accepted by A. For every state q, we denote by $\text{Loop}(q)$ the set $\{w \in \Sigma^+ \mid \delta_L(q, w) = q\}$ of all non-empty words that allow to loop on q.

A *deterministic finite automaton (DFA)* is an NFA such that I is a singleton and for all $q \in Q, \sigma \in \Sigma$: $|\delta(q, \sigma)| \leq 1$. Let L be a regular language. We denote by $A_L = (Q_L, \Sigma, i_L, F_L, \delta_L)$ the (complete) minimal DFA for L and by N the number $|Q_L|$ of states. For $q_0 \in Q$, we say that a *run from* q_0 of A on $w = a_1 \cdots a_n$ is a sequence $q_0 \to \cdots \to q_n$ of states such that $q_i \in \delta(q_{i-1}, a_i)$, for every $i \in \{1, \ldots, n\}$. When A is a DFA and q_0 its initial state, we also simply call it *the run of* A *on* w.

2.3 Graph Databases and Paths

We use edge-labeled directed and undirected multigraphs as abstractions for graph databases. An edge-labeled directed multigraph $G = (V, E, \mathcal{E})$ consists of a finite set of nodes V, a finite set of edges E, and a function $\mathcal{E} \colon E \to V \times \Gamma \times V$ that maps each edge identifier to a tuple (v_1, a, v_2) describing the origin, the label, and the destination node of the edge. We denote v_1 by $\text{origin}(e)$, a by $\text{lab}(e)$ and v_2 by $\text{destination}(e)$. We emphasize that $\mathcal{E}$ does not need to be injective, that is, there might be several edges with identical origin, label, and destination.

An edge-labeled undirected multigraph $G = (V, E, \mathcal{E})$ consists of a finite set of nodes V, a finite set of edges E, and a function $\mathcal{E} \colon E \to 2^{V \cup \Gamma}$ with $|\mathcal{E}(e) \cap \Gamma| = 1$ and $1 \leq |\mathcal{E}(e) \cap V| \leq 2$ for every $e \in E$. Given an edge $e \in E$, we denote by $\mathsf{Node}(e) = \mathcal{E}(e) \cap V$ its nodes and by $\text{lab}(e) = \mathcal{E}(e) \cap \Gamma$ its label. For convenience, in undirected multigraphs, we also denote $\mathcal{E}(e)$ as (u, a, v), where $\{u, v\} = \mathsf{Node}(e)$ and $a = \text{lab}(e)$. As such, (u, a, u) denotes a *self-loop* with label a on node u. With this notation, (u, a, v) and (v, a, u) are the same in undirected graphs, and we will order u and v in our notation such that it optimizes readability.

Given an (un-)directed multigraph $G = (V, E, \mathcal{E})$, the size of G is defined as $|V| + |E|$. A *(simple) graph* is a multigraph where $\mathcal{E}$ is injective. We sometimes denote $\mathcal{E}(e)$ as (u, v)

if the label does not matter.

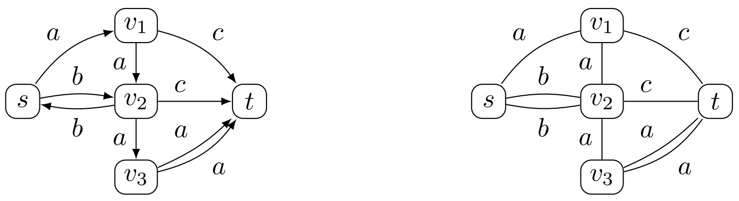


Figure 2.1: A directed multigraph (left) and an undirected multigraph (right)

A path p from s to t in an (un-)directed multigraph G is a sequence of edges $e_1 \cdots e_k$ in G such that $\mathcal{E}(e_1) \cdots \mathcal{E}(e_k)$ can be written as $(s, a_1, v_1)(v_1, a_2, v_2) \cdots (v_{k-1}, a_{k-1}, t)$ for some nodes $v_1, \ldots, v_{k-1} \in V$ and labels $a_1, \ldots, a_{k-1} \in \Sigma$. The set of *nodes of path* p is $V(p) = \{s, v_1, \ldots, v_{k-1}, t\}$. The *length* of p, denoted by $|p|$, is the number of edges in p. A path is a *trail* if every edge e appears at most once[2] and a *simple path* if all its nodes are different, that is, if $|V(p)| = |p| + 1$. We note that each simple path is a trail but not vice versa.

We denote $\text{lab}(e_1) \cdots \text{lab}(e_k)$ by $\text{lab}(p)$. Given a language $L \subseteq \Sigma^*$, path p *matches* L if $\text{lab}(p) \in L$. If r is a regular expression (respectively N is an NFA), we simplify notation and also say that p *matches* r when p matches $L(r)$. For a subset $E' \subseteq E$, path p is E'-restricted if every edge of p is in E'. We use *a-edge* to refer to an edge with label a (that is, with $\text{lab}(e) = a$) and *a-path* to refer to a path that consists only of a-edges. Given a trail p and two edges e_1 and e_2 in p, we denote the subpath of p from e_1 to e_2 by $p[e_1, e_2]$. The *concatenation* of paths $p_1 = e_1 \cdots e_k$ and $p_2 = e_{k+1} \cdots e_n$ is simply the concatenation $p_1 p_2$ of the two sequences. Notice that the last node of p_1 needs to be the same as the first node of p_2.

The product of a directed multigraph $G = (V, E, \mathcal{E})$ and NFA $A = (Q, \Sigma, I, F, \delta)$ is a directed multigraph $(V', E', \mathcal{E}')$ with $V' = V \times Q$, $E' = \{(e, (q_1, q_2)) \mid (q_1, \text{lab}(e), q_2) \in \delta\}$ and $\mathcal{E}'((e, (q_1, q_2))) = ((\text{origin}(e), q_1), \text{lab}(e), (\text{destination}(e), q_2))$.

We illustrate some of these notions on the directed and undirected multigraphs in Figure 2.1. The path $p = e_1 e_2$ with $\mathcal{E}(e_1) = (s, b, v_2)$, $\mathcal{E}(e_2) = (v_2, b, s)$ is a trail, both in the directed and the undirected multigraph, since $e_1 \neq e_2$. But it is no simple path because $|p| = 2$ and $V(p) = \{s, v_2\}$, and thus $|V(p)| \neq |p| + 1$. In both multigraphs there are two different a-paths from s to t that are simple paths. Both of them can be written as $(s, a, v_1)(v_1, a, v_2)(v_2, a, v_3)(v_3, a, t)$. Each simple path is also a trail. If we drop the restriction to simple paths, then there are infinitely many a-paths from s to t in the undirected multigraph, for example $(s, a, v_1)(v_1, a, s)(s, a, v_1)(v_1, a, v_2)(v_2, a, v_3)(v_3, a, t)$.

[2]We note that it is allowed that for $i \neq j$ it holds that $\mathcal{E}(e_i) = \mathcal{E}(e_j)$.

2.4 Main Problems

In this thesis we will study variants of the SimPath and Trail problem.

SimPath(L)	
Given:	A directed multigraph $G = (V, E, \mathcal{E})$, two nodes $x, y \in V$
Question:	Is there a simple path from x to y in G that matches L?

Trail(L)	
Given:	A directed multigraph $G = (V, E, \mathcal{E})$, two nodes $x, y \in V$.
Question:	Is there a trail from x to y in G that matches L?

Note that we study the *data complexity* of SimPath and Trail, that is, we assume that the language L (the query) in the problems SimPath(L) and Trail(L) is not part of the input, but fixed. Therefore, each language gives rise to a different computational problem. We note that in this setting it plays no role whether L is given as a DFA, NFA, or regular expression.

We will contrast these problems by also considering variants where the language is not fixed, but used as a parameter. We explain this in more detail in Section 5.1 and will denote parameterized versions by a preceding P.

In Part II we will study these problems on undirected multigraphs. Similarly to the parameterized version, we will denote variants on undirected multigraphs by adding a U.

2.5 Fundamental Subclasses of Regular Languages

We introduce some important languages which will be used throughout this thesis.

2.5.1 Downward Closed Languages (DC)

A language L is *downward closed*[3] (DC) if it is closed under taking subsequences. That is, for every word $w = a_1 \cdots a_n \in L$ and every sequence $0 < i_1 < \cdots < i_k < n+1$ of integers, we have that $a_{i_1} \cdots a_{i_k} \in L$. Perhaps surprisingly, *downward closed languages are always regular* [115]. Furthermore, they can be defined by a clean class of regular expressions (which was shown by Jullien [124] and later rediscovered by Abdulla et al. [1]), which is defined as follows.

Definition 2.5.1. An *atomic expression* over Σ is an expression of the form $(a + \varepsilon)$ or of the form $(a_1 + \cdots + a_n)^*$, where $a, a_1, \ldots, a_n \in \Sigma$. A *product* is a (possibly empty) concatenation $e_1 \cdots e_n$ of atomic expressions $e_1, \ldots, e_n$. A *simple regular expression* is of the form $p_1 + \cdots + p_n$, where $p_1, \ldots, p_n$ are products.

[3]The term *downward closed* comes from being closed under taking the smaller elements in the subsequence ordering which, due to Higman's Lemma, is a well quasi ordering.

Another characterization is by Mendelzon and Wood [160], who show that a regular language L is downward closed if and only if its minimal DFA $A_L = (Q_L, \Sigma, i_L, F_L, \delta_L)$ exhibits the *suffix language containment property*, which says that if $\delta_L(q_1, a) = q_2$ for some symbol $a \in \Sigma$, then we have $L_{q_2} \subseteq L_{q_1}$.[4] Since this property is transitive, it is equivalent to require that $L_{q_2} \subseteq L_{q_1}$ for every state q_2 that is reachable from q_1.

Theorem 2.5.2 ([1, 115, 124, 160])**.** *The following are equivalent:*

(1) L is a downward closed language.

(2) L is definable by a simple regular expression.

(3) The minimal DFA of L exhibits the suffix language containment property.

2.5.2 Tractable Class for Regular Simple Path Queries ($\mathsf{SP}_{\mathsf{tract}}$)

Bagan et al. [20] introduced[5] the class $\mathsf{SP}_{\mathsf{tract}}$, which characterizes the class of regular languages L for which the *regular simple path query (SimPath)* problem is tractable. While they studied SimPath on directed simple graphs, their results immediately carry over to multigraphs: since simple paths can use every node at most once, they cannot use more than one edge between any pair of nodes. Thus regarding simple path semantics, the results on a multigraph will be no different from the results on the underlying simple graph.

Theorem 2.5.3 (Theorem 3 in Bagan et al. [20])**.** *Let L be a regular language.*

(1) If L is finite, then $\mathsf{SimPath}(L) \in AC^0$.

(2) If $L \in \mathsf{SP}_{\mathsf{tract}}$ and L is infinite, then $\mathsf{SimPath}(L)$ is NL-complete.

(3) If $L \notin \mathsf{SP}_{\mathsf{tract}}$, then $\mathsf{SimPath}(L)$ is NP-complete.

One characterization of $\mathsf{SP}_{\mathsf{tract}}$ is the following (Theorem 6 in [20]):

Definition 2.5.4. $\mathsf{SP}_{\mathsf{tract}}$ is the set of regular languages L such that there exists an $i \in \mathbb{N}$ for which the following holds: for all $w_\ell, w, w_r \in \Sigma^*$ and $w_1, w_2 \in \Sigma^+$ we have that, if $w_\ell w_1^i w w_2^i w_r \in L$, then $w_\ell w_1^i w_2^i w_r \in L$.

Bagan et al. [20] also gave a characterization of $\mathsf{SP}_{\mathsf{tract}}$ in terms of regular expression:

Theorem 2.5.5. *Let L be a regular language. Then L belongs to $\mathsf{SP}_{\mathsf{tract}}$ if and only if L can be written as a union of regular expressions of the form*

$$w_\ell (w_1 + \varepsilon)(A_1^{\geq k_1} + \varepsilon)(w_2 + \varepsilon) \cdots (A_n^{\geq k_n} + \varepsilon) w_r$$

for some $n, k_1, \ldots, k_n \in \mathbb{N}$, words $w_\ell, w_1, \ldots, w_n, w_r \in \Sigma^*$, and sets $A_1, \ldots, A_n \subseteq \Sigma$.

[4] They restrict q_1, q_2 to be on paths from i_L to some state in F_L, but the property trivially holds for q_2 being a sink-state.

[5] They called the class $\mathsf{C}_{\mathsf{tract}}$, which stands for "tractable class". We distinguish between $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$ here to avoid confusion between simple paths and trails.

2.5.3 Aperiodic Languages (**FO**[<])

There are many characterizations of aperiodic languages [185]. A regular language L is *aperiodic* if and only if its minimal DFA A_L satisfies $\delta_L(q, w^{N+1}) = \delta_L(q, w^N)$ for every state q and word w, where N is the number of states of A_L. Equivalently, a regular language L is aperiodic if and only if its minimal DFA does not have a simple cycle labeled w^k for $k > 1$ and $w \neq \varepsilon$. Thus, for "large enough n" we have: $uw^nv \in L$ if and only if $uw^{n+1}v \in L$. So, a language like $(aa)^*$ is not aperiodic (take $w = a$ and $k = 2$), but $(ab)^*$ is.

Let **FO**[<] be the set of languages definable in first-order logic with unary predicates P_a for all $a \in \Sigma$ and the binary predicate $<$. Here, P_a denotes the positions carrying the symbol a while $<$ denotes the order relation among position. It follows from McNaughton and Papert [159] and Schützenberger [185] that the aperiodic languages are exactly those definable in **FO**[<].[6]

[6]McNaughton and Papert [159] showed the equivalence of first-order definability and star-freeness for finite words. The equivalence of star-freeness and aperiodicity for finite words is due to Schützenberger [185].

Part I

Evaluation on Directed Multigraphs

Chapter 3

A Trichotomy for Regular Trail Queries

In this chapter, we inspect for which regular languages L the problem $\mathsf{Trail}(L)$ can be answered in polynomial time. To this end, we define and characterize the class $\mathsf{T_{tract}}$ and prove that it contains exactly the languages for which $\mathsf{Trail}(L)$ is tractable (under the assumption NP $\neq$ NL). We compare the expressiveness of $\mathsf{T_{tract}}$ to yardstick languages such as aperiodic languages, $\mathsf{SP_{tract}}$, and downward closed languages, and study its closure properties.

3.1 The Tractable Class $\mathsf{T_{tract}}$

In this section, we define and characterize a class of languages of which we will prove that it is exactly the class of regular languages L for which $\mathsf{Trail}(L)$ is tractable (if NL $\neq$ NP).

The following definitions are the basis of the class of languages for which $\mathsf{Trail}(L)$ is tractable.

Definition 3.1.1. An NFA A satisfies the *left-synchronized containment property* if there exists an $n \in \mathbb{N}$ such that the following implication holds for all $q_1, q_2 \in Q$:

$$\text{If } q_1 \rightsquigarrow q_2 \text{ and if } w_1 \in \text{Loop}(q_1), w_2 \in \text{Loop}(q_2) \text{ with } w_1 = aw_1' \text{ and } w_2 = aw_2', \quad \text{then } w_2^n L_{q_2} \subseteq L_{q_1}.$$

Similarly, A satisfies the *right-synchronized containment property* if the same condition holds with $w_1 = w_1'a$ and $w_2 = w_2'a$.

We note that every minimal DFA of a downward closed language L satisfies the left-synchronized containment property.

The *left-synchronizing length* of an NFA A is the smallest value n such that the implication in Definition 3.1.1 for the left-synchronized containment property holds. We define the *right-synchronizing length* analogously.

Observation 3.1.2. *Let n_0 be the left-synchronizing length of an NFA A. Then the implication of Definition 3.1.1 is satisfied for every $n \geq n_0$. The reason is that $w_2 \in$ Loop(q_2), so $w_2 L_{q_2} \subseteq L_{q_2}$.*

Definition 3.1.3. A regular language L is *closed under left-synchronized power abbreviations* (respectively, *closed under right-synchronized power abbreviations*) if there exists an $n \in \mathbb{N}$ such that for all words $w_\ell, w_m, w_r \in \Sigma^*$ and all words $w_1 = aw_1'$ and $w_2 = aw_2'$ (respectively, $w_1 = w_1'a$ and $w_2 = w_2'a$) we have that $w_\ell w_1^n w_m w_2^n w_r \in L$ implies $w_\ell w_1^n w_2^n w_r \in L$.

We note that Definition 3.1.3 is equivalent to requiring that there exists an $n \in \mathbb{N}$ such that the implication holds for all $i \geq n$. The reason is that, given $i > n$ and a word of the form $w_\ell w_1^i w_m w_2^i w_r$, we can write it as $w_\ell' w_1^n w_m w_2^n w_r'$ with $w_\ell' = w_\ell w_1^{i-n}$ and $w_r' = w_2^{i-n} w_r$, for which the implication holds by Definition 3.1.3.

Lemma 3.1.4. *Consider a minimal DFA $A_L = (Q_L, \Sigma, i_L, F_L, \delta_L)$ with N states. Then the following is true:*

(1) If A_L satisfies the left-synchronized containment property, then the left-synchronizing length is at most N.

(2) If A_L satisfies the right-synchronized containment property, then the right-synchronizing length is at most N.

Proof. We first prove (1). Let $n \in \mathbb{N}$ be the left-synchronizing length. Since A_L satisfies the left-synchronized containment property, n is well defined. If $n \leq N$, we are done, therefore we assume $n > N$. By Definition 3.1.1, it holds that: If $q_1, q_2 \in Q_A$ such that $q_1 \rightsquigarrow q_2$ and if $w_1 \in \text{Loop}(q_1), w_2 \in \text{Loop}(q_2)$ with $w_1 = aw_1'$ and $w_2 = aw_2'$, then we have $w_2^n L_{q_2} \subseteq L_{q_1}$.

Since $n > N$ and $w_2^n L_{q_2} \subseteq L_{q_1}$, there must be a loop in the w_2^n part that generates multiples of w_2. Thus we can ignore the loop and obtain that $w_2^i L_{q_2} \subseteq L_{q_1}$ for an $i < n$. This is a contradiction to n being the left-synchronizing length (that is, the minimality of n).

The proof of (2) is analogous. We only need to replace $w_1 = aw_1'$ and $w_2 = aw_2'$ with $w_1 = w_1'a$, $w_2 = w_2'a$, and *left* with *right.* ☐

From Definition 3.1.1, Observation 3.1.2, and Lemma 3.1.4, we get the following corollary.

Corollary 3.1.5. *Let A be a minimal DFA with N states, $q_1, q_2 \in Q_A$ with $q_1 \rightsquigarrow q_2$, $w_1 \in \text{Loop}(q_1)$, and $w_2 \in \text{Loop}(q_2)$. If A satisfies the*

- *left-synchronized containment property, $w_1 = aw_1'$, and $w_2 = aw_2'$, then $w_2^N L_{q_2} \subseteq L_{q_1}$.*
- *right-synchronized containment property, $w_1 = w_1'a$, and $w_2 = w_2'a$, then $w_2^N L_{q_2} \subseteq L_{q_1}$.*

We need two lemmas to prove Theorem 3.1.9. And their proofs require the following lemma:

Lemma 3.1.6 (Implicit in [20], Lemma 3 proof)**.** *Every minimal DFA satisfying*

$$\textit{for all } q_1, q_2 \in Q_L \textit{ such that } q_1 \rightsquigarrow q_2 \textit{ and } \mathrm{Loop}(q_1) \cap \mathrm{Loop}(q_2) \neq \emptyset : L_{q_2} \subseteq L_{q_1} \qquad \text{(P)}$$

accepts an aperiodic language.

Lemma 3.1.7. *If A_L has the left-synchronized containment property or right-synchronized containment property, then L is aperiodic.*

Proof. Let A_L satisfy the left- or right-synchronized containment property. We show that L satisfies Property (P), restated here for convenience.

$$L_{q_2} \subseteq L_{q_1} \text{ for all } q_1, q_2 \in Q_L \text{ such that } q_1 \rightsquigarrow q_2 \text{ and } \mathrm{Loop}(q_1) \cap \mathrm{Loop}(q_2) \neq \emptyset \qquad \text{(P)}$$

This proves the lemma since all languages satisfying Property (P) are aperiodic, see Lemma 3.1.6. Let $q_1, q_2 \in Q_L$ and w satisfy $q_1 \rightsquigarrow q_2$ and $w \in \mathrm{Loop}(q_1) \cap \mathrm{Loop}(q_2)$. By Corollary 3.1.5 we then have that $w^N L_{q_2} \subseteq L_{q_1}$. Since $w \in \mathrm{Loop}(q_1)$, we have that $\delta(q_1, w^N) = q_1$, which in turn implies that $L_{q_2} \subseteq L_{q_1}$. □

Lemma 3.1.8. *If L is closed under left- or right-synchronized power abbreviations, then L is aperiodic.*

Proof. Let L be closed under left- or right-synchronized power abbreviations and $n \in \mathbb{N}$ be as in Definition 3.1.3. We show that $A_L = (Q_L, \Sigma, i_L, F_L, \delta_L)$ satisfies the Property (P). The aperiodicity then follows from Lemma 3.1.6.

Let $q_1, q_2 \in Q_L$ and w satisfy $q_1 \rightsquigarrow q_2$ and $w \in \mathrm{Loop}(q_1) \cap \mathrm{Loop}(q_2)$. Let $w_\ell, w_m \in \Sigma^*$ be such that $q_1 = \delta_L(i_L, w_\ell)$ and $q_2 = \delta_L(q_1, w_m)$. Let $w_r \in L_{q_2}$. Then, $w_\ell w^* w_m w^* w_r \subseteq L$ by construction. Especially, $w_\ell w^n w_m w^n w_r \in L$ and, by Definition 3.1.3, also $w_\ell w^n w^n w_r \in L$. Since $\delta_L(i_L, w_\ell w^n w^n) = q_1$, this means that $w_r \in L_{q_1}$. Therefore, $L_{q_2} \subseteq L_{q_1}$. □

Next, we show that all conditions defined in Definitions 3.1.1 and 3.1.3 are equivalent for DFAs.

Theorem 3.1.9. *For a regular language L with minimal DFA A_L, the following are equivalent:*

(1) A_L satisfies the left-synchronized containment property.

(2) A_L satisfies the right-synchronized containment property.

(3) L is closed under left-synchronized power abbreviations.

(4) L is closed under right-synchronized power abbreviations.

Proof. Let $A_L = (Q_L, \Sigma, i_L, F_L, \delta_L)$ and $N = |Q_L|$. (1) ⇒ (3): Let A_L satisfy the left-synchronized containment property. We will show that if there exists a word $w_\ell w_1^i w_m w_2^i w_r \in L$ with $i = N + N^2$ and w_1 and w_2 starting with the same letter, then $w_\ell w_1^i w_2^i w_r \in L$. To this end, let $w_\ell w_1^i w_m w_2^i w_r \in L$. Due to the pumping lemma, there are states q_1, q_2 and integers $h, j, k, \ell, m, n \leq N$ with $j, m \geq 1$ satisfying: $q_1 = \delta(i_L, w_\ell w_1^h)$, $q_1 = \delta(q_1, w_1^j)$, $q_2 = \delta(q_1, w_1^k w_m w_2^\ell)$, $q_2 = \delta(q_2, w_2^m)$, and $w_2^n w_r \in L_{q_2}$. This implies that

$$w_\ell w_1^h (w_1^j)^* w_1^k w_m w_2^\ell (w_2^m)^* w_2^n w_r \subseteq L \ .$$

Since A_L satisfies the left-synchronized containment property and by Corollary 3.1.5, we have $(w_2^m)^N L_{q_2} \subseteq L_{q_1}$ and therefore

$$w_\ell w_1^h (w_1^j)^* (w_2^m)^N w_2^n w_r \subseteq L \ .$$

Now we use that L is aperiodic, see Lemma 3.1.7:

$$w_\ell w_1^h (w_1^j)^N (w_1)^* (w_2^m)^N (w_2)^* w_2^n w_r \subseteq L$$

And finally, we use that $i = N + N^2$ and $h, j, m, n \leq N$ to obtain $w_\ell (w_1)^i (w_2)^i w_r \in L$.

(3) $\Rightarrow$ (4): Let L be closed under left-synchronized power abbreviations and let $j \in \mathbb{N}$ be the maximum of $|A_L|$ and $n+1$, where the n is from Definition 3.1.3. We will show that if $w_\ell (w_1 a)^j w_m (w_2 a)^j w_r \in L$, then $w_\ell (w_1 a)^j (w_2 a)^j w_r \in L$. If $w_\ell (w_1 a)^j w_m (w_2 a)^j w_r \in L$, then we also have $w_\ell (w_1 a)^j w_m (w_2 a)^{j+1} w_r \in L$ since L is aperiodic, see Lemma 3.1.8, and $j \geq |A_L|$. This can be rewritten as

$$w_\ell w_1 (a w_1)^{j-1} a w_m w_2 (a w_2)^{j-1} (a w_2 a w_r) \in L \ .$$

As L is closed under left-synchronized power abbreviations, and $n < j$, this implies

$$w_\ell w_1 (a w_1)^{j-1} (a w_2)^{j-1} (a w_2 a w_r) \in L \ .$$

This can be rewritten into $w_\ell (w_1 a)^j (w_2 a)^j w_r \in L$.

(4) $\Rightarrow$ (2): Let L be closed under right-synchronized power abbreviations. We will prove that A_L satisfies the right-synchronized containment property, that is, if there are two states q_1, q_2 in A_L with $q_1 \rightsquigarrow q_2$ and $w_1 \in \text{Loop}(q_1)$, $w_2 \in \text{Loop}(q_2)$, such that w_1 and w_2 end with the same letter, then $(w_2 a)^N L_{q_2} \subseteq L_{q_1}$. Let q_1, q_2 be such states. Then there exist w_ℓ, w_m with $q_1 = \delta_L(i_L, w_\ell)$ and $q_2 = \delta_L(q_1, w_m)$. If $L_{q_2} = \emptyset$, we are done. So let us assume there is a word $w_r \in L_{q_2}$. We define $w_r' = w_2^N w_r$. Due to construction, we have $w_\ell w_1^* w_m w_2^* w_r' \subseteq L$. Since L is closed under right-synchronized power abbreviations, there is an $i \in \mathbb{N}$ such that $w_\ell w_1^i w_2^i w_r' \in L$. Since we have a deterministic automaton and $q_1 = \delta_L(i_L, w_\ell w_1^i)$ this implies that $w_2^i w_r' = w_2^i w_2^N w_r \in L_{q_1}$. We now use that L is aperiodic due to Lemma 3.1.8 to infer that $w_2^N w_r \in L_{q_1}$.

(2) $\Rightarrow$ (1): We will show that if there exist states $q_1, q_2 \in Q_L$ and words $w_1, w_2 \in \Sigma^*$ with $a w_1 \in \text{Loop}(q_1)$ and $a w_2 \in \text{Loop}(q_2)$ and $q_1 \rightsquigarrow q_2$, then $(a w_2)^N L_{q_2} \subseteq L_{q_1}$. Let q_1, q_2 be such states and w_1, w_2 as above. We define $q_1' = \delta_L(q_1', w_1)$ and $q_2' = \delta_L(q_2', w_2)$. Since A_L is deterministic, the construction implies that $w_1 a \in \text{Loop}(q_1')$ and $w_2 a \in \text{Loop}(q_2')$. Furthermore, it holds that (i) $L_{q_1'} = a^{-1} L_{q_1}$ and (ii) $w_2 L_{q_2} \subseteq L_{q_2'}$. With this we will show that $(w_2 a)^N L_{q_2'} \subseteq L_{q_1'}$ implies $(a w_2)^N L_{q_2} \subseteq L_{q_1}$. Let $(w_2 a)^N L_{q_2'} \subseteq L_{q_1'}$. Adding an a left hand, yields $(a w_2)^N a L_{q_2'} \subseteq a L_{q_1'} \subseteq L_{q_1}$ because of (i). We use (ii) to replace $L_{q_2'}$ to get: $(a w_2)^{N+1} L_{q_2} \subseteq L_{q_1}$. Since L is aperiodic, see Lemma 3.1.7, this is equivalent to $(a w_2)^N L_{q_2} \subseteq L_{q_1}$. $\square$

Corollary 3.1.10. *If a regular language L satisfies Definition 3.1.3 and $N = |A_L|$ then, for all $i > N^2 + N$ and for all words $w_\ell, w_m, w_r \in \Sigma^*$ and all words $w_1 = aw_1'$ and $w_2 = aw_2'$ (respectively, $w_1 = w_1'a$ and $w_2 = w_2'a$) we have that $w_\ell w_1^i w_m w_2^i w_r \in L$ implies $w_\ell w_1^i w_2^i w_r \in L$.*

Proof. This immediately follows from the proof of (1) $\Rightarrow$ (3). $\square$

In Theorem 3.4.1 we will show that, if NL $\neq$ NP, the languages L that satisfy the above properties are precisely those for which $\mathsf{Trail}(L)$ is tractable. To simplify terminology, we will henceforth refer to this class as $\mathsf{T}_{\mathsf{tract}}$.

Definition 3.1.11. A regular language L belongs to $\mathsf{T}_{\mathsf{tract}}$ if L satisfies one of the equivalent conditions in Theorem 3.1.9.

For example, $(ab)^*$ and $(abc)^*$ are in $\mathsf{T}_{\mathsf{tract}}$, whereas a^*ba^*, $(aa)^*$ and $(aba)^*$ are not. The following property immediately follows from the definition of $\mathsf{T}_{\mathsf{tract}}$.

Observation 3.1.12. *Every regular expression for which each alphabet symbol under a Kleene star occurs at most once in the expression defines a language in $\mathsf{T}_{\mathsf{tract}}$.*

A special case of these expressions are those in which every alphabet symbol occurs at most once. These are known as *single-occurrence regular expressions (SORE)* [41]. SOREs were studied in the context of learning schema languages for XML [41], since they occur very often in practical schema languages.

3.2 A Syntactic Characterization of $\mathsf{T}_{\mathsf{tract}}$

As we have seen before, regular expressions in which every symbol occurs at most once define languages in $\mathsf{T}_{\mathsf{tract}}$. We will define a similar notion on automata.

Definition 3.2.1. A component C of some NFA A is called *memoryless*, if for each symbol $a \in \Sigma$, there is at most one state q in C, such that there is a transition (p, a, q) with p in C.

In this section, we will prove the following theorem which provides (in a non-trivial proof that requires several steps) a syntactic condition for languages in $\mathsf{T}_{\mathsf{tract}}$. The syntactic condition is item (4) of the theorem, which we define after its statement. Condition (5) emposes an additional restriction on condition (4).

Theorem 3.2.2. *For a regular language L, the following properties are equivalent:*

(1) $L \in \mathsf{T}_{\mathsf{tract}}$

(2) There exists an NFA A for L that satisfies the left-synchronized containment property.

(3) There exists an NFA A for L that satisfies the left-synchronized containment property and only has memoryless components.

(4) There exists a detainment automaton for L with consistent jumps.

(5) There exists a detainment automaton for L with consistent jumps and only memoryless components.

To define detainment automata, we use *finite automata with counters or CNFAs* from Gelade et al. [103], which we slightly adapt to make the construction easier. For convenience, we first recall the definition of counter NFAs from Gelade et al. [103]. We introduce a minor difference, namely that counters count down instead of up, since this makes our construction easier to describe. Furthermore, since our construction only requires a single counter, zero tests, and setting the counter to a certain value, we immediately simplify the definition to take this into account.

Let c be a *counter variable*, taking values in $\mathbb{N}$. A *guard* on c is a statement γ of the form true or $c = 0$. We denote by $c \models \gamma$ that c satisfies the guard γ. In the case where γ is true, this is trivially fulfilled and, in the case where γ is $c = 0$, this is fulfilled if c equals 0. By G we denote the set of guards on c. An *update* on c is a statement of the form $c := c - 1$, $c := c$, or $c := k$ for some constant $k \in \mathbb{N}$. By U we denote the set of updates on c.

Definition 3.2.3. A *non-deterministic counter automaton* (CNFA) with a single counter is a 6-tuple $A = (Q, I, c, \delta, F, \tau)$ where Q is the finite set of states; $I \subseteq Q$ is a set of initial states; c is a counter variable; $\delta \subseteq Q \times \Sigma \times G \times Q \times U$ is the transition relation; and $F \subseteq Q$ is the set of accepting states. Furthermore, $\tau \in \mathbb{N}$ is a constant such that every update is of the form $c := k$ with $k \leq \tau$.

Intuitively, A can make a transition $(q, a, \gamma; q', \pi)$ whenever it is in state q, reads a, and $c \models \gamma$, that is, guard γ is true under the current value of c. It then updates c according to the update π, in a way we explain next, and moves into state q'. To explain the update mechanism formally, we introduce the notion of configuration. A *configuration* is a pair (q, ℓ) where $q \in Q$ is the current state and $\ell \in \mathbb{N}$ is the value of c. Finally, an update π defines a function $\pi : \mathbb{N} \to \mathbb{N}$ as follows. If $\pi = (c := k)$ then $\pi(\ell) = k$ for every $\ell \in \mathbb{N}$. If $\pi = (c := c - 1)$ then $\pi(\ell) = \max(\ell - 1, 0)$. Otherwise, that is, if $\pi = (c := c)$, then $\pi(\ell) = \ell$. So, counters never become negative.

An *initial configuration* is $(q_0, 0)$ with $q_0 \in I$. A configuration (q, ℓ) is *accepting* if $q \in F$ and $\ell = 0$. A configuration $\alpha' = (q', \ell')$ *immediately follows* a configuration $\alpha = (q, \ell)$ by reading $a \in \Sigma$, denoted $\alpha \to_a \alpha'$, if there exists $(q, a, \gamma; q', \pi) \in \delta$ with $c \models \gamma$ and $\ell' = \pi(\ell)$.

For a word $w = a_1 \cdots a_n$ and two configurations α and α', we denote by $\alpha \Rightarrow_w \alpha'$ that $\alpha \to_{a_1} \cdots \to_{a_n} \alpha'$. A configuration α is *reachable* if there exists a word w such that $\alpha_0 \Rightarrow_w \alpha$ for some initial configuration α_0. A word w is *accepted* by A if $\alpha_0 \Rightarrow_w \alpha_f$ where α_0 is an initial configuration and α_f is an accepting configuration. We denote by $L(A)$ the set of words accepted by A. It is easy to see that CNFA accept precisely the regular languages. (Due to the value τ, counters are always bounded by a constant.)

We are now ready to define detainment automata. Let A be a CNFA with one counter c. Initially, the counter has value 0. The automaton has transitions of the form

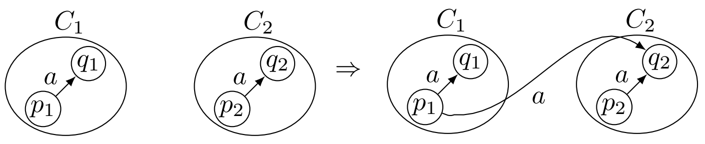


Figure 3.1: Consistent jump property (simplified, that is, without preconditions, counter and update) used in Theorem 3.3.3. C_1 and C_2 are components (not necessarily different) such that C_2 is reachable from C_1.

$(q_1, a, P; q_2, U)$ where P is a precondition on c and U an update operation on c. For instance, the transition $(q_1, a, c = 5; q_2, c := c - 1)$ means: if A is in state q_1, reads a, and the value of c is five, then it can move to q_2 and decrease c by one. If we decrease a counter with value zero, its value remains zero. We denote the precondition that is always fulfilled by true.

We say that A is a *detainment automaton* if, for every component C of A:

- every transition inside C is of the form $(q_1, a, \mathsf{true}; q_2, c := c - 1)$;
- every transition that leaves C is of the form $(q_1, a, c = 0; q_2, c := k)$ for some $k \in \mathbb{N}$;[1]

Intuitively, if a detainment automaton enters a non-trivial component C, then it must stay there for at least some number of steps, depending on the value of the counter c. The counter c is decreased for every transition inside C and the automaton can only leave C once $c = 0$. We say that A has the *consistent jump property* if, for every pair of components C_1 and C_2, if $C_1 \rightsquigarrow C_2$ and there are transitions $(p_i, a, \mathsf{true}; q_i, c := c - 1)$ inside C_i for all $i \in \{1, 2\}$, then there is also a transition $(p_1, a, P; q_2, U)$ for some $P \in \{\mathsf{true}, c = 0\}$ and some update U.[2] We note that C_1 and C_2 may be the same component. The consistent jump property is the syntactical counterpart of the left-synchronized containment property. The *memoryless* condition carries over naturally to CNFAs, ignoring the counter.

The next lemmas characterize the internal language of a component and are needed to show the "memoryless" in Theorem 3.2.2.

Lemma 3.2.4. *Let $L \in \mathsf{T}_{\mathsf{tract}}$, $a \in \Sigma$, C be a component of A_L, and $q_1, q_2 \in C$. If there exist $w_1 a \in \mathrm{Loop}(q_1)$ and $w_2 a \in \mathrm{Loop}(q_2)$, then, for all $\sigma \in \Sigma$, we have that $\delta_L(q_1, \sigma) \in C$ if and only if $\delta_L(q_2, \sigma) \in C$.*

Proof. Let $q_1 \neq q_2$ be two states in C. Let σ satisfy $\delta_L(q_1, \sigma) \in C$ and let $w \in \mathrm{Loop}(q_1) \cap \sigma\Sigma^* a$. Such a w exists since $\delta_L(q_1, \sigma) \in C$ and $\delta_L(q_1, w_1 a) = q_1$. Let $q_3 = \delta_L(q_2, w^N)$. We will prove that $q_1 = q_3$, which implies that $\delta_L(q_2, \sigma) \in C$. As L is aperiodic, $w \in \mathrm{Loop}(q_3)$. Consequently, there is an $n \in \mathbb{N}$ such that $w^n L_{q_3} \subseteq L_{q_1}$ by Definition 3.1.1. Since $w \in \mathrm{Loop}(q_1)$, this also implies $L_{q_3} \subseteq L_{q_1}$. Furthermore, q_2 has

[1] If q_2 is a trivial component, then k should be 0 for the transition to be useful.
[2] The values of P and U depend on whether C_1 is the same as C_2 or not.

a loop ending with a and A_L satisfies the right-synchronized containment property, so $w^N L_{q_1} \subseteq L_{q_2}$ by Corollary 3.1.5. Hence, $L_{q_1} \subseteq (w^N)^{-1} L_{q_2}$ and, by definition of q_3, we have $(w^N)^{-1} L_{q_2} = L_{q_3}$. So we showed $L_{q_3} \subseteq L_{q_1}$ and $L_{q_1} \subseteq L_{q_3}$ which, by minimality of A_L, implies $q_1 = q_3$. □

The following is a direct consequence thereof.

Corollary 3.2.5. *Let $L \in \mathsf{T}_{\mathsf{tract}}$, $a \in \Sigma$, C be a component of A_L, and $q_1, q_2 \in C$. If there exist $w_1 a \in \mathrm{Loop}(q_1)$ and $w_2 a \in \mathrm{Loop}(q_2)$, then $\delta_L(q_1, w) \in C$ if and only if $\delta_L(q_2, w) \in C$ for all words $w \in \Sigma^*$.*

Lemma 3.2.6. *Let A_L satisfy the left-synchronized containment property. If states q_1 and q_2 belong to the same component of A_L and $\mathrm{Loop}(q_1) \cap \mathrm{Loop}(q_2) \neq \emptyset$, then $q_1 = q_2$.*

Proof. Let q_1, q_2 be as stated and let w be a word in $\mathrm{Loop}(q_1) \cap \mathrm{Loop}(q_2)$. According to Definition 3.1.1, there exists an $n \in \mathbb{N}$ such that $w^n L_{q_2} \subseteq L_{q_1}$. Since $w \in \mathrm{Loop}(q_1)$, this implies that $L_{q_2} \subseteq L_{q_1}$. By symmetry, we have $L_{q_2} = L_{q_1}$, which implies $q_1 = q_2$, since A_L is the minimal DFA. □

To this end, we use the following synchronization property for A_L.

Lemma 3.2.7. *Let $L \in \mathsf{T}_{\mathsf{tract}}$, let C be a component of A_L, let $q_1, q_2 \in C$, and let w be a word of length N^2. If $\delta_L(q_1, w) \in C$ and $\delta_L(q_2, w) \in C$, then $\delta_L(q_1, w) = \delta_L(q_2, w)$.*

Proof. Assume that $w = a_1 \cdots a_{N^2}$. For each i from 0 to N^2 and $\alpha \in \{1, 2\}$, let $q_{\alpha,i} = \delta_L(q_\alpha, a_1 \cdots a_i)$. Since there are at most N^2 distinct pairs $(q_{1,i}, q_{2,i})$, there exist i, j with $0 \leq i < j \leq N^2$ such that $q_{1,i} = q_{1,j}$ and $q_{2,i} = q_{2,j}$. Since $\delta_L(q_1, w) \in C$ and $\delta_L(q_2, w) \in C$, $q_{1,i}, q_{2,i} \in C$. Let $w' = a_{i+1} \cdots a_j$. We have $w' \in \mathrm{Loop}(q_{1,i}) \cap \mathrm{Loop}(q_{2,i})$, hence $q_{1,i} = q_{2,i}$ by Lemma 3.2.6. As a consequence, $\delta_L(q_1, w) = \delta_L(q_2, w)$. □

Furthermore, we show that every language in $\mathsf{T}_{\mathsf{tract}}$ satisfies an inclusion property which is stronger than indicated by Definition 3.1.1. That is, we show that it is not necessary to repeat some word w_2 multiple times. Instead, we show that any word w that stays in a component, given that w is long enough and starts with a suitable symbol, already implies an inclusion property.

Lemma 3.2.8. *Let $L \in \mathsf{T}_{\mathsf{tract}}$ and let q_1, q_2 be two states such that $q_1 \rightsquigarrow q_2$ and $\mathrm{Loop}(q_1) \cap a\Sigma^* \neq \emptyset$. Let C be the component of A_L that contains q_2. Then,*

$$L_{q_2} \cap L^a_{q_2} \Sigma^* \subseteq L_{q_1}$$

where $L^a_{q_2}$ is the set of words w of length N^2 that start with a and such that $\delta_L(q_2, w) \in C$.

Proof. If $\mathrm{Loop}(q_2) = \emptyset$, then $L_{q_2} \cap L^a_{q_2} \Sigma^* = \emptyset$ and the inclusion trivially holds. Therefore we assume from now on that $\mathrm{Loop}(q_2) \neq \emptyset$. Since the proof of this lemma requires a number of different states and words, we provide a sketch in Figure 3.2. Let $w \in L_{q_2} \cap L^a_{q_2} \Sigma^*$, u be the prefix of w of length N^2 and w' be the suffix of w such that

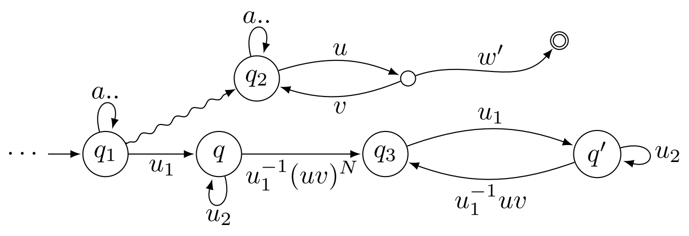


Figure 3.2: Sketch of the proof of Lemma 3.2.8

$w = uw'$. Since q_2 and $\delta_L(q_2, u)$ are both in the same component C, there exists a word v with $uv \in \mathrm{Loop}(q_2)$. Corollary 3.1.5 implies that

$$(uv)^N L_{q_2} \subseteq L_{q_1} \; . \tag{3.1}$$

Let $q_3 = \delta_L(q_1, (uv)^N)$. Due to aperiodicity we have $uv \in \mathrm{Loop}(q_3)$. Since A_L is deterministic, this implies $L_{q_3} = ((uv)^N)^{-1} L_{q_1}$ and, together with Equation (3.1) that

$$L_{q_2} \subseteq L_{q_3} \; . \tag{3.2}$$

We now show that there is a prefix u_1 of u such that $\delta_L(q_1, u_1) = q$ and $\delta_L(q_3, u_1) = q'$ with $\mathrm{Loop}(q) \cap \mathrm{Loop}(q') \neq \emptyset$. Assume that $u = a_1 \cdots a_{N^2}$. Let $q_{\alpha,0} = q_\alpha$ and, for each i from 1 to N^2 and $\alpha \in \{1, 3\}$, let $q_{\alpha,i} = \delta_L(q_\alpha, a_1 \cdots a_i)$. Since there are at most N^2 distinct pairs $(q_{1,i}, q_{3,i})$, there exist i, j with $0 \leq i < j \leq N^2$ such that $q_{1,i} = q_{1,j}$ and $q_{3,i} = q_{3,j}$. Let $u_1 = a_1 \cdots a_i$ and $u_2 = a_{i+1} \cdots a_j$. We have $u_2 \in \mathrm{Loop}(q_{1,i}) \cap \mathrm{Loop}(q_{3,i})$. We define $q := \delta_L(q_1, u_1)$ and $q' = \delta_L(q_3, u_1)$. Since $q \rightsquigarrow q'$ and $u_2 \in \mathrm{Loop}(q) \cap \mathrm{Loop}(q')$, Corollary 3.1.5 implies $u_2^N L_{q'} \subseteq L_q$. Since $u_2 \in \mathrm{Loop}(q)$, we also have that

$$L_{q'} \subseteq L_q \; . \tag{3.3}$$

By definition of q and the determinism of A_L, we have that $L_q = u_1^{-1} L_{q_1}$. Thus, Equation (3.3) implies $L_{q'} \subseteq u_1^{-1} L_{q_1}$. The definition of q' implies that $L_{q'} = u_1^{-1} L_{q_3}$, so $u_1^{-1} L_{q_3} \subseteq u_1^{-1} L_{q_1}$. In other words, we have $L_{q_3} \cap u_1 \Sigma^* \subseteq L_{q_1} \cap u_1 \Sigma^*$. Since u_1 is a prefix of u, and by Equation (3.2), we also have $L_{q_2} \cap u\Sigma^* \subseteq L_{q_1}$. This implies that $w \in L_{q_1}$, which concludes the proof. $\square$

The following lemma is the implication (1) $\Rightarrow$ (5) from Theorem 3.2.2

Lemma 3.2.9. *If* $L \in \mathsf{T}_{\mathsf{tract}}$*, then there exists a detainment automaton for* L *with consistent jumps and only memoryless components.*

Proof. Let $A_L = (Q_L, \Sigma, i_L, F_L, \delta_L)$ be the minimal DFA for L. The proof goes as follows: First, we define a CNFA A with two counters. Second, we show that we can convert A to an equivalent CNFA A' with only one counter that is a detainment automaton with consistent jumps and only memoryless components. This conversion is done by simulating one of the counters using a bigger set of states. Last, we show that $L(A) = L(A_L)$, which shows the lemma statement as $L(A) = L(A')$.

Before we start we need some additional notation. For a number $k \in \mathbb{N}$, we abbreviate $\{0, \ldots, k\}$ with $[0, k]$. We write $p_1 \curvearrowright^a q_2$ to denote that $C(p_1) \rightsquigarrow C(q_2)$ and there are states $q_1 \in C(p_1)$ and $p_2 \in C(q_2)$ such that $(p_i, a, q_i) \in \delta_L$ for $i \in \{1, 2\}$. Let q be a state, then we write $\Sigma^{\circlearrowright}(q)$ to denote the set of symbols a, such that there is a word $w = aw' \in \text{Loop}(q)$.

Let $\sim \subseteq Q_L \times Q_L$ be the smallest equivalence relation over Q_L that satisfies $p \sim q$ if $C(p) = C(q)$ and $\Sigma^{\circlearrowright}(p) \cap \Sigma^{\circlearrowright}(q) \neq \emptyset$. For $q \in Q_L$, we denote by $[q]$ the equivalence class of q. By $[Q_L]$ we denote the set of all equivalence classes. We also write $[C]$ to denote the equivalence classes that only use states from some component C. We extend the notion $C(q)$ to $[Q_L]$, that is, $C([q]) = C(q)$ for all $q \in Q_L$.

We will use the following observation that easily follows from Lemma 3.2.4 using the definition of $\sim$.

Observation 3.2.10. *Let q_1, q_2 be states with $[q_1] = [q_2]$, then for all $a \in \Sigma$ it holds that $\delta_L(q_1, a) \in C(q_1)$ if and only if $\delta_L(q_2, a) \in C(q_1)$.*

We define a CNFA $A = (Q, I, c, d, \delta, F, N^2)$ that has two counters c and d. The counter c is allowed to have any initial value from $[0, N^2]$, while the counter d has initial value 0. We note that we will eliminate counter c when converting to a one counter automaton, thus this is not a contradiction to the definition of CNFA with one counter that we use.

We use $Q' = Q_L \cup [Q_L]$, that is, we can use the states from A_L and the equivalence classes of the equivalence relation $\sim$. The latter will be used to ensure that components are memoryless, while the former will only be used in trivial components. We use $I = \{i_L, [i_L]\}$ and $F = F_L$.

$$
\begin{aligned}
\delta^1_{\circlearrowright} &= \{(q_1, a, \{c > 0, d = 0\}; q_2, \{c := c - 1\}) \mid (q_1, a, q_2) \in \delta_L, C(q_1) = C(q_2)\} \\
\delta^2_{\circlearrowright} &= \{([q_1], a, \{c = N^2\}; [q_2], \{d := d - 1\}) \mid (q_1, a, q_2) \in \delta_L, C(q_1) = C(q_2)\} \\
\delta^3_{\circlearrowright} &= \{([q_1], a, \{c = N^2, d = 0\}; q_2, \{c := c - 1\}) \mid (q_1, a, q_2) \in \delta_L, C(q_1) = C(q_2)\} \\
\delta^1_{\rightarrow} &= \{(q_1, a, \{c = 0, d = 0\}; q_2, \{c := i\}) \mid (q_1, a, q_2) \in \delta_L, C(q_1) \neq C(q_2), i \in [0, N^2 - 1]\} \\
\delta^2_{\rightarrow} &= \{(q_1, a, \{c = 0, d = 0\}; [q_2], \{c := N^2\}) \mid (q_1, a, q_2) \in \delta_L, C(q_1) \neq C(q_2)\} \\
\delta_{\curvearrowright} &= \{([q_1], a, \{c = N^2, d = 0\}; [q_2], \{d := N^2\}) \mid q_1 \curvearrowright^a q_2, C(q_1) \neq C(q_2)\} \\
\delta \enspace &= \delta^1_{\circlearrowright} \cup \delta^2_{\circlearrowright} \cup \delta^3_{\circlearrowright} \cup \delta^1_{\rightarrow} \cup \delta^2_{\rightarrow} \cup \delta_{\curvearrowright}
\end{aligned}
$$

We say that a component C of A_L is a *long run component* of a given word $w = a_1 \cdots a_n$, if $|\{i \mid \delta(i_L, a_1 \cdots a_i) \in C\}| > N^2$, that is, if the run stays in C for more than N^2 symbols. All other components are *short run components.*

For short run components, we use states from Q_L. We use the counter c to enforce that these parts are indeed short. For long run components, we first use states in $[Q_L]$. Only the last N^2 symbols in the component are read using states from Q_L. The left-synchronized containment property guarantees that for long run components the precise state is not important, which allows us to make these components memoryless.

The transition relation is divided into transitions between states from the same component of A_L (indicated by $\delta_{\circlearrowright} = \delta^1_{\circlearrowright} \cup \delta^2_{\circlearrowright} \cup \delta^3_{\circlearrowright}$) and transitions between different components (indicated by $\delta_{\rightarrow} = \delta^1_{\rightarrow} \cup \delta^2_{\rightarrow}$). Transitions in $\delta_{\curvearrowright}$ are added to satisfy the consistent jumps property. They are the only transitions that increase the counter d. This is necessary, as the left-synchronized containment property only talks about the language of the state reached after staying in the component for some number of symbols. If we added the transitions in $\delta_{\curvearrowright}$ without using the counter, we would possibly add additional words to the language. This concludes the definition of A.

We now argue that the automaton $A' = (Q' \times [0, N^2], i_L, d, \delta', F \times \{0\}, N^2)$ derived from A by pushing the counter c into the states is a detainment automaton with consistent jumps that only has memoryless components. The states of A' have two components, first the state of A and second the value of the second counter that is bounded by N^2. We do not formally define δ'. It is derived from δ in the obvious way, that is, by doing the precondition checks that depend on c on the second component of the state. Similarly, updates of c are done on the second component of the states.

It is straightforward to see that A' is a detainment automaton with consistent jumps that only has memoryless components using the following observations:

- Every transition in A that does not have $c = N^2$ before and after the transition requires $d = 0$.

- Let Cuts be the set of components of A, then the set of components of A' is $\{[C] \times \{N^2\} \mid C \in \mathsf{Cuts}\}$.

The consistent jumps are guaranteed by the transitions in $\delta_{\curvearrowright}$. As A' only has memoryless components, the consistent jump property is trivially satisfied for states inside the same component.

We now show that $L(A_L) \subseteq L(A)$. Let $w = a_1 \cdots a_n$ be some word in $L(A_L)$ and $q_0 \to \cdots \to q_n$ be the run of A_L on w. $\mathsf{countdown}\colon \mathbb{N} \to \mathbb{N}$ that gives us how long we stay inside some component as $\mathsf{countdown}\colon i \mapsto j - i$, where j is the largest number such that $C(q_j) = C(q_i)$.

It is easy to see by the definitions of the transitions in $\delta_{\rightarrow}$ and $\delta_{\circlearrowright}$, that the run

$$\big(p_0, \min(N^2, \mathsf{countdown}(0)), 0\big) \;\to\; \cdots \;\to\; \big(p_n, \min(N^2, \mathsf{countdown}(n)), 0\big)$$

is an accepting run of A, where p_i is q_i if $c_i < N^2$ and $[q_i]$ otherwise. We note that the counter d is always zero, as we do not use any transitions from $\delta_{\curvearrowright}$. The transitions in $\delta_{\curvearrowright}$ are only there to satisfy the consistent jumps property. This shows $L(A_L) \subseteq L(A)$.

Towards the lemma statement, it remains to show that $L(A) \subseteq L(A_L)$. Let therefore $w = a_1 \cdots a_n$ be some word in $L(A)$, $(p_0, c_0, d_0) \to \cdots \to (p_n, c_n, d_n)$ be an accepting run of A, and $q_0 \to \cdots \to q_n$ be the unique run of A_L on w.

We now show by induction on i that there are states $\hat{q}_1, \ldots, \hat{q}_n$ in Q_L such that the following claim is satisfied. The claim easily yields that $q_n \in F_L$, as both counters have to be zero for the word to be accepted.

$$L_{\hat{q}_i} \cap a_{i+1} \cdots a_{i+d_i}\Sigma^* \subseteq L_{q_i} \text{ and } \hat{q}_i \in \begin{cases} \{p_i\} & \text{if } c_i = d_i = 0 \\ [p_i] & \text{if } c_i + d_i > 0 \text{ and } p_i \in Q_L \\ p_i & \text{if } c_i + d_i > 0 \text{ and } p_i \in [Q_L] \end{cases}$$

The base case $i = 0$ is trivial by the definition of I. We now assume that the induction hypothesis holds for i and are going to show that it holds for $i + 1$. Let $\rho = (p_{i-1}, a_i, P; p_i, U)$ be the transition used to read a_i. We distinguish several cases depending on ρ.

Case $\rho \in \delta_{\rightarrow}$: In this case, $c_i = 0$ by the definition of $\delta_{\rightarrow}$. Therefore, the claim for $i+1$ follows with $\hat{q}_{i+1} = p_{i+1}$, as $\hat{q}_i = p_i$ by the induction hypothesis and $(p_i, a, p_{i+1}) \in \delta_L$ by the definition of $\delta_{\rightarrow}$.

Case $\rho \in \delta^2_{\circlearrowright}$: We note that $p_i, p_{i+1} \in [Q_L]$. The claim for $i + 1$ follows with $\hat{q}_{i+1} = \delta(\hat{q}_i, a_{i+1})$ using $C(q') = C(\delta(q', a_{i+1})$ for all $q' \in [p_i]$ (by Observation 3.2.10), $C(p_i) = C(p_{i+1})$ (by definition of $\delta_{\circlearrowright}$), and $\hat{q}_i \in p_i$ (by the induction hypothesis).

Case $\rho \in \delta^3_{\circlearrowright}$: We want to show that $L_{p_{i+N^2}} \subseteq L_{q_{N^2}}$ establishing the claim directly for the position $i + N^2$ using $\hat{q}_{i+N^2} = p_{i+N^2}$. Therefore, we first want to apply Lemma 3.2.7 to show that $\delta(\hat{q}_i, a_{i+1} \cdots a_{i+N^2}) = p_{i+N^2}$. The preconditions of the lemma require us to show that (i) $C(\hat{q}_i) = C(p_i)$, (ii) $C(p_i) = C(p_{i+N^2})$, and (iii) $C(\hat{q}_i) = C(\delta_L(\hat{q}_i, a_{i+1} \cdots a_{i+N^2}))$. Precondition (i) is given by the induction hypothesis, precondition (ii) is by the definition of $\delta_{\circlearrowright}$, that is, that all transitions in $\delta_{\circlearrowright}$ are inside the same component of A_L, and precondition (iii) is by the fact that each transition in $\delta_{\circlearrowright}$ has a corresponding transition in δ_L that stays in the same component. Therefore, we can actually apply Lemma 3.2.7 to conclude that $\delta(\hat{q}_i, a_{i+1} \cdots a_{i+N^2}) = p_{i+N^2}$. As we furthermore have that $L_{\hat{q}_i} \cap a_{i+1} \cdots a_{i+d_i}\Sigma^* \subseteq L_{q_i}$ by the induction hypotheses, we can conclude that $L_{p_{i+N^2}} \subseteq L_{q_{N^2}}$. This establishes the claim for position $i + N^2$ using $\hat{q}_{i+N^2} = p_{i+N^2}$. As we only need the claim for position n (and not for all smaller positions), we can continue the induction at position $i + N^2$. Especially there is no need to look at the case where $\rho \in \delta^1_{\circlearrowright}$.

Case $\rho \in \delta_{\curvearrowright}$: By the definition of $\delta_{\curvearrowright}$, we have that $p_i, p_{i+1} \in [Q_L]$. Furthermore, there are transitions (p_i, a_{i+1}, p') and (p'', a_{i+1}, p_{i+1}) in δ_L such that $C(p') = C(p_i)$, $C(p'') = C(p_{i+1})$, and $p' \rightsquigarrow p''$. This (and the fact that $\hat{q}_i \in p_i$ by the induction hypothesis) allows us to apply Observation 3.2.10 , which yields $\delta(\hat{q}_i, a_{i+1}) \in C(p_i)$. From $p' \rightsquigarrow p''$ and $\hat{q}_i \in C(p')$ we can conclude that $\hat{q}_i \rightsquigarrow p''$. We now can apply Lemma 3.2.8 that gives us $L_{p''} \cap L^{a_{i+1}}_{p''}\Sigma^* \subseteq L_{\hat{q}_i}$.

Now we argue that the subword $a_{i+2} \cdots a_{i+N^2+1}$ is in $L^{a_{i+1}}_{p''}$. By the definition of $\delta_{\curvearrowright}$, we have $d_{i+1} = N^2$, enforcing that the next N^2 transitions are all from $\delta^2_{\circlearrowright}$, as these are the only transitions that allow $d > 0$ in the precondition. Applying Observation 3.2.10 N^2 times yields that $\delta(p'', a_{i+2} \cdots a_{i+N^2+1}) \in C(p'')$ and therefore $a_{i+2} \cdots a_{i+N^2+1} \in L^{a_{i+1}}_{p''}$. Using this and $L_{p''} \cap L^{a_{i+1}}_{p''}\Sigma^* \subseteq L_{\hat{q}_i}$, we get that $L_{\delta(p'', a_{i+1})} \cap a_{i+2} \cdots a_{i+N^2+1}\Sigma^* \subseteq L_{\delta(\hat{q}_i, a_{i+1})}$ yielding the claim for $i + 1$. This concludes the proof of the lemma. □

Proof sketch of Theorem 3.2.2. The implications (3) $\Rightarrow$ (2) and (5) $\Rightarrow$ (4) are trivial. We sketch the proofs of (1) $\Rightarrow$ (5) $\Rightarrow$ (3) and (4) $\Rightarrow$ (2) $\Rightarrow$ (1) below, establishing the theorem.

(1) $\Rightarrow$ (5) uses a very technical construction that essentially exploits that—if the automaton stays in the same component for a long time—the reached state only depends on the last N^2 symbols read in the component. This is formalized in Lemma 3.2.7 and allows us to merge any pair of two states p, q which contradict that some component is memoryless. To preserve the language, words that stay in some component C for less than N^2 symbols have to be dealt with separately, essentially avoiding the component altogether. Finally, the left-synchronized containment property allows us to simply add transitions required to satisfy the consistent jumps property without changing the language.

(5) $\Rightarrow$ (3) and (4) $\Rightarrow$ (2): We convert a given CNFA to an NFA by simulating the counter (which is bounded) in the set of states. The consistent jump property implies the left-synchronized containment property on the resulting NFA. The property that all components are memoryless is preserved by the construction.

(2) $\Rightarrow$ (1): One can show that the left-synchronized containment property is invariant under the powerset construction.

Proof of Theorem 3.2.2. We show (1) $\Rightarrow$ (5) $\Rightarrow$ (3) $\Rightarrow$ (2) $\Rightarrow$ (1) and (5) $\Rightarrow$ (4) $\Rightarrow$ (2).

(1) $\Rightarrow$ (5): Holds by Lemma 3.2.9.

(5) $\Rightarrow$ (3) and (4) $\Rightarrow$ (2): Let $A = (Q, I, c, \delta, F, \ell)$ be a detainment automaton with consistent jumps. We compute an equivalent NFA $A' = (Q \times \{0, \dots, \ell\}, \delta', I \times \{0\}, F \times \{0\})$ in the obvious way, that is, $((p, i), a, (q, j)) \in \delta'$ if and only if A can go from configuration (p, i) to configuration (q, j) reading symbol a. By the definition of detainment automata, we get that the components of A' are

$$\{\ C \times \{0\} \ \mid\ C \text{ is a component of } A\ \}$$

This directly shows that A' only has memoryless components if A only has memoryless components.

To prove the left-synchronizing containment property, we choose $n = \ell$. Let now $(q_1, c_1), (q_2, c_2) \in Q \times \{0, \dots, \ell\}$, $a \in \Sigma$, and $w_1', w_2' \in \Sigma^*$ be such that $(q_1, c_1) \rightsquigarrow (q_2, c_2)$, $w_1 = aw_1' \in \text{Loop}((q_1, c_1))$, and $w_2 = aw_2' \in \text{Loop}((q_2, c_2))$. We have to show that

$$w_2^n L_{(q_2,c_2)} \ \subseteq\ L_{(q_1,c_1)}\ . \tag{3.4}$$

We distinguish two cases. If q_1 and q_2 are in the same component, we know that there is a transition $(q_1, a, \mathsf{true}; q_3; c := c - 1) \in \delta$, as A has consistent jumps. Therefore, there is a transition $((q_1, 0), a, (q_2, 0)) \in \delta'$, which directly yields (3.4).

If q_1 and q_2 are in different components, then there is a transition $(q_1, a, c = 0; q_3; c := k) \in \delta$, as A has consistent jumps. Therefore, there is a transition $((q_1, 0), a, (q_2, k)) \in \delta'$ for some $k \in [0, \ell]$. We have $w_2 \in \text{Loop}(q_2)$. The definition of detainment automata requires that every transition inside a component—thus every transition used to read

w_2 using the loop—is of the form $(p, a, \mathsf{true}; q, c := c-1)$, that is, it does not have a precondition and it decreases the counter by one. Therefore in A', we have that $\delta'((q_2, k), w^\ell) \supseteq \delta((q_2, 0), w^\ell)$. This concludes the proof of (5) ⇒ (3) and (4) ⇒ (2)

(5) ⇒ (4) and (3) ⇒ (2): Trivial.

(2) ⇒ (1): Let $A = (Q, \Sigma, \delta, I, F)$ be an NFA satisfying the left-synchronized containment property and A_L be the minimal DFA equivalent to A. We show that A_L satisfies the left-synchronized containment property establishing (1).

Let M be the left synchronizing-length of A and $q_1, q_2 \in Q_L$ be states of A_L such that

- $q_1 \rightsquigarrow q_2$; and
- there are words $w_1 \in \text{Loop}(q_1)$ and $w_2 \in \text{Loop}(q_2)$ that start with the same symbol a.

We need to show that there exists an $n \in \mathbb{N}$ with $w_2^n L_{q_2} \subseteq L_{q_1}$. Let w be a word such that $\delta(q_1, w) = q_2$. Let $P_1 \subseteq Q$ be a state of the powerset automaton of A with $L_{P_1} = L_{q_1}$ and let $P_2 = \delta(P_1, ww_2^*)$ be the state in the powerset automaton of A that consists of all states reachable from P_1 reading some word from ww_2^*.

It holds that $L_{P_2} = L_{q_2}$, as $\delta(q_1, ww_2^*) = q_2$ and $L_{q_1} = L_{P_1}$.

We define

$$\begin{aligned} P_2' &= \{\, p \in P_2 \mid w_2^i \in \text{Loop}(p) \text{ for some } i > 0 \,\} \\ P_2'' &= \delta(P_1, w_2^{|A|}) \end{aligned}$$

We obviously have $P_2'' \subseteq P_2' \subseteq P_2$. Furthermore, we have

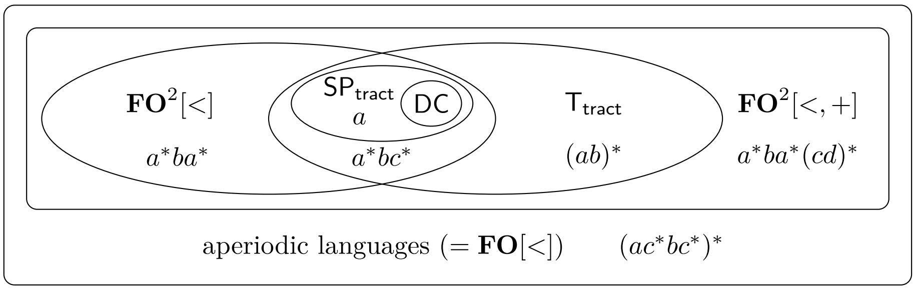
$$L_{P_2} \;=\; L_{q_2} \;=\; L_{\delta(q_2, w_2^{|A|})} \;=\; L_{P_2''}$$

The second equation is by $\delta(q_2, w_2^{|A|}) = q_2$. We can conclude that $L_{q_2} = L_{P_2'}$.

Let $\rho\colon Q \to Q$ be a function that selects for every state $p_2 \in P_2'$ a state $p_1 \in P_1$ such that $p_1 \rightsquigarrow p_2$. By definition of P_2', such states exist. Using the fact that A satisfies the left-synchronized containment property, we get that $w_2^M L_{p_2} \subseteq L_{\rho(p_2)}$ for each $p_2 \in P_2$. We can conclude

$$w_2^M L_{q_2} \;=\; w_2^M L_{P_2'} \;=\; \bigcup_{p_2 \in P_2'} w_2^M L_{p_2} \;\subseteq\; \bigcup_{p_2 \in P_2'} L_{\rho(p_2)} \;\subseteq\; L_{P_1} \;=\; L_{q_1}$$

and therefore $w_2^{|A|+M} L_{q_2} \subseteq L_{q_1}$. So A_L satisfies the left-synchronized containment property with $n = M$, where M is the left synchronizing-length of A. This concludes the proof for (2) ⇒ (1) and thus the proof of the theorem. □



$\mathbf{FO}^2[<]$
a^*ba^*
$\mathsf{SP}_{\mathsf{tract}}$
a
DC
a^*bc^*
$\mathsf{T}_{\mathsf{tract}}$
$(ab)^*$
$\mathbf{FO}^2[<,+]$
$a^*ba^*(cd)^*$
aperiodic languages ($= \mathbf{FO}[<]$)
$(ac^*bc^*)^*$

Figure 3.3: Expressiveness of subclasses of the aperiodic languages

3.3 Comparison of $\mathsf{T}_{\mathsf{tract}}$ to Other Classes

From Definition 2.5.4 it is easy to see that every language in $\mathsf{SP}_{\mathsf{tract}}$ is also in $\mathsf{T}_{\mathsf{tract}}$, since our definition imposes an extra "synchronizing" condition on w_1 and w_2, namely that they share the same first (or last) symbol (Definition 3.1.3).

We now fully classify the expressiveness of $\mathsf{T}_{\mathsf{tract}}$ and $\mathsf{SP}_{\mathsf{tract}}$ compared to yardsticks as DC, $\mathbf{FO}[<]$, $\mathbf{FO}^2[<]$, and $\mathbf{FO}^2[<,+]$ (see also Figure 3.3). Some of these classes are used throughout the work and defined in Section 2.5. Here, $\mathbf{FO}^2[<]$ and $\mathbf{FO}^2[<,+]$ are the two-variable restrictions of $\mathbf{FO}[<]$ and $\mathbf{FO}[<,+]$ over words, respectively. By $\mathbf{FO}[<,+]$ we mean the first-order logic with unary predicates P_a for all $a \in \Sigma$ (denoting positions carrying the letter a) and the binary predicates $+1$ and $<$ (denoting the successor relation and the order relation among positions). $\mathbf{FO}[<]$ is $\mathbf{FO}[<,+]$ without the binary predicate $+1$.

While some of the following proofs are relatively easy: for example, every downward closed (DC) language is in $\mathsf{T}_{\mathsf{tract}}$, since $\mathsf{T}_{\mathsf{tract}}$ relaxed the containment property, others need some algebraic techniques. We refer the reader to the book [177] for a general overview of syntactic semigroups and the different hierarchies. We will need the following notation. The syntactic preorder of a language L of Σ^* is the relation $\leq_L$ defined on Σ^* by $x \leq_L y$ if and only if for all $u, v \in \Sigma^*$ we have $uxv \in L \Rightarrow uyv \in L$. The syntactic congruence of L is the associated equivalence relation $\sim_L$ defined by $x \sim_L y$ if and only if $x \leq_L y$ and $y \leq_L x$. The quotient $\Sigma^+/\sim_L$ ($\Sigma^*/\sim_L$) is called the syntactic semigroup (monoid) of L. A word $e \in \Sigma^*$ is *idempotent* if $e^2 = e$. Given a finite semigroup S, it is folklore that there is an integer $\omega(S)$ (denoted by ω when S is understood) such that for all $s \in S$, s^ω is idempotent. More precisely, s^ω is the limit of the Cauchy sequence $(s^{n!})_{n \geq 0}$.

Using this notation, we can give yet another characterization of $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$.

Observation 3.3.1.

(1) A language L is in $\mathsf{SP}_{\mathsf{tract}}$ if and only if its syntactic semigroup satisfies $x^\omega u y^\omega \leq x^\omega y^\omega$, for all non-empty words x, y.

(2) A language L is in $\mathsf{T}_{\mathsf{tract}}$ if and only if its syntactic semigroup satisfies $(xy)^\omega u(xz)^\omega \leq (xy)^\omega(xz)^\omega$, for every non-empty word x.

Proof. Item (1) follows from Definition 2.5.4 and the observation that if there exists an i for which Definition 2.5.4 holds, then it also holds for each $i' \geq i$. This can easily be seen by choosing $w'_\ell = w_\ell w_1^{i'-i}$ and $w'_r = w_2^{i'-i} w_r$. Item (2) follows from Theorem 3.1.9, Definition 3.1.3 and the paragraph after it. □

With this notation we can now prove the next lemma.

Lemma 3.3.2. $\mathsf{T}_{\mathsf{tract}} \subseteq \mathbf{FO}^2[<,+]$

Proof. Let L be a language in $\mathsf{T}_{\mathsf{tract}}$ and A_L be the minimal DFA of L. We use the characterization of $\mathbf{FO}^2[<,+]$ from Place and Segoufin [179]: A language L is definable in $\mathbf{FO}^2[<,+]$ if and only if the syntactic semigroup of L satisfies

$$(esete)^\omega = (esete)^\omega t (esete)^\omega \qquad (\ddagger)$$

for each $e, s, t \in \Sigma^+$ with e being idempotent.

In particular, L is expressible in $\mathbf{FO}^2[<,+]$ if the above equivalence holds for all $e, s, t \in \Sigma^+$, that is, dropping the condition that e is an idempotent.

Let $L \in \mathsf{T}_{\mathsf{tract}}$. We show that there exists a n' such that for each $n \geq n'$ and all words $u, v \in \Sigma^*$ and all $e, s, t \in \Sigma^*$ it holds that $u(esete)^n v = u(esete)^n t (esete)^n v \in L$.

We choose $n \geq 2N^2$. Let $q = \delta(i_L, u(esete)^{n/2})$ be the state after reading $u(esete)^{n/2}$ in A_L. By standard pumping arguments, we know that we have read the last copy of $esete$ inside some nontrivial component C of A_L. By Corollary 3.2.5, we can conclude that $\delta(q, (esete)^{n/2}) \in C$, $\delta(q, (esete)^{n/2} t) \in C$ and $\delta(q, (esete)^{n/2} t (esete)^n) \in C$. By Lemma 3.2.7, we can conclude that $\delta(q, (esete)^{n/2}) = \delta(q, (esete)^{n/2} t (esete)^n)$, which yields ($\ddagger$). □

Theorem 3.3.3.

(a) $\mathsf{DC} \subsetneq \mathsf{SP}_{\mathsf{tract}} \subsetneq (\mathbf{FO}^2[<] \cap \mathsf{T}_{\mathsf{tract}})$

(b) $\mathsf{T}_{\mathsf{tract}} \subsetneq \mathbf{FO}^2[<,+]$

(c) $\mathsf{T}_{\mathsf{tract}}$ *and* $\mathbf{FO}^2[<]$ *are incomparable*

Since $\mathbf{FO}^2[<,+] \subsetneq \mathbf{FO}[<]$, we also have $\mathsf{T}_{\mathsf{tract}} \subsetneq \mathbf{FO}[<]$. Thus, every language in $\mathsf{T}_{\mathsf{tract}}$ is aperiodic.

Proof. We first show (a). As DC is definable by simple regular expressions, we have for each downward closed language L that $w_\ell w_1^i w w_2^i w_r \in L$ implies $w_\ell w_1^i w_2^i w_r \in L$ for every integer $i \in \mathbb{N}$ and all words $w_\ell, w_1, w, w_2, w_r \in \Sigma^*$. Therefore, $L \in \mathsf{SP}_{\mathsf{tract}}$ by Definition 2.5.4. The language $\{a\}$ is not downward closed, but in $\mathsf{SP}_{\mathsf{tract}}$ using Definition 2.5.4 with $i = 1$.

As $\mathsf{SP}_{\mathsf{tract}} \subseteq \mathsf{T}_{\mathsf{tract}}$ by definition and a^*bc^* is a language that is not in $\mathsf{SP}_{\mathsf{tract}}$, but in $\mathsf{T}_{\mathsf{tract}}$ and in $\mathbf{FO}^2[<]$, it only remains to show $\mathsf{SP}_{\mathsf{tract}} \subseteq \mathbf{FO}^2[<]$.

Thérien and Wilke [198] proved that $\mathbf{DA}$=$\mathbf{FO}^2[<]$, where $\mathbf{DA}$ is defined by the identity $(xyz)^\omega y(xyz)^\omega = (xyz)^\omega$. Thus we only have to prove that each syntactic semigroup of a language in $\mathsf{SP}_{\mathsf{tract}}$ satisfies this identity. Let $L \in \mathsf{SP}_{\mathsf{tract}}$. By Observation 3.3.1, it immediately follows that the syntactic semigroup of L satisfies $(xyz)^\omega y(xyz)^\omega \leq (xyz)^\omega$. Thus it remains to show that there exists an n' such that for each $n \geq n'$ and all $u, v, x, y, z \in \Sigma^*$ it holds that:
$u(xyz)^n y(xyz)^n v \in L$ if $u(xyz)^n v \in L$.

For this direction, we use that Bagan et al. [20, Theorem 6] give a definition of $\mathsf{SP}_{\mathsf{tract}}$ in terms of regular expressions, showing that each component can be represented as $(A^{\geq k} + \varepsilon)$ for some set $A \subseteq \Sigma$ and $k \in \mathbb{N}$. So if there is $xyz \in \Sigma^*$ with $u(xyz)^M v \in L$ for some $u, v \in \Sigma^*$, then we also have $u(xyz)^M(\mathrm{Alph}(xyz))^*(xyz)^M v \subseteq L$, where $\mathrm{Alph}(x)$ denotes the set of symbols x uses. Thus we especially have $u(xyz)^M y(xyz)^M v \in L$, which proves the other direction. The same holds for each $M' \geq M$. This concludes the proof of (a).

Statement (b) follows from Lemma 3.3.2 and the observation that a^*ba^* is a language in $\mathbf{FO}^2[<,+]$ but not in $\mathsf{T}_{\mathsf{tract}}$.

It remains to show (c), which simply follows from the facts that the language a^*ba^* is in $\mathbf{FO}^2[<]$ but not in $\mathsf{T}_{\mathsf{tract}}$ whereas the language $(ab)^*$ is in $\mathsf{T}_{\mathsf{tract}}$ but not in $\mathbf{FO}^2[<]$. $\square$

Next, we show where $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$ are in the Straubing-Thérien hierarchy [190, 197]) and the dot-depth hierarchy (also known as Brzozowski hierarchy [68]). Both hierarchies are particular instances of concatenation hierarchies, which means that they can be built through a uniform construction scheme. Pin [178] summarized numerous results and conjectures around these hierarchies. We note that Jean-Éric Pin and Luc Segoufin helped with the proof of Part 3.3.4(b).

Proposition 3.3.4.

(a) $\mathsf{SP}_{\mathsf{tract}}$ *is in* $\mathcal{V}_{3/2}$*, the 3/2th level of the Straubing-Thérien hierarchy.*

(b) Every language L *in* $\mathsf{T}_{\mathsf{tract}} \cap \Sigma^+$ *is in* $\mathcal{B}_1$*, the 1st level of the dot-depth hierarchy.*

Proof. We start with (a). Let $L \in \mathsf{SP}_{\mathsf{tract}}$. The 3/2th level of the Straubing-Thérien hierarchy is defined by the profinite inequality $x^\omega \leq x^\omega y x^\omega$ where $\mathrm{Alph}(x) = \mathrm{Alph}(y)$ [177, Theorem 8.9]. This means that we have to show that there exists an n' such that for all $n \geq n'$ and words w_ℓ, w_r it holds: if $w_\ell x^n w_r$ in L, then also $w_\ell x^n y x^n w_r$ in L. We can easily see that every language in $\mathsf{SP}_{\mathsf{tract}}$ satisfies this: The components have the form $(A^{\geq k} + \varepsilon)$ for some set of symbols A by the definition of $\mathsf{SP}_{\mathsf{tract}}$ in terms of regular expressions, see [20, Theorem 6]. Therefore, the implication immediately holds for all y with $\mathrm{Alph}(y) = \mathrm{Alph}(x)$.

For (b), we first note that $\mathcal{B}_1$ is defined over languages of Σ^+, thus excluding ε, see [177, Proposition 8.17]. We will therefore show that the syntactic semigroup of each language in $\mathsf{T}_{\mathsf{tract}}$ satisfies the Knast identity: $(exfye)^\omega exfte(esfte)^\omega = (exfye)^\omega(esfte)^\omega$, where f, e are non-empty idempotents. The algebraic characterization of $\mathsf{T}_{\mathsf{tract}}$ by the profinite inequality $(xy)^\omega s(xz)^\omega \leq (xy)^\omega(xz)^\omega$ (see Observation 3.3.1) implies $ese \leq e$ for each non-empty idempotent e. So the $\leq$ is obvious.

For the other direction, we show how to rewrite $(exfye)^\omega(esfte)^\omega$. In the first step, we use that for the idempotent f, it holds that $f = ff$. Then we use the aperiodicity of languages in $\mathsf{T}_{\mathsf{tract}}$ and afterwards the definition of $\mathsf{T}_{\mathsf{tract}}$, namely $(xy)^\omega s(xz)^\omega \leq (xy)^\omega(xz)^\omega$. Then we again use the aperiodicity of languages in $\mathsf{T}_{\mathsf{tract}}$ and the equation $f = ff$.

$$\begin{aligned}(exfye)^\omega(esfte)^\omega &= (exffye)^\omega(esffte)^\omega\\ &= ex(fyeexf)^\omega yees(fteesf)^\omega fte\\ &\leq ex(fyeexf)^\omega(fteesf)^\omega fte\\ &= (exfye)^\omega exfte(esfte)^\omega\end{aligned}$$

Thus L satisfies the Knast identity and is therefore in $\mathcal{B}_1$. □

Thus Proposition 3.3.4 implies that every language in $\mathsf{SP}_{\mathsf{tract}}$ can be described by a formula in $\Sigma_2[<]$ and every language in tractable language $\mathsf{T}_{\mathsf{tract}}$ by a boolean combination of formulas in $\Sigma_1[<, +, \min, \max]$, see Pin [178, Theorem 4.1].

3.4 The Trichotomy

This section is devoted to the proof of the following theorem.

Theorem 3.4.1. *Let L be a regular language.*

(1) If L is finite then $\mathsf{Trail}(L) \in AC^0$.

(2) If $L \in \mathsf{T}_{\mathsf{tract}}$ and L is infinite, then $\mathsf{Trail}(L)$ is NL-complete.

(3) If $L \notin \mathsf{T}_{\mathsf{tract}}$, then $\mathsf{Trail}(L)$ is NP-complete.

3.4.1 Finite Languages

We now turn to proving Theorem 3.4.1. We start with Theorem 3.4.1(1). Clearly, we can express every finite language L as a FO-formula. Since we can also test in **FO** that no edge e is used more than once, the multigraphs for which $\mathsf{Trail}(L)$ holds are **FO**-definable. By Immerman [122], this implies that $\mathsf{Trail}(L)$ is in AC^0.

3.4.2 Languages in $\mathsf{T}_{\mathsf{tract}}$

We now sketch the proof of Theorem 3.4.1(2). We note that we define several concepts (trail summary, local edge domains, admissible trails) that have a natural counterpart for simple paths in Bagan et al.'s proof of the trichotomy for simple paths [20]. However, the underlying proofs of the technical lemmas are quite different. For instance, components of languages in $\mathsf{SP}_{\mathsf{tract}}$ behave similarly to A^* for some $A \subseteq \Sigma$, while components of languages in $\mathsf{T}_{\mathsf{tract}}$ are significantly more complex. Furthermore, the trichotomy for trails leads to a strictly larger class of tractable languages.

Let L be a regular language and N the number of states of a minimal DFA for L. For the remainder of this section, we fix the constant $K = N^2$.

We will show that in the case where L belongs to $\mathsf{T_{tract}}$, we can identify a number of edges that suffice to check if the path is (or can be transformed into) a trail that matches L. This number of edges only depends on L and is therefore constant for the $\mathsf{Trail}(L)$ problem. These edges will be stored in a *summary*. We will define summaries formally and explain how to use them to check whether a trail between the input nodes that matches L exists. To this end, we need a few definitions.

Definition 3.4.2. Let $p = e_1 \cdots e_m$ be a path and $r = q_0 \to \cdots \to q_m$ the run of A_L over $\text{lab}(p)$. For a set C of states of A_L, we denote by left_C the first edge e_i with $q_{i-1} \in C$ and by right_C the last edge e_j with $q_j \in C$. A component C of A_L is a *long run component of* p if left_C and right_C are defined and $|p[\mathsf{left}_C, \mathsf{right}_C]| > K$.

Next, we want to reduce the amount of information that we require for trails. The synchronization property, see Lemma 3.2.7, motivates the use of *summaries*, which we define next.

Definition 3.4.3. Let Cuts denote the set of components of A_L and $\mathsf{Abbrv} = \mathsf{Cuts} \times (V \times Q) \times E^K$. A *component abbreviation* $(C, (v, q), e_K \cdots e_1) \in \mathsf{Abbrv}$ consists of a component C, a node v of G and state $q \in C$ to start from, and K edges $e_K \cdots e_1$. A trail π *matches* a component abbreviation, denoted $\pi \models (C, (v, q), e_K \cdots e_1)$, if $\delta_L(q, \pi) \in C$, it starts at v, and its suffix is $e_K \cdots e_1$. Given an arbitrary set of edges E', we write $\pi \models_{E'} (C, (v, q), e_K \cdots e_1)$ if $\pi \models (C, (v, q), e_K \cdots e_1)$ and all edges of π are from $E' \cup \{e_1, \ldots, e_K\}$. For convenience, we write $e \models_\emptyset e$.

If p is a trail, then the *summary* S_p of p is the sequence obtained from p by replacing, for each long run component C the subsequence $p[\mathsf{left}_C, \mathsf{right}_C]$ by the abbreviation $(C, (v, q), p_{\mathsf{suff}})$, where v is the source node of the edge left_C, q is the state in which A_L is immediately before reading left_C, and p_{suff} is the suffix of length K of $p[\mathsf{left}_C, \mathsf{right}_C]$.

We note that the length of a summary is always bounded by $O(N^3)$, that is, a constant that depends on L. Indeed, A_L has at most N components and, for each of them, we store at most $K+3$ many things (namely, C, v, q, and K edges). Our goal is to find a summary S and replace all abbreviations with matching pairwise edge-disjoint trails which do not use any other edge in S, because this results in a trail that matches L. However, not every sequence of edges and abbreviations is a summary, because a summary needs to be obtained from a trail. So, we will work with candidate summaries instead.

Definition 3.4.4. A *candidate summary* S is a sequence of the form $S = \alpha_1 \cdots \alpha_m$ with $m \leq N$ where each α_i is either (1) an edge $e \in E$ or (2) an abbreviation $(C, (v, q), e_K \cdots e_1) \in \mathsf{Abbrv}$. Furthermore, all components in S are distinct and each edge e occurs at most once. A path p that is derived from S by replacing each $\alpha_i \in \mathsf{Abbrv}$ by a trail p_i such that $p_i \models \alpha_i$ is called a *completion* of the candidate summary S.

The following corollary is immediate from the definitions and Lemma 3.2.7, as the lemma ensures that the state after reading p inside a component does not depend on the whole path but only on the labels of the last K edges, which are fixed.

Corollary 3.4.5. *Let L be a language in $\mathsf{T}_{\mathsf{tract}}$. Let S be the summary of a trail p that matches L and let p' be a completion of S. Then, p' is a path that matches L.*

Together with the following lemmas, Corollary 3.4.5 can be used to obtain an NL algorithm[3] that gives us a completion of a summary S. The lemma heavily relies on other results on the structure of components in A_L.

Lemma 3.4.6. *There exists a NL algorithm that, given a directed graph G and nodes s and t, outputs a shortest path from s to t in G.*

Proof. We show that Algorithm 1 can output a shortest path in NL. Recall that nondeterministic algorithms with output either give up, or produce a correct output and that at least one computation does not give up. We note that Algorithm 1 is a mixture of the Immermann-Szelepscényi Theorem [122, 195] and reachability. To this end, $S(k)$ denotes the set of states reachable from s with k edges. Using the algorithm given by Immermann [122] and Szelepscényi [195] to show that non-reachability is in NL, we can find in lines 1–27 the smallest n such that a path from s to t of length n but none of length $n-1$ exists. Indeed, we only added a test in line 19 to find the smallest k for which $t \in S(k)$—this k is the length of a shortest path from s to t. After line 28 we then use the smallest k (which we name n) together with a standard reachability algorithm to nondeterministically output a path of this length. (If we are only interested in the length of a shortest path, we can return n instead.)

That Algorithm 1 is in NL follows from the Immermann-Szelepscényi Theorem and reachability being in NL. □

We explain how to use the algorithm described in Lemma 3.4.6 to output a shortest path that satisfies some additional constraints.

Lemma 3.4.7. *Let $L \in \mathsf{T}_{\mathsf{tract}}$, let $(C, (v, q), e_K \cdots e_1)$ be an abbreviation and $E' \subseteq E$. There exists an NL algorithm that outputs a shortest trail p such that $p \models_{E'} (C, (v, q), e_K \cdots e_1)$ if it exists and rejects otherwise.*

Proof Sketch. We use the algorithm described in Lemma 3.4.6 to search and output a shortest path p from (v, q) to (t, q') for t being the target node of e_1 and some $q' \in C$ in the product of G (restricted to the edges $E' \cup \{e_1, \ldots, e_K\}$) and C such that $e_K, \ldots, e_1$ are only used once, and $e_K \cdots e_1$ is the suffix of p. Since K is a constant, this is in NL. □

Proof. Let G be a directed (labeled) multigraph. In order to find a path in G that matches C, ends on $e_K \cdots e_1$, and uses edges $\{e_1, \ldots, e_K\}$ only once, we use Algorithm 1 on the product of G (restricted to the edges $E' \cup \{e_1, \ldots, e_K\}$) and C extended with numbers $\ell \in [K]$. Since we cannot store the product in $O(\log n)$, we will construct it on-the-fly. Intuitively, the value of ℓ will tell us if we are in the last K edges and if so,

[3]That is, a nondeterministic Turing Machine with read-only input and write-only output that only uses $O(\log n)$ space on its working tapes.

Algorithm 1: Extension of the Immermann-Szelepscényi Theorem

```
   Input: A directed graph $G = (V, E, \mathcal{E})$, nodes $s, t$ in $G$, $s \neq t$
   Output: A shortest path from $s$ to $t$ in $G$ or “no” if no path from $s$ to $t$ exists
 1 $n \leftarrow -1$          ▷ $n$ will be the length of a shortest path from $s$ to $t$
 2 $|S(0)| \leftarrow 1$
 3 for $k = 1, 2, \dots, |V| - 1$ do                 ▷ Compute $|S(k)|$ from $|S(k-1)|$
 4     $\ell \leftarrow 0$
 5     foreach $u \in V$ do                                   ▷ Test if $u \in S(k)$
 6         $m \leftarrow 0$
 7         $reply \leftarrow$ false
 8         foreach $v \in V$ do                           ▷ Test if $v \in S(k-1)$
 9             $w_0 \leftarrow s$
10             for $p = 1, \dots, k - 1$ do
11                 guess a node $w_p$
12                 if $(w_{p-1}, w_p)$ is not an edge in $G$ then
13                     give up
14             if $w_{k-1} \neq v$ then
15                 give up
16             $m \leftarrow m + 1$
17             if $(v, u)$ is an edge in $G$ then
18                 $reply \leftarrow$ true
19                 if $u = t$ then
20                     $n \leftarrow k$
21                     continue in line 28
22         if $m < |S(k-1)|$ then
23             give up
24         if $reply =$ true then
25             $\ell \leftarrow \ell + 1$
26     $|S(k)| \leftarrow \ell$
27 return “no”                                       ▷ $t \notin S(k)$ for any $k$
28 $w_0 \leftarrow s$
29 for $p = 1, \dots, n - 1$ do
30     guess a node $w_p$
31     if $(w_{p-1}, w_p)$ is not an edge in $G$ then
32         give up
33     output an edge from $w_{p-1}$ to $w_p$ in $G$
34 if $w_{n-1} \neq t$ then
35     give up
```

then which of the last K edges we expect next. The "product" of G, C, and $[K]$ is a directed multigraph $G^* = (V^*, E^*, \mathcal{E}^*)$ defined as follows: $V^* = V \times Q \times [K]$ and

$$\begin{aligned} E^* = \{(e_\ell, (q_1, q_2, \ell)) \mid \ell \in [K] \text{ and } (q_1, \text{lab}(e_\ell), q_2) \in C\} \\ \cup \{(e, (q_1, q_2, K)) \mid e \in E' - \{e_1, \dots, e_K\} \text{ and } (q_1, \text{lab}(e), q_2) \in C\} \end{aligned}$$

$$\begin{aligned} \mathcal{E}^*((e, (q_1, q_2, K))) &= (\text{origin}(e), q_1, K), \text{lab}(e), (\text{destination}(e), q_2, K)) \text{ if } e \neq e_K \\ \mathcal{E}^*((e, (q_1, q_2, K))) &= ((\text{origin}(e), q_1, K), \text{lab}(e), (\text{destination}(e), q_2, K-1)) \text{ if } e = e_K \\ \mathcal{E}^*((e_\ell, (q_1, q_2, \ell))) &= ((\text{origin}(e_\ell), q_1, \ell), \text{lab}(e_\ell), (\text{destination}(e_\ell), q_2, \ell-1)) \text{ if } \ell < K \end{aligned}$$

Since K is a constant, the size of each state in V^* is logarithmic in the input, and for two states $x, y \in V^*$, we can test in logarithmic space if there is an edge $e \in E^*$ such that $\mathcal{E}^*(e) = (x, \text{lab}(e), y)$. This is necessary for lines 12, 17, and 31.

We then output a shortest path from (v, q, K) to $(t, q', 1)$ for t being the target node of e_1 and some $q' \in C$.[4] More precisely, since we want a path in G and not in the product, we project away the unnecessary state and number and only output the corresponding edge in G in line 33.

It remains to show that p is a trail (in G). Assume towards contradiction that $p = d_1 \cdots d_m e_K \cdots e_1$ is not a trail. Then there exists an edge $d_i = d_j$ that appears at least twice in p. Note that d_j is not in the suffix $e_K \cdots e_1$ by definition of p. We define

$$p' = d_1 \cdots d_i d_{j+1} \cdots d_m e_K \cdots e_1$$

and show that p' is a shorter than p but meets all requirements. Let $q_1 = \delta(q, d_1 \cdots d_i)$ and $q_2 = \delta(q, d_1 \cdots d_j)$. By definition, $q_1, q_2 \in C$ and both have an incoming edge with label $\text{lab}(d_i) = \text{lab}(d_j)$. This allows us to use Corollary 3.2.5 to ensure that

$$\delta(q_1, d_{j+1} \cdots d_m e_K \cdots e_1) \in C.$$

We can then apply Lemma 3.2.7 to prove that

$$\delta(q_1, d_{j+1} \cdots d_m e_K \cdots e_1) = \delta(q_2, d_{j+1} \cdots d_m e_K \cdots e_1)\ .$$

So p' is indeed a trail satisfying $p' \models_{E'} (C, (v, q), e_K \cdots e_1)$. Furthermore, p' is shorter than p, contradicting our assumption. ☐

Using the algorithm of Lemma 3.4.7 we can, in principle, output a completion of S that matches L using nondeterministic logarithmic space. However, such a completion does not necessarily correspond to a trail. The reason is that, even though each trail p_C we guess for some abbreviation involving a component C is a trail, the trails for different components may not be disjoint. Therefore, we will define pairwise disjoint subsets of edges that can be used for the completion of the components.

[4]Algorithm 1 can also output a shortest path from s to some node in a set T by testing $u \in T$ in line 19 and $w_{n-1} \notin T$ in line 34.

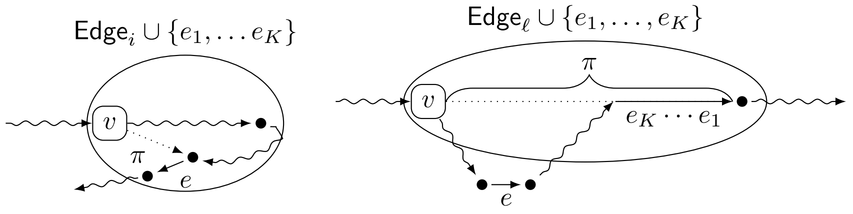

$\mathsf{Edge}_i \cup \{e_1, \ldots e_K\}$
v
π
e
$\mathsf{Edge}_\ell \cup \{e_1, \ldots, e_K\}$
π
v
$e_K \cdots e_1$
e

Figure 3.4: Sketch of case (1) and (2) in the proof of Lemma 3.4.10

The following definition fulfills the same purpose as the local domains on nodes in Bagan et al. [20, Definition 7]. Since our components can be more complex, we require extra conditions on the states (the $\delta_L(q,\pi) \in C$ condition).

Definition 3.4.8 (Local Edge Domains)**.** Let $S = \alpha_1 \cdots \alpha_k$ be a candidate summary and $E(S)$ be the set of edges appearing in S. We define the local edge domains $\mathsf{Edge}_i \subseteq E_i$ inductively for each i from 1 to k, where E_i are the remaining edges defined by $E_1 = E \setminus E(S)$ and $E_{i+1} = E_i \setminus \mathsf{Edge}_i$. If there is no trail p such that $p \models \alpha_i$ or if α_i is a single edge, we define $\mathsf{Edge}_i = \emptyset$.

Otherwise, let $\alpha_i = (C, (v,q), e_K \cdots e_1)$. We denote by m_i the minimal length of a trail p with $p \models_{E_i} \alpha_i$ and define Edge_i as the set of edges used by trails π that start at v, only use edges in E_i, are of length at most $m_i - K$, and satisfy $\delta_L(q,\pi) \in C$.

By definition of Edge_i, we can conclude that $\mathcal{E}(e_i) \neq \mathcal{E}(e_j)$ for all $e_i \in \mathsf{Edge}_i, e_j \in \mathsf{Edge}_j, i \neq j$, as $e_i \in \mathsf{Edge}_i$ and $\mathcal{E}(e_i) = \mathcal{E}(e_j)$ imply that $e_j \in \mathsf{Edge}_i$. We note that a shortest trail using e_i but not e_j can use e_j instead of e_i. We note that the sets $E(S)$ and $(\mathsf{Edge}_i)_{i \in [k]}$ are always disjoint.

Definition 3.4.9 (Admissible Trail)**.** We say that a trail p is *admissible* if there exist a candidate summary $S = \alpha_1 \cdots \alpha_k$ and trails $p_1, \ldots, p_k$ such that $p = p_1 \cdots p_k$ is a completion of S and $p_i \models_{\mathsf{Edge}_i} \alpha_i$ for every $i \in [k]$.

We show that *shortest* trails that match L are always admissible. Thus, the existence of a trail is equivalent to the existence of an admissible trail.

Lemma 3.4.10. *Let G and (s,t) be an instance for $\mathsf{Trail}(L)$, with $L \in \mathsf{T}_{\mathsf{tract}}$. Then every shortest trail from s to t in G that matches L is admissible.*

Proof sketch. We assume towards a contradiction that there is a shortest trail p from s to t in G that matches L and is not admissible. That means there is some $\ell \in \mathbb{N}$, and an edge e used in p_ℓ with $e \notin \mathsf{Edge}_\ell$. There are two possible cases: (1) $e \in \mathsf{Edge}_i$ for some $i < \ell$ and (2) $e \notin \mathsf{Edge}_i$ for any i. In both cases, we construct a shorter trail p that matches L, which then leads to a contradiction. We depict the two cases in Figure 3.4. We construct the new trail by substituting the respective subtrail with π. ☐

Proof. In this proof, we use the following notation for trails. By $p[e_1, e_2)$ we denote the prefix of $p[e_1, e_2]$ that excludes the last edge (so it can be empty). Analogously, $p(e_1, e_2]$

denotes the suffix of $p[e_1, e_2]$ that excludes the first edge. Notice that $p[e_1, e_2]$, $p[e_1, e_2)$, and $p(e_1, e_2]$ are always well-defined for trails. Let $p = d_1 \cdots d_m$ be a shortest trail from s to t that matches L. Let $S = \alpha_1 \cdots \alpha_k$ be the summary of p and let $p_1, \dots, p_k$ be trails such that $p = p_1 \cdots p_k$ and $p_i \models \alpha_i$ for all $i \in [k]$. We denote by left_i and right_i the first and last edge in p_i. By definition of p_i and the definition of summaries, left_i and right_i are identical with left_C and right_C if $\alpha_i \in \mathsf{Abbrv}$ is an abbreviation for the component C.

Assume that p is not admissible. That means there is some edge e used in p_ℓ such that $e \notin \mathsf{Edge}_\ell$. There are two possible cases:

(1) $e \in \mathsf{Edge}_i$ for some $i < \ell$; and

(2) $e \notin \mathsf{Edge}_i$ for any i.

In case (1), we choose i minimal such that some edge $e \in \mathsf{Edge}_i$ is used in p_j for some $j > i$. Among all such edges $e \in \mathsf{Edge}_i$, we choose the edge that occurs latest in p. This implicitly maximizes j for a fixed i. Especially no edge from Edge_i is used in $p_{j+1} \cdots p_k$.

Let $\alpha_i = (C_i, (v, q), e_K \cdots e_1)$. By definition of Edge_i, there is a trail π from v, ending with e, with $\delta_L(q, \text{lab}(\pi)) \in C_i$, and that is shorter than the subpath $p[\mathsf{left}_i, \mathsf{right}_i]$ and therefore shorter than $p[\mathsf{left}_i, e]$.

We now show that $p' = p_1 \cdots p_{i-1} \pi p(e, d_m]$ is a trail. Since p is a trail, it suffices to prove that the edges in π are disjoint with other edges in p'. We note that all intermediate edges of π belong to Edge_i. By minimality of i, no edge in $p_1 \cdots p_{i-1}$ can use any edge of Edge_i and by our choice of e, no edge in p after e can use any edge of Edge_i. This shows that p' is a trail.

We now show that p' matches L. Since e appears in p_j, there is a path from left_j to right_j over e that stays in C_j. Let q_1 and q_2 be the states of A_L before and after reading e in p and, analogously, q_1' and q_2' the states of A_L before and after reading e in p'. That is

$$\begin{aligned} q_1 &= \delta_L(i_L, p[d_1, e)) & \qquad q_2 &= \delta_L(q_1, e) \\ q_1' &= \delta_L(i_L, p'[d_1, e)) & \qquad q_2' &= \delta_L(q_1', e) \end{aligned}$$

We note that in p', e is at the end of the subtrail π.

We can conclude that the states q_1 and q_1' both have loops starting with $a = \text{lab}(e)$, as the transition $(q_1, \text{lab}(e), q_2)$ is read in C_j and the transition $(q_1', \text{lab}(e), q_2')$ is read in C_i. Furthermore, $q_1' \rightsquigarrow q_1$, since $q_1' \in C_i$ and $q_1 \in C_j$. Therefore, Lemma 3.2.8 implies that $L_{q_1} \cap L^a_{q_1}\Sigma^* \subseteq L_{q_1'}$ where $L^a_{q_1}$ denotes all words w of length K that start with a and such that $\delta_L(q_1, w) \in C_j$.

We have that $\text{lab}(p[e, d_m]) \in L_{q_1}$ by the fact that p matches L. We have that $\text{lab}(p[e, d_m]) \in L^a_{q_1}\Sigma^*$, as, by the definition of summaries, A_L stays in C_j for at least K more edges after reading e in p. We can conclude that $\text{lab}(p[e, d_m]) \in L_{q_1'}$, which proves that p' matches L.

This concludes case (1). For case (2), we additionally assume without loss of generality that there is no edge $e \in \mathsf{Edge}_i$ that appears in some p_j with $j > i$, that is, no edge satisfies case (1). By definition of Edge_ℓ, there is a trail π with $\pi \models_{\mathsf{Edge}_\ell} \alpha_\ell$ that is shorter than $p[\mathsf{left}_\ell, \mathsf{right}_\ell]$. We choose p' as the path obtained from p by replacing p_ℓ with π.

We now show that $p' = p_1 \cdots p_{\ell-1} \cdot \pi \cdot p_{\ell+1} \cdots p_k$ is a trail. Since p is a trail, it suffices to prove that the edges in π are disjoint with other edges in p'. We note that all intermediate edges of π belong to Edge_ℓ.

By definition of Edge_ℓ, no edge in $p_1 \cdots p_{\ell-1}$ is in Edge_ℓ. And by the assumption that there is no edge satisfying case (1), no edge in $p_{\ell+1} \cdots p_k$ is in Edge_ℓ. Therefore, p' is a trail.

It remains to prove that p' matches L. Let $(C, (v, \hat{q}), e_K \cdots e_1) = \alpha_\ell$ and let q and q' be the states in which A_L is before reading e_K in p and p', respectively. By definition of a summary, we have that $\delta_L(q, e_K \cdots e_1) \in C$ and, by definition of $\models$, we have that $\delta_L(q', e_K \cdots e_1) \in C$. By Lemma 3.2.7 we can conclude that $\delta_L(q, e_K \cdots e_1) = \delta_L(q', e_K \cdots e_1)$. As p matches L, we can conclude that also p' matches L. $\square$

So, if there is a solution to $\mathsf{Trail}(L)$, we can find it by enumerating the candidate summaries and completing them using the local edge domains. We next prove that testing if an edge is in Edge_i can be done in logarithmic space. We will name this decision problem $P_{\mathsf{edge}}(L)$ and define it as follows:

	$P_{\mathsf{edge}}(L)$
Given:	A multigraph $G = (V, E, \mathcal{E})$, nodes s, t, a candidate summary S, an edge $e \in E$ and an integer i.
Question:	Is $e \in \mathsf{Edge}_i$?

Lemma 3.4.11. *$P_{\mathsf{edge}}(L)$ is in NL for every $L \in \mathsf{T}_{\mathsf{tract}}$.*

Proof. The proof is similar to the proof of Lemma 17 by Bagan et al. [20], which is based on the following result due to Immerman [122]: $\mathrm{NL}^{\mathrm{NL}} = \mathrm{NL}$. In other words, if a decision problem P can be solved by an NL algorithm using an oracle in NL, then this problem P belongs to NL. Let, for each $k \geq 0$, $P^{\leq k}_{\mathsf{edge}}(L)$ be the decision problem $P_{\mathsf{edge}}(L)$ with the restriction $i \leq k$, that is, (G, s, t, S, e, i) is a positive instance of $P^{\leq k}_{\mathsf{edge}}(L)$ if and only if (G, s, t, S, e, i) is a positive instance of $P_{\mathsf{edge}}(L)$ and $i \leq k$. Notice that i belongs to the input of $P^{\leq k}_{\mathsf{edge}}(L)$ while this is not the case for k. Obviously, $P_{\mathsf{edge}}(L) = P^{\leq |S|}_{\mathsf{edge}}(L)$. We prove that $P^{\leq k}_{\mathsf{edge}}(L) \in \mathrm{NL}$ for each $k \geq 0$. If $k = 0$, $P^{\leq 0}_{\mathsf{edge}}(L)$ always returns false because Edge_i is not defined for $i = 0$. So $P^{\leq 0}_{\mathsf{edge}}(L)$ is trivially in NL. Assume, by induction, that $P^{\leq k}_{\mathsf{edge}}(L) \in \mathrm{NL}$. It suffices to show that there is an NL algorithm for $P^{\leq k+1}_{\mathsf{edge}}(L)$ using $P^{\leq k}_{\mathsf{edge}}(L)$ as an oracle. Since $\mathrm{NL}^{\mathrm{NL}} = \mathrm{NL}$, this implies that $P^{\leq k+1}_{\mathsf{edge}}(L) \in \mathrm{NL}$.

Let (G, s, t, S, e, i) be an instance of $P^{\leq k+1}_{\mathsf{edge}}(L)$. If $i \leq k$, we return the same answer as the oracle $P^{\leq k}_{\mathsf{edge}}(L)$. If $i = k+1$ and $\alpha_i \in E$, we return false, as $\mathsf{Edge}_i = \emptyset$. If $i = k+1$ and $\alpha_i \in \mathsf{Abbrv}$, we first compute the length m of a minimal trail p such that $p \models_{E_i} \alpha_i$ using the NL algorithm of Lemma 3.4.7. We note that we can compute E_i using the NL algorithm for $P^{\leq k}_{\mathsf{edge}}$.

To test whether the edge e can be used by a trail from some (v, q) in at most $m - K$ steps, we use the on-the-fly product of G and A_L restricted to the edges in E_i and states in C. We search for a shortest path[5] from (v, q) to some $(v', q') \in V \times C$ that ends with e. We remind that reachability is in NL.

We note that this trail in the product might correspond to a path p with a cycle in G. As we project away the states, some distinct edges in the product are actually the same edge in G. However, by Lemma 3.2.4, we can remove all cycles from p without losing the property that $\delta_L(q, p) \in C$. This concludes the proof. □

With this, we can finally give an NL algorithm that decides whether a candidate summary can be completed to an admissible trail that matches L.

Lemma 3.4.12. *Let L be a language in $\mathsf{T}_{\mathsf{tract}}$. There exists an NL algorithm that given an instance G, (s, t) of $\mathsf{Trail}(L)$ and a candidate summary $S = \alpha_1 \cdots \alpha_k$ tests whether there is a trail p from s to t in G with summary S that matches L.*

Proof. We propose the following algorithm, which consists of three tests:

(1) Guess, on-the-fly, a path p from S by replacing each α_i by a trail p_i such that $p_i \models_{\mathsf{Edge}_i} \alpha_i$ for each $i \in [k]$. This test succeeds if and only if this is possible.

(2) In parallel, check that p matches L.

(3) In parallel, check that S is a summary of p.

We first prove that the algorithm is correct. First, we assume that there is a trail with summary S from s to t that matches L. Then, there is also a shortest such trail and, by Lemma 3.4.10, this trail is admissible. Therefore, the algorithm will succeed.

Conversely, assume that the algorithm succeeds. Since $E(S)$ and all the sets Edge_i are mutually disjoint, the path p is always a trail. By tests (2) and (3), it is a trail from s to t that matches L.

We still have to check the complexity. We note that the sets Edge_i are not stored in memory: we only need to check on-the-fly if a given edge belongs to those sets, which only requires logarithmic space according to Lemma 3.4.11. Therefore, we use an on-the-fly adaption of the NL algorithm from Lemma 3.4.7, which requires a set Edge_i as input, which we will provide on-the-fly.

Testing if p matches L can simply be done in parallel to test (1) on an edge-by-edge basis, by maintaining the current state of A_L in memory. If we do so, we can also check in parallel if $S = \alpha_1 \cdots \alpha_k$ is a summary of p. This is simply done by checking, for each α_i of the form $(C, (v, q), e_K \cdots e_1)$ and $\alpha_{i+1} = e$, whether $e \notin C$. This ensures that, after being in C for at least K edges, the path p leaves the component C, which is needed for summaries. Furthermore, we test if there is no substring $\alpha_i \cdots \alpha_j$ in S that purely consists of edges that are visited in the same component C, but which is too long to fulfill the definition of a summary. Since this maximal length is a constant, we can check it in NL. □

We eventually show the main Lemma of this section, proving that $\mathsf{Trail}(L)$ is tractable for every language in $\mathsf{T}_{\mathsf{tract}}$.

[5]A shortest path is necessarily a trail.

Lemma 3.4.13. *Let $L \in \mathsf{T}_{\mathsf{tract}}$. Then, $\mathit{Trail}(L) \in$ NL.*

Proof. We simply enumerate all possible candidate summaries S with respect to (L, G, s, t), and apply on each summary the algorithm of Lemma 3.4.12. We return "yes" if this algorithm succeeds and "no" otherwise. Since the algorithm succeeds if and only if there exists an admissible path from s to t that matches L, and consequently, if and only if there is a trail from s to t that matches L (Lemma 3.4.10), this is the right answer. Since L is fixed, there is a polynomial number of candidate summaries, each of logarithmic size. Consequently, they can be enumerated within logarithmic space. ☐

Lemma 3.4.14. *Let $L \in \mathsf{T}_{\mathsf{tract}}$ and L be infinite. Then, $\mathit{Trail}(L)$ is NL-complete.*

Proof. The upper bound is due to Lemma 3.4.13, the lower due to reachability in directed graphs being NL-hard. ☐

Corollary 3.4.15. *Let $L \in \mathsf{T}_{\mathsf{tract}}$, G be a directed multigraph, and s, t be nodes in G. If there exists a trail from s to t that matches L, then we can output a shortest such trail in polynomial time (and in nondeterministic logarithmic space).*

Proof. For each candidate summary S, we first use Lemma 3.4.12 to decide whether there exists an admissible trail with summary S. With the algorithm in Lemma 3.4.7, we then compute the minimal length m_i of each p_i. The sum of these m_is then is the length of a shortest trail that is a completion of S. We will keep track of a summary of one of the shortest trails and finally recompute the overall shortest trail completing this summary and outputting it. Notice that this algorithm is still in NL since the summaries have constant size and overall counters never exceed $|E|$. ☐

3.4.3 Languages not in $\mathsf{T}_{\mathsf{tract}}$

In this section we prove the NP hardness for graphs, and therefore also for multigraphs. The proof of Theorem 3.4.1(3) is by reduction from the following NP-complete problem:

	TwoEdgeDisjointPaths
Given:	A directed graph $G = (V, E, \mathcal{E})$, and two pairs of nodes (s_1, t_1), (s_2, t_2).
Question:	Are there two paths p_1 from s_1 to t_1 and p_2 from s_2 to t_2 such that p_1 and p_2 are edge-disjoint?

The proof is very close to the corresponding proof for simple paths by Bagan et al. [20, Lemma 4] (which is a reduction from the two vertex-disjoint paths problem).

Lemma 3.4.16 (Fortune et al. [98])**.** *TwoEdgeDisjointPaths is NP-complete.*

To prove the lower bound, we first show that every regular language that is not in $\mathsf{T}_{\mathsf{tract}}$ admits a witness for hardness, which is defined as follows.

Definition 3.4.17. A *witness for hardness* is a tuple (q, w_m, w_r, w_1, w_2) with $q \in Q_L$, $w_m, w_r, w_1, w_2 \in \Sigma^*$, $w_1 \in \text{Loop}(q)$ and there exists a symbol $a \in \Sigma$ with $w_1 = aw_1'$ and $w_2 = aw_2'$ and satisfying

- $w_m(w_2)^* w_r \subseteq L_q$, and
- $(w_1 + w_2)^* w_r \cap L_q = \emptyset$.

Before we prove that each regular language that is not in $\mathsf{T}_{\mathsf{tract}}$ has such a witness, recall Property P:

$$L_{q_2} \subseteq L_{q_1} \text{ for all } q_1, q_2 \in Q_L \text{ such that } q_1 \rightsquigarrow q_2 \text{ and } \text{Loop}(q_1) \cap \text{Loop}(q_2) \neq \emptyset$$

Lemma 3.4.18. *Let L be a regular language that does not belong to $\mathsf{T}_{\mathsf{tract}}$. Then, L admits a witness for hardness.*

Proof. Let L be a regular language that does not belong to $\mathsf{T}_{\mathsf{tract}}$. Then there exist $q_1, q_2 \in Q_L$ and words w_1, w_2 with $w_1 = aw_1'$ and $w_2 = aw_2'$ such that $w_1 \in \text{Loop}(q_1), w_2 \in \text{Loop}(q_2)$, and $q_1 \rightsquigarrow q_2$ such that $w_2^M w_r' \notin L_{q_1}$ for a $w_r' \in L_{q_2}$. Let w_m be a word with $q_2 = \delta_L(q_1, w_m)$. We set $w_r = w_2^M w_r'$.

We now show that the so defined tuple $(q_1, w_m, w_r, w_1, w_2)$ is a witness for hardness. By definition, we have $w_m(w_2)^* w_r \subseteq L_{q_1}$. We distinguish two cases, depending on whether L satisfies Property P or not. If L does not satisfy P, we can assume without loss of generality that in our tuple we have $w_1 = w_2$ and since $w_2^M w_r' \notin L_{q_1}$, we also have $w_2^* w_r \cap L_{q_1} \neq \emptyset$, so it is indeed a witness for hardness.

Otherwise, L is aperiodic, see Lemma 3.1.6. We assume without loss of generality that $w_1 = (w_1')^M$ for some word w_1'. Then, we claim that $L_{q'} \subseteq L_{q_1}$ for every q' in $\delta_L(q_1, \Sigma^* w_1)$. Indeed, every $q' \in \delta_L(q_1, \Sigma^* w_1)$ loops over w_1 by the pumping lemma and aperiodicity of L, hence $w_1 \in \text{Loop}(q_1) \cap \text{Loop}(q')$ and therefore $L_{q'} \subseteq L_{q_1}$ due to Property P.

It remains to prove that $(w_1 + w_2)^* w_r \cap L_{q_1} = \emptyset$. Every word in $(w_1 + w_2)^* w_r$ can be decomposed into uv with $u \in \varepsilon + (w_1 + w_2)^* w_1$ and $v \in w_2^* w_r$. For $q' = \delta_L(q_1, u)$ we have proved that $L_{q'} \subseteq L_{q_1}$, so it suffices to show that $v \notin L_{q_1}$. This is immediate from $w_r = w_2^M w_r' \notin L_{q_1}$ and the aperiodicity of L. So we have $uv \notin L_{q_1}$ and the tuple (q, w_m, w_r, w_1, w_2) is indeed a witness for hardness. □

We can now show the following

Lemma 3.4.19. *Let L be a regular language that does not belong to $\mathsf{T}_{\mathsf{tract}}$. Then, $\mathsf{Trail}(L)$ is NP-complete.*

Proof. The proof is almost identical to the reduction from two node-disjoint paths to the $\mathsf{SimPath}(L)$ problem by Bagan et al. [20]. Clearly, $\mathsf{Trail}(L)$ is in NP for every regular language L, since we only need to guess a trail of length at most $|E|$ from s to t and verify that the word on the trail is in L. Let $L \notin \mathsf{T}_{\mathsf{tract}}$. We exhibit a reduction from $\mathsf{TwoEdgeDisjointPaths}$ to $\mathsf{Trail}(L)$. According to Lemma 3.4.18, L admits a witness for hardness (q, w_m, w_r, w_1, w_2). Let w_ℓ be a word such that $\delta_L(i_L, w_\ell) = q$. By definition of a witness we get $w_\ell(w_1 + w_2)^*w_r \cap L = \emptyset$ and $w_\ell w_1^* w_m w_2^* w_r \subseteq L$. Let $a \in \Sigma$ and $w_1', w_2' \in \Sigma^*$ such that $w_1 = aw_1'$ and $w_2 = aw_2'$. If w_1' or w_2' is empty, we replace it with a.

In the following construction, whenever we say that we a add a path from v_0 to v_n labeled by a word $w = a_1 \cdots a_n$, denoted by $v_0 \overset{w}{\rightsquigarrow} v_n$ we mean that we add $n-1$ new nodes $v_1, \ldots, v_{n-1}$ and n new edges $e_1, \ldots, e_n$ such that $\mathcal{E}'(e_i) = (v_{i-1}, a_i, v_i)$.

Let $G = (V, E, \mathcal{E})$ be a directed (unlabeled) graph for the $\mathsf{TwoEdgeDisjointPaths}$ problem and s_1, t_1, s_2, t_2 be nodes in V. We build from G a directed, labeled graph $G' = (V', E', \mathcal{E}')$ such that (G, s_1, t_1, s_2, t_2) is a yes-instance of $\mathsf{TwoEdgeDisjointPaths}$ if and only if there is a trail from s to t matching L in G'. We start with the nodes from G and add two new nodes s and t and three paths $s \overset{w_\ell}{\rightsquigarrow} s_1$, $t_1 \overset{w_m}{\rightsquigarrow} s_2$, and $t_2 \overset{w_r}{\rightsquigarrow} t$. Furthermore, for each edge (v_1, v_2) in G, we add a new node v_{12} and three paths $v_1 \overset{a}{\rightsquigarrow} v_{12}$, $v_{12} \overset{w_1'}{\rightsquigarrow} v_2$, and $v_{12} \overset{w_2'}{\rightsquigarrow} v_2$. An example for the language $da^*c(abc)^*ef$ and some graph G can be seen in Figure 3.5.

By construction, two edge-disjoint paths p_1 and p_2 in G going from s_1 to t_1 and from s_2 to t_2 correspond to a trail p from s to t in G' that contains the path $t_1 \overset{w_m}{\rightsquigarrow} s_2$. Such a trail p matches a word in $w_\ell(w_1 + w_2)^*w_m(w_1 + w_2)^*w_r$. And, as w_1 and w_2 can be used interchangeably, we can find a path p' matching $w_\ell w_1^* w_m w_2^* w_r$. Thus p' is a trail matching L.

For the other direction, we have to show two things. First, we show that every trail p in G' from s to t that uses the path $t_1 \overset{w_m}{\rightsquigarrow} s_2$ proves the existence of two edge disjoint paths p_1 and p_2 in G from s_1 to t_1 and from s_2 to t_2. Indeed p_1 and p_2 can be computed from p by keeping only those nodes that are from G and splitting p between t_1 and s_2. The paths are disjoint, as otherwise some edge v_i to v_{ij} has to be used twice by p. Second, we show that there can be no trail p from s to t in G' that matches L and does not use the path $t_1 \overset{w_m}{\rightsquigarrow} s_2$. Indeed, every trail p from s to t in G' that does not contain the path $t_1 \overset{w_m}{\rightsquigarrow} s_2$ matches a word in $w_\ell(w_1 + w_2)^*w_r$. By definition of witness for hardness, no such word is in L. Thus, $\mathsf{Trail}(L)$ returns "yes" for (G', s, t) if and only if there is a trail from s to t in G' that contains the edge (t_1, w_m, s_2) that is, if and only if $\mathsf{TwoEdgeDisjointPaths}$ returns "yes" for (G, s_1, t_1, s_2, t_2). □

3.5 Recognition and Closure Properties of $\mathsf{T}_{\mathsf{tract}}$

The following theorem establishes the complexity of deciding if a regular language is in $\mathsf{T}_{\mathsf{tract}}$.

Before we establish the complexity of deciding for a regular language L whether $L \in \mathsf{T}_{\mathsf{tract}}$, we need some lemmas. The first has been adapted from the simple path case (Lemma 6 in [20]).

Lemma 3.5.1. *Let L be a regular language. Then, L belongs to $\mathsf{T}_{\mathsf{tract}}$ if and only if for all pairs of states $q_1, q_2 \in Q_L$ and symbols $a \in \Sigma$ such that $q_1 \rightsquigarrow q_2$ and* $\text{Loop}(q_1) \cap a\Sigma^* \neq \emptyset$, *the following statement holds:* $(\text{Loop}(q_2) \cap a\Sigma^*)^N L_{q_2} \subseteq L_{q_1}$.

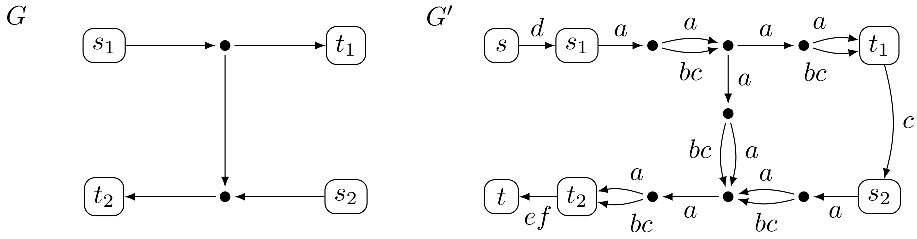


Figure 3.5: Example of the reduction in Lemma 3.4.19 for the language $da^*c(abc)^*ef$. We use $w_\ell = d$, $w_m = c$, $w_r = ef$, $w_1 = aa$, and $w_2 = abc$ for the construction. For the ease of readability, we omit the intermediate nodes on the bc and ef paths.

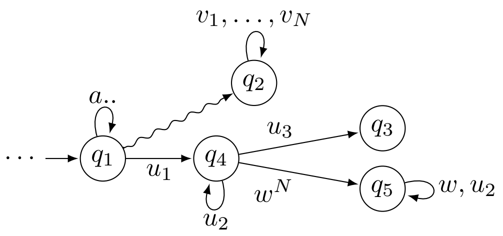


Figure 3.6: Sketch of the proof of Lemma 3.5.1

Proof. The (if) implication is immediate by Corollary 3.1.5. Let us now prove the (only if) implication. Since the proof of this lemma requires a number of different states and words, we provide a sketch in Figure 3.6. Assume $L \in \mathsf{T}_{\mathsf{tract}}$. Let q_1, q_2 be two states such that $\text{Loop}(q_1) \cap a\Sigma^* \neq \emptyset$ and $q_1 \rightsquigarrow q_2$. If $\text{Loop}(q_2) \cap a\Sigma^* = \emptyset$, the statement follows immediately. So let us assume without loss of generality that $\text{Loop}(q_2) \cap a\Sigma^* \neq \emptyset$. Let $v_1, \ldots, v_N \in (\text{Loop}(q_2) \cap a\Sigma^*)$ be arbitrary words and $q_3 = \delta_L(q_1, v_1 \cdots v_N)$. We want to prove $L_{q_2} \subseteq L_{q_3}$. For some i, j with $0 \leq i < j \leq N$, we get $\delta_L(q_1, v_1 \cdots v_i) = \delta_L(q_1, v_1 \cdots v_j)$ due to the pumping Lemma. (We have $\delta_L(q_1, v_1 \cdots v_i) = q_1$ for $i = 0$.) Let $u_1 = v_1 \cdots v_i, u_2 = v_{i+1} \cdots v_j$ and $u_3 = v_{j+1} \cdots v_k$. Let $q_4 = \delta_L(q_1, u_1)$.

We claim that $L_{q_2} \subseteq L_{q_4}$. The result then follows from $L_{q_2} = u_3^{-1} L_{q_2} \subseteq u_3^{-1} L_{q_4} = L_{q_3}$. To prove the claim, let $w = u_1 u_2^N$ and $q_5 = \delta_L(q_1, w^N)$. As $w \in \text{Loop}(q_2)$, we can use Corollary 3.1.5 to obtain $w^N L_{q_2} \subseteq L_{q_1}$. Together with $L_{q_5} = (w^N)^{-}1 L_{q_1}$ this implies $L_{q_2} \subseteq L_{q_5}$. Furthermore, u_2 belongs to $\text{Loop}(q_5)$ because L is aperiodic. To conclude the proof, we observe that $L_{q_5} \subseteq L_{q_4}$, by Corollary 3.1.5 with q_5, q_4 and u_2, and because $\delta_L(q_4, u_2^N) = q_4$ and $u_2 \in \text{Loop}(q_5)$. ☐

Theorem 3.5.2. *Testing whether a regular language L belongs to $\mathsf{T}_{\mathsf{tract}}$ is*

(1) NL-complete if L is given by a DFA and

(2) PSPACE-complete if L is given by an NFA or by a regular expression.

Proof. The proof is inspired by Bagan et al. [20]. The upper bound for (1) needs several adaptions, the lower bound for (1) and the proof for (2) works exactly the same as in [19], a preliminary version of [20] (just replacing $\mathsf{SP}_{\mathsf{tract}}$ by $\mathsf{T}_{\mathsf{tract}}$).

We first prove (1). W.l.o.g., we can assume that L is given by the minimal DFA A_L, as testing Nerode-equivalence of two states is in NL.

By Lemma 3.5.1, we need to check for each pair of states q_1, q_2 and symbol $a \in \Sigma$ whether

(i) $q_1 \rightsquigarrow q_2$;

(ii) $\text{Loop}(q_1) \cap a\Sigma^* \neq \emptyset$; and

(iii) $(\text{Loop}(q_2) \cap a\Sigma^*)^N L_{q_2} \setminus L_{q_1} = \emptyset$.

Statements (i) and (ii) are easily verified using an NL algorithm for transitive closure. For (iii), we test emptiness of $(\text{Loop}(q_2) \cap a\Sigma^*)^N L_{q_2} \setminus L_{q_1}$ using an NL algorithm for reachability in the product automaton of A_L with itself, starting in the state (q_2, q_1). More precisely, the algorithm checks whether there does not exist a word that is in L_{q_2}, is not in L_{q_1}, starts with an a, and leaves the state q_2 (in the left copy of A_L) at least N times with an a-transition.

The remainder of the proof is from [19] and only included for self containedness.

For the lower bound of (1), we give a reduction from the Emptiness problem. Let $L \subseteq \Sigma^*$ be an instance of Emptiness given by a DFA A_L. W.l.o.g. we assume that $\varepsilon \notin L$, since this can be checked in constant time. Furthermore, we assume that the symbol 1 does not belong to Σ. Let $L' = 1^+L1^+$. A DFA $A_{L'}$ that recognizes L' can be obtained from A_L as follows. We add a state q_I that will be the initial state of $A_{L'}$. and a state q_F that will be the unique final state of $A_{L'}$. The transition function $\delta_{L'}$ is the extension of δ_L defined as follows:

- $\delta_{L'}(q_I, 1) = q_I$ and $\delta_{L'}(q_I, a) = i_L$ for every symbol $a \in \Sigma$.
- For every final state $q \in F_L$, $\delta_{L'}(q, 1) = q_F$.
- $\delta_{L'}(q_F, 1) = q_F$.

We will show that $L' \in \mathsf{T}_{\mathsf{tract}}$ if and only if L is empty. If L is empty, then $L' = \emptyset$ belongs to $\mathsf{T}_{\mathsf{tract}}$. For the other direction, assume that L is not empty. Let $w \in L$. Then, for every $n \in \mathbb{N}$, $1^n w 1^n \in L'$ and $1^n 1^n \notin L'$. Thus $L' \notin \mathsf{T}_{\mathsf{tract}}$.

For the upper bound of (2), we first observe the following fact: Let A, B be two problems such that $A \in$ NL and let t be a reduction from B to A that works in polynomial space and produces an exponential output. Then B belongs to PSPACE. Thus, we can apply the classical powerset construction for determinization on the NFA and use the upper bound from (1).

For the lower bound of (2), we give a reduction from Universality. Let $L \subseteq \{0,1\}^*$ be an instance of Universality given by an NFA or a regular expression. Consider

$L' = (0+1)^*a^*ba^* + La^*$ over the alphabet $\{0,1,a,b\}$. We show that $L = \{0,1\}^*$ if and only if $L' \in \mathsf{T}_{\mathsf{tract}}$. Our reduction associates L' to L and keeps the same representation (NFA or regular expression). If $L' = \{0,1\}^*$, then $L' = (0+1)^*a^*(b+\varepsilon)a^*$ and thus $L' \in \mathsf{T}_{\mathsf{tract}}$. Conversely, assume $L \neq \{0,1\}^*$. Let $w \in \{0,1\}^* \setminus L$. Then, for every $n \in \mathbb{N}$, $wa^nba^n \in L'$ and $wa^na^n \notin L'$. Thus $L' \notin \mathsf{T}_{\mathsf{tract}}$. □

We wondered if, similarly to Theorem 2.5.2, it could be the case that languages closed under left-synchronized power abbreviations are always regular, but this is not the case. For example, the (infinite) Thue-Morse word [164, 200] has no subword that is a cube (that is, no subword of the form w^3) [200, Satz 6]. The language containing all prefixes of the Thue-Morse word thus trivially is closed under left-synchronized power abbreviations (with $i = 3$), yet it is not regular.

We now give (and repeat) some closure properties of $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$. Bagan et al. [20] showed that $\mathsf{SP}_{\mathsf{tract}}$ is closed under union, intersection, and reversal.

Lemma 3.5.3 ((i), (ii), (iii) for $\mathsf{SP}_{\mathsf{tract}}$ in Bagan et al. [20])**.** *Both classes* $\mathsf{SP}_{\mathsf{tract}}$ *and* $\mathsf{T}_{\mathsf{tract}}$ *are closed under* (i) *finite unions,* (ii) *finite intersections,* (iii) *reversal,* (iv) *left and right quotients,* (v) *inverses of non-erasing morphisms,* (vi) *removal and addition of individual words.*

Proof. Let $\mathcal{C} \in \{\mathsf{SP}_{\mathsf{tract}}, \mathsf{T}_{\mathsf{tract}}\}$ and $L_1, L_2 \in \mathcal{C}$. Let $n = \max(n_1, n_2)$, where $n_i \in \mathbb{N}$ is the smallest number such that $w_\ell w_1^{n_i} w_m w_2^{n_i} w_r \in L_i$ implies $w_\ell w_1^{n_i} w_m w_2^{n_i} w_r \in L_i$ for $i \in \{1,2\}$.

The proofs for (i) to (vi) all establish that closure under left-synchronized power abbreviations (or the analogous property for $\mathsf{SP}_{\mathsf{tract}}$) is preserved under the operations. Let therefore $w_\ell, w_m, w_r \in \Sigma^*$ and $w_1, w_2 \in \Sigma^+$ (if $\mathcal{C} = \mathsf{SP}_{\mathsf{tract}}$) or $w_1, w_2 \in a\Sigma^*$ for some $a \in \Sigma$ (if $\mathcal{C} = \mathsf{T}_{\mathsf{tract}}$).

(i) If $w_\ell w_1^n w_m w_2^n w_r \in L_1 \cup L_2$, then there exists $i \in \{1,2\}$ with $w_\ell w_1^n w_m w_2^n w_r \in L_i$ and thus $w_\ell w_1^n w_2^n w_r \in L_i$. So, $w_\ell w_1^n w_2^n w_r \in L_1 \cup L_2$.

(ii) If $w_\ell w_1^n w_m w_2^n w_r \in L_1 \cap L_2$, then $w_\ell w_1^n w_m w_2^n w_r \in L_i$ and thus $w_\ell w_1^n w_2^n w_r \in L_i$ for $i \in \{1,2\}$. So, $w_\ell w_1^n w_2^n w_r \in L_1 \cap L_2$.

(iii) The "reversal" of the definitions define the same class of languages, see Definition 2.5.4 (Definition 3.1.11 and Theorem 3.1.9, respectively).

(iv) Let $w \in \Sigma^*$. If $w_\ell w_1^n w_m w_2^n w_r \in w^{-1}L_1$, then $(ww_\ell)w_1^n w_m w_2^n w_r \in L_1$ and therefore $(ww_\ell)w_1^n w_2^n w_r \in L_1$. This implies $w_\ell w_1^n w_2^n w_r \in w^{-1}L_1$. Closure under right-quotients follows from closure under left-quotients together with closure under reversal.

(v) Let h be a non-erasing morphism. Let $w_\ell w_1^n w_m w_2^n w_r \in h^{-1}(L_1)$. Then we have $h(w_\ell w_1^n w_m w_2^n w_r) \in L_1$, so $h(w_\ell)h(w_1)^n h(w_m)h(w_2)^n h(w_r) \in L_1$. Since $h(w_1), h(w_2)$ are nonempty in the case $\mathcal{C} = \mathsf{SP}_{\mathsf{tract}}$ and $h(w_1), h(w_2)$ are in $h(a)\Sigma^*$ in the case $\mathcal{C} = \mathsf{T}_{\mathsf{tract}}$, it follows that $h(w_\ell)h(w_1)^n h(w_2)^n h(w_r) \in L_1$. This implies $w_\ell w_1^n w_2^n w_r \in h^{-1}(L_1)$.

(vi) Let w' be any word from L_1. Here we choose $n = \max(n_1, |w'|)$. Let $w_\ell w_1^n w_m w_2^n w_r \in L_1 - \{w'\}$. As $w_\ell w_1^n w_m w_2^n w_r \in L_1$ we have $w_\ell w_1^n w_2^n w_r \in L_1$ and therefore $w_\ell w_1^n w_2^n w_r \in L_1 - \{w'\}$. We note that $|w'| < |w_\ell w_1^{n'} w_2^{n'} w_r|$. The proof for $L \cup \{w'\}$ is analogous. $\square$

This lemma implies that $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$ each are a positive C_{ne}-variety of languages, that is, a positive variety of languages that is closed under inverse non-erasing homomorphisms.

Lemma 3.5.4. *The classes* $\mathsf{SP}_{\mathsf{tract}}$ *and* $\mathsf{T}_{\mathsf{tract}}$ *are not closed under complement.*

Proof. Let $\Sigma = \{a, b\}$. The language of the expression b^* clearly is in $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$. Its complement is the language L containing all words with at least one a. It can be described by the regular expression $\Sigma^* a \Sigma^*$. Since $b^i a b^i \in L$ for all i, but $b^i b^i \notin L$ for any i, the language L is neither in $\mathsf{SP}_{\mathsf{tract}}$ nor in $\mathsf{T}_{\mathsf{tract}}$. $\square$

It is an easy consequence of Lemma 3.5.3 (vi) that there do not exist best lower or upper approximations for regular languages outside $\mathsf{SP}_{\mathsf{tract}}$ or $\mathsf{T}_{\mathsf{tract}}$.

Corollary 3.5.5. *Let* $\mathcal{C} \in \{\mathsf{SP}_{\mathsf{tract}}, \mathsf{T}_{\mathsf{tract}}\}$. *For every regular language* L *such that* $L \notin \mathcal{C}$ *and*

- *for every upper approximation* L'' *of* L *(that is,* $L \subsetneq L''$*) with* $L'' \in \mathcal{C}$ *it holds that there exists a language* $L' \in \mathcal{C}$ *with* $L \subsetneq L' \subsetneq L''$*;*
- *for every lower approximation* L'' *of* L *(that is,* $L'' \subsetneq L$*) it holds that there exists a language* $L' \in \mathcal{C}$ *with* $L'' \subsetneq L' \subsetneq L$*.*

The corollary implies that Angluin-style learning of languages in $\mathsf{SP}_{\mathsf{tract}}$ or $\mathsf{T}_{\mathsf{tract}}$ is not possible. However, learning algorithms for single-occurrence regular expressions (SOREs) exist [41] and can therefore be useful for an important subclass of $\mathsf{T}_{\mathsf{tract}}$.

Conclusion In the next section, we will see that the types of RPQs that users ask are different from those that lead to high worst-case complexity.

Chapter 4

Regular Path Queries in Practice

In the previous chapter we have seen that there are languages for which $\mathsf{Trail}(L)$ is NP-complete. The picture for $\mathsf{SimPath}(L)$ is analogous. In this chapter we study real-world query logs to understand why database systems are not regularly brought to their knees by NP-complete queries. We define the class of *simple transitive expressions* which captures 99.99% of the queries found in the logs and study their complexity in detail in the next chapter.

4.1 SPARQL Logs

To the best of our knowledge, the first study on huge logs of structured queries was done by Bonifati et al. [53]. The study had a total of about 180 million SPARQL queries which came almost exclusively from DBpedia, Semantic Web Dog Food, LinkedGeoData, BioPortal, OpenBioMed, and the British Museum, ranging from 2009 until 2016.

The study of Bonifati et al. [53] had 247k SPARQL property paths in unique queries, which gave us a first impression what kind of RPQs actually occur. Syntactically, SPARQL property paths are extensions of RPQs. This is important, because it means that the types of regular expressions we will see are not syntactically constrained by the query language. On top of the ordinary operators for RPQs, SPARQL allows operators for wildcards and for following edges in the reverse direction. This would not be the case for Cypher, for example. (In Neo4j's Cypher 3.2 manual, only single labels or wildcards were allowed below Kleene stars [168]. Cypher 9 is becoming more liberal and allows disjunction below a Kleene star, see [99, Figure 3: Syntax of Cypher patterns]. In the near future, Cypher plans to support full regular path queries [99].)

In Table 4.1, we provide a summary of the types of property paths found in the data of [53]. That is, Table 4.1 is not the table appearing in [53], but we went over the raw data again and aggregated the types of expressions slightly differently. We use the following conventions:

- Lower case letters denote single symbols.
- Upper case letters denote sets of symbols.

Expression Type	ℓ	*Relative*	*STE?*	*Expression Type*	ℓ	*Relative*	*STE?*
$(a_1 + \cdots + a_\ell)^*$	2–4	29.10%	yes	abc^*		$< 0.01\%$	yes
$\sqcup$		25.48%	yes$^{(*)}$	$A_1 \cdots A_\ell$	2–6	$< 0.01\%$	yes
a^*		19.66%	yes	$(a_1 + a_2)?$		$< 0.01\%$	yes
$a_1 \cdots a_\ell$	2–6	8.66%	yes	$\sqcup^*$		$< 0.01\%$	yes$^{(*)}$
a^*b		7.73%	yes	$\sqcup b^*$		$< 0.01\%$	yes$^{(*)}$
$(a_1 + \cdots + a_\ell)$	1–6	6.61%	yes	$\sqcup?$		$< 0.01\%$	yes$^{(*)}$
$(a_1 + \cdots + a_\ell)^+$	1–2	1.54%	yes	$(ab^*) + c$		$< 0.01\%$	no
$a_1?a_2? \cdots a_\ell?$	1–5	1.15%	yes	$a^* + b$		$< 0.01\%$	no
$a(b_1 + b_2)?$		0.01%	yes	$a + b^+$		$< 0.01\%$	no
$a_1 a_2? \cdots a_\ell?$	2–3	0.01%	yes	$a^+ + b^+$		$< 0.01\%$	no
$a^*b?$		$< 0.01\%$	yes	$(ab)^*$		$< 0.01\%$	no

Table 4.1: Structure of the 247,404 SPARQL property paths that were also used in the query logs investigated by Bonifati et al. [53]. The structure is sometimes in terms of a variable $\ell \in \mathbb{N}$, for which the second column indicates the values that were found in the logs. *Relative* indicates which percentage of the 247,404 property paths have this structure.

- We denote a wildcard test by $\sqcup$.[1]
- We do not distinguish between following an edge in the forward or backward direction.[2]
- Each expression type also encompasses its symmetric form. For instance, when we write a^*b, we count the expressions of the form a^*b and ba^*. We always list the variant that occurred most often in the data. That is, a^*b occurred more often than ba^*.

Under *Expression Type*, the table summarizes which types of expressions are in Bonifati et al.'s data set, sometimes parameterized by a number ℓ for which the next column describes the values that were found. *Relative* describes which percentage of the 247,404 expressions fall into this expression type. We discuss *STE?* in the next section.

In Table 4.1 we can immediately observe that the property paths found in the query logs of Bonifati et al. are not very complex and almost all are in $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$. In fact, the query $(ab)^*$ occured only once and we found out that this query was posed by a theoretician testing the robustness of the engine [Vrgoč, personal communication].

Another thing to keep in mind is how to interpret the classification in Table 4.1. After all, property paths do not occur often in the logs of Bonifati et al. [53]: only about 0.4%

[1]We treat every expression of the form $!a$ ("match every label that is not a") as a wildcard. In the total corpus, 17 expressions use the operator "!" in a slightly more complex way than just $!a$, for instance, $(!a+!b)^*$ or $(a+!a)^*$, which boil down to reachability tests in the graph and both of which we classified as $\sqcup^*$.

[2]That is, we treat the property path a the same way as $\hat{}a$. The operator $\hat{}$ was used in 306 expressions.

of the queries have them. However, this seems to be an artifact of the underlying data. Most of the property paths appear in DBpedia queries, but DBpedia was designed when property paths were not yet part of SPARQL. In a more recent study on Wikidata query logs, containing 35 million unique queries, a drastically larger 38.94% of the queries use property paths [55]. Moreover, the structure of these property paths shows a picture similar to what we see in Table 4.1 [55].

4.2 Simple Transitive Expressions

We now define a class of RPQs called *simple transitive expressions (STEs)*, with the intent of capturing the vast majority of the expressions in Table 4.1, while avoiding languages like $(aa)^*$ and a^*ba^* for which SimPath and Trail are NP-complete. Intuitively, simple transitive expressions aim at capturing very basic navigation in graphs: first do some *local navigation*, followed by an optional *transitive step*, and finally again some *local navigation*. The rationale is that, if we want to connect entities in a graph database, then this is a natural way to navigate. Let us again consider our introductory example of people who are artists. When we want to find out if a Person is an artist, we first need to do some local navigation (following an `occupation`-edge) and then perform a transitive reflexive step (following an arbitrarily long path of `subclassof`-edges). More precisely, simple transitive expressions allow to:

1. first follow a path of length *exactly* k_1 or *at most* k_1 (for some $k_1 \in \mathbb{N}$),
2. then do a (reflexive) transitive closure step,
3. finally, follow a path of length *exactly* k_2 or *at most* k_2 (for some $k_2 \in \mathbb{N}$).

All three steps are subject to label tests. Furthermore, any step can be omitted, so a simple transitive expression can also express that paths must have length between k_1 and $k_1 + k_2$. In the following definition, we use sets $A = \{a_1, \dots, a_\ell\} \subseteq \Sigma$ to abbreviate disjunctions $(a_1 + \cdots + a_\ell)$.

Definition 4.2.1. An *atomic expression* is of the form $A \subseteq \Sigma$ with $A \neq \emptyset$. A *bounded expression* is a regular expression of the form $A_1 \cdots A_k$ or $A_1? \cdots A_k?$, where $k \geq 0$ and each A_i is an atomic expression. Finally, a *simple transitive expression (STE)* is a regular expression

$$B_{\text{pre}} T^* B_{\text{suff}},$$

where B_{pre} and B_{suff} are bounded expressions and T is ε or an atomic expression.

A minor technicality is that we can take $T = \varepsilon$. This means that T^* will only match the empty word, and therefore the STE defines a finite language. In Table 4.1 the column *STE?* indicates whether the expression is an STE. Here, we write "yes$^{(*)}$" to indicate that the expression is an STE if a wildcard is treated the same as a set of labels A. (Our algorithms indeed can be generalized to incorporate wildcards.)

In total, we saw that only 20 property paths are not STEs or trivially equivalent to an STE (by taking $T = \varepsilon$ in the definition of STEs, for example). For instance, the expression type $a_1 a_2? \cdots a_\ell?$ is equivalent to an STE where $B_{\text{pre}} = a_1$, $T = \varepsilon$, and $B_{\text{suff}} = a_2? \cdots a_\ell?$. In this sense, 99.992% of the property paths in Table 4.1 correspond to STEs.

In fact, *all* expressions in the table except for $(ab)^*$ are unions of STEs. Unions of STEs can actually be handled in the same way than STEs, by applying the STE evaluation algorithm to each part of the union.

Chapter 5

Fine-grained Dichotomy for STEs

In the last chapter we have defined the class of simple transitive expressions (STEs). Since these expressions are relevant in practice, we want to study their complexity in more detail. To this end, we will study SimPath and Trail from a parameterized complexity perspective. The reason why we focus on parameterized complexity is that we want to obtain a more precise view on the complexity of different languages L, but at the same time, if we give L as part of the input, then SimPath is trivially NP-complete because it encompasses the NP-complete Hamilton Path problem. Indeed, given a directed graph G with n nodes and only a-edges, nodes s and t, and RPQ a^{n-1}, the SimPath problem asks if there is a Hamiltonian path from s to t in G. Using Lemma 5.5.1(3), NP-completeness also follows for Trail.

On the other hand, the RPQ a is trivially tractable. This example tells us that we should also take the size of RPQs into account, which is why we parameterize our problems with the size of the RPQ. Alon et al. [7] proved that SimPath for graphs with n nodes and RPQs of the form a^k is fixed-parameter tractable in k, using their famous color-coding technique. We note that a precise view on the parameterized complexity of SimPath subsumes long-standing open problems. For instance, Alon et al. [7] showed that SimPath is in P if $k = \log n$, but the question if SimPath is in P if $k = \log^2 n$ has been open since 1995 [7].[1]

5.1 Parameterized Complexity

We give a quick overview on *parameterized complexity* and the parameterized versions of SimPath and Trail that we consider. We follow the exposition of Cygan et al. [75] and refer to their work for further details. A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$ where, as before, Σ is a fixed, finite alphabet. For an instance $(x, p) \in \Sigma^* \times \mathbb{N}$, we call p the *parameter*. The *size* $|(x, p)|$ of an instance (x, p) is defined as $|x| + p$. A parameterized problem L is called *fixed-parameter tractable* if there exists an algorithm $\mathcal{A}$, a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, p) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$

[1] Björklund et al. [48] showed that, under the Exponential Time Hypothesis (ETH), for any nondecreasing polynomial time computable function f that tends to infinity there is no P algorithm that can decide if there exists a simple path of length $\Omega(f(n) \log^2 n)$ between two nodes in a graph of size n. Chen and Flum [66, Theorem 12] showed that, under the ETH, deciding whether there exists a simple path of length $\log^2 n$ in an undirected graph cannot be in polynomial time.

correctly decides whether $(x,p) \in L$ in time at most $f(p) \cdot |(x,p)|^c$. The complexity class containing exactly the fixed-parameter tractable problems is called FPT.

In terms of parameterized complexity, Downey and Fellows [86] introduced the W-hierarchy, where FPT = W[0] and $\mathrm{W}[i] \subseteq \mathrm{W}[j]$ for all $i \leq j$. It is a standard assumption in parameterized complexity theory that FPT $\neq$ W[1]. In order to prove W[1] hardness, we need the notion of *fpt-reduction*. If L and L' are two parameterized problems, an *fpt-reduction* from L to L' is an algorithm $\mathcal{R}$ that, given an instance (x,k) of L, outputs an instance (x',k') of L' such that

- (x,k) is a yes-instance of L if and only if (x',k') is a yes-instance of L',
- $k' \leq g(k)$ for some computable function g, and
- the running time of $\mathcal{R}$ is $f(k) \cdot |x|^{O(1)}$ for some computable function f.

Let $\mathcal{R}$ be a class of regular expressions. We will consider the following parameterized variants of SimPath and Trail.

PSimPath($\mathcal{R}$)	
Given:	A directed multigraph $G = (V,E,\mathcal{E})$, two nodes $x,y \in V$, $r \in \mathcal{R}$
Parameter:	$\|r\|$
Question:	Is there a simple path from x to y in G that matches r?

PTrail($\mathcal{R}$)	
Given:	A directed multigraph $G = (V,E,\mathcal{E})$, two nodes $x,y \in V$.
Parameter:	$\|r\|$
Question:	Is there a trail from x to y in G that matches r?

If $\mathcal{R}$ is just a single regular expression r, then we simply write PSimPath(r) instead of PSimPath($\{r\}$), and analogously for PTrail.

5.2 Dichotomies for STEs

Our main technical results of this chapter are two dichotomies for evaluating STEs under simple path and trail semantics. That is, we precisely characterize for which classes $\mathcal{R}$ of STEs the problems PSimPath($\mathcal{R}$) and PTrail($\mathcal{R}$) are easy and for which classes these problems are difficult. Here, "easy" and "difficult" refer to complexities in parameterized complexity, namely *fixed-parameter tractable* and W[1]-hard. Our results will imply that PSimPath and PTrail are "easy" for the types of expressions in Table 4.1—except for $(ab)^*$. Furthermore, the parameters on which the complexity can exponentially depend are small.

Some Examples and Intuition

We give a bit of intuition about our results. Throughout the example, we use the following notation. The input graph is always denoted as G, and it has n nodes and m edges. We always denote the start and end nodes in the input of the PSimPath problem by s and t, respectively. We will abbreviate long concatenations with a power notation, that is, we use r^k to denote a sequence of k times the expression r. For instance a^4 denotes the expression $aaaa$. Let a^k denote the class $\{a^k \mid k \in \mathbb{N}\}$ of STEs. We define the classes $(a?)^k$, a^ka^*, ba^ka^*, and a^kba^* analogously. (We do this to be able to discuss some classes of expressions, using a simple notation. If we use this convention, we will consistently denote the variable by "k".)

We now discuss the complexities of PSimPath for these classes. As a first example, we consider PSimPath for $(a?)^k$. This problem is easy to solve: one can simply use an algorithm that tests reachability with a-labeled edges. The crux is that loops do not matter: if there is a path from s to t that matches $(a?)^k$ then there is also a simple such path, since removing loops does not change matching $(a?)^k$.

This technique does not work for our second example: PSimPath for a^k. However, Alon et al.'s color coding technique [7] can solve this problem in time $2^{O(k)}m \log n$. Color coding therefore shows that PSimPath for a^k is fixed-parameter tractable, where the parameter is the size k of the RPQ: it is an algorithm with complexity $f(k) \cdot p(|G| + k)$, where f is a computable function and p is a polynomial. The function f is even single exponential in this case. Notice that, if P $\neq$ NP, we cannot hope for f to be a polynomial function, because PSimPath for a^k is at least as difficult as the Hamiltonian Path problem. (Indeed, the cases of PSimPath for a^k where we give a graph G with only a-labeled edges and the RPQ a^{m+1} are equivalent to the Hamiltonian Path problem for G.)

As a third example, we consider PSimPath for a^ka^*. This problem requires yet another technique, since color coding is designed to work for fixed-length paths. It can be solved in time $2^{O(k)}(n^2 + mn)$, however, using the representative sets technique of Fomin et al. [97]. The representative sets technique is nontrivial and addresses the following problem. Assume that we try to deal with a^ka^* naively by considering all simple paths P of length k that start in s. For each such path P, assuming it ends in some node x_P, we could then test reachability from x_P to t while avoiding the nodes of P. But this algorithm is too inefficient. We may have up to n^k different possibilities for P, which means that the running time is not of the form $f(k) \cdot p(|G| + k)$ for a polynomial p and computable function f. In other words, it does not show that the problem is fixed-parameter tractable. This is where the representative sets technique is useful. It shows that the number of different paths P we have to consider can be limited to $2^{O(k)}n$, which makes the problem fixed-parameter tractable. The representative sets technique can even be adapted so that it *enumerates* all the simple paths.

We turn to two cases where the edge labels become important. First, consider PSimPath(ba^ka^*). Here, we can simply enumerate all b-edges that start in x and then use the algorithm for PSimPath(a^ka^*) from there (and making sure that we don't visit x). This shows that PSimPath(ba^ka^*) is fixed-parameter tractable.

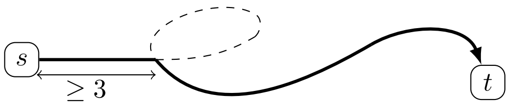


Figure 5.1: Intuition behind cuttability, using $bbba^*$

Second, take $\mathsf{PSimPath}(a^k ba^*)$. At its core, this problem is a variant of the Two Disjoint Paths problem. We are essentially searching for two nodes x and y such that there is a path P_1 of length k from s to x and a path P_2 from y to t. Moreover, P_1 and P_2 should be node-disjoint and there should be a b-edge from x to y. Since we can prove that this Two Disjoint Paths problem (with parameter k) is W[1]-hard, see Theorem 5.4.7, it turns out that $\mathsf{PSimPath}(a^k ba^*)$ is hard as well.

The central notion in our dichotomy for $\mathsf{PSimPath}$ is *cut borders* of STEs. We first explain this notion intuitively, based on two simple examples. Consider the expressions $r_1 = aaaa^*$ and $r_2 = aaab^*$. Assume that, as in Figure 5.1, we found a path p (that may contain a loop) from s to t that matches r_1. Intuitively, if we want to test if the simple path p' obtained from p by deleting all loops still matches r_1, we just need to test if p' has length at least three. For r_2, however, we additionally need to test that the loop does not occur in the prefix of length 3 of p. For this reason, the cut border of r_2 will be equal to 3. We can prove that this notion of cut border is indeed the crucial one for the complexity of $\mathsf{PSimPath}$.

Dichotomy for Simple Paths

We first define the notions that we need for the dichotomy for simple paths.

Definition 5.2.1. Let $r = B_{\text{pre}} T^* B_{\text{suff}}$ be an STE. If $B_{\text{pre}} = A_1 \cdots A_{k_1}$, then the *left cut border* c_1 *of* r is the largest value such that $T \not\subseteq A_{c_1}$ if it exists and zero otherwise. If $B_{\text{pre}} = A_1? \cdots A_{k_1}?$, then the left cut border is zero. Symmetrically, if $B_{\text{suff}} = A'_{k_2} \cdots A'_1$, then the *right cut border* c_2 *of* r is the largest value such that $T \not\subseteq A'_{c_2}$ if it exists and zero otherwise. (Notice that the indices in B_{suff} are reversed.) If $B_{\text{suff}} = A'_{k_2}? \cdots A'_1?$, then the right cut border is zero.

We explain the intuition behind cut borders in Figure 5.2. For $c \in \mathbb{N}$, an expression is *c-bordered* if the sum of its left and right cut borders is c. We call a class $\mathcal{R}$ of STEs *cuttable* if there exists a constant $c \in \mathbb{N}$ such that each expression in $\mathcal{R}$ is c'-bordered for some $c' \leq c$.

We can now prove a dichotomy on the complexity of $\mathsf{PSimPath}(\mathcal{R})$ for classes of STEs $\mathcal{R}$, if $\mathcal{R}$ satisfies the following mild condition. We say that $\mathcal{R}$ *can be sampled* if there exists an algorithm that, given $k \in \mathbb{N}$, returns an expression in $\mathcal{R}$ that is k'-bordered with $k' \geq k$, and “no” if there is no such expression. We need the condition that $\mathcal{R}$ can be sampled to prove the W[1]-hardness. For this reason, this condition is no longer needed in the upper bound results (Lemma 5.3.16 and Theorem 9.3.2).

We now state our main result on PSimPath and explain the cut borders, cuttability, and the sampling condition after its statement. (We only require the condition that $\mathcal{R}$ can be sampled for the lower bound proof in part (b).)

Here, FPT is the class of problems that is *fixed-parameter tractable.* It is a standard assumption in parameterized complexity theory that FPT $\neq$ W[1]. This assumption has a similar calibre as the P $\neq$ NP assumption in terms of decision problems.

We now explain cut borders, cuttability, and the condition that $\mathcal{R}$ can be sampled. To this end, the *left* (respectively, *right*) *cut border* of an STE $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ is the largest value i such that T has a symbol that is not in A_i (respectively, A'_i). If we have $A_1? \cdots A_{k_1}?$ (respectively, $A'_{k_2}? \cdots A'_1?$), then the left (respectively, right) cut border is 0. The *cut border* of r is the sum of its left and right cut border. A class $\mathcal{R}$ of STEs is *cuttable* if there exists a $c \in \mathbb{N}$ such that the cut border of each expression $r \in \mathcal{R}$ is at most c. The intuition of cut borders is explained in Figure 5.2: they characterize parts of paths in which it is not allowed to remove loops to obtain a simple path that still matches the expression.

Finally, we say that $\mathcal{R}$ *can be sampled* if there exists an algorithm that, given a number k in unary, returns an expression from $\mathcal{R}$ that has cut border at least k. Notice that this is a very weak restriction on $\mathcal{R}$.

Theorem 5.2.2. *Let $\mathcal{R}$ be a class of STEs that can be sampled.*

(a) If $\mathcal{R}$ is cuttable, then PSimPath($\mathcal{R}$) is in FPT and

(b) otherwise, PSimPath($\mathcal{R}$) is W[1]-hard.

The result will follow immediately from Lemma 5.3.16 and Lemma 5.4.8.

Notice that the difference between cuttable and non-cuttable classes of STEs can be subtle. For instance, $a^k b^*$ and $a^k(a+b)^*$ are non-cuttable, but $(a+b)^k a^*$ is cuttable. Looking back at Table 4.1, we see that abc^* is 2-bordered and all other STEs are either 0-bordered or 1-bordered. It therefore seems that cut borders in practice are small and over 99% of the expressions fall on the tractable side of Theorem 5.2.2.

Dichotomy for Trails

We now present a similar dichotomy for trails. The dichotomy is, perhaps surprisingly, different from the one in Theorem 5.2.2 in the sense that more classes fall on the tractable side. For instance, PSimPath($a^k b^*$) is intractable, whereas PTrail($a^k b^*$) is fixed parameter tractable because the a-path and the b-path can be evaluated independent of each other (no a-edge will be equal to a b-edge).

Before we can give the dichotomy, we first need some definitions.

Definition 5.2.3. Let $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ be an STE with left cut border c_1 and right cut border c_2. We say that A_i with $i \leq c_1$ (respectively, A'_j with $j \leq c_2$) is a *conflict position* if $A_i \cap T \neq \emptyset$ (respectively, $A'_j \cap T \neq \emptyset$). We say that a class $\mathcal{R}$ of STEs is *almost conflict-free* if there exists a constant c such that each $r \in \mathcal{R}$ has at most c conflict positions and *conflict-free* if this constant is 0.

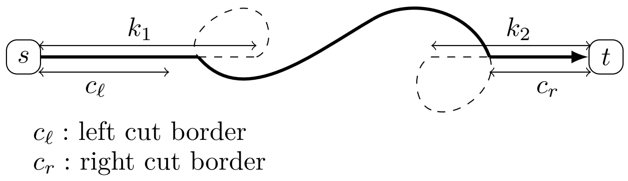


Figure 5.2: Assume $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ has left and right cut borders c_1 and c_2, respectively. Assume that an arbitrary path from s to t matches r such that its length k_1 prefix and length k_2 suffix do not have loops and are node disjoint. If, after removing all loops, (1) the length c_1 prefix and length c_2 suffix are still the same and (2) the path still has length at least $k_1 + k_2$, then it matches r.

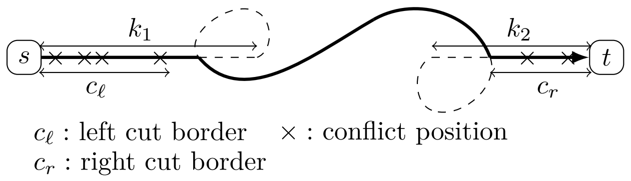


Figure 5.3: Visualization of the effect of conflict positions in a path that matches an STE r. If we start with an arbitrary path and remove loops, we mainly need to be careful about labels behind the cut borders that can be identical to labels in the transitive part.

In Figure 5.3 we give a visual intuition about the meaning of conflict positions. The class $a^k b^*$ is not cuttable, but it is conflict-free because $\{a\}$ and $\{b\}$ have an empty intersection. The point is that an edge labeled by some symbol in $\{a\}$ can never be the same than an edge labeled by some symbol in $\{b\}$, since their labels must be different. Therefore, we can evaluate a^k and b^* separately.

We say that $\mathcal{R}$ *can be conflict-sampled* if there exists an algorithm that, given $k \in \mathbb{N}$, returns an expression in $\mathcal{R}$ that has k' conflict positions with $k' \geq k$, and "no" if there is no such expression. Our main dichotomy for trails is the following.

Theorem 5.2.4. *Let $\mathcal{R}$ be a class of STEs that can be conflict-sampled.*

(a) If $\mathcal{R}$ is almost conflict-free, then $\mathsf{PTrail}(\mathcal{R})$ is in FPT and

(b) otherwise, $\mathsf{PTrail}(\mathcal{R})$ is W[1]-hard.

This theorem follows immediately from Lemma 5.6.3 and Lemma 5.6.5.

Core Techniques At the core of our tractability results lies the representative sets technique of Fomin et al. [97]. This technique can be used to find simple paths and trails of length at least k in time $2^{O(k)}(n^2 + nm)$, given a graph and the number k. If regular path queries are involved, the technique is only compatible with certain languages, such as cuttable or conflict-free STEs. The compatible languages have the property that we only need to guard a constant number of nodes/edges at the beginning and at the end of the path, to make sure that the rest of the path does not re-use the same nodes/edges.

Indeed, we can show that for languages violating this property, the problem becomes intractable. The reason is that it becomes at least as hard as a parameterized version of the two-disjoint paths problem. This parameterized problem asks: given a graph G, node pairs (x_1, y_1) and (x_2, y_2), and parameter $k \in \mathbb{N}$, are there two disjoint paths p_1 from x_1 to y_1 and p_2 from x_2 to y_2 such that p_1 has length k. (One can consider node-disjoint or edge-disjoint paths here.) We prove that this problem is W[1]-hard, both when node- or edge disjointness is required.

What Does This Mean?

If we interpret Theorems 5.2.2 and 5.2.4 in the light of the real world property paths in Table 4.1 we can observe the following. Let n and m be the number of nodes and edges of the graph, respectively.

Concerning simple paths semantics, Theorem 5.2.2 tells us that $\mathsf{PSimPath}(\mathcal{R})$ is fixed-parameter tractable for cuttable classes $\mathcal{R}$. This result, together with the observation that the largest cut border in Table 4.1 is two, and therefore very small, can be seen as an explanation why, in practice, simple path semantics usually does not bring systems to their knees, even though this would theoretically be possible using regular expressions such as $(aa)^*$. Since the evaluation problem under simple path semantics generalizes the Hamilton Path problem (if $|r| = |V| - 1$), we cannot hope for a significantly better complexity unless P = NP.

Looking closer, we prove that $\mathsf{PSimPath}$ is in time $2^{O(|r|)} \cdot |V|^{c+3} \cdot |E|$ in the worst case (Lemma 5.3.16), where $|r|$ is the size of the RPQ, c is the largest cut border in $\mathcal{R}$, and $|V|$ and $|E|$ are the number of nodes and edges in the graph, respectively. In Table 4.1, the largest value of c in STEs or unions thereof is two (for abc^*), and $|r|$ is relatively small. One should keep in mind that this is a worst-case bound. In most practical settings, we expect that the run-time of even more naive evaluation algorithms will not come close to requiring n^{c+3} time for these simple expressions. For instance, the n^c factor comes from considering all paths that start in a given node s and obey a label constraint. For instance, for the expression abc^*, these are just the paths that start in s and are labeled ab. While this can, in the worst case, be $|V|^2$ many paths, we expect this to be much less in real databases.

The story for trails is similar. Here our upper bound admittedly gives less efficiency guarantees than the one for simple paths, but this is mainly because we have developed our methods for simple paths and then adapted them for trails. Furthermore, the dichotomy shows that it is easier to deal with trails than with simple paths: for every class of queries for which we have fixed-parameter tractable algorithms for simple path semantics, we also have them for trail semantics, but not vice versa.

5.3 Main Upper Bound for PSimPath

In this section we will prove part (a) of Theorem 5.2.2. To this end, we first need some preliminary results.

Preliminary Technical Result: Downward Closed Languages

We first recall a useful result which we will use to deal with downward closed parts of STEs, to be more precise, with bounded expressions of the form $A_1? \cdots A_k?$ and the transitive part T^*. Note that this lemma is restricted to downward closed languages, since simple paths in the product of a directed multigraph and an NFA may use nodes $(u, q_1) \neq (u, q_2)$ and may therefore correspond to non-simple paths in G. Recall that downward closed languages were introduced in Section 2.5.1. We will use the following lemma.

Lemma 5.3.1 (Theorem 5 in [160]). *Let N be an NFA for a downward closed language. Let G be a directed multigraph and s and t be nodes in G. Then we can decide if there is a simple path from s to t that matches N in time $O(|N||G|)$.*

Proof. The algorithm consists of two steps. First construct the product between N and G, which takes time $O(|N||G|)$. Then, test if (t, f) is reachable from (s, i) for some accepting state f and initial state i. Indeed, (t, f) is reachable from (s, i), if and only if there exists some path p from s to t that matches N. Since $L(N)$ is downward closed, the simple path obtained from p by removing all loops still matches N. ☐

Representative Sets and Simple Paths with Length Constraints

To prove Theorem 5.2.2(a), we need the representative sets technique [97]. At their core, this technique can be used to prove that the following parameterized problems are in FPT:

- PSimPathLength: Given a directed multigraph G, nodes s, t, and a parameter $k \in \mathbb{N}$, is there a simple path from s to t of length exactly k in G?
- PSimPathLength$^{\geq}$: Given a directed multigraph G, nodes s, t, and a parameter $k \in \mathbb{N}$, is there a simple path from s to t of length at least k in G?

Before we explain the representative sets technique, we first restate some important results on these problems: Alon et al. [7] proved that PSimPathLength is in FPT, using their famous color coding technique. For the theorem statement, we assume that $G = (V, E, \mathcal{E})$ is a directed multigraph.

Theorem 5.3.2 (Alon et al. [7]). *PSimPathLength is in time $2^{O(k)}|E| \log |V|$ and therefore in FPT.*

Bagan et al. [19, Theorem 7] combine color coding and dynamic programming to prove that, given a directed graph G, nodes s, t, an NFA N, and a parameter $k \in \mathbb{N}$, deciding if there is a simple path from s to t of length at most k that matches $L(N)$ can be done in time $2^{O(k)}|N||G|\log|G|$. In their proof they actually show that it is in time $2^{O(k)}|N||G|\log|V|$. From this, the following can be inferred.

Lemma 5.3.3 (Immediate consequence of Corollary 1 in Bagan et al. [19])**.** *Let $G = (V, E, \mathcal{E})$ be a directed multigraph, s, t be nodes of G, and N be an NFA accepting a finite language. It can be decided in time $2^{O(|N|)}|G|\log|V|$ if there exists a simple path from s to t in G, labeled with a word from $L(N)$.*

Corollary 5.3.4. *Let $\mathcal{R}$ be a class of STEs defining finite languages. Then PSimPath($\mathcal{R}$) is in FPT or, more precisely, in time $2^{O(|r|)}|G|\log|V|$.*

PSimPathLength$^{\geq}$ can be shown to be in FPT by adapting methods from Fomin et al. [97]. They proved that testing the existence of simple directed cycles of length at least k is in FPT and discovered that their technique also works for paths [82]. The following theorem is therefore by the authors of [97].

Theorem 5.3.5 (Similar to Theorem 5.3 in [97])**.** *PSimPathLength$^{\geq}$ is in FPT. More precisely, it is in time $2^{O(k)} \cdot |E||V|\log|V|$.*

We received a proof sketch of the result from Holger Dell [82] (who attributed the result to Fomin et al., the authors of [97]). Next, we provide a self-contained generalization of Theorem 5.3.5 that deals with edge labels, based on the proof sketch we received. Our contribution is the generalization of the approach towards the extra condition that checks the labels of the path. We emphasize that the most complex part of the proof concerns the length constraints and is due to the authors of [97].

One way to test whether there exists a simple path from s to t of length at least k is to find a simple path p_k of length exactly k such that there is a path from the last node of p_k to t that avoids p_k. But the number of such paths p_k is $n!/(k!(n-k)!)$. So naively testing and enumerating all paths is not fixed-parameter tractable in k. We therefore need a way to decrease the number of such paths we need to consider. We can do this using the following notion, originally introduced by Monien [163].

Definition 5.3.6 (k-representative family [97])**.** Given a set of nodes V, an integer $k \in \mathbb{N}$, and a set $\mathcal{S}$ containing subsets of V, all of size ℓ, for some $\ell \in \mathbb{N}$, we say that a subfamily $\hat{\mathcal{S}} \subseteq \mathcal{S}$ is *k-representative for $\mathcal{S}$* if the following holds: for every set $Y \subseteq V$ of size at most k, if there is a set $X \in \mathcal{S}$ disjoint from Y, then there is a set $\hat{X} \in \hat{\mathcal{S}}$ disjoint from Y. We abbreviate this by $\hat{\mathcal{S}} \subseteq^k_{\text{rep}} \mathcal{S}$.

Intuitively, if one needs to be able to avoid k-element sets, it is sufficient to store a k-representative set. Notice that each set $\mathcal{S}$ is trivially k-representative for itself. The crux is that we want to be able to compute k-representative sets that are small. The condition that all sets in $\mathcal{S}$ have the same size is just a technicality that allows us to simplify proofs later.

In the following, s, v are nodes and r is a regular expression of the form $A_1 \cdots A_k$ for some $k \in \mathbb{N}$. We define

$$P^r_{s,v} := \{V(p) \mid \text{there is a simple path } p \text{ from } s \text{ to } v \text{ in } G \text{ that matches } r\}.$$

Notice that, by definition of r, these simple paths from s to v in G have length k. Therefore, all sets in $P^r_{s,v}$ have exactly $k+1$ elements.

We next show that representative sets $\hat{P}^r_{s,v} \subseteq^{k+1}_{\text{rep}} P^r_{s,v}$ exist for each node $v \in V$ and can be constructed in fixed parameter tractable time. We restate the relevant parts of Lemma 3.3 and Corollary 4.16 from [97] since we need them in the proof. We note that while they only considered simple graphs, their results immediately hold for (unlabeled) multigraphs. Lemma 5.3.7 states that the relation "is a k-representative set for" is transitive. Corollary 5.3.8 gives a rough time and space bound for computing k-representative sets.

Lemma 5.3.7 (Lemma 3.3 in [97] for directed graphs)**.** *Given a directed multigraph $G = (V, E, edge)$ and a family $\mathcal{S}$ of subsets of V. If $\hat{\mathcal{S}} \subseteq^k_{rep} \mathcal{S}'$ and $\mathcal{S}' \subseteq^k_{rep} \mathcal{S}$, then $\hat{\mathcal{S}} \subseteq^k_{rep} \mathcal{S}$.*

Corollary 5.3.8 (Corollary 4.16 in [97], without weight function)**.** *There is an algorithm that, given a family $\mathcal{A}$ of sets of size ℓ over a set V of nodes and an integer k, computes in time*

$$O\left(|\mathcal{A}| \cdot \left(\frac{k+\ell}{k}\right)^k \cdot 2^{o(k+\ell)} \cdot \log |V|\right)$$

a subfamily $\hat{\mathcal{A}} \subseteq^k_{rep} \mathcal{A}$ such that $|\hat{\mathcal{A}}| \leq \binom{k+\ell}{\ell} \cdot 2^{o(k+\ell)}$.

We now adapt Lemma 5.2 in Fomin et al. [97] to show a time and space bound for representative sets $\hat{P}^r_{s,v} \subseteq^k_{\text{rep}} P^r_{s,v}$ under label constraints. We will need this to deal with the bounded parts of STEs later.

Lemma 5.3.9. *For each regular expression $r = A_1 \cdots A_\ell$ and $k \geq \ell$, there is a collection of families $\hat{P}^r_{s,v} \subseteq^k_{rep} P^r_{s,v}$ with $v \in V \setminus \{s\}$, each of size at most $\binom{k+\ell+1}{\ell+1} \cdot 2^{o(k+\ell)}$. This collection of families can be computed in time $O(8^{k+o(k)}|E| \log |V| + |r||E|)$.*

Proof. Fomin et al. use in their complexity analysis that, given (u, v), one can test if there exists an edge from u to v in the graph in constant time. We first preprocess the multigraph so that, given $(u, v) \in V \times V$ and $i \in \{1, \ldots, \ell\}$, we can test in constant time whether there is an edge from u to v with a label in A_i. Such preprocessing consists of annotating each edge with a ℓ-bit vector and takes time $O(|r||E|)$. (For each edge, and each A_i, test if the edge label is in A_i.)

We describe a dynamic programming algorithm. We assume without loss of generality that the nodes in V are named $\{s, v_1, \ldots, v_{n-1}\}$. Let D be an $\ell \times (n-1)$ matrix where the rows are indexed with integers in $1, \ldots, \ell$ and the columns are indexed with nodes in $\{v_1, \ldots, v_{n-1}\}$. For $i = 1, \ldots, \ell$, we will denote by r_i the prefix $A_1 \cdots A_i$ of r. The entry $D[i, v]$ will store a family $\hat{P}^{r_i}_{s,v} \subseteq^{k+\ell-i}_{\text{rep}} P^{r_i}_{s,v}$ of size at most $\binom{k+\ell+1}{i+1} \cdot 2^{o(k+\ell)}$. We fill the entries in the matrix D in increasing order of rows. For $i = 1$, we set $D[1, v] = \{\{s, v\}\}$ if G has an edge (s, a, v) with $a \in A_1$, and $D[1, v] = \emptyset$ otherwise. Assume that we have filled all the entries until row $i - 1$. For two families of sets $\mathcal{A}$ and $\mathcal{B}$, we define

$$\mathcal{A} \bullet \mathcal{B} = \{X \cup Y \mid X \in \mathcal{A},\ Y \in \mathcal{B}, \text{ and } X \cap Y = \emptyset\}.$$

We denote by $\exists(u, A_i, v)$ that there exists an edge (u, a, v) with $a \in A_i$. Let

$$\mathcal{N}^{r_i}_{s,v} = \bigcup_{\exists(u,A_i,v)} \hat{P}^{r_{i-1}}_{s,u} \bullet \{v\}.$$

Before we continue, we adapt Claim 5.1 in [97] such that it takes r into account, that is:

Claim 5.3.10. $\mathcal{N}^{r_i}_{s,v} \subseteq^{k+\ell-i}_{rep} P^{r_i}_{s,v}$

Proof. The proof is by induction on i. Let $S \in P^{r_i}_{s,v}$ and Y be a set of size at most $k+\ell-i$ such that $S \cap Y = \emptyset$. We will show that there exists a set $S' \in \mathcal{N}^{r_i}_{s,v}$ such that $S' \cap Y = \emptyset$. This will imply the desired result. Since $S \in P^{r_i}_{s,v}$, there exists a simple path $P = (s, u_1) \cdots (u_{i-1}, v)$ in G such that $S = V(P)$ and the predicate $\exists(u_{i-1}, A_i, v)$ is true. The existence of the path $P[0, i-1]$, the subpath of P from s to u_{i-1}, implies that $X' = S \setminus \{v\} \in P^{r_{i-1}}_{s,u_{i-1}}$. Take $Y' = Y \cup \{v\}$. Observe that $X' \cap Y' = \emptyset$ and $|Y'| \le k+\ell-i+1$. Since $\hat{P}^{r_{i-1}}_{s,u_{i-1}} \subseteq^{k+\ell-i+1}_{\text{rep}} P^{r_{i-1}}_{s,u_{i-1}}$ by induction, there exists a set $\hat{X}' \in \hat{P}^{r_{i-1}}_{s,u_{i-1}}$ such that $\hat{X}' \cap Y' = \emptyset$. However, since $\exists(u_{i-1}, A_i, v)$ and $v \notin \hat{X}'$ (because $\hat{X}' \cap Y' = \emptyset$), we have $\hat{X}' \bullet \{v\} = \hat{X}' \cup \{v\}$ and $\hat{X}' \cup \{v\} \in \mathcal{N}^{r_i}_{s,v}$. Taking $S' = \hat{X}' \cup \{v\}$ suffices for our purpose. This completes the proof of the claim. ☐

We fill the entry for $D[i, v]$ for $i \ge 2$ as follows. Observe that

$$\mathcal{N}^{r_i}_{s,v} = \bigcup_{\exists(u,A_i,v)} D[i-1, u] \bullet \{v\}.$$

Let us denote by $d^-(v)$ the indegree of v, that is, the number of edges that end in v. We already have computed the family corresponding to $D[i-1, u]$ for all u. By construction, we have $|\hat{P}^{r_{i-1}}_{s,u}| \le \binom{k+\ell+1}{i} 2^{o(k+\ell)}$ and thus also $|\mathcal{N}^{r_i}_{s,v}| \le d^-(v)\binom{k+\ell+1}{i} 2^{o(k+\ell)}$. Furthermore, we can compute $\mathcal{N}^{r_i}_{s,v}$ in time $O\left(d^-(v)\binom{k+\ell+1}{i} 2^{o(k+\ell)}\right)$. Recall that, due to the preprocessing, we can test if there's an edge with label in A_i in constant time. Now, we use Corollary 5.3.8 on $\mathcal{N}^{r_i}_{s,v}$, which contains sets of size $(i+1)$, to obtain a $(k+\ell+1-(i+1))$-representative, that is, $(k+\ell-i)$-representative subfamily $\hat{\mathcal{N}}^{r_i}_{s,v}$ of size at most $\binom{k+\ell+1}{i+1} \cdot 2^{o(k+\ell)}$ in time

$$O\left(d^-(v)\binom{k+\ell+1}{i} 2^{o(k+\ell)} \cdot \left(\frac{(k+\ell-i)+(i+1)}{k+\ell-i}\right)^{k+\ell-i} \cdot 2^{o((k+\ell-i)+(i+1))} \cdot \log|V|\right).$$

By Claim 5.3.10, we know that $\mathcal{N}^{r_i}_{s,v} \subseteq^{k+\ell-i}_{\text{rep}} P^{r_i}_{s,v}$. Thus, Lemma 5.3.7 implies that $\hat{\mathcal{N}}^{r_i}_{s,v} \subseteq^{k+\ell-i}_{\text{rep}} P^{r_i}_{s,v}$. We define $\hat{P}^{r_i}_{s,v} = \hat{\mathcal{N}}^{r_i}_{s,v}$ and assign this family to $D[i, v]$. This completes the description and the correctness of the algorithm.

Notice that, if we keep the elements in the sets in the order in which they were built using the $\bullet$ operation, then they directly correspond to paths. As such, every ordered set in our family represents a path in the graph.

Algorithm 2: FLPS Algorithm with restricted STE

```
   Input: A directed multigraph $G = (V, E, \mathcal{E})$, nodes $s, t$ in $G$, regular expression
          $rT^*$ with $r = A_1 \cdots A_k$ and $T \subseteq A_i$ for all $i$
   Output: Decide if there exists a simple path from $s$ to $t$ that matches $rT^*$
 1 for every $v \in V$ do
 2     Compute $\hat{P}^r_{s,v} \subseteq^{k+1}_{\text{rep}} P^r_{s,v}$
 3     for every $X \in \hat{P}^r_{s,v}$ do
 4         $V' \leftarrow (V \setminus X) \cup \{v\}$
 5         $E' \leftarrow \{e \in E \mid \mathcal{E}(e) \in V' \times T \times V'\}$
 6         if there exists a path from $v$ to $t$ in $(V', E', \mathcal{E}|_{E'})$ then
 7             return "yes"
 8 return "no"
```

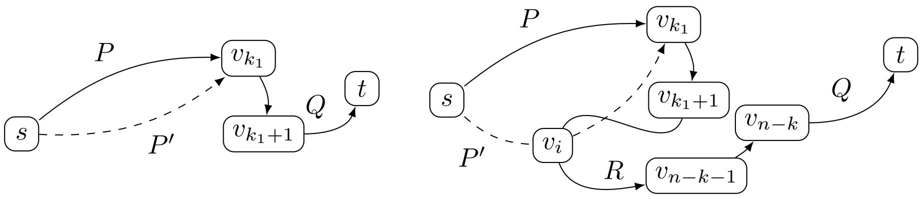


Figure 5.4: This figure shows how we partition a shortest simple path p in the proof of Lemma 5.3.11 if p is short (left) or if p is long (right). Notice that $V(P), V(Q)$, and $V(R)$ are pairwise disjoint.

Since our only change was that we test $\exists(u, A_i, v)$ instead of the existence of an edge (u, v), the time bound $O\left(8^{k+o(k)}|E|\log|V|\right)$ [97, Lemma 5.2] carries over, modulo the additive $O(|r||E|)$ term for preprocessing that we used to test $\exists(u, A_i, v)$ in constant time. The size bound is still guaranteed by Corollary 5.3.8. $\square$

Notice that Claim 5.3.10 will not work for arbitrary regular expressions. We used in the claim that if there exists an edge (u_{i-1}, a, v) with $a \in A_i$, then we can add v to any set $\hat{X}' \in \hat{P}^{r_{i-1}}_{s,u_{i-1}}$ to obtain a valid set in $\mathcal{N}^{r_i}_{s,v}$. For arbitrary regular expressions this is not the case, an example being $(aa + bb)$.

5.3.1 Algorithms for PSimPath

We now present an algorithm that solves PSimPath for the case where the RPQ is of the form $A_1 \cdots A_k T^*$ and is 0-bordered, that is, $T \subseteq A_i$ for all i, see Algorithm 2. The algorithm computes, for every node v, a $(k+1)$-representative set $\hat{P}^r_{s,v}$ in line 2 (for $r = A_1 \cdots A_k$) and subsequently iterates over each set of nodes X in $\hat{P}^r_{s,v}$ to test if there is a path from v to t that avoids X.

For the correctness of the algorithm, the next lemma is crucial.

Lemma 5.3.11. *Let r_1T^* be a 0-bordered expression with $r_1 = A_1 \cdots A_{k_1}$ and let $L(r_2)$ be an arbitrary finite language with words up to length k_2. We define $k = k_1 + k_2$. Then, a directed multigraph $G = (V, E, \mathcal{E})$ has a simple path from s to t that matches $r_1T^*r_2$ if and only if there exists a node $v \in V$ and $X \in \hat{P}^{r_1}_{s,v} \subseteq^{k+1}_{rep} P^{r_1}_{s,v}$, such that G has a simple path from s to t that matches $r_1T^*r_2$ and with the first $k_1 + 1$ nodes belonging to X.*

Proof. The if direction is straightforward. For the only-if direction, let $p = e_1 \cdots e_n$ be a shortest simple path from s to t that matches $r_1T^*r_2$. We first give the intuition of the proof. We will partition p as depicted in Figure 5.4, depending on whether p is short or long. Here, p is the path consisting of the solid edges. Since P and Q are disjoint, we will find a path P' with $V(P') \in \hat{P}^{r_1}_{s,v_{k_1}}$ that is node-disjoint from Q. We then show that, if p is long, P' and R must be disjoint, otherwise it will contradict p being a shortest path.

More precisely, we make the following case distinction. If $|p| \leq 2k_1 + k_2 + 1$, we define $P = e_1 \cdots e_{k_1}$ and $Q = e_{k_1+2} \cdots e_n$. Clearly, P matches r_1 and $e_{k_1+1} \cdot Q$ matches T^*r_2. Let $v_{k_1} = \text{destination}(e_{k_1})$. We have that $V(P) \in P^{r_1}_{s,v_{k_1}}$, $|V(Q)| \leq k + 1$, and $V(P) \cap V(Q) = \emptyset$. Let $\hat{P}^{r_1}_{s,v_{k_1}}$ be a $(k+1)$-representative set of $P^{r_1}_{s,v_{k_1}}$. Then there exists a set $X \in \hat{P}^{r_1}_{s,v_{k_1}}$ with $X \cap V(Q) = \emptyset$. By definition of $P^{r_1}_{s,v_{k_1}}$, there exists a simple path P' from s to v_{k_1} with $V(P') = X$ that matches r_1. Therefore, $P' \cdot e_{k_1+1} \cdot Q$ is a simple path from s to t that matches $r_1T^*r_2$.

Otherwise, we have $|p| > 2k_1 + k_2 + 1$. We define $P = e_1 \cdots e_{k_1}$, $R = e_{k_1+2} \cdots e_{n-k-1}$, and $Q = e_{n-k+1} \cdots e_n$. We thus have

$$p = P \cdot e_{k_1+1} \cdot R \cdot e_{n-k} \cdot Q.$$

Since p matches $r_1T^*r_2$, we furthermore know that P matches r_1, R matches T^*, and Q matches $T^*T^{k_1}r_2$.[2] Let $v_{k_1} = \text{destination}(e_{k_1})$. Since $|V(Q)| = k + 1$, $V(P) \in P^{r_1}_{s,v_{k_1}}$, and $V(P) \cap V(Q) = \emptyset$, the definition of $\hat{P}^{r_1}_{s,v_{k_1}} \subseteq^{k+1}_{\text{rep}} P^{r_1}_{s,v_{k_1}}$ guarantees, similar as in the previous case, the existence of a path P' from s to v_{k_1} that matches r_1 with $V(P') \in \hat{P}^{r_1}_{s,v_{k_1}}$ and $V(P') \cap V(Q) = \emptyset$. Let $P' = e'_1 \cdots e'_{k_1}$. If $V(P')$ is disjoint from $V(R)$, the path

$$p' = P' \cdot e_{k_1+1} \cdot R \cdot e_{n-k} \cdot Q$$

is a simple path matching $r_1T^*r_2$, and we are done.

We show that P' must be disjoint from R. Towards a contradiction, assume that there exists an $i \in [k_1-1]$ such that $\text{destination}(e'_i) = \text{origin}(e_j)$ for some $j \in \{k_1+2, \ldots, n-k\}$.[3] We choose i minimal and build a new simple path $p' = e'_1 \cdots e'_i e_j \cdots e_n$. This path matches $A_1 \cdots A_i T^* T^{k_1} r_2$. And since r_1T^* is 0-bordered, we have $T \subseteq A_i$ for all $1 \leq i \leq k_1$, so the new path matches $r_1T^*r_2$. Finally, we note that p' does not contain the edge e_{k_1+1}, so p' is shorter than p, which contradicts the assumption that p was a shortest simple path from s to t that matches $r_1T^*r_2$. So P' must be disjoint from R. □

[2]The path Q does not necessarily match $T^{k_1}r_2$, since r_2 might contain words shorter than k_2.

[3]Since P and R are disjoint, we have $s, v_{k_1} \notin V(R)$.

Notice that we allow $T = \emptyset$ in Lemma 5.3.11. Since $L(\emptyset^*) = \{\varepsilon\}$, this means that the lemma also deals with the case where the expression is just $A_1 \cdots A_{k_1}$. From the proof of Lemma 5.3.11 we can also infer the following corollary, which states that shortest matching paths can also be found with this method. It will be useful when considering enumeration problems in Part III.

Corollary 5.3.12. *Let r_1T^* be a 0-bordered expression with $r_1 = A_1 \cdots A_{k_1}$ and let $L(r_2)$ be an arbitrary finite language with words up to length k_2. We define $k = k_1 + k_2$. Then, a directed multigraph $G = (V, E, \mathcal{E})$ has a simple path from s to t that matches $r_1T^*r_2$ if and only if there exists a node $v \in V$ and $X \in \hat{P}^{r_1}_{s,v} \subseteq^{k+1}_{rep} P^{r_1}_{s,v}$, such that G has a shortest simple path from s to t that matches $r_1T^*r_2$ and with the first $k_1 + 1$ nodes belonging to X.*

The following lemma states that Algorithm 2 is correct and runs in fixed parameter tractable time.

Lemma 5.3.13. *PSimPath($\mathcal{R}$) is in FPT for the class $\mathcal{R}$ of 0-bordered STEs of the form $r = A_1 \cdots A_kT^*$. More precisely, it is in time $2^{O(|r|)} \cdot |E||V|^2$.*

Proof. The problem can be solved using Algorithm 2. Its correctness follows directly from Lemma 5.3.11 with $r_2 = \varepsilon$. Using Lemma 5.3.9, we now show that the algorithm is indeed an FPT algorithm.

We obtain from Lemma 5.3.9 that line 2 takes $O\left(8^{k+o(k)}|E|\log|V| + |r||E|\right)$ time for each $v \in V$. Since we need to consider at most $|V| \cdot \binom{2(k+1)}{k+1} \cdot 2^{o(2(k+1))}$ sets X in line 3, the number of such sets we need to consider throughout the entire algorithm is at most $O(|V|4^{k+o(k)})$. Finally, line 6 can be checked by a reachability test (say, depth-first search) in time $O(|V| + |E|)$, so the overall running time of Algorithm 2 is bounded by

$$O\left(|V| \cdot (8^{k+o(k)}|E|\log|V| + |r||E|) + 4^{k+o(k)} \cdot (|V|^2 + |E||V|)\right),$$

which is clearly in FPT for the parameter k. □

We now extend the algorithm to 0-bordered STEs of the form $A_1 \cdots A_{k_1}T^*A'_{k_2} \cdots A'_1$. Since STEs allow bounded expressions on both sides, we need to do more than simply apply Algorithm 2. Instead, we will use a nesting thereof, which we present in Algorithm 3. The next Lemma shows the correctness and running time of Algorithm 3.

Lemma 5.3.14. *Let $\mathcal{R}$ be the class of 0-bordered STEs of the form $A_1 \cdots A_{k_1}T^*A'_{k_2} \cdots A'_1$. Then PSimPath($\mathcal{R}$) is in FPT. More precisely, it is solvable in time $2^{O(|r|)} \cdot |V|^3|E|$.*

Proof. We prove that Algorithm 3 solves the problem in the required time. Recall that

$$P^r_{s,v} := \{V(p) \mid \text{there is a simple path } p \text{ from } s \text{ to } v \text{ in } G \text{ that matches } r\}.$$

We first show correctness. Let $k = k_1 + k_2$. Obviously, $k \leq |r|$. Using Lemma 5.3.11 with $r_1 = A_1 \cdots A_{k_1}$ and $r_2 = A'_{k_2} \cdots A'_1$, it suffices to consider paths in which the first $k_1 + 1$

Algorithm 3: Algorithm for 0-bordered STEs

```
Input: A directed multigraph $G = (V, E, \mathcal{E})$, nodes $s, t$ in $G$, and 0-bordered
       regular expression $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$
Output: Does there exist a simple path from $s$ to $t$ matching $r$?
1  for all $v \in V$ do
2      Compute $\hat{P}^{r_1}_{s,v} \subseteq^{k_1+k_2+1}_{\text{rep}} P^{r_1}_{s,v}$ in $G$ with $r_1 = A_1 \cdots A_{k_1}$.
3      for all sets $X \in \hat{P}^{r_1}_{s,v}$ do
4          $V' \leftarrow (V \setminus X) \cup \{v\}$
5          $E' \leftarrow \{e \in E \mid \mathcal{E}(e) \in V' \times \Sigma \times V'\}$
6          for all $u \in V'$ do
7              Compute $\hat{P}^{r_2}_{u,t} \subseteq^{k_2+1}_{\text{rep}} P^{r_2}_{u,t}$ in $(V', E', \mathcal{E}|_{E'})$ with $r_2 = A'_{k_2} \cdots A'_1$.
8              for all sets $X' \in \hat{P}^{r_2}_{u,t}$ do
9                  $V'' \leftarrow (V' \setminus X') \cup \{u\}$
10                 $E'' \leftarrow \{e \in E \mid \mathcal{E}(e) \in V'' \times T \times V''\}$    ▷ $(V'', E'')$ has only
                   $T$-edges
11                 if there exists a path from $v$ to $u$ in $(V'', E'', \mathcal{E}|_{E''})$ then
12                     return "yes"
13 return "no"
```

nodes belong to a set $X \in \hat{P}^{r_1}_{s,v} \subseteq^{k+1}_{\text{rep}} P^{r_1}_{s,v}$ for some $v \in V$. Then we need to find the rest of the path, that is, a simple path from v to t that matches $T^* A'_{k_2} \cdots A'_1$ and that does not use nodes in $X \setminus \{v\}$.

We can apply Lemma 5.3.11 on the graph obtained from $(V', E', \mathcal{E}|_{E'})$ by reversing all edges and using the expression $A'_1 \cdots A'_{k_2} T^* \varepsilon$. Hence, if such a path exists in (V', E'), then there exists a node u such that its last $k_2 + 1$ nodes belong to a set $X' \in \hat{P}^{r_2}_{u,t} \subseteq^{k_2+1}_{\text{rep}} P^{r_2}_{u,t}$. It then remains to test if there is a path from v to u that matches T^* and avoids the nodes in $(X \cup X') \setminus \{u, v\}$, which is done in line 11. This concludes the correctness proof.

We next show that the algorithm is indeed in FPT. Lemma 5.3.9 allows us to compute, after the preprocessing phase which takes $O(|r||E|)$ time, $\hat{P}^{r_1}_{s,v}$ on line 2 in time $O(8^{k+o(k)}|E| \log |V|)$ and such that its size is at most $\binom{2k_1+k_2+2}{k_1+1} \cdot 2^{o(k_1+k_2)}$. Similarly, we can compute $\hat{P}^{r_2}_{u,t}$ on line 7 in time $O(8^{k+o(k)}|E| \log |V|)$ and such that its size is at most $\binom{2k_2+2}{k_2+1} \cdot 2^{o(k_2)}$.

This means that we need to consider $O(|V| \cdot 4^{k+o(k)})$ many sets in line 3. Computing $\hat{P}^{r_2}_{u,t}$ takes time $O\left(8^{k+1+o(k+1)}|E| \log |V|\right)$ for each $u \in V$, so we have $O(|V|^2 \cdot 4^{k+o(k)} \cdot 8^{k+o(k)}|E| \log |V|)$ time for this part and need to consider at most $O(|V|^2 \cdot 4^{k+o(k)} \cdot 4^{k+o(k)})$ many sets in line 8. Finally, the reachability test in line 11 is in $O(|V| + |E|)$, so in sum we obtain a running time of

$$O\Big(|r||E| + |V|^2 \cdot 4^{k+o(k)} \cdot \Big(8^{k+o(k)}|E| \log |V| + 4^{k+o(k)} \cdot (|V| + |E|)\Big)\Big).$$

□

The previous lemma only dealt with 0-bordered STEs of the form $A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$. The next lemma generalizes this to all 0-bordered STEs.

Lemma 5.3.15. *Let $\mathcal{R}$ be the class of 0-bordered STEs. Then* PSimPath($\mathcal{R}$) *is in FPT. More precisely, it is solvable in time $2^{O(|r|)} \cdot |V|^3|E|$.*

Proof. We prove the lemma by case distinction on the form of r. Recall that $r = B_{\text{pre}} T^* B_{\text{suff}}$. We differentiate between the forms of B_{pre} and B_{suff}. There are two possible forms, that is (1) $B_1? \cdots B_\ell?$ with $\ell \geq 0$ or (2) $B_1 \cdots B_\ell$ with $\ell \geq 1$. If B_{pre} and B_{suff} are of form (1), the language of r is downward closed. Therefore the entire problem reduces to a reachability problem on a product between G and an NFA for r. According to Lemma 5.3.1, this problem can be solved in time $O(|G||r|)$, since it is possible to compute an NFA of size $|r|$ for each STE r.

If B_{pre} and B_{suff} are both of form (2), the result follows from Lemma 5.3.14, which internally uses Algorithm 3, in time $2^{O(|r|)} \cdot |V|^3|E|$. We now explain how Algorithm 3 can be changed to work if B_{pre} is of form (2) and B_{suff} of form (1). Assume we have $r = A_1 \cdots A_{k_1} T^* A'_{k_2}? \cdots A'_1?$. Then we replace everything from line 6 to line 12 with a test for a simple path from v to t matching the downward closed language $T^* A'_{k_2}? \cdots A'_1?$. The correctness is again by Lemma 5.3.11. For the running time we observe that testing if there is a simple path matching $T^* A'_{k_2}? \cdots A'_1?$ is in time $O(|G||r|)$ by Lemma 5.3.1, since the language is downward closed. The running time in this case is therefore

$$O\left(|r||E| + |V| \cdot \left(8^{|r|+o(|r|)}|E| \log |V| + 4^{|r|+o(|r|)} \cdot |G||r|\right)\right).$$

The case $r = A_1? \cdots A_{k_1}? T^* A'_{k_2} \cdots A'_1$ is symmetric. To see this, notice that it is equivalent to deciding if there is a simple path from t to s that matches the regular expression $A'_1 \cdots A'_{k_2} T^* A_{k_1}? \cdots A_1?$ in the multigraph G with all edges reversed. □

Lemma 5.3.16. *Let $c \in \mathbb{N}$ be a constant and let $\mathcal{R}$ be the class of STEs with cut border at most c. Then* SimPath($\mathcal{R}$) *is in FPT. More precisely, it is in time $2^{O(|r|)} \cdot |V|^{c+3}|E|$.*

Proof. Let $r \in \mathcal{R}$ and let c_1 and c_2 be the left and right cut border of r, respectively. Hence, $r = A_1 \ldots A_{c_1} r' A'_{c_2} \cdots A'_1$. (If $c_i = 0$, then the respective part of r is simply missing.) Let $G = (V, E, \mathcal{E})$ a directed multigraph. We can compute, for all $u, v \in V$, all paths p_1 from s to u matching $A_1 \cdots A_{c_1}$ and all paths p_2 from v to t matching $A'_{c_2} \cdots A'_1$ in time $O(|V|^c)$.[4] We then do a loop over all pairs (p_1, p_2) of such paths that are node-disjoint. For the remainder of the proof, fix such a pair (p_1, p_2). We delete in G all nodes in $(V(p_1) \setminus \{u\}) \cup (V(p_2) \setminus \{v\})$. In the remaining multigraph, we search a path from u to v that matches the rest of the regular expression. The rest r' can have one of the following forms.

[4]For the purpose of the proof, it suffices to compute the paths without the edge labels here. For deciding whether there exists a simple path, it suffices to know that there exist node-disjoint simple paths matching $A_1 \cdots A_{c_1}$ and $A'_{c_2} \cdots A'_1$ and which nodes they use. We dropped the exact labels to have $O(|V|^c)$ complexity.

- $r' = A_{c_1+1} \cdots A_{k_1} T^* A'_{k_2} \cdots A'_{c_2+1}$,
- $r' = A_1? \cdots A_{k_1}?T^* A'_{k_2} \cdots A'_{c_2+1}$,
- $r' = A_{c_1+1} \cdots A_{k_1} T^* A'_{k_2}? \cdots A'_1?$, or
- $r' = A_1? \cdots A_{k_1}?T^* A'_{k_2}? \cdots A'_1?$.

These are the only possibilities and each of them is 0-bordered. Thus, we can use Lemma 5.3.15, which allows us to solve PSimPath(r') in time $2^{O(|r'|)} \cdot |V|^3|E|$. Since $|r'| \leq |r|$, this shows that PSimPath($\mathcal{R}$) is in FPT. So we need $2^{O(|r'|)} \cdot |V|^3|E|$ time for each set of nodes of size c_1+c_2, and therefore have an overall time of $2^{O(|r|)} \cdot |V|^{c+3}|E|$. □

5.4 Main Lower Bound for PSimPath

We prove part (b) of Theorem 5.2.2 by a reduction from variants of the TwoDisjointPaths problem. A *two-colored graph* is a directed graph in which every edge is labeled a or b. We consider the following parameterized problems:

- PTwoDisjointPaths: Given a directed graph G, nodes s_1, t_1, s_2, t_2, and parameter $k \in \mathbb{N}$, are there simple paths p_1 from s_1 to t_1 and p_2 from s_2 to t_2 such that p_1 and p_2 are node-disjoint and p_1 has length k?
- PTwoColorDisjointPaths: Given a two-colored directed graph G, nodes s_a, t_a, s_b, t_b, and parameter $k \in \mathbb{N}$, is there a simple a-path p_a from s_a to t_a and a simple b-path p_b from s_b to t_b such that p_a and p_b are node-disjoint and p_a has length k?

It is well-known that TwoDisjointPaths, the non-parameterized version of PTwoDisjoint-Paths, is NP-complete [98].

We will prove that both PTwoColorDisjointPaths and PTwoDisjointPaths are W[1]-hard. The latter result is stronger, but we start by proving W[1]-hardness for PTwoColorDisjoint-Paths, because it makes the proof for PTwoDisjointPaths, which relies on it, easier to understand.

5.4.1 Parameterized Two Color Disjoint Paths

In this section, we prove the following theorem.

Theorem 5.4.1. *PTwoColorDisjointPaths is W[1]-hard.*

To prove the theorem, we use an adaptation of a proof of Slivkins [188, Theorem 2.1], who proved that Edge-Disjoint-Paths parameterized with the number of disjoint paths is W[1]-hard in directed acyclic graphs (DAG). Furthermore, we use the idea of *control nodes* by Grohe and Grüber [110, Lemma 16], who showed that Slivkins' construction can be extended to show that finding disjoint cycles in a directed graph is W[1]-hard when parameterized by the number of cycles.

The proof is by reduction from the parameterized version of Clique defined as follows:

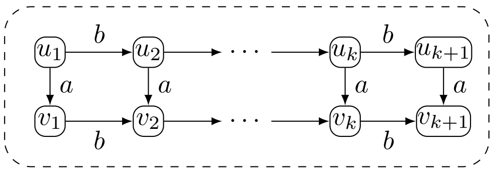


Figure 5.5: Internal structure of each of the gadgets $G_{i,j}$.

- PClique: Given an undirected graph G and a parameter $k \in \mathbb{N}$, is there a clique of size k in G?

It is well-known that PClique is W[1]-complete [87].

Construction 5.4.2. (Construction of G_{col} and k_{col}.) Given an input instance (G, k) of PClique, we construct a directed two-colored graph G_{col}, nodes s_a, t_a, s_b, t_b, and parameter k_{col} such that $(G, k) \in$ PClique if and only if $(G_{\text{col}}, s_a, t_a, s_b, t_b, k_{\text{col}}) \in$ PTwoColorDisjoint-Paths. Let n be the number of nodes of G. The graph G_{col} contains kn gadgets $G_{i,j}$ with $i \in [k]$ and $j \in [n]$, each consisting of $2(k+1)$ nodes. Gadgets will be ordered in k rows, where row i has the gadgets $G_{i,1}, \ldots, G_{i,n}$. Furthermore, G_{col} contains $k+1$ additional nodes $r_1, \ldots, r_{k+1}$ that link the rows together, and $k + 1 + k(k-1)/2$ *control nodes* $c_1, \ldots c_{k+1}$ and c_{i_1,i_2} with $1 \le i_1 < i_2 \le k$ that will limit the number of disjoint paths from row $i-1$ to row i or from row i_1 to i_2, respectively. (The edge cases, c_1 and c_{k+1}, do not link rows together but just serve as start and end node, respectively.) We define $s_a = c_1$, $t_a = c_{k+1}$, $s_b = r_1$, and $t_b = r_{k+1}$. We will now explain how the nodes are connected in G_{col}. We will denote by $u \xrightarrow{a} v$ that there is an a-edge from u to v (similar for b-edges). Each gadget $G_{i,j}$ contains a disjoint copy of $2(k+1)$ nodes which we call $u_1, u_2, \ldots, u_{k+1}$ and $v_1, v_2, \ldots, v_{k+1}$. To simplify notation, we sometimes give these nodes the same name (for example in Figures 5.6, 5.7, and 5.8), even though they are different. One such gadget is depicted in Figure 5.5. To avoid ambiguity, we may also refer to node u_ℓ in gadget $G_{i,j}$ by $G_{i,j}[u_\ell]$. Each gadget contains edges $u_\ell \xrightarrow{a} v_\ell$ (for every $\ell \in [k+1]$) and $u_\ell \xrightarrow{b} u_{\ell+1}$ and $v_\ell \xrightarrow{b} v_{\ell+1}$ (for every $\ell \in [k]$).

We now explain how the gadgets $G_{i,j}$ are connected within the same row, see Figure 5.6. In each row $i \in [k]$, node r_i has two outgoing edges $r_i \xrightarrow{b} G_{i,1}[u_1]$ and $r_i \xrightarrow{b} G_{i,2}[v_1]$. We also have two incoming edges for r_{i+1}, namely $G_{i,n-1}[u_{k+1}] \xrightarrow{b} r_{i+1}$ and $G_{i,n}[v_{k+1}] \xrightarrow{b} r_{i+1}$. Furthermore, we have the edges $G_{i,j}[u_{k+1}] \xrightarrow{b} G_{i,j+1}[u_1]$ and $G_{i,j}[v_{k+1}] \xrightarrow{b} G_{i,j+1}[v_1]$ for every $j \in [n-1]$. We also add edges $G_{i,j}[u_{k+1}] \xrightarrow{b} G_{i,j+2}[v_1]$ for every $j[n-2]$. The latter edges ensure that every b-labeled path from r_i to r_{i+1} "skips" exactly one gadget $G_{i,j}$ for some $j \in [n]$.

We now explain how the gadgets $G_{i,j}$ are connected in different rows via the control nodes c_i and c_{i_1,i_2} (Figure 5.7). We first consider the edges from row i to $i+1$. In each row $i = 1, \ldots, k-1$, and every $j = 1, \ldots, n$, we add the edges $G_{i,j}[v_{k+1}] \xrightarrow{a} c_{i+1}$ and $c_{i+1} \xrightarrow{a} G_{i+1,j}[u_{i+2}]$. Furthermore, we add the edges $c_1 \xrightarrow{a} G_{1,j}[u_2]$ and $G_{k,j}[v_{k+1}] \xrightarrow{a} c_{k+1}$.

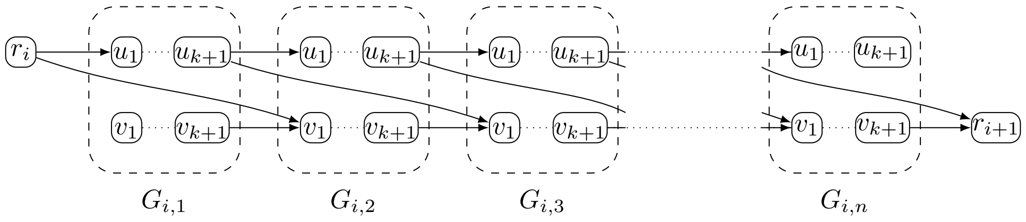


Figure 5.6: The b-edges in row i. The internal structure of the $G_{i,j}$ is as in Figure 5.5.

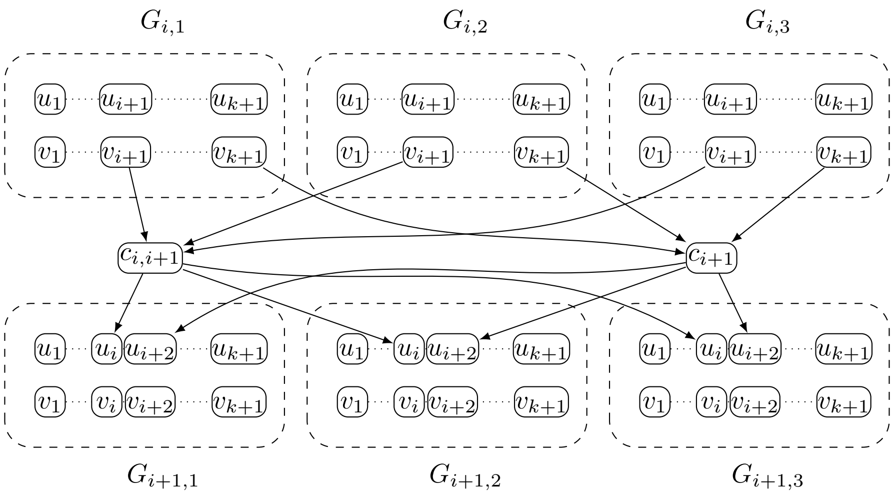


Figure 5.7: The a-edges from row i to row $i + 1$. (We assume $n = 3$ in the picture).

We connect two rows i_1, i_2, with $1 \leq i_1 < i_2 \leq k$, by adding the edges $G_{i_1,j}[v_{i_2}] \xrightarrow{a} c_{i_1,i_2}$, and $c_{i_1,i_2} \xrightarrow{a} G_{i_2,j}[u_{i_1}]$ for all $j = 1, \ldots, n$.

The edges of the original graph G are modeled in G_{col} by adding the edge $G_{i_2,x}[v_{i_1}] \xrightarrow{a} G_{i_1,y}[u_{i_2+1}]$ if and only if $1 \leq i_1 < i_2 \leq k$, $x \neq y$, and $(x, y) \in E$. This is illustrated in Figure 5.8.

Finally, we define $k_{\text{col}} = k(k-1)/2 \cdot 5 + 3k$. □

We denote by G^a_{col} the subgraph of G_{col} from Construction 5.4.2 that contains only the a-edges. We now prove a lemma that summarizes useful properties of G^a_{col}.

Lemma 5.4.3. *The directed graph G^a_{col} has the following properties:*

(a) G^a_{col} is a DAG. Moreover, there is a strict total order $<_c$ on all control nodes $\mathcal{C}$ such that, for every path from a node $v \in \mathcal{C}$ to another node $v' \in \mathcal{C}$ where no intermediate vertex is in $\mathcal{C}$, node v' is the successor of v in $<_c$. The smallest and largest nodes in $<_c$ are c_1 to c_{k+1}, respectively.

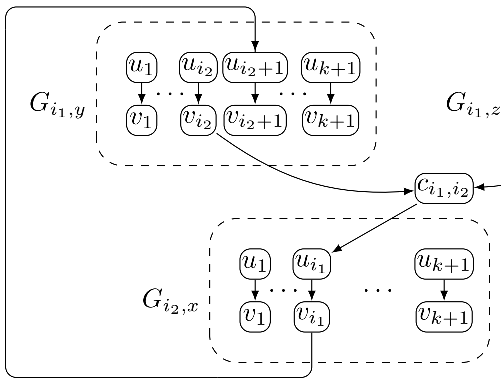


Figure 5.8: The a-edges in the gadgets and between gadgets $G_{i_1,y}$, $G_{i_1,z}$ and $G_{i_2,x}$, with $i_1 < i_2 - 1$, under the assumption that $(x, y) \in E$ and $(x, z) \notin E$.

(b) Each path in G^a_{col} from c_1 to c_{k+1} visits all control nodes, that is, it contains all c_i and c_{i_1,i_2}, with $i \in [k+1]$ and $1 \le i_1 < i_2 \le k$. Furthermore, it visits the control nodes in the order $<_c$.

(c) Each path in G^a_{col} has length at most k_{col}. Its length is exactly k_{col} if and only if it is from c_1 to c_{k+1}.

(d) Each path in G^a_{col} of length k_{col} has at least one edge $u_\ell \xrightarrow{a} v_\ell$ in every row of G^a_{col}.

Proof. First observe that G^a_{col} contains a fixed part that depends only on n and k, plus a set of edges that represent edges in G, that is, edges that are present in G_{col} if and only if there exists a corresponding edge in G. Therefore, every possible graph G_{col} that the reduction produces is a subgraph of the case where G is a complete graph (that is, if G has n nodes, it is the n-clique). Let G^a_{clique} denote the graph G^a_{col} in the case where G is the n-clique.

We first prove part (a). We show that, if G^a_{clique} has a cycle, then this cycle must contain a control node. Indeed, within the same row, the graph G^a_{clique} only has the edges from u_i to v_i in all the gadgets. So, there cannot be a cycle that only contains nodes from a single row. Therefore, the cycle must contain a path from some node in a row i_1 to a node in row i_2, for $i_1 < i_2$. Since every path in G^a_{clique} from row i_1 to i_2 with $i_1 < i_2$ contains at least one control node by construction, we have that every cycle in G^a_{clique} must contain a control node. It therefore remains to show that G^a_{clique} contains no cycle that uses a control node. To this end, observe that the relation $\prec$ where $n_1 \prec n_2$ if and only if $n_1 \neq n_2$ and n_2 is reachable from n_1 is a strict total order

$$c_1 \prec c_{1,2} \prec c_{1,3} \prec \ldots \prec c_{1,k} \prec c_2 \prec c_{2,3} \prec \ldots \prec c_{k-2,k} \prec c_{k-1,k} \prec c_k \prec c_{k+1} \qquad (\dagger)$$

on the control nodes $\mathcal{C}$. That is, the order is such that control nodes are reachable in G^a_{clique} from all "smaller" control nodes and none of the "larger" control nodes. Notice that $\prec$ satisfies the requirements for $<_c$. Part (b) follows from (a). By (a), the smallest and largest nodes in $<_c$ are c_1 and c_{k+1}, respectively. Assume that p is a path from c_1 to c_{k+1}. Again by (a), p must visit every control node, in the order $<_c$.

We now prove part (c). First we prove that, between two consecutive[5] control nodes in G^a_{clique}, each path has a fixed length that depends only on the kind of control nodes. Then, since G^a_{clique} is a DAG by part (a), we can simply concatenate paths to obtain the length of paths from c_1 to c_{k+1}, showing (c). In this proof, when we consider a path that visits nodes in row i in G^a_{clique}, then by construction of G^a_{clique}, the length of this path is independent of the gadget $G_{i,j}$ that the path visits. That is, the path's length is the same for every $j \in [n]$. To simplify notation, we therefore omit the j in $G_{i,j}[u]$ and write $G_i[u]$ instead.

We first consider the length of paths between consecutive control nodes in the ordering (†). Therefore, fix two such consecutive control nodes n_1 and n_2. We make a case distinction:

- $n_1 = c_i$ and $n_2 = c_{i,i+1}$: Each path from c_i to $c_{i,i+1}$ uses exactly the nodes $c_i, G_i[u_{i+1}], G_i[v_{i+1}], c_{i,i+1}$ in that order and therefore has length 3.

- $n_1 = c_{i,j}$ and $n_2 = c_{i,j+1}$: Each path from $c_{i,j}$ to $c_{i,j+1}$ with $1 \leq i < j \leq k-1$ uses exactly the nodes $c_{i,j}, G_j[u_i], G_j[v_i], G_i[u_{j+1}], G_i[v_{j+1}], c_{i,j+1}$ in that order and therefore has length 5.

- $n_1 = c_{i,k}$ and $n_2 = c_{i+1}$: Each path from $c_{i,k}$ to c_{i+1} uses exactly the nodes $c_{i,k}, G_k[u_i], G_k[v_i], G_i[u_{k+1}], G_i[v_{k+1}], c_{i+1}$ in that order and therefore has length 5.

- $n_1 = c_k$ and $n_2 = c_{k+1}$: Each path from c_k to c_{k+1} uses exactly the nodes $c_k, G_k[u_{k+1}], G_k[v_{k+1}], c_{k+1}$ in that order and therefore has length 3.

Since $\prec$ is a strict total order, this means that each path from c_1 to c_{k+1} in G^a_{clique} has the same length. We show that this length is exactly $k(k-1)/2 \cdot 5 + 3k = k_{\text{col}}$. The paths c_i to $c_{i,i+1}$ (for all $i \in [k-1]$) and c_k to c_{k+1} sum up to length $3k$. For a fixed i we have $5 \cdot (k-i-1)$ paths from $c_{i,i+1}$ to $c_{i,k}$, which sum up to length $5(k(k-1)/2) - 5k + 5$ for $i \in [k-2]$. Finally, we need to consider the paths from $c_{i,k}$ to c_{i+1}, which, for $i \in [k-1]$, sum up to length $5k-5$. This shows that each path G^a_{clique} from c_1 to c_{k+1} has length exactly k_{col}.

Since G^a_{clique} is a DAG and every node in G^a_{clique} is reachable from c_1, and c_{k+1} is reachable from all nodes and does not have outgoing edges in G^a_{clique}, the longest paths in G^a_{clique} are from c_1 to c_{k+1}. This shows (c).

Due to (b) and (c) each path of length k_{col} in G^a_{clique} contains c_i for $i \in [k+1]$. Since each path from c_i to the next control node contains $(G_{i,j}[u_{i+1}], G_{i,j}[v_{i+1}])$, for a $j \in [n]$ we also have (d). $\square$

We are now ready to prove Theorem 5.4.1.

[5]Control nodes x and y such that $x <_c y$ and there are no other control nodes in between.

Proof of Theorem 5.4.1. We prove that (G, k) is a yes-instance of PClique if and only if $(G_{\text{col}}, s_a, t_a, s_b, t_b, k_{\text{col}})$ is a yes-instance of PTwoColorDisjointPaths. Let us first assume that the undirected graph G has a k-clique with nodes $\{n_1, \ldots, n_k\}$. Then an a-path can go from c_1 to c_{k+1} using only the gadgets G_{i,n_i} with $i \in [k]$. The reason is that, since $(n_{i_1}, n_{i_2}) \in E$, the edges $G_{i_2,n_{i_2}}[v_{i_1}] \xrightarrow{a} G_{i_1,n_{i_1}}[u_{i_2+1}]$ exist for all $i_1 < i_2$. Due to Lemma 5.4.3(c), this path has exactly k_{col} edges. The b-path, on the other hand, can go from r_1 to r_{k+1} and skip exactly G_{i,n_i} for all $i \in [k]$ (using the diagonal edges in Figure 5.6). Since it skips these G_{i,n_i}, it is node-disjoint from the a-path and therefore we have a solution for PTwoColorDisjointPaths.

For the other direction let us assume that there exists a simple a-path p_a from c_1 to c_{k+1} and a simple b-path p_b from r_1 to r_{k+1} in G_{col} such that p_a and p_b are node-disjoint and p_a has length k_{col}. We show that G has a k-clique. Since every b-path from r_1 to r_{k+1} goes through each row, that is, from r_i to r_{i+1} for all $i \in [k]$, this is also the case for p_b. By construction, p_b must also skip exactly one gadget in each row, using the diagonal edges in Figure 5.6. Indeed, this is the only way to move from r_i to r_{i+1} using only b-edges. Furthermore, for each gadget $G_{i,j}$ that p_b visits, it must be the case that it either visits all nodes $u_1, \ldots, u_{k+1}$ or all nodes $v_1, \ldots, v_{k+1}$. (This is immediate from Figure 5.5, showing all internal edges of a gadget.) Therefore, since p_a and p_b are node-disjoint, the path p_a cannot visit any gadget $G_{i,j}$ already visited by p_b. Therefore, p_a, which goes from c_1 to c_{k+1}, can only do so through the k skipped gadgets, call them G_{i,n_i} for $i \in [k]$. Recall that the edges $G_{i_2,n_{i_2}}[v_{i_1}] \xrightarrow{a} G_{i_1,n_{i_1}}[u_{i_2+1}]$ with $i_1 < i_2$ only exist if $(n_{i_1}, n_{i_2}) \in E$. As these edges are necessary for the existence of the a-path from c_1 to c_{k+1} that uses only the skipped gadgets, all nodes n_i must be pairwise adjacent in G. That is, they form a clique of size k in G. □

Parameterized Two Disjoint Paths

The two colors in the proof of Theorem 5.4.1 play a central role: since the a-path cannot use any b-edges and vice versa, we have much control over where the two paths can be. We now show that the construction in Theorem 5.4.1 can be strengthened so that we do not need the two colors. To this end, we replace the b-edges by long paths to ensure that all paths from s_a to t_a that have length at most k_{col} cannot use b-edges.

Construction 5.4.4. We construct the directed graph G_{node} from G_{col} by replacing each b-edge with a b-path of length k_{col}. (Even though PTwoDisjointPaths does not care about a-edges or b-edges, we keep them to simplify the reasoning in the remainder of the proof.) We define $s_1 = s_a$, $t_1 = t_a$, $s_2 = s_b$, and $t_2 = t_b$. (Notice that G_{col} has $O(k^2 n)$ nodes while G_{node} has $O(k^2 n \cdot k_{\text{col}})$ nodes.) □

Lemma 5.4.5. *In G_{node}, we have that*

(a) every path from s_1 to t_1 has length at least k_{col} and

(b) every path from s_1 to t_1 has length exactly k_{col} if and only if it is an a-path.

(c) Furthermore, all properties of G^a_{col} from Lemma 5.4.3 also hold for G^a_{node}.

Proof. For part (a) we have two cases. If a path from s_1 to t_1 is an a-path, the result is immediate from Lemma 5.4.3(c). If it uses at least one b-edge, then it uses at least k_{col} b-edges by construction. Thus, the path will have length at least k_{col}.

For part (b), if a path from s_1 to t_1 has length exactly k_{col}, it uses at least one a-edge since t_1 only has incoming a-edges. If it used at least one b-edge, it would therefore use at least $k_{\text{col}} + 1$ edges which contradicts that the length is k_{col}. The converse direction is immediate from Lemma 5.4.3(c). The last point is obvious since G^a_{col} and G^a_{node} are the same. □

Lemma 5.4.6. *If $(G_{node}, s_1, t_1, s_2, t_2, k_{col}) \in$ PTwoDisjointPaths, then each solution p_1, p_2 is such that p_1 is an a-labeled path and p_2 a b-labeled path.*

Proof. It follows from Lemma 5.4.5 that p_1 can only use a-edges. We now show that the path p_2 from s_2 to t_2 can only use b-edges, that is, we show that it cannot use a-edges. There are three types of a-edges in G_{node}: (i) the ones from and to control nodes, (ii) "upward" edges that connect row i_2 to row i_1 with $i_1 < i_2$, and (iii) edges from u_ℓ to v_ℓ in one gadget.

Notice that, by construction, p_2 must visit nodes in row 1 and later also nodes in row k. To do so, p_2 cannot use edges from or to control nodes (type (i)), since, due to Lemma 5.4.3(b), p_1 already visits all control nodes. So p_2 cannot go from row i to a row j with $i < j$ via a-edges. This means that, if $i < j$, then p_2 can only go from row i to row j through r_{i+1} (and through nodes in row $i + 1$), since every remaining path from row i to a larger row goes through r_{i+1}. So, in order to go from row 1 to row k, path p_2 needs to visit all nodes $r_2, \dots, r_k$, in that order. This means that it is also impossible for p_2 to use edges of type (ii). Indeed, if p_2 used an edge from row j to row i with $j > i$, then it would need to visit r_{i+1} a second time to arrive back in row j. Finally, if p_2 used an edge of type (iii) in row i, then, by construction, it would have to visit every gadget in this row. But since p_1 already uses at least one edge from u_ℓ to v_ℓ in each row, see Lemma 5.4.3(d), this means that p_2 cannot be node-disjoint with p_1. This completes the proof. □

Theorem 5.4.7. *PTwoDisjointPaths is W[1]-hard.*

Proof. We reduce from PTwoColorDisjointPaths, which is W[1]-hard due to Theorem 5.4.1.

We show that $(G_{\text{col}}, s_a, t_a, s_b, t_b, k_{\text{col}}) \in$ PTwoColorDisjointPaths if and only if $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}}) \in$ PTwoDisjointPaths. If $(G_{\text{col}}, s_a, t_a, s_b, t_b, k_{\text{col}}) \in$ PTwoColorDisjointPaths, then we can use the corresponding paths in G_{node} (where we follow the longer b-paths in G_{node} instead of the b-edges in G_{col}).

Conversely, if $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}}) \in$ PTwoDisjointPaths, it follows from Lemma 5.4.6 that the paths p_1 and p_2 correspond to paths p_a and p_b that are solutions for $(G_{\text{col}}, s_a, t_a, s_b, t_b, k_{\text{col}}) \in$ PTwoColorDisjointPaths. □

Reduction to PSimPath

We are now ready to proof the hardness side of Theorem 5.2.2, that is, Theorem 5.2.2(b).

Lemma 5.4.8. *Let $\mathcal{R}$ be a class of STEs that can be sampled. If $\mathcal{R}$ is not cuttable, then the problem $\mathsf{PSimPath}(\mathcal{R})$ is W[1]-hard.*

Proof. Let $\mathcal{R}$ be an arbitrary but fixed class of STEs that is not cuttable and that can be sampled. We show that $\mathsf{PSimPath}(\mathcal{R})$ is W[1]-hard by giving an FPT reduction from $\mathsf{PTwoDisjointPaths}$ restricted to instances of the form $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}})$ from Construction 5.4.4. The problem $\mathsf{PTwoDisjointPaths}$ is W[1]-hard due to Theorem 5.4.7.

Consider an input $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}})$ of $\mathsf{PTwoDisjointPaths}$. We construct an input $(G_{\text{lab}}, s, t, r)$ for $\mathsf{PSimPath}(\mathcal{R})$ such that $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}}) \in \mathsf{PTwoDisjointPaths}$ if and only if $(G_{\text{lab}}, s, t, r) \in \mathsf{PSimPath}(\mathcal{R})$.

Since $\mathcal{R}$ is not cuttable and can be sampled, a k'-bordered expression $r \in \mathcal{R}$ for some $k' \geq 2k_{\text{col}} + 1$ can be computed within time $f(k_{\text{col}})$, for some computable function f. Since r can be computed in time $f(k_{\text{col}})$, we know that $|r| \leq f(k_{\text{col}})$. Let k_{lab} be the maximum of the left and right cut border of r. Since k' is the sum of the left and right cut borders, $k_{\text{lab}} \geq k_{\text{col}} + 1$. Here we only consider the case that the left cut border is k_{lab}, that is, $T \not\subseteq A_{k_{\text{lab}}}$, the other case is symmetric. We therefore know that r is of the form

$$r = A_1 \cdots A_{k_{\text{col}}} \cdots A_{k_{\text{lab}}} \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$$

or

$$r = A_1 \cdots A_{k_{\text{col}}} \cdots A_{k_{\text{lab}}} \cdots A_{k_1} T^* A'_{k_2}? \cdots A'_1? \,.$$

We now construct (G_{lab}, s, t). Fix three words w_1, w_2, and w_3 such that

- $w_1 \in L(A_1 \cdots A_{k_{\text{col}}})$,
- $w_2 \in L(A_{k_{\text{col}}+1} \cdots A_{k_{\text{lab}}} \cdots A_{k_1})$, and
- $w_3 \in L(A'_{k_2} \cdots A'_1)$.[6]

Notice that such words indeed exist. For the construction of G_{lab}, we start with the graph G_{node}. The main idea is to have at most one edge with a label in $A_{k_{\text{lab}}}$ that is reachable from s by a path of length $k_{\text{lab}} - 1$. More formally, fix an $x \in (T \setminus A_{k_{\text{lab}}})$, which must exist due to choice of k_{lab}.

- We replace each b-edge in G_{node} with an x-path of length k_{lab} (using $k_{\text{lab}} - 1$ new nodes for each replacement). We need to do this, because k_{lab} is potentially much larger than k_{col}.
- We change the labels of the a-edges in G_{node} such that each path from s_1 to t_1 is labeled w_1. Notice that the label for each such edge is well-defined. Indeed, by Lemma 5.4.3(c) we have that each a-path from s_1 to t_1 has length exactly k_{col}. If there were an edge e on an a-path from s_1 to t_1 that is reachable from s_1 through n_1 edges and also through n_2 edges, with $n_1 \neq n_2$, then, since t_1 is reachable from e, it means that there would be paths of different lengths from s_1 to t_1.

[6]We use $w_3 \in L(A'_{k_2} \cdots A'_1)$ in case that r ends with $A'_{k_2} \cdots A'_1$ but also if it ends with $A'_{k_2}? \cdots A'_1?$.

- We add a path labeled w_2 from t_1 to s_2. We refer to this path as *the w_2-labeled path* in the remainder of the proof.
- We add a path labeled w_3 from t_2 to a new node t, to which we will refer as the *w_3-labeled path* in the remainder of the proof.

The resulting tuple $(G_{\text{lab}}, s_1, t, r)$ serves as input for $\mathsf{PSimPath}(\mathcal{R})$. This concludes the construction.

We now show that the reduction is correct. Therefore, we show that $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}}) \in \mathsf{PTwoDisjointPaths}$ if and only if $(G_{\text{lab}}, s_1, t, r) \in \mathsf{PSimPath}(\mathcal{R})$. If $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}}) \in \mathsf{PTwoDisjointPaths}$ with solution p_1 and p_2, then there exists a (unique) simple path from s_1 to t in G_{lab} that contains the nodes $V(p_1) \cup V(p_2)$ and matches r.

Conversely, if $(G_{\text{lab}}, s_1, t, r) \in \mathsf{PSimPath}(\mathcal{R})$, then there exists a simple path p from s_1 to t in G_{lab} that matches r. We will now prove the following:

(i) Consider the graph G^a_{node}, obtained from G_{node} by deleting all b-edges and nodes that have no adjacent a-edges. The nodes of $p[0, k_{\text{col}}]$ form a simple path from s_1 to t_1 in G^a_{node}.

(ii) The path $p[0, k_1]$ ends in s_2 and is labeled $w_1 w_2$.

(iii) The path p is labeled $w_1 w_2 w' w_3$ with $w' \in L(T^*)$. Its suffix of length $|w_3|$ starts in t_2 and ends in t.

(iv) The subpath of p from s_2 to t_2 is an x-path.

We prove (i). By definition of r, the edge $p[k_{\text{lab}} - 1, k_{\text{lab}}]$ is labeled by some symbol in $A_{k_{\text{lab}}}$. Therefore, this symbol cannot be x. By construction of G_{lab}, this edge is either an edge that was labeled a in G_{node}, an edge on the w_2-labeled path, or an edge on the w_3-labeled path (since all other edges are labeled x).

The w_3-labeled path is not reachable from s_1 with a path of length smaller than k_{lab}, so this cannot be the case. Furthermore, the w_2-labeled path starts in t_1 and is therefore only reachable with a path of length at least k_{col} (see Lemma 5.4.5), so we can also exclude that. Therefore, the first $k_{\text{col}} + 1$ nodes must form an a-path in G_{node}. From Lemma 5.4.3(c), we know that each path in G^a_{node} of length k_{col} goes from s_1 to t_1 which implies (i). Since all nodes (except s_2) that belong to the w_2-labeled path of length $k_1 - k_{\text{col}}$ have only one outgoing edge, we have that $p[0, k_1]$ ends in s_2 and must match $w_1 w_2$. This shows (ii).

Since p matches $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ or $r = A_1 \cdots A_{k_1} T^* A'_{k_2}? \cdots A'_1?$, and since each word in $A_1 \cdots A_{k_1}$ has length k_1, it follows that $\text{lab}(p) = w_1 w_2 w'$ with $w' \in L(T^* A'_{k_2} \cdots A'_1) \cup L(T^* A'_{k_2}? \cdots A'_1?)$.

By construction of G_{lab}, the w_3-labeled path is the unique path of length $|w_3|$ leading to t. Therefore, each path from s_1 to t in G_{lab} must end with the w_3-labeled path which is from t_2 to t. Since $w_3 \in L(A'_{k_2} \cdots A'_1)$ and $|w_3|$ is the length of every word in $L(A'_{k_2} \cdots A'_1)$, we have that $\text{lab}(p) = w_1 w_2 w' w_3$ where $w' \in L(T^*)$. So we have (iii). Let p' be the part of p labeled w'. It follows from (ii) and (iii) that p' is a path from s_2 to t_2.

Since it must be node-disjoint from $p[0, k_{\text{col}}]$, which is entirely in G^a_{node}, it follows from Lemma 5.4.6 that p' cannot use edges that correspond to ones in G^a_{node}.

Therefore, p' consists only of edges labeled x. This shows that G_{node} and k_{col} are in PTwoDisjointPaths, because $p[0, k_{\text{col}}]$ corresponds to a path p_1 and p' to p_2, which are solutions to PTwoDisjointPaths.

Finally, we note that the construction of $(G_{\text{lab}}, s_1, t, r)$ can indeed be done in FPT since the expression $r \in \mathcal{R}$ can be determined in time $f(k_{\text{col}})$ for a computable function f, and all changes we made to the graph are in time $h(k_{\text{col}}) \cdot |G_{\text{node}}|$, for a computable function h, which is FPT. Indeed, we only relabeled all edges, replaced each edge at most once with k_{lab} new edges and added other paths of length at most $|r|$. Since $|r| \leq f(k_{\text{col}})$, we indeed have an fpt-reduction. □

5.5 Connection Between Simple Paths and Trails

LaPaugh and Rivest [141, Lemma 1 and Lemma 2] and Perl and Shiloach [174, Theorem 2.1 and Theorem 2.2] showed that there is a strong correspondence between trail and simple path problems that we will use extensively and therefore revisit here. Since the statements of the results do not precisely capture what we need, we have to be a bit more precise.

The Split Graph The following construction is from LaPaugh and Rivest [141, Proof of Lemma 1]. Let $(G, s_1, t_1, \ldots, s_k, t_k)$ be a directed multigraph G together with nodes $s_1, t_1, \ldots, s_k, t_k$. We define $\text{split}(G, s_1, t_1, \ldots, s_k, t_k)$ as the tuple $(G', s'_1, t'_1, \ldots, s'_k, t'_k)$ obtained as follows. The directed multigraph G' is obtained from G by replacing each node v by two nodes v^{in} and v^{out}. A directed edge is added from v^{in} to v^{out}. All incoming edges of v become incoming edges of v^{in} and all outgoing edges of v become outgoing edges of v^{out}. For every s_i and t_i, we define $s'_i = s^{\text{in}}_i$ and $t'_i = t^{\text{out}}_i$. For a path p in G, we denote by $\text{split}(p)$ the path in G' that is obtained from p by replacing for each edge e the node $\text{origin}(e)$ by $\text{origin}(e)^{\text{out}}$, $\text{destination}(e)$ by $\text{destination}(e)^{\text{in}}$, and adding edges of the form $(u^{\text{in}}, u^{\text{out}})$ to the path. We give an example. If p can be written as $(u_1, u_2) \cdots (u_{n-1}, u_n)$, then $\text{split}(p)$ can be written as

$$(u^{\text{in}}_1, u^{\text{out}}_1)(u^{\text{out}}_1, u^{\text{in}}_2) \cdots (u^{\text{out}}_{n-1}, u^{\text{in}}_n)(u^{\text{in}}_n, u^{\text{out}}_n).$$

The following Lemma is immediate from LaPaugh and Rivest's construction.

Lemma 5.5.1. *Let $(G', s'_1, t'_1, \ldots, s'_k, t'_k) = \text{split}(G, s_1, t_1, \ldots, s_k, t_k)$. Then the following hold:*

(1) For every $i \in [k]$, the path p is a simple path from s_i to t_i in G if and only if $\text{split}(p)$ is a trail from s'_i to t'_i in G'.

(2) For every $i \in [k]$, the number of simple paths from s_i to t_i in G equals the number of trails from s'_i to t'_i in G'.

(3) There exist pairwise node disjoint simple paths of length k_i from s_i to t_i in G for every $i \in [k]$ if and only if there exist pairwise edge disjoint trails of length $2k_i + 1$ from s'_i to t'_i in G' for every $i \in [k]$.

The Line Graph We denote by line$(G, s_1, t_1, \dots, s_k, t_k)$ a variation on the *line graph of G* [141, Proof of Lemma 2]. The line graph construction was used by LaPaugh and Rivest to reduce the edge disjoint subgraph homeomorphism problem to the node disjoint subgraph homeomorphism problem. We adapt it to multigraphs. More precisely, we denote by line$(G, s_1, t_1, \dots, s_k, t_k)$ the tuple $(G', s_1, t_1, \dots, s_k, t_k)$ obtained as follows. Let $G = (V, E, \mathcal{E})$ be a directed (unlabeled) multigraph. We construct the directed graph $G' = (V', E', \mathcal{E}')$ were $V' = \{v_e \mid e \in E\} \cup \{s_1, t_1, \dots, s_k, t_k\}$ and E' is the disjoint union of

- $\{(v_{e_1}, v_{e_2}) \mid e_1, e_2 \in E$ and destination$(e_1) =$ origin$(e_2)\}$,
- $\{(s_i, v_e) \mid i \in [k], e \in E$, and $s_i =$ origin$(e)\}$, and
- $\{(v_e, t_i) \mid i \in [k], e \in E$, and $t_i =$ destination$(e)\}$.

For completeness, we define $\mathcal{E}'((x, y)) = (x, y)$ for every edge $(x, y) \in E'$.

Lemma 5.5.2. *Let $(G', s_1, t_1, \dots, s_k, t_k) =$ line$(G, s_1, t_1, \dots, s_k, t_k)$. Then the following hold:*

(1) For every $i \in [k]$, the path $e_1 \cdots e_n$ is a trail from s_i to t_i in G if and only if

$$(s_i, v_{e_1})(v_{e_1}, v_{e_2}) \cdots (v_{e_{n-1}}, v_{e_n})(v_{e_n}, t_i)$$

is a simple path in G'.

(2) For every $i \in [k]$, the number of trails from s_i to t_i in G equals the number of simple paths from s_i to t_i in G'.

(3) There exist pairwise edge-disjoint trails of length k_i from s_i to t_i in G for every $i \in [k]$ if and only if there exist pairwise node-disjoint simple paths of length $k_i + 1$ from s_i to t_i in G' for every $i \in [k]$.

Proof. Properties (1) and (2) are immediate from the construction. Property (3) follows from (1): if we have edge-disjoint trails, then the same simple paths as obtained in (1) are node-disjoint and the other way around. If they were not node-disjoint, at least two would share a node, say, v_e in G', but they only contain this node both if the corresponding trails in G have the edge e, contradicting the edge-disjointness of the trails in G. $\square$

Adding Edge Labels If we additionally consider edge labels and RPQs, the correspondence between simple paths and trails is a bit more complex. We prove here that upper bounds transfer from simple path problems to trail problems. That is, we give a variant of Lemma 5.5.2 for labeled graphs.

Notice that strengthening Lemma 5.5.1 for labeled graphs without changing the language of the RPQ is impossible if FPT $\neq$ W[1]. To see this, we note that the expression $a^k b^*$ is conflict-free, but not cuttable. This implies that $\mathsf{PTrail}(a^k b^*)$ is in FPT while $\mathsf{PSimPath}(a^k b^*)$ is W[1]-hard (see Theorem 5.2.2 and Theorem 5.2.4). Since a strengthened version of Lemma 5.5.1 would imply that $\mathsf{PSimPath}(a^k b^*)$ is at most as hard as $\mathsf{PTrail}(a^k b^*)$, such a lemma can only exist if FPT = W[1].

Lemma 5.5.3. *Let r be an RPQ, let σ be an arbitrary symbol in Σ, let G be a directed multigraph with labels in Σ, and s, t nodes in G. Then there exist a directed graph H and nodes s', t' such that there exists a trail from s to t in G that matches r if and only if there exists a simple path from s' to t' in H that matches the RPQ $\sigma \cdot r$. Furthermore, H, s', and t' can be computed using logarithmic space and $H = (V_H, E_H, \mathcal{E}_H)$ with $|V_H| = O(|E|)$ and $|E_H| = O(|E|^2)$.*

Proof. Given a directed multigraph $G = (V, E, \mathcal{E})$ and nodes s, $t \in V$, we will construct a directed graph H and nodes s' and t' such that there exists a simple path from s' to t' in H matching the RPQ $\sigma \cdot r$ if and only if there exists a trail from s to t matching r in G. In fact, $(H, s', t') = \mathit{line}(G, s, t)$ with labels. More precisely, let $\sigma \in \Sigma$ be fixed. Let $H = (V_H, E_H, \mathcal{E}_H)$ with $V_H = \{v_e \mid e \in E\} \cup \{s', t'\}$ and $E_H = \{(v_{e_1}, \text{lab}(e_1), v_{e_2}) \mid e_1, e_2 \in E \text{ and } \text{destination}(e_1) = \text{origin}(e_2)\} \cup \{(s', \sigma, v_e) \mid e \in E \text{ and } \text{origin}(e) = s\} \cup \{(v_e, \text{lab}(e), t') \mid e \in E \text{ and } \text{destination}(e) = t\}$. For completeness, we define $\mathcal{E}_H((x, a, y)) = (x, a, y)$ for every edge $(x, a, y) \in E_H$. An example of this reduction can be seen in Figure 5.9. Note that in the figure the nodes are named $v_{\mathcal{E}(e)}$ instead of v_e to give a better overview. From this construction, it immediately follows that $|V_H| = O(|E|)$ and $|E_H| = O(|E|^2)$.

We argue that this construction is correct. Indeed, assume there exists a path $p = e_1 \cdots e_n$ from s to t in G that matches r and has pairwise disjoint edges. Then the path

$$p' = (s', \sigma, v_{e_1})(v_{e_1}, \text{lab}(e_1), v_{e_2})(v_{(e_2}, \text{lab}(e_2), v_{e_3}) \cdots (v_{e_n}, \text{lab}(e_n), t')$$

is a simple path from s' to t' in H that matches $\sigma \cdot r$. The other direction follows analogously since each path from s' to t' in H that matches $\sigma \cdot r$ has this form and we can therefore find the corresponding path from s to t in G. ☐

We note that, in the proof of Lemma 5.5.3, there is a clear correspondence between nodes in H and edges in G. To be more precise, each node in H, except for s' and t', corresponds to exactly one edge in G. We therefore obtain the following corollary:

Corollary 5.5.4. *Let r be an RPQ, G a directed multigraph, and s, t nodes in G. Let (H, s', t') and $\sigma \cdot r$ be the instance obtained from G, s, and t as in Lemma 5.5.3. Then there exists a bijection f_{sp} from the set of trails from s to t in G to the set of simple paths from s' to t' in H such that $\sigma \cdot lab(p) = lab(f_{sp}(p))$. Moreover, f_{sp} and f_{sp}^{-1} are computable in linear time.*

Proof. Let $p = e_1 \cdots e_n$ be a trail from s to t in G. Then we define

$$f_{sp}(p) = (s', \sigma, v_{e_1})(v_{e_1}, \text{lab}(e_1), v_{e_2})(v_{(e_2}, \text{lab}(e_2), v_{e_3}) \cdots (v_{e_n}, \text{lab}(e_n), t')$$

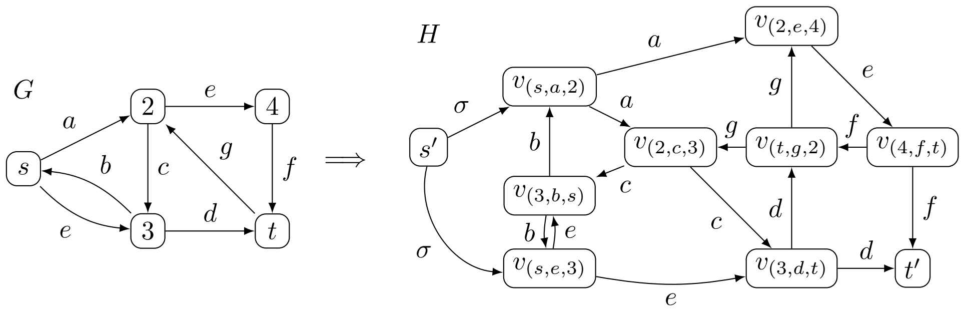


Figure 5.9: Example of a part of the reduction in Lemma 5.5.3. There exists a trail from s to t matching r in the left graph G if and only if there exists a simple path from s' to t' matching $\sigma \cdot r$ in the right graph H. So we effectively have to find a simple path that matches r in the right graph (H) from $s'_1 = v_{(s,a,2)}$ to t' or from $s'_2 = v_{(s,e,3)}$ to t'.

in H. Since all edges in p are pairwise different, the nodes $v_{e_1}, v_{e_2}, \dots, v_{e_n}$ (and s' and t') must be pairwise different. The mapping f_{sp} is a bijection since each simple path p' from s' to t' in H has such a form and we can therefore find the corresponding unique path $f_{sp}^{-1}(p')$ from s to t in G. □

5.6 Bounds for PTrail

In this section, we prove Theorem 5.2.4. To this end, we first consider the following fundamental parameterized problems for trails:

- PTrailLength: Given a directed multigraph G, nodes s and t, and parameter $k \in \mathbb{N}$, is there a trail from s to t of length exactly k in G?
- PTrailLength$^{\geq}$: Given a directed multigraph G, nodes s and t, and parameter $k \in \mathbb{N}$, is there a trail from s to t of length at least k in G?

By Lemma 5.5.2, the complexities of Theorems 5.3.2 and 5.3.5 carry over from simple paths to trails.

Theorem 5.6.1. *PTrailLength and PTrailLength$^{\geq}$ are in FPT. More precisely, PTrailLength is in time $2^{O(k)} \cdot |E|^3$ and PTrailLength$^{\geq}$ in time $2^{O(k)} \cdot |E|^4 \log |E|$.*

Similarly, we can consider the trail version of the parameterized two disjoint paths problem, where we require the paths to be edge-disjoint trails.

- PTwoDisjointTrails: Given a directed graph G, nodes s_1, t_1, s_2, t_2, and parameter $k \in \mathbb{N}$, are there trails p_1 from s_1 to t_1 and p_2 from s_2 to t_2 such that p_1 and p_2 are edge-disjoint and p_1 has length k?

The following theorem is immediate from Theorem 5.4.7 and Lemma 5.5.1(3).

Theorem 5.6.2. *PTwoDisjointTrails is W[1]-hard.*

Cai and Ye [59] proved that PTwoDisjointTrails is in FPT for *undirected graphs.* They left the node-disjoint and edge-disjoint cases for directed graphs open [59, Problems 3 and 4]. These two problems are W[1]-hard, due to Theorems 5.4.7 and 5.6.2. Next we will prove our dichotomy for trails.

Upper Bound for PTrail

Lemma 5.6.3. *Let $c \in \mathbb{N}$ be a constant and let $\mathcal{R}$ be the class of STEs with at most c conflict positions, that is, $\mathcal{R}$ is almost conflict-free. Then, PTrail($\mathcal{R}$) is in FPT. More precisely, it is in time $2^{O(|r|)} \cdot |E|^{c+6}$.*

Proof. We use the construction from the proof of Lemma 5.5.3 on the directed multigraph G to obtain a directed graph $H = (V_H, E_H, \mathcal{E}_H)$. By construction, there is a trail from s to t matching r in G if and only if there is a simple path from s' to t' matching $\sigma \cdot r$ in H (we can take σ to be an arbitrary label). So we need to decide whether there exists a simple path matching $\sigma \cdot r$ in H. To this end, we will do the following:

(1) We relabel the expression r to a conflict-free expression $\tilde{r}$. Then we enumerate all possible sets S of nodes of size up to c and relabel H depending on S, obtaining the graph H_S. We show that there is a simple path from s' to t' in H that matches $\sigma \cdot r$ if and only if there is a set S such that there is a simple path from s' to t' in H_S that matches $\sigma \cdot \tilde{r}$.

(2) Using a simple brute force algorithm, we can get rid of σ.

(3) We prove that Algorithm 3 does not only work for 0-bordered STEs, but also for conflict-free STEs when we restrict the graphs such that every node has only outgoing edges with the same label. Such graphs are obtained from the construction in Lemma 5.5.3. This allows us to use the methods from Lemma 5.3.15 to decide whether there exists a simple path matching $\tilde{r}$.

From (1)–(3) we can then conclude that deciding whether there exists a trail from s to t matching r with at most c conflict positions can be done using $|V_H|^{c+1}$ applications of Lemma 5.3.15, more precisely, $|V_H|^c$ times for all different sets S and $|V_H|$ times from the brute force algorithm to get rid of the σ. Since the time needed to find a simple path in Lemma 5.3.15 is $2^{O(|r|)} \cdot |V_H|^3|E_H|$, and V_H and E_H are of size $O(|E|)$ and $O(|E|^2)$, respectively (Lemma 5.5.3), we obtain a running time of $2^{O(|r|)} \cdot |E|^{c+6}$.

We start with (1). Let $r_1 = B_{\text{pre}}$ and $r_2 = B_{\text{suff}}$ with $r = r_1 T^* r_2$. We change r_1 and r_2 by relabeling the labels in conflict positions. Let c_1 and c_2 denote the left and right cut borders of r. In r_1, we replace each conflict position A_i, where $i \leq c_1$, with $\tilde{A}_i$. Here, $\tilde{A}_i$ is $(A_i \setminus T) \cup \{\tilde{a} \mid a \in A_i \cap T\}$, where we assume without loss of generality that $\tilde{a}$ is a new symbol, not occurring in r. Analogously, we replace each A'_j, where $j \leq c_2$ with $\tilde{A}'_j$, where $\tilde{A}'_j = (A'_j \setminus T) \cup \{\tilde{a} \mid a \in A'_j \cap T\}$. We name the resulting expressions $\tilde{r}_1, \tilde{r}_2$, and $\tilde{r} = \tilde{r}_1 T^* \tilde{r}_2$ to avoid confusion. Notice that the relabeling affects only conflict positions, thus at most c many A_i or A'_j.

Then, we enumerate all subsets of up to c nodes in H. For each possible subset S, we generate the graph H_S by changing each edge (u, a, v) with $u \in S$ and $a \in T$ to $(u, \tilde{a}, v)$.

We prove that there is a simple path from s' to t' in H that matches $\sigma \cdot r$ if and only if there is a set S such that there is a simple path from s' to t' in H_S that matches $\sigma \cdot \tilde{r}$. Assume that there is a simple path $p = (s', \sigma, v_1)(v_1, a_1, v_2) \cdots (v_\ell, a_\ell, t')$ from s' to t' in H that matches $\sigma \cdot r$. We choose $I_1 = \{i \mid a_i \in A_i \cap T \text{ and } i \leq c_1\}$ and $I_2 = \{\ell + 1 - i \mid a_{\ell+1-i} \in A'_i \cap T \text{ and } i \leq c_2\}$ and $S = \{v_i \mid i \in I_1 \cup I_2\}$. Then, the path in H_S consisting of the same nodes as p, in the same order, is a simple path from s' to t' matching $\sigma \cdot \tilde{r}$ in H_S. Conversely, if there is a simple path from s' to t' matching $\sigma \cdot \tilde{r}$ in H_S, for some set S, the path using the same nodes in the same order in H will match $\sigma \cdot r$. This concludes (1).

For (2), we enumerate all nodes $v \in V_H$ with $(s', \sigma, v) \in E_H$. Since s' has no incoming edges by construction, we cannot reach s' (unless we start in s') and therefore we do not need to explicitly delete s'.

For (3), we prove in Lemma 5.6.4 that Algorithm 3 also works for conflict-free STEs when the graphs are restricted to those where every node has only outgoing edges with the same label. Its proof is similar to the one of Lemma 5.3.11. The crucial part is that $\tilde{A}_1 \cdots \tilde{A}_{c_1}$ (where c_1 is the left cut border) and T have different labels. Since every node in H_S has only outgoing edges with the same labels, the first c_1 nodes of a path matching $\tilde{A}_1 \cdots \tilde{A}_{c_1}$ must therefore be node-disjoint from every path matching T^*.

Thus, we can use the methods[7] from Lemma 5.3.15 to decide whether, for any set S from (1) and node v from (2), there exists a simple path matching $\tilde{r}$ from v to t' in H_S. □

The following lemma is the counterpart to Lemma 5.3.11 and needed to complete the proof of Lemma 5.6.3.

Lemma 5.6.4 (similar to Lemma 5.3.11)**.** *Let $\mathcal{H}$ be the class of directed graphs in which each node has only outgoing edges with the same label. Let $r_1 = A_1 \cdots A_{k_1}$ be such that $r_1 T^*$ has no conflict positions and let $L(r_2)$ be an arbitrary finite language with length of longest word k_2. We define $k = k_1 + k_2$ and $r = r_1 T^* r_2$. Then, $H = (V_H, E_H, \mathcal{E}_H) \in \mathcal{H}$ has a simple path from s to t that matches r if and only if there exists a node $v \in V_H$ and a set of nodes $X \in \hat{P}^{r_1}_{s,v} \subseteq^{k+1}_{rep} P^{r_1}_{s,v}$, such that H has a simple path from s to t that matches r and with the first $k_1 + 1$ nodes belonging to X.*

Proof. The if direction is straightforward. For the only-if direction, let $p = e_1 \cdots e_n$ be a shortest simple path from s to t that matches r. We make the following case distinction on the length of p.

If $|p| \leq 2k_1 + k_2 + 1$, we define $P = e_1 \cdots e_{k_1}$ and $Q = e_{k_1+2} \cdots e_n$. Clearly, P matches r_1 and $e_{k_1+1} \cdot Q$ matches $T^* r_2$. Let $v_{k_1} = \text{destination}(e_{k_1})$. We have that $V(P) \in P^{r_1}_{s,v_{k_1}}$,

[7] That is, depending on the form of $\tilde{r}$, we use a simple reachability test, Algorithm 3, or a mixture of both.

$|V(Q)| \leq k+1$, and $V(P) \cap V(Q) = \emptyset$. Let $\hat{P}^{r_1}_{s,v_{k_1}}$ be a $(k+1)$-representative set of $P^{r_1}_{s,v_{k_1}}$. Then there exists a set $X \in \hat{P}^{r_1}_{s,v_{k_1}}$ with $X \cap V(Q) = \emptyset$. By definition of $P^{r_1}_{s,v_{k_1}}$, there exists a simple path P' from s to v_{k_1} with $V(P') = X$ that matches r_1. Therefore, $P' \cdot e_{k_1+1} \cdot Q$ is a simple path from s to t that matches $r_1T^*r_2$.

Otherwise, we have $|p| > 2k_1 + k_2 + 1$. $P = e_1 \cdots e_{k_1}$, $R = e_{k_1+2} \cdots e_{n-k-1}$, and $Q = e_{n-k+1} \cdots e_n$. We thus have

$$p = P \cdot e_{k_1+1} \cdot R \cdot e_{n-k} \cdot Q.$$

Since p matches r, we furthermore know that P matches r_1, R matches T^*, and Q matches $T^*T^{k_1}r_2$.[8] Let $v_{k_1} = \text{destination}(e_{k_1})$. Since $|V(Q)| = k+1$, $V(P) \in P^{r_1}_{s,v_k}$, and $V(P) \cap V(Q) = \emptyset$, the definition of $\hat{P}^{r_1}_{s,v_{k_1}} \subseteq^{k+1}_{\text{rep}} P^{r_1}_{s,v_{k_1}}$ guarantees, similar as in the previous case, the existence of a simple path P' from s to v_{k_1} that matches r_1 with $V(P') \in \hat{P}^{r_1}_{s,v_{k_1}}$ and $V(P') \cap V(Q) = \emptyset$. Let $P' = e'_1 = \cdots e'_{k_1}$. If P' is disjoint from R, the path

$$p' = P' \cdot e_{k_1+1} \cdot R \cdot e_{n-k} \cdot Q$$

is a simple path matching r, and we are done.

We show that P' must be disjoint from R. Let c_1 be the left cut border of $r_1T^*r_2$. Clearly, the paths P' and R cannot intersect in the first c_1 nodes of P' since those nodes only have outgoing edges that have labels not in T because of the definition of $H \in \mathcal{H}$ and r_1T^* has no conflict positions.

Towards a contradiction, assume that there is an $i \in \{c_1, \ldots, k_1 - 1\}$ such that $\text{destination}(e'_i) = \text{origin}(e_j)$ for some $j \in \{k_1 + 2, \ldots, n-k\}$.[9]

We choose i minimal and build a new simple path $p' = e'_1 \cdots e'_i e_j \cdots e_n$. This path matches $A_1 \cdots A_{c_1} \cdots A_i T^* T^{k_1} r_2$. Since c_1 is the left cut border of r, we have $T \subseteq A_i$ for all $c_1 + 1 \leq i \leq k_1$, so the new path matches r. Finally, we note that p' does not contain the edge e_{k_1+1}, so p' is shorter than p, which contradicts that p was a shortest simple path from s to t matching r. So P' must be disjoint from R. □

Lower Bound for PTrail

Lemma 5.6.5. *Let $\mathcal{R}$ be a class of STEs that can be conflict-sampled. If $\mathcal{R}$ is not almost conflict-free, then PTrail($\mathcal{R}$) is W[1]-hard.*

Proof. The proof follows the lines of Lemma 5.4.8, that is, we give a reduction from PTwoDisjointPaths restricted to instances of the form $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}})$ from Construction 5.4.4. Let $(G_{\text{node}}, s_1, t_1, s_2, t_2, k_{\text{col}})$ be an instance from PTwoDisjointPaths. Since $\mathcal{R}$ is not almost conflict-free and can be conflict-sampled, we can find an $r \in \mathcal{R}$ with at least $4k_{\text{col}} + 1$ conflict positions in time $f(k_{\text{col}})$, for some computable function f.

Let us assume that we have at least $2k_{\text{col}} + 1$ conflict positions in $A_1 \cdots A_{c_1}$, where c_1 is the left cut border of r. The case where we have at least $2k_{\text{col}} + 1$ conflict positions in

[8]Since r_2 might have shorter words, we cannot simply write $T^{k_1}r_2$.

[9]Since P and R are disjoint, we have $s, v_{k_1} \notin V(R)$.

$A'_{c_2} \cdots A'_1$ is symmetric. Therefore, r is of the form

$$A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1 \text{ or } A_1 \cdots A_{k_1} T^* A'_{k_2}? \cdots A'_1?$$

Starting from G_{node}, we will now split the nodes as in Lemma 5.5.1, and relabel the directed graph depending on r. More precisely, fix three words w_1, w_2, and w_3 such that

- $w_1 = a_1 \cdots a_{c_1} \in L(A_1 \cdots A_{c_1})$, such that $|w_1| \geq 2k_{\text{col}} + 1$ and $a_i \in A_i \cap T$ in at least $2k_{\text{col}}$ positions $i \in [c_1 - 1]\}$,
- $w_2 \in L(A_{c_1+1} \cdots A_{k_1})$, and
- $w_3 \in L(A'_{k_2} \cdots A'_1)$.[10]

Notice that, since $A_1 \cdots A_{c_1}$ has at least $2k_{\text{col}} + 1$ conflict positions, we can indeed choose w_1 such that $|w_1| \geq 2k_{\text{col}} + 1$ and $a_i \in A_i \cap T$ in at least $2k_{\text{col}}$ positions with $i \leq c_1 - 1$. We will refer to the first $2k_{\text{col}}$ such positions as the *conflict indices of* w_1. If i is a conflict index, we refer to the symbol a_i as *conflict symbol.*

We explain how G_{node} is changed. By definition of cut borders, we have that $T \not\subseteq A_{c_1}$. So we can fix an $x \in (T \setminus A_{c_1})$.

- As in Lemma 5.5.1, we split each node v into v^{in} and v^{out}. Furthermore, if v has an adjacent (incoming or outgoing) a-edge in G_{node}, we label the edge from v^{in} to v^{out} with a. Otherwise, we label it b. Observe that the resulting graph is the split graph of G_{node}, with some additional labels. We therefore call the resulting graph $\text{split}(G_{\text{node}})$.
- We replace each b-edge of $\text{split}(G_{\text{node}})$ by an x-path of length c_1.
- We will now relabel the a-edges in $\text{split}(G_{\text{node}})$ such that the resulting paths from s_1^{in} to t_1^{out} match w_1. We do this in several steps. The conflict positions on w_1 play a crucial role in the graph and the substrings of w_1 between conflict indices will serve as "padding" on the paths. Recall that w_1 has exactly $2k_{\text{col}}$ conflict indices $\{i_1, \ldots, i_{2k_{\text{col}}}\}$. Furthermore, $2k_{\text{col}}$ is the length of every a-path from s_1^{in} to t_1^{in} in $\text{split}(G_{\text{node}})$ (due to the construction in Lemma 5.5.1 and Lemma 5.4.3(c)). Therefore, on each path from s_1^{in} to t_1^{in}, we can label the ℓ-th edge with the conflict symbol a_{i_ℓ} from w_1.

  Since we only used the conflict indices of w_1 until now, we will still need to add padding to the paths to ensure that every path from s_1^{in} to t_1^{out} matches w_1. Furthermore, for the reduction to be correct, this padding needs to be done in a particular way, which we explain next. We label $(t_1^{\text{in}}, a_{c_1}, t_1^{\text{out}})$ with $a_{c_1} \in A_{c_1}$. (Since w_1 has $2k_{\text{col}} + 1$ conflict positions, c_1 is not a conflict index.) All paths from s_1^{in} to t_1^{out} are of the form

$$(u_1^{\text{in}}, u_1^{\text{out}})(u_1^{\text{out}}, u_2^{\text{in}})(u_2^{\text{in}}, u_2^{\text{out}}) \cdots (u_{k_{\text{col}}+1}^{\text{in}}, u_{k_{\text{col}}+1}^{\text{out}})$$

[10]We use $w_3 \in L(A'_{k_2} \cdots A'_1)$ in case that r ends with $A'_{k_2} \cdots A'_1$ but also if it ends with $A'_{k_2}? \cdots A'_1?$.

for some nodes $u_1, \dots, u_{k_{\mathrm{col}}+1}$ from G_{node}. For the correctness of the reduction, it will be crucial that, for each $j = 2, \dots, k_{\mathrm{col}}$, the edge from u_j^{in} to u_j^{out} is labeled with a conflict symbol, so we can only replace the edges from u_j^{out} to u_{j+1}^{in} with longer paths. Therefore, for every $j \in [k_{\mathrm{col}} - 1]$, we replace each such edge $(u_j^{\mathrm{out}}, a_{i_\ell}, u_{j+1}^{\mathrm{in}})$ with a path labeled $w[i_{\ell-1}+1, i_{\ell+1}-1]$ (where all internal nodes on these paths are new). Notice that, for each such edge, we have that $2 \leq \ell \leq 2k_{\mathrm{col}} - 1$, so $i_{\ell-1}$ and $i_{\ell+1}$ are indeed conflict indices of w_1. Additionally we replace $(u_{k_{\mathrm{col}}}^{\mathrm{out}}, a_{i_\ell}, u_{k_{\mathrm{col}}+1}^{\mathrm{in}})$ with the word $w_1[i_{2k_{\mathrm{col}}}, |w| - 1]$. If the word $w_1[1, i_1 - 1]$ is non-empty, we replace the edge $(s_1^{\mathrm{in}}, a_{i_1}, s^{\mathrm{out}})$ with a new path labeled $w_1[1, i_1]$. As a result, every path from s_1^{in} to t_1^{out} is now labeled with w_1.

- We add a path labeled w_2 from t_1^{out} to s_2^{in}, which we will call *the w_2-labeled path*, and a path labeled w_3 from t_2^{out} to a new node t, which we will call *the w_3-labeled path*.

This completes the construction. Call the resulting directed graph G_{edge}.

We will now prove correctness, that is, $(G_{\mathrm{node}}, s_1, t_1, s_2, t_2, k_{\mathrm{col}})$ is a yes-instance from PTwoDisjointPaths if and only if there is a trail from s_1^{in} to t matching r in G_{edge}.

For the direction from left to right, let $p_1 = u_1, \dots, u_{k_{\mathrm{col}}+1}$ be a simple path of length k_{col} from s_1 to t_1 and p_2 a simple path from s_2 to t_2 in G_{node}, such that p_1 and p_2 are node-disjoint. By Lemma 5.5.1 the path split(p_1) is a trail in split(G_{node}). By construction, there is a unique path P_1 from s_1^{in} to t_1^{out} in G_{edge} that contains all the edges of split(p_1). (Indeed, P_1 is the path split(p_1) with the extra padding.) Moreover, this path P_1 is a trail that matches w_1. Likewise, the path $P_2 = \text{split}(p_2)$ is a trail in split(G_{node}) and, by construction, also a trail in G_{edge}. Moreover, it matches T^* because every edge is either labeled x or labeled with a conflict symbol. Since p_1 and p_2 are node-disjoint, P_1 and P_2 are also node-disjoint and therefore edge-disjoint. Finally, if P_{w_2} and P_{w_3} are the w_2- and w_3-labeled paths respectively, then $P_1 P_{w_2} P_2 P_{w_3}$ is a trail from s_1^{in} to t that matches r.

For the other direction, let p be a trail from s_1^{in} to t in G_{edge} that matches r. We need some additional notions. For a path p in G_{edge}, we denote by contract(p) the path in G_{node} obtained from p by removing the padding and contracting node pairs $(u^{\mathrm{in}}, u^{\mathrm{out}})$ back to u. Formally, if $p = e_1 \cdots e_n$, then such a path is obtained from p by removing all edges of the form $(u^{\mathrm{in}}, ^{\mathrm{out}})$ and replacing, in the remaining edges, all nodes named u^{in} or u^{out} by u. By definition of G_{edge}, the resulting sequence of nodes forms a path in G_{node}.

We will prove:

(i) The path $p_1 = \text{contract}(p[0, c_1])$ is a simple path from s_1 to t_1 in G_{node}. Moreover, p_1 has length k_{col} and each edge in p_1 is labeled a (so it is even a path in G_{node}^a).

(ii) The prefix of p of length k_1 ends in s_2^{in} and is labeled $w_1 w_2$.

(iii) The path p is labeled $w_1 w_2 w' w_3$ with $w' \in L(T^*)$. The w_3-labeled suffix of p starts in t_2^{out} and ends in t.

We prove (i). By definition of r, the edge $p[c_1-1,c_1]$ in G_{edge} is labeled by some symbol in A_{c_1}. Therefore, this symbol cannot be x. By construction of G_{edge}, the only edges that are not labeled x are either on some w_1-labeled path from s_1^{in} to t_1^{out}, on the w_2-labeled path, or on the w_3-labeled path. Since the w_3-labeled path is not reachable from s_1 by a path of length at most c_1 and the w_2-labeled path starts in t_1^{out} and is therefore only reachable from s_1 with a path of length at least c_1, the edge $p[c_1-1,c_1]$ must be on one of the w_1-labeled paths from s_1^{in} to t_1^{out}. Furthermore, the entire path $p[0,c_1]$ must be a prefix of some w_1-labeled path from s_1^{in} to t_1^{out}. Indeed, if this were not be the case, then $p[0,c_1]$ would have to contain an x-path of length c_1 (since we replaced every b-edge in $\text{split}(G_{\text{node}})$ by an x-path of length c_1), which is impossible because it is too short for that.

This means that $p_1 = \text{contract}(p[0,c_1])$ is indeed a path in G_{node} and every edge of p_1 is labeled a. Therefore, it is a path in G^a_{node}. Since $p[0,c_1]$ has precisely $2k_{\text{col}}$ conflict indices and additionally contains the edge $(t_1^{\text{in}}, a_{c_1}, t_1^{\text{out}})$, it contains precisely $2k_{\text{col}}+1$ edges of the form $(u^{\text{in}}, u^{\text{out}})$ or $(u^{\text{out}}, v^{\text{in}})$ for some nodes $u, v \in V_{\text{node}}$. Since, for each path p in G_{node}, the length of $\text{split}(p)$ is $2|p|+1$, this means that the length of p_1 is precisely k_{col}. This implies (i).

Since all nodes that belong to the w_2-labeled path have only one outgoing edge, and since the path has length $k_1 - c_1$, we have that $p[0,k_1]$ ends in s_2^{in} and must match $w_1 w_2$. This shows (ii).

Since p matches $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ (the case $r = A_1 \cdots A_{k_1} T^* A'_{k_2}? \cdots A'_1?$ is analogous) and each word in $A_1 \cdots A_{k_1}$ has length k_1, it follows that $\text{lab}(p) = w_1 w_2 w'$ with $w' \in L(T^* A'_{k_2} \cdots A'_1)$. By construction of G_{edge}, the w_3-labeled path is the unique path of length $|w_3|$ leading to t. Therefore, each path from s_1^{in} to t in G_{edge} must end with the w_3-labeled path which is from t_2^{out} to t. Since $w_3 \in L(A'_{k_2} \cdots A'_1)$ and $|w_3|$ is the length of every word in $L(A'_{k_2} \cdots A'_1)$, we have that $\text{lab}(p) = w_1 w_2 w' w_3$ where $w \in L(T^*)$. So we have (iii).

Let p' be the part of p labeled w'. It follows from (ii) and (iii) that p' is a path from s_2^{in} to t_2^{out}. Let $p_2 = \text{contract}(p')$. First note that, by definition of G_{edge}, the resulting sequence of nodes is indeed a path in G_{node}. We show that p_1 and p_2 are node-disjoint. We first note that $p[0,c_1]$ and p' contain v^{in} if only if they contain v^{out}, since they start in s_1^{in} and s_2^{in} and end in t_1^{out} and t_2^{out}, respectively. Indeed, this is since v^{in} has only one outgoing edge and v^{out} only one incoming edge. So, if v^{out} belongs to $p[0,c_1]$, it cannot be part of p', otherwise $p[0,c_1]$ and p' both contain the edge $(v^{\text{in}}, v^{\text{out}})$, which would contradict that p is a trail. The same holds for nodes v^{out} that belong to p'. This implies that p_1 and p_2 cannot share a node and are therefore node-disjoint. Together with (i), we know that $|p_1| = k_{\text{col}}$, which implies that p_1 and p_2 are solutions to PTwoDisjointPaths.

Finally, we note that the construction can indeed be done in FPT since the expression $r \in \mathcal{R}$ can be determined in time $f(k_{\text{col}})$ for a computable function f, and all changes we made to the graph G_{node} are in time $h(k_{\text{col}}) \cdot |G_{\text{node}}|$, for a computable function h, which is FPT. Indeed, we only relabeled all edges, replaced each edge at most once with c_1 new edges, split each node at most once into two new ones, and added other paths of length at most $|r|$. Since $|r| \le f(k_{\text{col}})$, we have an fpt-reduction. □

Part II

Evaluation on Undirected Multigraphs

Chapter 6

Towards a Dichotomy for Regular Simple Path and Trail Queries

While the last part focused on directed multigraphs, we now turn to undirected multigraphs. This will help us to understand the data complexity of RPQs on graph databases with undirected or bidirectional edges, which is supported by the major systems. Furthermore, it helps us to understand the complexity of SimPath and Trail for *two-way languages*, that is, regular languages over $\Sigma \uplus \{\bar{a} \mid a \in \Sigma\}$, where $\uplus$ denotes disjoint union and the symbols $\bar{a}$ allow to match edges in *reverse* direction. More precisely, this means that, whenever there is an edge (u, a, v) in a directed graph, we are also allowed to consider it as the edge $(v, \bar{a}, u)$. For instance, in the directed graph in Figure 2.1 (left), the word $b\bar{a}c$ matches the path from s to t going through v_2 and v_1.

While interesting in their own right, RPQs over undirected multigraphs also teach us something about two-way RPQs on directed multigraphs. To see this, it is useful to consider for a directed multigraph its *underlying undirected multigraph*, which we do not define formally but illustrate in Figure 2.1. (Essentially, it is obtained by "forgetting" the direction of the edges.)

If we denote by h the homomorphism that maps every Σ-symbol a to $(a + \bar{a})$, then a language $L(r)$ is tractable under simple path semantics on undirected multigraphs if and only if the language $L(h(r))$ is tractable under simple path semantics on directed multigraphs. For example, simple path evaluation of the two-way regular expression $((a + \bar{a})(a + \bar{a}))^*$ corresponds to finding a simple a-path of even length in the underlying undirected graph. The situation for trails is analogous.

6.1 Definitions and Main Problems

Additionally to the preliminaries provided in Chapter 2, we need some more notation.

A path p from v to v is a *simple cycle* if $|V(p)| = |p|$. An example of a simple cycle is the path $p = e_1e_2$ with $\mathcal{E}(e_1) = (s, b, v_1)$, $\mathcal{E}(e_2) = (v_1, b, s)$ in Figure 2.1.

For an undirected multigraph $G = (V, E, \mathcal{E})$ and a set $X \subseteq V$, the *induced subgraph* of G on X is the multigraph $G' = (X, E', \mathcal{E}\big|_{E'})$ with $E' = \{e \mid e \in E \text{ and } \mathsf{Node}(e) \subseteq X\}$. For a label a, we denote by G_a the subgraph of $G = (V, E, \mathcal{E})$ restricted to edges labeled a, that is, $G_a = (V, E_a, \mathcal{E}\big|_{E_a})$ is a multigraph with $E_a = \{e \mid e \in E \text{ and } \mathrm{lab}(e) = a\}$.

Main Problems

We now consider the following SimPath and Trail on undirected graphs:

USimPath(L)	
Given:	An undirected multigraph G, nodes s, t.
Question:	Is there a simple path from s to t in G that matches L?

UTrail(L)	
Given:	An undirected multigraph G, nodes s, t.
Question:	Is there a trail from s to t in G that matches L?

6.2 Context

The task of understanding USimPath and UTrail for all regular languages (which is a major step towards understanding SimPath and Trail for all two-way regular languages) is very general and subsumes a long open standing problem to which we will get later in this section. First, consider the following problems.

kDisjointPaths	
Given:	A multigraph G, node pairs $(s_1, t_1), \dots, (s_k, t_k)$.
Question:	Are there pairwise disjoint paths from s_i to t_i in G for every $i \in [k]$?

These problems come in four variants for each k: for multigraphs that are directed or undirected, and for paths that are required to be node-disjoint (no common node) or edge-disjoint (no common edge).

Mod-k-Path	
Given:	A multigraph G, nodes s, t.
Question:	Is there a path of length 0 modulo k from s to t in G?

These problems also come in four variants: for multigraphs that are directed or undirected, and for simple paths or trails.

These problems are very relevant to SimPath and Trail on directed and undirected multigraphs: the Mod-k-Path problem is equivalent to deciding if there is a simple path/trail from s to t that matches $(a^k)^*$ in G. Let $L_{\text{equiv}} = a^* a_1 a_k a^* a_2 a_{k+1} ... a^* a_{k-1} a_{2k-2} a^*$, where $a, a_1, \dots, a_{2k-2} \in \Sigma$ are pairwise different. The kDisjointPaths problem is equivalent to deciding if there is a simple path/trail from s_1 to t_k that matches L_{equiv} in a G.[1] We make this equivalence more explicit.

[1] We use two different symbols $a_i a_{k-1+i}$ to simulate a directed edge even if G is an undirected multigraph. If G is a directed multigraph, the langugage can be simplified.

- We can find a simple path/trail from s to t in G that matches L_{equiv} as follows: We iterate over all tuples $(p_1, \ldots, p_{k-1})$ of node/edge-disjoint simple paths/trails p_i such that p_i matches $a_i a_{k-1-i}$. Assume that path p_i is from u_i to v_i. It then remains to test for k node/edge-disjoint paths matching a^* from s to u_1, v_i to u_{i+1}, and from v_{k-1} to t. This is equivalent to solving kDisjointPaths in the subgraph G_a of G.

- We can solve kDisjointPaths as follows: (Re)label every edge in G with a. Add new nodes and edges labeled $a_i a_{k-1+i}$ from t_i to s_{i+1} for each $i \in [k-1]$. Then this is equivalent to deciding if there is a simple path/trail from s_1 to t_k in G that matches L_{equiv}.

Since Mod-k-Path and kDisjointPaths are NP-complete for $k \geq 2$ on directed graphs [140, 98], these problems can be used to show NP-hardness. For example, Mendelzon and Wood [160] use TwoDisjointPaths to show that SimPath(a^*ba^*) is NP-complete. Bagan et al. [20] also use a reduction from TwoDisjointPaths to show that SimPath(L) is NP-complete for regular languages $L \notin \mathsf{SP}_{\mathsf{tract}}$. We use a similar reduction from the edge-disjoint variant in Lemma 3.4.19. Even our W[1]-hardness proofs in Lemmas 5.4.8 and 5.6.5 rely on the W[1]-hardness of the parameterized version of TwoDisjointPaths.

On the other hand, some of these problems are tractable on undirected multigraphs. This allows us to prove tractability results for USimPath and UTrail by reductions to cases in which Mod-k-Path and kDisjointPaths are tractable.

We now discuss in detail what is known about Mod-k-Path and kDisjointPaths on undirected (multi-)graphs.

Unlabeled, Undirected The famous *minor theorem* [182] implies that kDisjointPaths is tractable for every fixed k on undirected graphs, independent of whether we require node-disjoint or edge-disjoint paths. Indeed, for node-disjoint paths, this problem is equivalent to deciding if the set of k distinct edges $(s_1, t_1), \ldots, (s_k, t_k)$ is a minor of a given undirected graph. We note that for node-disjoint paths, it does not play a role whether we consider multigraphs or restrict ourselves to graphs, since every node can only be used once and therefore, no edge between a pair of nodes can be used more than once. For edge-disjoint paths, Jarry and Pérennes [123, Lemma 2] show how to decide if k edge-disjoint paths in undirected, unlabeled multigraphs in polynomial time exist: They split each edge before applying the line graph construction and then use the minor theorem.

Therefore, the following Proposition follows from Robertson and Seymour's Graph Minor Project [182] and Jarry and Pérennes [123, Lemma 2]:

Proposition 6.2.1. *kDisjointPaths on undirected multigraphs is in polynomial time for node- and edge-disjoint paths.*

The Mod-k-Path problem for $k = 2$ is tractable for simple paths [140].[2] The situation for $k \geq 3$ is rather intriguing. Arkin et al. [17] proved that, for every fixed $k > 1$, one can

[2]They attribute the first algorithm for this problem to Jack Edmonds due to private communication.

test in polynomial time whether there is an undirected simple path of length *different from 0 mod k* between two given nodes. Although this result allows to solve Mod-k-Path for $k = 2$, there is no clear reduction from Mod-k-Path to this problem if $k > 2$. Indeed, the complexity of Mod-k-Path for $k = 3$ has been open for 30 years [17].

When it comes to parity (mod 2) conditions, there has been recent progress. For instance, the minor theorem has been extended to incorporate parities [119, 132], which was a non-trivial effort. As a consequence, we now know that we can test in polynomial time if a given undirected graph has, k node-disjoint simple paths of even length.

Concerning edge-disjoint trails, Kawarabayashi and Kobayashi [130] defined the *extended line graph* construction, which maintains parity information and allows to transfer from known results on node-disjoint simple paths with parity constraints.

The construction replaces every vertex with a clique in which every edge is subdivided into two edges. The size of the clique replacing v is the number of edges adjacent to v.

We show how this idea can be tweaked to cope with arbitrary modulos. If we want to test path lengths modulo m, we subdivide every edge of the new cliques into m edges. Furthermore, start- and end-nodes $(s_1, t_1), \dots, (s_k, t_k)$ can be incorporated by adding extra nodes.

Lemma 6.2.2. *Let G be an undirected multigraph. There are trails from s_i to t_i of length j_i mod m_i in G which are pairwise edge-disjoint if and only if there are simple paths from s_i^* to t_i^* of length j_i mod m_i in the extended line-graph of G with respect to $(s_i, t_i)_{i \in [k]}$ modulo $m_1 \cdot m_2 \cdots m_k$ which are pairwise node-disjoint.*

Proof. We start with the exact definition of extension to the extended line graph from Kawarabayashi and Kobayashi [130] which now incorporates node-pairs and arbitrary modulo m conditions. Let an undirected multigraph $G = (V, E, \mathcal{E})$, and node-pairs $(s_1, t_1), \dots, (s_k, t_k)$ be given. Let $<$ be some order on E. We can assume without loss of generality that $s_i \neq t_i$ or $j_i \neq 0$ for each i (otherwise, this pair is trivially satisfied and can thus be removed). The extended line graph

$$L_{m_1 \cdot m_2 \cdots m_k}(G, (s_j, t_j)_{j \in [k]}) = ((G^*), (s_j^*, t_j^*)_{j \in [k]})$$

where $G^* = (V^*, E^*, \mathcal{E}^*)$ is an undirected graph defined by:

$$\begin{aligned}
V^* &= V_1^* \cup V_2^* \cup V_s^* \cup V_t^* \\
V_1^* &= \{(v, e) \mid v \in V, e \in E, e \text{ is adjacent to } v\} \\
V_2^* &= \{(v, \{e_1, e_2\}, i) \mid v \in V, e_1, e_2 \in E, e_1 \text{ and } e_2 \text{ are adjacent to } v \text{ and } i \in [m-1]\} \\
V_s^* &= \{s_j^* \mid j \in [k]\} \cup \{(s_j^*, (s_j, e), i) \mid e \text{ is adjacent to } s_j, j \in [k], i \in [m-1]\} \\
V_t^* &= \{t_j^* \mid j \in [k]\} \cup \{(t_j^*, (t_j, e), i) \mid e \text{ is adjacent to } t_j, j \in [k], i \in [m-1]\} \\
E^* &= E_1^* \cup E_2^* \cup E_s^* \cup E_t^* \\
E_1^* &= \{((v, e_1)(v, \{e_1, e_2\}, 1)), ((v, \{e_1, e_2\}, m-1), (v, e_2)) \mid \\
&\qquad\qquad e_1 < e_2, (v, e_1), (v, \{e_1, e_2\}) \in V^*, v \in V\} \\
&\quad \cup \{((v, \{e_1, e_2\}, i)(v, \{e_1, e_2\}, i+1)) \mid
\end{aligned}$$

$$
\begin{aligned}
&\qquad (v,\{e_1,e_2\},i),(v,\{e_1,e_2\},i+1) \in V^*, i \in [m-2]\}\\
E_2^* &= \{((v_1,e)(v_2,e)) \mid e \text{ is an edge connecting } v_1 \text{ and } v_2\}\\
E_s^* &= \{(s_j^*,(s_j^*,(s_j,e),1)),((s_j^*,(s_j,e),m-1),(s_j,e)) \mid\\
&\qquad s_j^*,(s_j^*,(s_j,e),1),(s_j,e) \in V^*, j \in [k]\}\\
&\quad \cup \{((s_j^*,(s_j,e),i),(s_j^*,(s_j,e),i+1)) \mid\\
&\qquad (s_j^*,(s_j,e),i),(s_j^*,(s_j,e),i+1) \in V^*, j \in [k], i \in [m-2]\}\\
E_t^* &= \{(t_j^*,(t_j^*,(t_j,e),1)),((t_j^*,(t_j,e),m-1),(t_j,e)) \mid\\
&\qquad t_j^*,(t_j^*,(t_j,e),1),(t_j,e) \in V^*, j \in [k]\}\\
&\quad \cup \{((t_j^*,(t_j,e),i),(t_j^*,(t_j,e),i+1)) \mid\\
&\qquad (t_j^*,(t_j,e),i),(t_j^*,(t_j,e),i+1) \in V^*, j \in [k], i \in [m-2]\}
\end{aligned}
$$

Furthermore, $\mathcal{E}^*(x,y) = (x,y)$ for all $(x,y) \in E^*$. We note that the sets $V_s^* \cup V_t^*$ and $E_s^* \cup E_t^*$ are empty if no distinguished nodes s_j, t_j exist.

We now prove the lemma. Let $((G^*),(s_j^*,t_j^*)_{j\in[k]}) = L_{m_1\cdot m_2\cdots m_k}(G,(s_j,t_j)_{j\in[k]})$. Since in G^* each path through each newly added clique has length 0 mod m_i, only edges of the form $((v_1,e)(v_2,e))$ count towards length modulo m_j. Given a simple path from s_j^* to t_j^* in G^*, one can construct a trail from s_j to t_j with the same length modulo m_j by replacing edges of the form $((v_1,e)(v_2,e))$ with e and omitting all edges of different forms. On the other hand, starting from a trail in G, one can add transitions (via the cliques) between each pair of edges to obtain a simple path in G^*. Since adding those transitions does not count towards the length modulo m_j, the so-constructed simple path has the same length modulo m_j. Finally, we note that node-disjoint paths in G^* cannot share edges of the form $((v_1,e)(v_2,e))$, thus their corresponding trails must be edge-disjoint and vice versa. ☐

This version of the extended line graph will be useful in Section 6.8.

Labeled, Undirected On undirected labeled graphs, the problem USimPath$((ab)^*)$ has been studied under the name *properly edge-colored (PEC) simple path* in a two-colored graph. Here, a path is defined to be PEC if its adjacent edges have different colors. It is decidable in polynomial time if a PEC simple path from s to t exists. For two colors, this result is attributed to Edmonds [149] and was generalized by Szeider [194] to any number of colors. Abouelaoualim et al. [5] give a polynomial time algorithm that decides in polynomial time if a PEC trail exists. Since their idea also works on multigraphs, USimPath$((ab)^*)$ and UTrail$((ab)^*)$ are in polynomial time.

Next, we discuss a number of results on two disjoint paths, since we will sometimes rely on them later in the thesis. To this end, for two languages L_1 and L_2, the problem of finding *node-* (respectively, *edge-*) *disjoint* L_1/L_2 *simple paths* (respectively, *trails*) refers to finding two node- (respectively, edge-) disjoint simple paths (respectively, trails) p_1

and p_2 such that $\text{lab}(p_1) \in L_1$ and $\text{lab}(p_2) \in L_2$ between given nodes (s_1, t_1), (s_2, t_2).[3] As such, finding disjoint a^*/b^* paths is equivalent to finding two monochromatic disjoint paths in an undirected graph with two edge colors. For edge-disjointness, the latter problem is in P as a-paths will always be edge-disjoint from b-paths. The problem therefore reduces to reachability. For node-disjointness, the problem is NP-complete, see [105, Theorem 16]. Finding node- or edge-disjoint $(ab)^*/(ab)^*$ paths is NP-hard [5] (a closely related problem was studied in [67]). Finally, node-disjoint $(ab)^*/a^*$ simple paths is NP-complete by Gourvès et al. [105, Proof of Corollary 10].

6.3 First Observations

Let $\mathsf{USP}_{\mathsf{tract}}$ be the class of regular languages for which $\mathsf{USimPath}$ is in P and let $\mathsf{UT}_{\mathsf{tract}}$ be the corresponding class for UTrail. While $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$ are closed under intersection and union, see Lemma 3.5.3, $\mathsf{USP}_{\mathsf{tract}}$ and $\mathsf{UT}_{\mathsf{tract}}$ are not closed under intersection if $\text{P} \neq \text{NP}$.

Theorem 6.3.1. *The following hold if P $\neq$ NP.*

(a) $\mathsf{USP}_{\mathsf{tract}}$ *and* $\mathsf{UT}_{\mathsf{tract}}$ *are closed under (finite) union.*

(b) $\mathsf{USP}_{\mathsf{tract}}$ *and* $\mathsf{UT}_{\mathsf{tract}}$ *are not closed under intersection.*

(c) $\mathsf{USP}_{\mathsf{tract}}$ *and* $\mathsf{UT}_{\mathsf{tract}}$ *are not closed under complement.*

(d) $\mathsf{USP}_{\mathsf{tract}}$ *and* $\mathsf{UT}_{\mathsf{tract}}$ *are closed under taking derivatives, that is, if L is tractable, then so is $w^{-1}L = \{u \mid wu \in L\}$.*

(e) $\mathsf{USP}_{\mathsf{tract}}$ *and* $\mathsf{UT}_{\mathsf{tract}}$ *are closed under reversal.*

Proof. We first prove (a). If we have two languages L_1, L_2 for which $\mathsf{USimPath}(L_1)$ and $\mathsf{USimPath}(L_2)$ ($\mathsf{UTrail}(L_1)$ and $\mathsf{UTrail}(L_2)$, respectively) are in P, we obtain a P algorithm for $\mathsf{USimPath}(L_1 \cup L_2)$ ($\mathsf{UTrail}(L_1 \cup L_2)$, respectively) by using the two P algorithms for L_1 and L_2. If any of them answers "yes", there is a simple path (trail, respectively) matching $L_1 \cup L_2$.

We start the proof of (b) with $\mathsf{USP}_{\mathsf{tract}}$. By Theorem 6.8.1 for the language containing an even number of a's and arbitrary number of bs $\mathsf{USimPath}$ is tractable, that is $\mathsf{USimPath}(b^*(ab^*ab^*)^*)$ is in P. Since a^*b^* is downwardclosed, $\mathsf{USimPath}(a^*b^*)$ is in P. On the other hand, $\mathsf{USimPath}((aa)^*b^*)$ is NP-hard. NP hardness follows from $G_{3\text{SAT}}$ with words $w_{\mathsf{s}} = a$, $w_{\mathsf{b}} = aa$, $w_{\mathsf{m}} = a$, $w_{\mathsf{r}} = b$, $w_{\mathsf{o}} = w_{\mathsf{t}} = \varepsilon$ and Theorem 6.4.2.

We now turn to $\mathsf{UT}_{\mathsf{tract}}$. By Theorem 6.8.1 the language containing an even number of a's and arbitrary number of b, c, and ds is tractable for UTrail, that is $\mathsf{UTrail}((b+c+d)^*(a(b+c+d)^*a(b+c+d)^*)^*)$ is in P. Since $a^*(b+c)^*a^*(b+d)^*$ is downwardclosed, $\mathsf{UTrail}(a^*(b+c)^*a^*(b+d)^*)$ is in P. Let L be the intersection of both languages, that is,

[3]The relationship between such problems and ours is that finding node-disjoint L_1/L_2 simple paths is closely related to finding a single simple path labeled L_1aL_2, for some label a (similar for edge-disjoint paths and trails).

$L = \{a^n(b+c)^*a^m(b+d)^* | n+m \text{ being even}\}$. We show that $\mathsf{UTrail}(L)$ is NP-hard. NP hardness follows from $G_{3\text{SAT}}$ with words $w_{\mathsf{s}} = a$, $w_{\mathsf{b}} = c$, $w_{\mathsf{m}} = a$, $w_{\mathsf{o}} = b$, $w_{\mathsf{r}} = d$, $w_{\mathsf{t}} = \varepsilon$ and Theorem 6.4.2.

For (c), we first observe that by (a), $\mathsf{USP}_{\mathsf{tract}}$ and $\mathsf{UT}_{\mathsf{tract}}$ are closed under union. If they were also closed under complement, we could simulate closure under intersection, which would contradict (b).

We now turn to prove (d). Let w be an arbitrary word. We first prove that there is a word w' of constant length such that $w^{-1}L = (w')^{-1}L$. Let L be a regular language. Let A an DFA for the language L, and w an arbitrary word. Then a DFA A' for $w^{-1}L$ can be obtained by changing the start state of A. (If $w^{-1}L = \emptyset$, we can choose a sink state as start.) Let q be the start state of A and q' be the start state of A'. Now we can find a word w' of length at most $|A|$ such that $\delta(q, w') = q'$.

We are now ready to prove (d). Let a graph G with nodes s and t be given. We can find a simple path (or trail) from s to t matching $w^{-1}L$ as follows: we add a new node s' and a path labeled w' from s' to s. Then there exists a simple path (trail, respectively) from s to t matching L' if and only if there exists a simple path (respectively trail) from s' to t matching L. Thus, if $L \in \mathsf{USP}_{\mathsf{tract}}$ (in $\mathsf{UT}_{\mathsf{tract}}$, respectively), it follows that $L' \in \mathsf{USP}_{\mathsf{tract}}$ (in $\mathsf{UT}_{\mathsf{tract}}$, respectively).

Part (e) is trivial because the question concerns undirected graphs. □

Although $\mathsf{USimPath}$ and UTrail are tractable for every language for which $\mathsf{SimPath}$ is tractable, UTrail and Trail are incomparable. An intuitive reason is that a trail for the language $(abc)^*$ in directed multigraphs is easy to find since loops which repeat edges can always be removed. On the other hand, we cannot use the same argument on undirected multigraphs since every edge can be used in one or the other direction and we can only remove loops if the joint edge is used in the same direction.

Theorem 6.3.2.

(a) $\mathsf{SP}_{\mathsf{tract}} \subseteq \mathsf{USP}_{\mathsf{tract}}$.

(b) $\mathsf{SP}_{\mathsf{tract}} \subseteq \mathsf{UT}_{\mathsf{tract}}$.

(c) If $P \neq NP$, then $\mathsf{T}_{\mathsf{tract}}$ *and* $\mathsf{UT}_{\mathsf{tract}}$ *are incomparable.*

Proof. We first prove (a). Let $G' = (V, E', \mathcal{E}')$ be the directed multigraph obtained from the undirected graph $G = (V, E, \mathcal{E})$ by replacing every undirected edge $e \in E$ with the two directed edges e_1, e_2 such that if $\mathcal{E}(e) = (u, a, v)$, then $\mathcal{E}'(e_1) = (u, a, v)$ and $\mathcal{E}'(e_2) = (v, a, u)$. Since a simple path can use at most one edge between any pair of nodes, a simple path in G' can use either e_1 or e_2, but not both. Thus, there is a simple path from s to t that matches L in G if and only if there is a simple path from s to t that matches L in G'.

We now prove (b). We will use that Bagan et al. [20, Theorem 6], see Theorem 2.5.5, give a definition of $\mathsf{SP}_{\mathsf{tract}}$ in terms of regular expressions, showing that a language is in $\mathsf{SP}_{\mathsf{tract}}$ if and only if it can be expressed as a union of regular expressions of the form

$$w_1(A_1^{\geq k_1} + \varepsilon)(w_2 + \varepsilon)(A_2^{\geq k_2} + \varepsilon) \cdots (w_n + \varepsilon)(A_n^{\geq k_n} + \varepsilon)w_{n+1} \qquad (\diamond)$$

for some $n \in \mathbb{N}$, words $w_j \in \Sigma^*$ with $j \in [n+1]$, sets $A_i \subseteq \Sigma$ and numbers $k_i \in \mathbb{N}$ with $i \in [n]$. Since $\mathsf{UT}_{\mathsf{tract}}$ is closed under union by Theorem 6.3.1, it suffices to prove that $\mathsf{UTrail}(r)$ is tractable for each regular expression r of the form $(\diamond)$.

Let $G = (V, E, \mathcal{E})$ be an undirected multigraph, $s, t \in V$, and r of the form $(\diamond)$. We will construct in polynomial time a directed graph G' and regular expression r' such that there exists a trail matching r from s to t in G if and only if there is a trail from s to t matching r' in G' that satisfies some additional restrictions. We then show that its existence can be tested in polynomial time.

Let $\$_1, \$_2$ be two symbols which occur neither in G nor in r. We construct from $G = (V, E, \mathcal{E})$ a new directed graph $G' = (V', E')$ with $V' = V \cup \{x_e, y_e \mid e \in E\}$, and $E' = \{(u, \$_1, x_e), (v, \$_2, x_e), (x_e, a, y_e), (y_e, \$_2, u), (y_e, \$_1, v) \mid e \in E, \mathcal{E}(e) = \{u, a, v\}\}$. Intuitively, we replace every edge e with the gadget presented in Figure 6.1. We note that these gadgets introduce loops labeled $\$_1 a \$_1$ from u to u and labeled $\$_2 a \$_2$ from v to v.

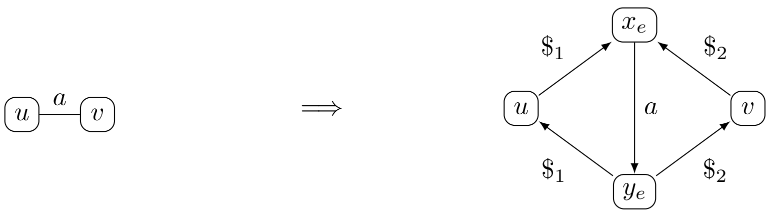


Figure 6.1: Illustration of the construction of the directed graph G' in the proof of Theorem 6.3.2(b).

Let $h\colon \Sigma \to \Sigma \cup \{\$_1, \$_2\}$ be a substitution with $h(\sigma) = (\$_1\sigma\$_2 + \$_2\sigma\$_1)$ for each $\sigma \in \Sigma$. That is, if $w = a_1 a_2 \cdots a_\ell \in \Sigma^+$, then $h(w)$ is a set of words, namely, $h(w) = (\$_1 a_1 \$_2 + \$_2 a_1 \$_1)(\$_1 a_2 \$_2 + \$_2 a_2 \$_1) \cdots (\$_1 a_\ell \$_2 + \$_2 a_\ell \$_1)$, furthermore, $h(\varepsilon) = \varepsilon$, and $h(A) = (\$_1 A \$_2 + \$_2 A \$_1)$ for every set A. Depending on r, we define

$$\tilde{r} = h(w_1)((h(A_1))^{\geq k_1} + \varepsilon)(h(w_2) + \varepsilon)((h(A_2))^{\geq k_2} + \varepsilon) \cdots$$
$$(h(w_n) + \varepsilon)((h(A_n))^{\geq k_n} + \varepsilon)h(w_{n+1}) .$$

Note that n, w_j for all $j \in [n+1]$, and A_i, k_i for all $i \in [n]$ are defined by r.

We now prove that there is a trail p matching r from s to t in G if and only if there is a trail p' matching $\tilde{r}$ in G'. Let $p = e_1 \cdots e_n$ be a trail from s to t in G that matches r. Then a trail p' can be obtained from p by replacing every edge e_i with its corresponding path matching $\$_1\mathrm{lab}(e_i)\$_2$ or $\$_2\mathrm{lab}(e_i)\$_1$ in G'. The so-constructed path clearly is a trail from s to t matching $\tilde{r}$ in G' and does not use subpaths labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$. On the other hand, let p' be a trail from s to t that matches $\tilde{r}$ in G'. By construction of G' and the definition of $\tilde{r}$, p' is a concatenation of paths of the form $(u, \$_1, x)(x, \sigma, y)(y, \$_2, v)$ or $(u, \$_2, x)(x, \sigma, y)(y, \$_1, v)$ for nodes $u, v \in V$ and $x, y \in V' - V$ and some symbol $\sigma \in \Sigma$. By construction of G', each such path corresponds to a unique edge in G, thus we can replace each such subpath of length 3 with the corresponding edge to obtain a trail from s to t matching r in G.

Unfortunately, expressions of the form $\tilde{r}$ are in general neither in $\mathsf{SP_{tract}}$ nor in $\mathsf{T_{tract}}$. The reason is that the subexpressions $(\$_1 A \$_2 + \$_2 A \$_1)^*$ do not satisfy the criteria in Definitions 2.5.4 or 3.1.11. Indeed, $\mathsf{SimPath}(\tilde{r})$ and $\mathsf{Trail}(\tilde{r})$ are NP-hard in general, but the graph G' has a very special form. To prove tractability, we first consider a similar expression r' defined as follows:

$$r' = h(w_1)((A_1 \cup \{\$_1, \$_2\})^{\geq 3k_1} + \varepsilon)(h(w_2) + \varepsilon)((A_2 \cup \{\$_1, \$_2\})^{\geq 3k_2} + \varepsilon) \cdots \\ (h(w_n) + \varepsilon)((A_n \cup \{\$_1, \$_2\})^{\geq 3k_n} + \varepsilon)h(w_{n+1}) \, .$$

The connection between $\tilde{r}$ and r' is as follows: because of the special form of G', there is a trail from s to t matching $\tilde{r}$ in G' if and only if there is a trail p' from s to t matching r' in G' and p' does not have a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$.

In order to show that we can decide in polynomial time whether a trail p' from s to t matching r' that does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for some symbol σ exists in G', we adapt the methods from Section 3.4.2. Indeed, $r' \in \mathsf{SP_{tract}}$ by Theorem 2.5.5 and since $\mathsf{SP_{tract}} \subseteq \mathsf{T_{tract}}$ by Theorem 3.3.3 it follows that $r' \in \mathsf{T_{tract}}$. We can thus use the results from Section 3.2. Let N be the size of the minimal DFA for r'. For convenience, we choose $K = N^2 + 4$ (instead of $K = N^2$ as in Section 3.4.2. We need the additional 4 edges to ensure that the path is "long enough" even if we remove some loops).

We now adapt some definitions to enforce that each shortest trail is "admissible". To this end, we define an *extended abbreviation* to be of the form $\mathsf{Cuts} \times (V \times Q) \times E^2 \times E^{K-2}$. An example is $(C, (v, q), e_K e_{K-1}, e_{K-2} \cdots e_1)$. A trail π matches $(C, (v, q), e_K e_{K-1} e_{K-2}, e_{K-3} \cdots e_1)$ if $\delta_L(q, \pi) \in C$, it starts in v with prefix $e_K e_{K-1}$ and its suffix is $e_{K-2} \cdots e_1$. We denote this with $\pi \models (C, (v, q), e_K e_{K-1}, e_{K-2} \cdots e_1)$. For an arbitrary set E' we write $\pi \models_{E'} (C, (v, q), e_K e_{K-1}, e_{K-2} \cdots e_1)$ if $\pi \models (C, (v, q), e_K e_{K-1}, e_{K-2} \cdots e_1)$ and all edges of π are from $E' \cup \{e_1, \ldots, e_K\}$. In the *extended summary* of a trail, every long run component is replaced by an extended abbreviation. An *extended candidate summary* is an extended summary of the form $S = \alpha_1 \cdots \alpha_m$ where each α_i is an edge or an extended abbreviation and all edges occurring in S are distinct.

Since all paths matching $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ are loops in G', a shortest path will not use such a subpath. Using this, we can use the NL algorithm from Lemma 3.4.7[4] to show the following

Lemma 6.3.3. *Let $r', G' = (V', E', \mathcal{E}')$ be as in the proof of Lemma 6.3.2(b), $\alpha = (C, (v, q), e_K e_{K-1}, e_{K-2} \cdots e_1)$ be an extended abbreviation, and $E'' \subseteq E'$. Then there is an NL algorithm that outputs a shortest trail p such that $p \models_{E''} \alpha$ if it exists and rejects otherwise. Furthermore, if a shortest path π from v to* destination(e_1) *with suffix $e_{K-2} \cdots e_1$ and with $\delta_L(q, \pi) \in C$ exists, for which $\pi \models_{E''} \alpha$ holds and such that π does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$, then p does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$.*

We postpone the proof for readability.

[4]We can adapt the algorithm such that the returned path starts with $e_K e_{K-1}$.

We now turn to local edge domains. Let E_i be as defined in Definition 3.4.8. We define Edge_i to be the set of edges used by trails π that start with $e_K e_{K-1}$, use only edges in E_i, and are of length at most $m_i - K + 2$. A trail p is *extended admissible* if there exists an extended candidate summary $S = \alpha_1 \cdots \alpha_k$ and trails $p_1, \ldots, p_k$ such that $p = p_1 \cdots p_k$ is a completion of S and $p_i \models_{\mathsf{Edge}_i} \alpha_i$ for every $i \in [k]$. Then the counterpart of Lemma 3.4.10 holds.

Lemma 6.3.4. *Let r', G' be as in the proof of Lemma 6.3.2(b). Then each shortest trail p from s to t that matches r' in G' and such that no subpath matches $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ is extended admissible.*

We postpone the proof of Lemma 6.3.4 for readability.

We can now give an NL algorithm similar to Lemma 3.4.13, that is, we enumerate all possible extended candidate summaries S with respect to (r', G', s, t) and apply on each extended summary the following algorithm which consists of four tests:

1. Guess, on-the-fly, a path p from S by replacing each α_i by a trail p_i such that $p_i \models_{\mathsf{Edge}_i} \alpha_i$ for each $i \in [k]$. This test succeeds if and only if this is possible.
2. In parallel, check that p matches r'.
3. In parallel, check that S is an extended summary of p.
4. In parallel, check that p does not contain a subpath matching $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$.

If all tests succeed on some candidate summary, then we answer "yes", and if on each candidate summary at least one test fails, the answer is "no".

To prove correctness, let there be a shortest trail p' from s to t matching r' that does not contain a subpath matching $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any symbol σ. Then, there is also a shortest such trail, and by Lemma 6.3.4 this trail is extended admissible. Conversely, if the algorithm succeeds, the path p is a trail because $E(S)$ and the sets Edge_i are mutually disjoint. By tests (2), (3), and (4), it is a trail from s to t that matches r' and does not contain a subpath matching $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$.

For the complexity, we note that compared to the NL algorithm in Lemma 3.4.13 we only need to additionally test (4), which can clearly be done in NL.

We now prove (c). On the one hand, $\mathsf{UTrail}(a^*ba^*)$ is in polynomial time by Theorem 6.5.2, while $a^*ba^* \notin T_{\text{tract}}$. On the other hand, $(abc)^*$ is in T_{tract} but $\mathsf{UTrail}((abc)^*)$ is NP-hard, see Theorem 6.6.1. □

To conclude the proof of part (b), we still need to prove Lemmas 6.3.3 and 6.3.4.

Proof of Lemma 6.3.3. Let $\alpha = (C, (v, q), e_K e_{K-1}, e_{K-2} \cdots e_1)$ be an extended abbreviation, and $E'' \subseteq E'$. Let $L = L(r')$. We use the NL algorithm from Lemma 3.4.7 to obtain a shortest trail p with $p \models_{E''} (C, (\text{destination}(e_{K-1}), \delta_L(q, e_K e_{K-1})), e_{K-2} \cdots e_1)$. Then $e_K e_{K-1} p$ is a shortest trail with $e_K e_{K-1} p \models_{E''} \alpha$.

Furthermore, let us assume that there exists a shortest path π from v to destination(e_1) with suffix $e_{K-4} \cdots e_1$ and with $\delta_L(q, \pi) \in C$ exists, π does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$, and such that $\pi \models_{E''} \alpha$.

Let $e_K e_{K-1} p = e_K e_{K-1} d_1 \cdots d_n e_{K-2} \cdots e_1$. We assume towards contradiction that $e_K e_{K-1} p$ contains a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$. Since π does not contain such a subpath, it cannot be in $e_{K-2} \cdots e_1$. Thus the subpath(s) labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ can only be in $e_K e_{K-1} d_1 d_2$, $d_1 \cdots d_n$, or $d_{n-1} d_n e_{K-2} e_{K-3}$. By definition of G', we can remove the loops labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ from these to obtain a trail p'' from v to destination(e_1). Since components of r' have the form $A^{\geq k_i}$ for some $k_i \in \mathbb{N}$ and some set of symbols A, $\delta_L(q, p'') \in C$. Furthermore, p'' has the suffix $e_{K-4} \cdots e_1$. Since $e_K e_{K-1} p$ is a shortest trail with $e_K e_{K-1} p \models_{E''} \alpha$, we have $|e_K e_{K-1} p| = |\pi|$. Since p'' is obtained from $e_K e_{K-1} p$ by removing edges, $|p''| < |\pi|$, contradicting the choice of π. $\square$

Proof of Lemma 6.3.4. We use the notion of extended abbreviation, extended (candidate) summary, and extended admissible from the proof of Theorem 6.3.2. The majority of the proof is similar to the proof of Lemma 3.4.10, indeed, we only have to additionally prove that the resulting paths p' do not have subpaths matching $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ or can be replaced with shorter paths that do not have such subpaths.By $p(e_1, e_2]$ we denote the suffix of $p[e_1, e_2]$ that excludes the first edge (so it can be empty). Notice that $p[e_1, e_2]$ and $p[e_1, e_2)$ are always well-defined for trails.

Let L be the language of r'. Let $p = d_1 \cdots d_m$ be a shortest trail from s to t that matches r' in G' and such that no subpath matches $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$. Let $S = \alpha_1 \cdots \alpha_k$ be the extended summary of p. Let $p_1, \ldots, p_k$ be trails such that $p = p_1 \cdots p_k$ and $p_i \models \alpha_i$ for all $i \in [k]$. We denote by left_i and right_i the first and last edge in p_i. By definition of p_i and the definition of extended summaries, left_i and right_i are identical with left_C and right_C if $\alpha_i \in \mathsf{Abbrv}$ is an extended abbreviation for the component C.

Assume that p is not extended admissible. That means there is some edge e used in p_ℓ such that $e \notin \mathsf{Edge}_\ell$. There are two possible cases:

(1) $e \in \mathsf{Edge}_i$ for some $i < \ell$; and

(2) $e \notin \mathsf{Edge}_i$ for any i.

In case (1), we choose i minimal such that some edge $e \in \mathsf{Edge}_i$ is used in p_j for some $j > i$. Among all such edges $e \in \mathsf{Edge}_i$, we choose the edge that occurs latest in p. This implicitly maximizes j for a fixed i. Especially no edge from Edge_i is used in $p_{j+1} \cdots p_k$.

Let $\alpha_i = (C_i, (v, q), e_K e_{K-1}, e_{K-2} \cdots e_1)$. By definition of Edge_i, there is a trail π from v, starting with $e_K e_{K-1}$ and ending with e, with $\delta_L(q, \mathrm{lab}(\pi)) \in C_i$, and that is shorter than the subpath $p[\mathsf{left}_i, \mathsf{right}_i]$ and therefore shorter than $p[\mathsf{left}_i, e]$. Let π be a shortest such path.

It was shown in Lemma 3.4.10 that $p' = p_1 \cdots p_{i-1} \pi p(e, d_m]$ is a trail matching r' and that the subpath $e'_{K-2} \cdots e'_1$ from $(C_j, (v', q'), e'_K e'_{K-1}, e'_{K-2} \cdots e'_1)$ is used in $p(e, d_m]$. In order to contradict the choice of p, we additionally need that p' does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$.

To this end, recall that all paths labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ in G' are loops.

- By choice of p, neither $p_1 \cdots p_{i-1}e_K e_{K-1}$ nor $p[e, d_m]$ contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$.
- If $\pi(e_{K-1}, e]$ contained a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$, then its removal would yield a shorter path π' starting with $e_K e_{K-1}$ and ending with e, and, by definition of r', with $\delta_L(q, \mathrm{lab}(\pi')) \in C_i$, contradicting the choice of π.

Thus, only $\pi p(e, d_m]$ could contain subpath(s) labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$: either in the first four edges of π or in the last two edges of π and the first two of $p(e, d_m]$. For example, π could end on $\$_1 a$ while $p(e, d_m]$ starts with $\$_1$.

Since each path labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ is a loop in G', the path obtained from p' by removing all subpaths labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ is indeed a path, and more precisely, a trail. By definition of r', subpaths of p' that are labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ must be matched by a strongly connected component of r'. Thus, in order to prove that p'' matches r', we have to prove that enough edges in C_j are used. Thus, it suffices to prove that removing the subpath(s) did not remove an edge of $e'_{K-4} \cdots e'_1$. Since $e'_{K-2} \cdots e'_1$ is in $p(e, d_m]$, and the removal of subpaths matching $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ could only remove the first two edges in $p(e, d_m]$, their removal does not affect $e'_{K-4} \cdots e'_1$. Thus p'' still matches r'.

This concludes case (1). For case (2), we additionally assume without loss of generality that there is no edge $e \in \mathsf{Edge}_i$ that appears in some p_j with $j > i$, that is, no edge satisfies case (1). By definition of Edge_ℓ, there is a trail π with $\pi \models_{\mathsf{Edge}_\ell} \alpha_\ell$ that is shorter than $p[\mathsf{left}_\ell, \mathsf{right}_\ell]$. We choose p' as the path obtained from p by replacing p_ℓ with a shortest such π.

It was shown in Lemma 3.4.10 that $p' = p_1 \cdots p_{\ell-1} \cdot \pi \cdot p_{\ell+1} \cdots p_k$ is a trail matching r'. Again, in order to contradict the choice of p, we additionally need that p' does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$. Let $(C, (v, \hat{q}), e_K e_{K-1}, e_{K-2} \cdots e_1) = \alpha_\ell$. Since p does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$, neither $p_1 \cdots p_{i-1} e_K e_{K-1}$ nor $e_{K-2} \cdots e_1 p_{\ell+1} \cdots p_k$ contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$.

Thus, subpaths labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ can only occur in $\pi[e_K, e_{K-3}]$. Because all paths labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ are loops in G', we can remove these paths from π and the resulting path π' is still a trail. Furthermore, π' is a trail from v that ends with $e_{K-4} \cdots e_1$ and, by definition of r' and its components, with $\delta_L(q, \pi') \in C$. Since $|e_{K-4} \cdots e_1| = N^2$, Lemma 3.2.7 implies that $\delta_L(q, \pi') = \delta_L(q, \pi)$. Thus $p'' = p_1 \cdots p_{\ell-1} \cdot \pi' \cdot p_{\ell+1} \cdots p_k$ is a trail matching r' that has no subpaths labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ and is shorter than p, contradicting the choice of p. ☐

An immediate consequence of Theorems 3.3.3 and 6.3.2 is the following:

Corollary 6.3.5. *USimPath*(L) *and UTrail*(L) *are in P for every downward closed language L.*

We present in Figure 6.2 an overview of the inclusion properties. Similar to Figure 3.3, the regular expressions provided in this figure can be used to distinguish the classes from one another. For example, the class $\mathsf{UT}_{\mathsf{tract}}$ can be distinguished from the class $\mathsf{USP}_{\mathsf{tract}}$ by the language a^*bc^*, while the language $(abc)^*$ is neither in $\mathsf{USP}_{\mathsf{tract}}$ nor in $\mathsf{UT}_{\mathsf{tract}}$. It

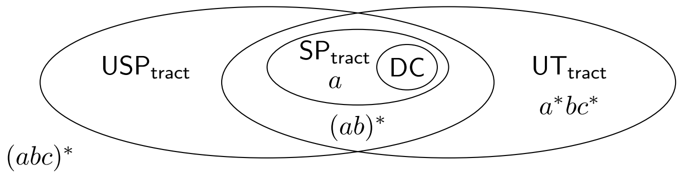


Figure 6.2: Expressiveness of $\mathsf{USP}_{\mathsf{tract}}$ and $\mathsf{UT}_{\mathsf{tract}}$.

is not known if there exists a language in $\mathsf{USP}_{\mathsf{tract}}$ that is not in $\mathsf{UT}_{\mathsf{tract}}$, which is why we do not give a language in that case.

Proposition 6.3.6. *Let L be a regular language and F_1, F_2 be finite languages. Then*

(a) if $L \in \mathsf{UT}_{\mathsf{tract}}$, then $F_1LF_2 \in \mathsf{UT}_{\mathsf{tract}}$ and

(b) if $L \in \mathsf{USP}_{\mathsf{tract}}$, then $F_1LF_2 \in \mathsf{USP}_{\mathsf{tract}}$.

Proof. To prove (a), let $L \in \mathsf{UT}_{\mathsf{tract}}$. Then there exists a P algorithm $\mathcal{A}$ which given nodes x and y, decides if there is a trail from x to y matching L. We can use $\mathcal{A}$ to decide if there exists a trail from s to t matching F_1LF_2 as follows: We iterate over all possible pairs of nodes $(x, y) \in V^2$ and all possible edge-disjoint trails (p_1, p_2) with p_1 from s to x matching F_1 and p_2 from y to t matching F_2. Then we use $\mathcal{A}$ to decide if there is a trail matching L from x to y in G without the edges in (p_1, p_2).

To prove (b), let $L \in \mathsf{USP}_{\mathsf{tract}}$. Then there exists a P algorithm $\mathcal{A}$ which given nodes x and y, decides if there is a simple path from x to y matching L. We can use $\mathcal{A}$ to decide if there exists a simple path from s to t matching F_1LF_2 as follows: We enumerate over all possible pairs of nodes $(x, y) \in V^2$ and all possible node-disjoint simple paths (p_1, p_2) with p_1 from s to x matching F_1 and p_2 from y to t matching F_2. Then we use $\mathcal{A}$ to decide if there is a simple path matching L from x to y in G without the nodes in (p_1, p_2). □

As a corollary, all languages definable by *simple transitive expressions*, see Definition 4.2.1, are in $\mathsf{UT}_{\mathsf{tract}}$ and $\mathsf{USP}_{\mathsf{tract}}$.

6.4 The Gadget G_{3SAT} for Lower Bounds

In this section, we construct a gadget for obtaining NP-hardness results throughout this thesis. We will reduce from 3SAT, which is well known to be NP-complete. An instance is a 3CNF formula $\varphi = \wedge_{i=1}^{m} C_i$ using variables $\{x_1, \dots, x_n\}$. The question is if φ is *satisfiable*, that is, there exists an assignment $\alpha : \{x_1, \dots, x_n\} \to \{\mathsf{true}, \mathsf{false}\}$ that satisfies φ. In fact, it is known that 3SAT is NP-complete, even if every variable appears exactly twice negated and twice unnegated in φ [79].

We will explain how to construct a generic undirected graph G_{3SAT} that we will later provide with labels to show NP-completeness of $\mathsf{USimPath}(L)$ and $\mathsf{UTrail}(L)$ for various languages L. The definition of $G_{3\text{SAT}}$ is somewhat technical[5] and is inspired on a gadget that was used by Eilam-Tzoreff [89] to reduce 3SAT to a variant of the disjoint paths problem with length constraints.

Construction 6.4.1. (Construction of $G_{3\text{SAT}}$.) Let φ be a formula in 3CNF with m clauses and n variables. In the following description, we will sometimes say that we will *add a w-path from u to v* for some word w. If $|w| \geq 1$ this means that, between the nodes u and v, we will add $|w| - 1$ new nodes and connect them such that the new nodes form a path from u to v that is labeled w. If $|w| = 0$, this means that u and v are merged together.

For each clause $(\ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3})$ in φ, we construct a *clause gadget* as in Figure 6.3 (left). For each variable x_j in φ, we construct a *variable gadget* as in Figure 6.3 (right). The words w_r and w_o are written on the paths in the usual "left-to-right" reading direction. We will refer to the edges on w_r-paths as *red edges*.

We now define a *switch* gadget that we will add for each occurrence of a variable, which leads to $3m$ such gadgets. Let $\ell_{i,k}$ be the kth literal in the ith clause. We add new nodes $u^1_{i,k}$ and $u^2_{i,k}$ and connect them as follows. We add a w_b-path from $u^1_{i,k}$ to $\ell^2_{i,k}$ and from $\ell^1_{i,k}$ to $u^2_{i,k}$. If $\ell_{i,k}$ is the pth negated occurrence of variable x_j, we additionally add w_b-paths from $u^1_{i,k}$ to $x^2_{j,p}$ and from $x^1_{j,p}$ to $u^2_{i,k}$. On the other hand, if $\ell_{i,k}$ is the pth unnegated occurrence of variable x_j, we additionally add w_b-paths from $u^1_{i,k}$ to $\overline{x}^2_{j,p}$ and from $\overline{x}^1_{j,p}$ to $u^2_{i,k}$.

Finally, we explain how to connect all gadgets. For each $i \in [m-1]$ we add a w_o-path from $c_{i,2}$ to $c_{i+1,1}$, from $c_{m,2}$ to $v_{1,1}$ and for each $j \in [n-1]$ we add a w_o-path from $v_{j,2}$ to $v_{j+1,1}$.

We then add w_o-paths from $u^2_{i,1}$ to $u^1_{i,2}$, from $u^2_{i,2}$ to $u^1_{i,3}$, and from $u^2_{i,3}$ to $u^1_{i+1,1}$. We set $s_2 = c_{1,1}, t_2 = v_{n,2}$, $s_1 = u^1_{1,1}$, and $t_1 = u^1_{m,3}$. Finally, we add a w_m-path from t_1 to s_2, new nodes s and t, a w_s-path from s to s_1, and a w_t-path from t_2 to t. We sketch the construction in Figure 6.4.

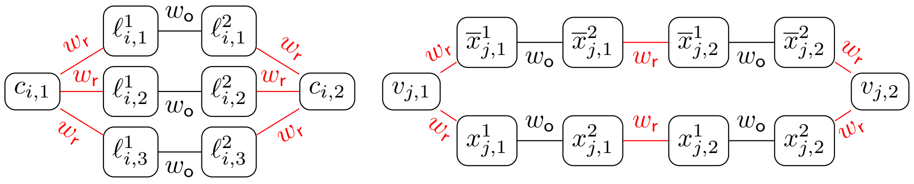


Figure 6.3: Clause gadget for the clause $C_i = (\ell_{i1} \vee \ell_{i2} \vee \ell_{i3})$ (left) and variable gadget for x_j (right). The paths are labeled such that the words w_r, w_o can be read from left to right.

Theorem 6.4.2. *Let $w_b, w_r \in \Sigma^+$ and G_{3SAT} as described in Construction 6.4.1. The following are equivalent:*

[5] In fact, some of the reductions in Gourvès et al. [105], which use a similar gadget, seem to be flawed, see Appendix A.

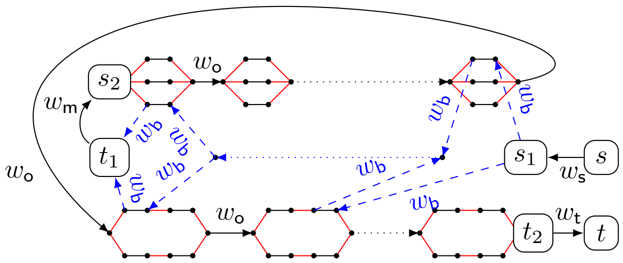


Figure 6.4: Sketch of the extension from two edge-disjoint paths to a single trail matching a language. The concrete placement of the (blue dashed) switch-edges depends on the occurrences of literals in clauses. The arrows indicate the "reading direction" of the words on the paths.

(a) φ is satisfiable.

(b) There exist node-disjoint paths p_1 from s_1 to t_1 and p_2 from s_2 to t_2 in G_{3SAT} such that p_1 does not use red edges.

(c) If $w_o \neq \varepsilon$ then there exist edge-disjoint paths p_1 from s_1 to t_1 and p_2 from s_2 to t_2 in G_{3SAT} such that p_1 does not use red edges.

(d) There exists a simple path p from s to t in G_{3SAT} that uses the w_m-path from t_1 to s_2 before using any red edge.

(e) If $w_o \neq \varepsilon$ then there exists a trail p from s to t in G_{3SAT} which reads the w_m-edge before using any red edge.

Proof. We first show (b) implies (a). If there exist two node-disjoint paths p_1 from s_1 to t_1 and p_2 from s_2 to t_2 in G_{3SAT} such that p_1 does not use red edges, then p_1 has to use all nodes $u^1_{i,k}$ and $u^2_{i,k}$ for each $i \in [m]$ and $k \in [3]$. Since p_2 is node-disjoint to p_1, it cannot use these vertices, so it can only pass through vertices of clause or variable gadgets. In the variable gadgets, it passes through one of two possible paths, if it passes through $\overline{x}^1_{j,1}$, we assign the variable x_j the value true, otherwise, we assign it the value false. We prove that this assignment satisfies φ. If p_2 used the path including $\overline{x}^1_{j,1}$, then p_1 cannot use this node, and thus has to use the nodes $\ell^1_{i,k}$ and $\ell^2_{i,k}$ with $\ell_{i,k} = x_j$ instead. Because of this, in clause gadgets, p_2 can only use edges which correspond literals which are set to true. Since p_2 has to traverse all clause gadgets in order to go from s_2 to t_2 while avoiding the nodes $u^1_{i,k}$ and $u^2_{i,k}$ for all $i \in [m]$ and $k \in [3]$, it follows that there must be at least one literal with value true in each clause gadget. Thus, φ is satisfiable.

The proof that (c) implies (a) is analogous.

We now prove that (a) implies (b) and (c): Let θ be a satisfying assignment of true values to variables in φ. We construct a path p_2 as follows: in each variable gadget, p_2 passes through the path with $\overline{x}_{j,1}$ if $\theta(x_j) = \mathsf{true}$ and through $x_{j,1}$ if $\theta(x_j) = \mathsf{false}$. In each clause gadget, p_2 passes through a path which corresponds to a literal which is set

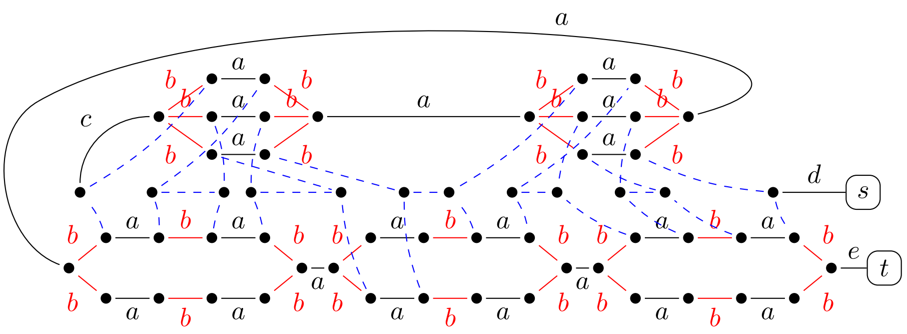


Figure 6.5: Example of the reduction from 3SAT with the boolean formula $(x_1 \lor x_1 \lor \overline{x_2}) \land (x_2 \lor x_3 \lor x_3)$ to the language $da^*c(a+b)^*e$. We use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = d, w_\mathsf{b} = w_\mathsf{o} = a, w_\mathsf{m} = c, w_\mathsf{r} = b, w_\mathsf{t} = e$. For readability, we colored the edges of the switch blue (and dashed) and omitted the labels on these edges (which all were a). Note that the path starting from s must use the c-labeled edge before it can use any of the *red edges*.

to true. This is possible since there is at least one in each clause. We can then construct a path p_1, which is (node- and edge-)disjoint from p_2. Between each pair of nodes $u^1_{i,k}$ and $u^2_{i,k}$, there are two possible paths which do not use red edges: one via a clause and one via a variable gadget. Let us assume that $\ell_{i,k}$ represents the literal ℓ. We have chosen p_2 in such a way that it uses at most one of these two edges, so p_1 uses the other. If ℓ is false, then p_2 uses the edge in the variable, but not in the clause gadget. If ℓ is true, then p_2 does not use the edge in the variable gadget.

Finally, we observe that (b) and (d) are equivalent: From node-disjoint paths p_1 and p_2 it is straight forward to construct a simple path from s to t by connecting p_1 via w_m to p_2. On the other hand, we can split a simple path p which does not use red edges before reading w_m into node-disjoint paths $p_1 = p[s_1, t_1]$ and $p_2 = p[s_2, t_2]$.

The proof that (c) and (e) are equivalent is analogous. □

Application of $G_{3\text{SAT}}$ for 2RPQs

We note that $G_{3\text{SAT}}$ can also be used to prove hardness for 2RPQs in which not every symbol is of the form $(a + \overline{a})$. More precisely, we can use $G_{3\text{SAT}}$ to show that the 2 disjoint paths problem where one path is directed and one path is undirected is NP-hard, even in a graph without labels/only a-labels.

Lemma 6.4.3. *Node-/Edge-disjoint $a^*/(a+\overline{a})^*$-paths in directed graphs is NP-complete.*

Proof. We use $G_{3\text{SAT}}$ with directed paths/edges. More precisely, we direct all $w_{\mathsf{b}} = a$ and $w_{\mathsf{o}} = a$ edges in direction of the usual word, while $w_{\mathsf{r}} = aa$ will be a path of length 2 where the directions point to the node in the middle. Graphically, w_{r} looks like: $\xrightarrow{a} \cdot \xleftarrow{a}$.

Then, the directed path from s_1 to t_1 cannot use w_{r}-paths. The correctness follows similar as in Theorem 6.4.2. □

This implies hardness for several "mixed" 2RPQs like $a^*b(a+\overline{a})^*$. Interestingly, $G_{3\text{SAT}}$ is much less complex than the gadget used in the hardness proof of 2 disjoint paths in the purely directed case [98].

We believe that $G_{3\text{SAT}}$ can be used to prove hardness for many more languages, and it would be an interesting direction for future work.

6.5 Generalizing Two Disjoint Paths

We already discussed the close relationship between 2-disjoint-path problems and UTrail and $\mathsf{USimPath}$ in Section 6.2, which we can now make more concrete. Indeed, using $G_{3\text{SAT}}$ from Construction 6.4.1 and Theorem 6.4.2, we can obtain the following:

Theorem 6.5.1. *Let A and B be non-empty subsets of Σ.*

- *The node-disjoint A^*/B^*-paths problem on undirected multigraphs is in P if $A = B$, and it is NP-complete otherwise.*
- *The edge-disjoint A^*/B^*-paths problem on undirected multigraphs is in P if $A = B$ or $A \cap B = \emptyset$, and it is NP-complete otherwise.*

Proof. For node-disjoint paths: If $A = B$, then we can use the minor theorem (see Proposition 6.2.1) to find node-disjoint paths in the multigraph restricted to A labels. If $A \neq B$, we can assume without loss of generality that $B \not\subseteq A$ (otherwise rename). Let $a \in A$ and $b \in B \setminus A$. We use $G_{3\text{SAT}}$ with $w_{\mathsf{b}} = a$, $w_{\mathsf{o}} = \varepsilon$, $w_{\mathsf{r}} = b$. Since the A^* path cannot use b-edges, Theorem 6.4.2 implies the NP hardness. Since $\mathsf{USimPath}(L)$ is in NP for every regular language, NP-completeness follows.

For edge-disjoint paths: If $A = B$, we can use the minor theorem (see Proposition 6.2.1) to find edge-disjoint paths in the multigraph restricted to A. Otherwise, if $A \cap B = \emptyset$, we can find paths in the subgraph restricted to A or restricted to B separately. If $A \neq B$ and $A \cap B \neq \emptyset$, we can assume without loss of generality that $B \not\subseteq A$ (otherwise rename). Let $a \in A \cap B$ and $b \in B \setminus A$. We use $G_{3\text{SAT}}$ with $w_{\mathsf{b}} = a$, $w_{\mathsf{o}} = a$, $w_{\mathsf{r}} = b$. Since the A^* path cannot use b-edges, Theorem 6.4.2 implies the NP hardness. Since $\mathsf{UTrail}(L)$ is in NP for every regular language, NP-completeness follows. □

This result can be used to completely classify the complexity of UTrail and $\mathsf{USimPath}$ for languages of the form A^*wB^*, where w is an arbitrary word. If $w = \varepsilon$, then the language is A^*B^*, which is downward-closed and therefore always tractable. The other cases are in the following theorem.

Theorem 6.5.2. *Let A, B non-empty subsets of Σ and $w = \sigma_1 \dots \sigma_n \in \Sigma^*$ with $n \geq 1$. Then USimPath(A^*wB^*) is in P if:*

(1) $A = B$; or

(2) $n = 1$ and $A - \{\sigma_1\} = B - \{\sigma_1\}$; or

(3) $\sigma_1 = \dots = \sigma_n$ and ($A = \{\sigma_1\}$ or $B = \{\sigma_1\}$); or

(4) $\sigma_1 = \dots = \sigma_i \neq \sigma_{i+1} = \dots = \sigma_n$ and $A = \{\sigma_1\}$ and $B = \{\sigma_n\}$; or

(5) $\sigma_1 \neq \sigma_2 = \dots = \sigma_n$ and $B = \{\sigma_n\}$ and $A = \{\sigma_1, \sigma_n\}$; or

(6) $\sigma_1 = \dots = \sigma_{n-1} \neq \sigma_n$ and $A = \{\sigma_1\}$ and $B = \{\sigma_1, \sigma_n\}$;

and it is NP-complete otherwise.

UTrail(A^*wB^*) is in P if one of (1)–(6) holds; or $A \cap B = \emptyset$; or $n = 1$ and $A \cap B = \{\sigma_1\}$; and NP-complete otherwise.

Proof. We first show that (1)–(6) imply tractability. To this end, we rewrite A^*wB^* in each case to a language of the form $w_1C^*w_2C^*w_3$ or $w_1C^*D^*w_2$ for $C, D \subseteq \Sigma$ and $w_1, w_2, w_3 \in \Sigma^*$ with $|w_1| + |w_2| + |w_3| \leq n$. Languages of these forms are tractable because we can iterate over all possible simple paths/trails matching w_1, w_2, w_3 and find in the subgraph without w_1, w_2, w_3 either a path matching a downward closed language, namely C^*D^*, from the end of w_1 to the start of w_2, or two node-/edge-disjoint C-paths from the end of w_1 to the start of w_2 and from the end of w_2 to the start of w_3 with Theorem 6.5.1. These tests are in polynomial time since there are at most $|E|^n$ many edges where n is a constant, Corollary 6.3.5, and Theorem 6.5.1.

In case (1), we can rewrite A^*wB^* into A^*wB^*, in case (2) into $(A \cup \{\sigma_1\})^*\sigma_1(A \cup \{\sigma_1\})^*$, in case (3) into $\sigma_1^n\sigma_1^*B^*$ or $A^*\sigma_1^*\sigma_1^n$, in case (4) into $\sigma_1^i\sigma_1^*\sigma_n^*\sigma_n^{n-i}$ for some $i \in [n]$, in case (5) into $A^*\sigma_1A^*\sigma_n^{n-1}$, and in case (6) into $\sigma_1^{n-1}B^*\sigma_nB^*$.

Furthermore, if $A \cap B = \emptyset$, UTrail(A^*wB^*) is also in P since we can first enumerate over all possible trails matching w and then find paths matching A^* and B^* separately, see Theorem 6.5.1. The case that UTrail$(A\sigma_1B)$ is in P if $A \cap B = \{\sigma_1\}$ is more complex and will be proved in Lemma 6.5.3.

On the other hand, if none of the conditions hold, we can prove NP completeness:

For every regular language L, UTrail(L) and SimPath(L) are in NP, thus we only need to prove NP-hardness. We first prove that if (1)–(6) fail, then USimPath(A^*wB^*) is NP-hard. Since $A \neq B$ and all rules are symmetric, we can assume without loss of generality that $B \not\subseteq A$, that is, $\exists b \in B \setminus A$. We perform a case distinction on w.

- if $n = 1$, then by $\neg(3)$ we know that there exist $a \in A, b \in B : a \neq \sigma_1 \neq b$. Since $B \not\subseteq A$ and by $\neg(2)$ $B \neq A \setminus \{\sigma_1\}$, there exist $a \in A, b \in B \setminus A$ with $a \neq \sigma_1 \neq b$. We use these symbols to label $G_{3\text{SAT}}$ as follows: $w_{\mathsf{s}} = \varepsilon, w_{\mathsf{b}} = a, w_{\mathsf{o}} = \varepsilon, w_{\mathsf{m}} = \sigma_1, w_{\mathsf{r}} = b, w_{\mathsf{t}} = \varepsilon$. Since the A^*-path starting in s cannot use b-edges, it has to follow the path until t_1 (the start of σ_1). By Theorem 6.4.2 this implies NP hardness.

- if $\exists i < j < k$ with $\sigma_i \neq \sigma_j \neq \sigma_k$, then take an arbitrary $a \in A$ and use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = a^n$, $w_\mathsf{r} = b^n$, $w_\mathsf{o} = \varepsilon$ and $w_\mathsf{m} = w$. Since w_b and w_r have length n, concatenations of w_b and w_r do not yield the substring w.

- if $\sigma_1 = \ldots = \sigma_n$ and $A \neq \{\sigma_1\} \neq B$: Then there exist $a \in A \setminus \{\sigma_1\}$, and, since $B \not\subseteq A$, there is $b \in B \setminus \{\sigma_1\}$ with $b \notin A$. We then use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = a$, $w_\mathsf{r} = b$, $w_\mathsf{o} = \varepsilon$ and $w_\mathsf{m} = w$.

- if $\sigma_1 = \ldots = \sigma_i \neq \sigma_j = \ldots = \sigma_n$: due to (4) we distinguish between (a) $A \neq \{\sigma_1\}$ and (b) $B \neq \{\sigma_n\}$.

  Case (a): If $A \neq \{\sigma_1\}$, we can pick $a \in A \setminus \{\sigma_1\}$ and use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = a^n$, $w_\mathsf{r} = b^n$, $w_\mathsf{o} = \varepsilon$ and $w_\mathsf{m} = w$. This works because $a \neq \sigma_1$, $b \notin A$, and $w \neq b^n$.

  Case (b): So let us assume that $A = \{\sigma_1\}$ and $B \neq \{\sigma_n\}$. If there exists a $b \in B \setminus A$ with $b \neq \sigma_n$, we can use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = \sigma_1$, $w_\mathsf{r} = b$, $w_\mathsf{o} = \varepsilon$ and $w_\mathsf{m} = w$. So let us assume that not such b exists, that is, $A = \{\sigma_1\}$ and $B = \{\sigma_1, \sigma_n\}$. If w contains 2 σ_n, we can use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = \sigma_1$, $w_\mathsf{r} = \sigma_1\sigma_n\sigma_1$, $w_\mathsf{o} = \varepsilon$ and $w_\mathsf{m} = w$. On the other hand, if w contains only a single σ_n, we are contradicting that (6) fails.

We now prove that if $A \cap B \neq \emptyset$, and in case $n = 1$ additionally $A \cap B \neq \{\sigma_1\}$, and (1)–(6) fail, then $\mathsf{UTrail}(A^*wB^*)$ is NP-hard. Since $A \neq B$ and all rules are symmetric, we can assume without loss of generality that $B \not\subseteq A$, that is, $\exists b \in B \setminus A$. We perform a case distinction on w.

- if $n = 1$, then we additionally know that $A \cap B \neq \{\sigma_1\}$. Thus, there exists $a \in A \cap B$ with $a \neq \sigma$. Together with $\neg(2)$ and $B \not\subseteq A$ this implies that there exist $a \in A \cap B$, $b \in B \setminus A$ with $a \neq \sigma_1 \neq b$. We can now prove NP-hardness with $G_{3\text{SAT}}$ using the labels $w_\mathsf{s} = \emptyset$, $w_\mathsf{b} = w_\mathsf{o} = a$, $w_\mathsf{m} = \sigma$, $w_\mathsf{r} = b$, $w_\mathsf{t} = \emptyset$. Since $b \notin A$, and $a \neq \sigma_1 \neq b$, the A-path starting from s is not allowed to use b-edges before reaching t_1. Therefore, Theorem 6.4.2 implies NP hardness.

- if $\exists i < j < k$ with $\sigma_i \neq \sigma_j \neq \sigma_k$, then take an arbitrary $a \in A \cap B$ and use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = a^n$, $w_\mathsf{r} = b^n$, $w_\mathsf{o} = a^n$ and $w_\mathsf{m} = w$. Since $w_\mathsf{b}, w_\mathsf{o}$, and w_r have length n, concatenations of w_b and w_r do not yield the substring w.

- if $\sigma_1 = \ldots = \sigma_n$ and $A \neq \{\sigma_1\} \neq B$: Then there exist $a \in A \setminus \{\sigma_1\}$, and, since $B \not\subseteq A$, there is $b \in B \setminus \{\sigma_1\}$ with $b \notin A$. Let $c \in A \cap B$ arbitrary. We then use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = a$, $w_\mathsf{r} = b$, $w_\mathsf{o} = c$ and $w_\mathsf{m} = w$. This works because $|w| \geq 2$ and w_o never appears twice in a row.

- if $\sigma_1 = \ldots = \sigma_i \neq \sigma_j = \ldots = \sigma_n$: due to (4) we distinguish between (a) $A \neq \{\sigma_1\}$ and (b) $B \neq \{\sigma_n\}$.

  Case (a): $A \neq \{\sigma_1\}$. If $A \cap B \neq \{\sigma_1\}$, we can pick $a \in A \cap B \setminus \{\sigma_1\}$ and use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = a^n$, $w_\mathsf{r} = b^n$, $w_\mathsf{o} = a^n$ and $w_\mathsf{m} = w$. This works because $a \neq \sigma_1$, $b \notin A$, and $w \neq b^n$. If $A \cap B = \{\sigma_1\}$, then, since $B \not\subseteq A$, $B \neq \{\sigma_1\}$. So there exist a, b with $\{\sigma_1, a\} \subseteq A$ and $\{\sigma_1, b\} \subseteq B$. We now perform a case distinction on σ_n.

- if $a = \sigma_n$, we use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = \sigma_1$, $w_\mathsf{r} = b$, $w_\mathsf{o} = \sigma_1$ and $w_\mathsf{m} = w$. Then σ_n only appears in w_m.
- if $b = \sigma_n$, we show that $\mathsf{UTrail}(B^* w^{\mathsf{rev}} A^*)$ is NP-hard. Therefore, we use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = \sigma_1$, $w_\mathsf{r} = a$, $w_\mathsf{o} = \sigma_1$ and $w_\mathsf{m} = w^{\mathsf{rev}}$. The result for $\mathsf{UTrail}(A^* w B^*)$ then follows since $\mathsf{UT}_{\mathsf{tract}}$ is closed under reversal, see Theorem 6.3.1.
- if $a \neq \sigma_n \neq b$, we use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = \sigma_1$, $w_\mathsf{r} = b$, $w_\mathsf{o} = \sigma_1$ and $w_\mathsf{m} = w$. Then σ_n only appears in w_m.

Case (b): So let us assume that $A = \{\sigma_1\}$ and $B \neq \{\sigma_n\}$. Since $A \cap B \neq \emptyset$, $\sigma_1 \in B$. If there exists a $b \in B \setminus A$ with $b \neq \sigma_n$, we can use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = \sigma_1$, $w_\mathsf{r} = b$, $w_\mathsf{o} = \sigma_1$ and $w_\mathsf{m} = w$. So let us assume that not such b exists, that is, $A = \{\sigma_1\}$ and $B = \{\sigma_1, \sigma_n\}$. If w contains 2 σ_n, we can use $G_{3\text{SAT}}$ with words $w_\mathsf{s} = w_\mathsf{t} = \emptyset$, $w_\mathsf{b} = \sigma_1$, $w_\mathsf{r} = \sigma_1\sigma_n\sigma_1$, $w_\mathsf{o} = \sigma_1$ and $w_\mathsf{m} = w$. On the other hand, if w contains only a single σ_n, we are contradicting that (6) fails.

Since $\mathsf{USimPath}(L)$ and $\mathsf{UTrail}(L)$ are in NP for every regular language, NP-completeness follows. □

To complete the proof of Theorem 6.5.2 it remains to prove the next lemma.

Lemma 6.5.3. *$\mathsf{UTrail}(A^*\sigma B^*)$ is in P if $A \cap B = \{\sigma\}$.*

Proof. If $A = \{\sigma\}$ or $B = \{\sigma\}$, then tractability follows from condition (3) in Theorem 6.5.2. So let $a \in A \setminus B$ and $b \in B \setminus A$. We denote by G_A the subgraph of G restricted to edges with labels in A and by G_B the subgraph of G restricted to edges with labels in B. We describe a polynomial time algorithm that solves $\mathsf{UTrail}(A^*\sigma B^*)$. Let $G = (V, E, \mathcal{E})$ be an undirected multigraph. We iterate over all tuples of nodes $(u_1, u_2) \in V \times V$ such that there is at least one σ-edge between u_1 and u_2. We name one such edge e_σ. Let G' be a copy of G without e_σ (we delete only a single edge, even if there are multiple σ-edges between u_1 and u_2). In G', we rename every σ-edge

- which is only on a trail from s to u_1 path in G'_A and not on a trail from u_2 to t path in G'_B to an a-edge,
- which is not on a trail from s to u_1 in G'_A, but on a trail from u_2 to t in G'_B to a b-edge,
- whose deletion would make t unreachable from u_2 in G'_B to a b-edge. That is, each σ-edge separating u_2 and t in G'_B is renamed to a b-edge.

If, after the renaming, there exist paths (not necessarily disjoint) from s to u_1 in G'_A and from u_2 to t in G'_B, we return true. If, after enumerating over all tuples (u_1, u_2) no such paths were found, we return false.

We now prove correctness. Let us assume the algorithm returned true. Then there exists a tuple of nodes (u_1, u_2), a σ-edge e_σ with endpoints u_1 and u_2, an A-path from s to u_1 and a B-path from u_2 to t in G'. Consider an arbitrary trail p from s to u_1 in G'_A. If p is σ-free, every path from u_2 to t in G'_B will be disjoint from p. Thus we can build a trail matching $A^*\sigma B^*$ by concatenating p with e_σ and a shortest path from u_2 to t in G'_B. Otherwise, let $p = e_1 \cdots e_\ell$ and e_i be the first σ-edge in p. Let $x \in \mathsf{Node}(e_i)$ be the destination of $e_1 \cdots e_i$. We will construct a trail p_2 from x to t in G'_B which does not use e_i. Since e_i was the first σ-edge in p and $A \cap B = \{\sigma\}$, the prefix of p will be disjoint from p_2 and therefore $e_1 \cdots e_i \cdot p_2$ will be a trail from s to t that matches $A^*\sigma B^*$.

Let $\mathsf{Node}(e_i) = \{x, y\}$. Since e_i was not relabeled, there is a trail from u_2 to t in G'_B which uses e_i. If this trail can be split into a trail from u_2 to y and from x to t, then the trail from x to t is our p_2. Otherwise, let us assume the B-path is split into a trail from u_2 to x and one from y to t. Since there is a trail from u_2 to t not using e_i (otherwise, this edge had been relabeled b), we can construct a B-path not using e_i by concatenating the path from x to u_2 with this path from u_2 to t. Removing cycles in which edges are used more than once then yields the trail p_2 from x to t which does not use e_i.

We now turn to the other direction. Let $p = e_1 \cdots e_\ell$ be a trail from s to t matching $A^*\sigma B^*$. Then there exists an $i \in [\ell]$ such that $e_1 \cdots e_{i-1}$ matches A^*, $\text{lab}(e_i) = \sigma$, and $e_{i+1} \cdots e_\ell$ matches B^*. Since the algorithm enumerates over all tuples of nodes, it will have enumerated over $\mathsf{Node}(e_i)$. Let (u_1, u_2) be the nodes of e_i in the order they appear in p. Since all edges in $e_1 \cdots e_{i-1}$ are on a trail from s to u_1, they will only be renamed to a-edges (if at all). With the same argument, the edges in $e_{i+1} \cdots e_\ell$ will only be renamed to b-edges (if at all). Thus, even after renaming the edges in G' as described in the algorithm, there are still paths from s to u_1 in G'_A and from u_2 to t in G'_B. Thus the algorithm will return true. ☐

6.6 Word Iterations

In this section we give an overview of the complexity of USimPath and UTrail for *word iterations*, that is, languages of the form w^*, where w is a word. This setting has essentially three cases. The first case, where $w = a^n$ with $n \geq 3$, has been an open problem since 1991 [17]. The other two cases are the following.

Theorem 6.6.1. *Let w be a word.*

(a) If $|w| \leq 2$, then USimPath(w^*) and UTrail(w^*) are in P.

(b) If $|w| \geq 3$ and w has at least 2 different symbols, then UTrail(w^*) and USimPath(w^*) are NP-complete.

Proof. We first prove (a). If $|w| = 1$, finding a simple path or trail is equivalent to finding an arbitrary path. If $|w| = 2$, then we can find simple paths labeled by a word in $L(w^*)$ in P using the graph duplication technique of Edmonds [140, 149]. We note that this technique also works on multigraphs. For trails, if $w = aa$, using the extended line graph construction, the problem reduces to the one for simple paths, see Lemma 6.2.2. If

$w = ab$, then Abouelaoualim et al. [5] give a polynomial time algorithm which builds on the work of Szeider [194] and also works on multigraphs. The case $w = ba$ is equivalent to $w = ab$. We now prove (b). Since $\mathsf{USimPath}(L)$ and $\mathsf{UTrail}(L)$ in NP for every regular language, is remains to prove NP hardness. Assume that $|w| \geq 3$ and w has at least 2 different symbols. We distinguish the following cases:

(1) w is periodic, that is, $w = w_1^i$ for some $i \geq 2$;

(2) w has at least 3 different symbols; or

(3) w is not periodic and has exactly 2 different symbols.

We note that we will use different methods here since we do not see how to handle $w = abab$ with a reduction from 3SAT, while showing hardness for $w = aab$ seems impossible with a reduction from the two node-/edge-disjoint paths problem on directed graphs.

(1) We use a reduction from TwoDisjointPaths or TwoEdgeDisjointPaths, respecitvely. Both problems are NP-complete on directed graphs [98]. Let the node pairs s_1, t_1 and s_2, t_2 be given and $G_D = (V_D, E_D, \mathcal{E}_D)$ be a directed graph. Similar to Chou et al. [67], the main idea is to replace the directed edges with undirected paths labeled with some word which implies the direction. Furthermore, we add paths from a new node s to s_1, from t_1 to s_2, and possibly from t_2 to a new node t, such that the languages enforce a valid path to take the path from t_1 to s_2.

For this reduction it is necessary that the paths may not be traversed in the opposite direction. If $w \neq w^{\mathsf{rev}}$, we can directly replace each directed edge $e \in E_D$ with an undirected path labeled w from origin(e) to destination(e), and add a path labeled $(w_1)^{i-1}$ from t_1 to s_2 and a new start node s with a path labeled w_1 to s_1. Furthermore, we define $t = t_2$.

In the case that $w = w^{\mathsf{rev}}$, we first need to "shift" w. Let $w = w_1^i$. Since w_1 has at least two different symbols, we can write w_1 in the form $w_\ell w_r$ such that w_r starts with a symbol which is different from the symbol on which w_ℓ ends. Let $w_2 = w_r w_\ell$. Then, $w_2 \neq w_2^{\mathsf{rev}}$ and $L(w^+) = L(w_\ell (w_2^i)^* w_2^{i-1} w_r) = L(w_\ell (w_2^i)^* w_2^{i-1} (w_2^i)^* w_r)$. Thus, we replace every edge $e \in E_D$ with an undirected path labeled w_2^i from origin(e) to destination(e). Furthermore, we add a path labeled w_ℓ from a new node s to s_1, an edge labeled w_2^{i-1} from t_1 to s_2, and an edge labeled w_r from t_2 to a new node t. Let G_U be the so-constructed undirected graph.

We now show that there are two node-/edge-disjoint paths from s_1 to t_1 and from s_2 to t_2 in G_D if and only if there exists a simple path/trail matching w^* from s to t in G_U. To this end, we first observe that for each edge $e \in E_D$ we find exactly one path of length $|w|$ from origin(e) to destination(e). (Depending on whether $w = w^{\mathsf{rev}}$, this path is either labeled w or w_2^i.) We name this path $\mathsf{corresp}(e)$. We note that for each $e \in E_D$, $\mathsf{corresp}(e)$ is simple and does not use any nodes of V_D besides origin(e) and destination(e). Let $p = e_1 \cdots e_n$ be a path. We define $\mathsf{corresp}(p) = \mathsf{corresp}(e_1) \cdots \mathsf{corresp}(e_n)$. If there exist two edge-disjoint trails p_1 from s_1 to t_1 and p_2 from s_2 to t_2 in G_D, then we obtain a trail p' from s to t in G_U matching w^* by concatenating the path from s to s_1, $\mathsf{corresp}(p_1)$, the

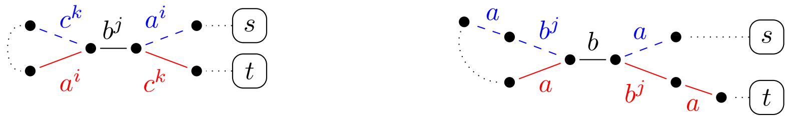


Figure 6.6: Parts of $G_{3\text{SAT}}$ for the proof of Theorem 6.6.1(b), cases (2) and (3). We show the "shared" w_o-path for the language $(a^i b^j c^k w')^*$ on the left and for the language $(abb^j w')^*$ (where w' is ε or starts with a) on the right. For orientation, we added dotted edges together with nodes s and t.

path from t_1 to s_2, $\mathsf{corresp}(p_2)$, and the path from t_2 to t. If p_1 and p_2 are node-disjoint simple paths, then the so-constructed path p' is a simple path by construction. On the other hand, if there is a path p' from s to t in G_U matching w^*, then it must start with the path from s to s_1, and end with the path from t_2 to t. Since p' matches w^*, and by definition of G_U, p' must use the path from t_1 to s_2. Furthermore, since $w \neq w^{\mathsf{rev}}$ or $w_2 \neq w_2^{\mathsf{rev}}$, the subpaths p_1' from s_1 to t_1 and p_2' from s_2 to t_2 must follow the paths in the "intended direction". Let p_1 and p_2 be the paths in G_D obtained from p_1' and p_2' by deleting nodes not in V_D and making its two neighbors adjacent. Furthermore, if p' is a simple path, then p_1 and p_2 must be node-disjoint simple paths. And if p' is a trail, then p_1 and p_2 must be edge-disjoint trails.

(2) Since w has at least 3 different symbols, say a, b, and c, we can write it as $w^+ = w_{\mathsf{s}}(a^i b^j c^k w')^* w_{\mathsf{t}}$ for some words $w_{\mathsf{s}}, w', w_{\mathsf{t}} \in \Sigma^*$ and numbers $i, j, k \geq 1$. We then use $G_{3\text{SAT}}$, see Construction 6.4.1, with the words $w_{\mathsf{s}} = a^i b^j, w_{\mathsf{o}} = b^j$, $w_{\mathsf{r}} = w_{\mathsf{b}} = c^k w' a^i$, $w_{\mathsf{t}} = b^j c^k w'$, $w_{\mathsf{m}} = b^j$ to prove NP-hardness.

Note that every path matching w^* that starts in s has use the w_{m}-path from t_1 to s_2 before it can use any red edge because before and after every "shared" b^j-path (that is, every undirected path labeled b^j from $\ell^1_{i,k}$ to $\ell^2_{i,k}$, from $x^1_{j,k}$ to $x^2_{j,k}$, or from $\overline{x}^1_{j,k}$ to $\overline{x}^2_{j,k}$) there is at most one a and at most one c-edge and no other b-edge, see Figure 6.6 (left). Thus the correctness follows from Theorem 6.4.2.

(3) Since w is not periodic and has exactly two different symbols, say a and b, it can be written as $w^+ = w_{\mathsf{s}}(abb^i w') w_{\mathsf{t}}$ for some words $w_{\mathsf{s}}, w', w_{\mathsf{t}} \in \{a, b\}^*$ and a number $j \geq 1$ such that $w' = \varepsilon$ or w' begins with a (that is, j is maximal).

We then use $G_{3\text{SAT}}$ with words $w_{\mathsf{o}} = b$, $w_{\mathsf{b}} = w_{\mathsf{r}} = b^j w' a$, $w_{\mathsf{s}} = ab$, $w_{\mathsf{t}} = bb^j$, $w_{\mathsf{m}} = b$.

Again, every path matching w^* that starts in s has use the w_{m}-path from t_1 to s_2 before it can use any red edge. The reason can be seen in Figure 6.6 (right): If the path starting in s would use a red edge before reading the w_{m}-path from t_1 to s_2, it would contain a substring $ab^j a$ or aba instead of $abb^j a$. Thus, this path would not be labeled w^*. Thus the correctness follows from Theorem 6.4.2. $\square$

6.7 Simple Chain Regular Expressions

We consider a simple variant of *chain regular expressions*, which were introduced to study static analysis of schemas for XML [152] and used for studying the complexity of SPARQL query evaluation [146].

Definition 6.7.1 (Simple Chain Regular Expression (SCRE))**.** A *factor* is a regular expression of the form a, a^*, or $a?$ where a is some symbol from Σ. A *simple chain regular expression (SCRE)* is a (possibly empty) concatenation of factors.

We use a similar shorthand notation for SCREs as in [152]. In short, we write $\mathsf{SCRE}(f_1, \dots, f_k)$ for the class of SCREs in which we allow factors $f_1, \dots, f_k$. For example, the expression $a^*b^*ab^*a?$ is in $\mathsf{SCRE}(a, a?, a^*, b^*)$. We will use a special symbol $\star$ to abbreviate “all alphabet symbols that were not listed yet”. For example, $\mathsf{SCRE}(a, a^*, \star?)$ is the class of SCREs that use factors in $\{a, a^*\} \cup \{b? \mid b \in \Sigma - \{a\}\}$. Next, we study $\mathsf{UTrail}(L)$ and $\mathsf{USimPath}(L)$ for languages L that are definable by SCREs.

Trails are Tractable

Remarkably, finding trails is tractable for every language definable by an SCRE.

Theorem 6.7.2. *$\mathsf{UTrail}(L)$ is in P for every language L definable by an SCRE.*

Proof. We can write every expression $r \in \mathsf{SCRE}(\star?, \star, \star^*)$ in the form $r = r_1 a_1^* r_2 \cdots a_{\ell-1}^* r_\ell$, where $r_i \in \mathsf{SCRE}(\star?, \star)$ for each $i \in [\ell]$. Since ℓ is a constant and since each path that matches r_i has constant length, we can iterate in polynomial time over all tuples $(p_1, \dots, p_\ell)$ of disjoint (sub)trails of the multigraph G such that each p_i matches r_i. Let G' be G without the edges of $(p_1, \dots, p_\ell)$. Assume that path p_i is from u_i to v_i (with $u_1 = s$ and $v_\ell = t$). In order to complete the subtrails to a trail that matches r, we will test, for each symbol $a \in \Sigma$ and all $i \in [\ell - 1]$, for edge-disjoint trails in G'_a. More precisely, let $k = |\{a_i \mid a_i = a\}|$. For each $a \in \Sigma$, we test if there exist k edge-disjoint paths $p^a_{i_1}, \dots, p^a_{i_k}$ such that $a_{i_j} = a$ and $p^a_{i_j}$ is a path from v_{i_j} to u_{i_j+1} in G'_a. Since k is a constant, their existence can be tested in polynomial time, see Proposition 6.2.1. Since G'_{a_i} and G'_{a_j} are mutually edge-disjoint graphs for all $a_i \neq a_j$, the so-constructed paths will be edge-disjoint. $\square$

Simple Paths are Not So Simple

The situation for finding simple paths is much more complex, however. In order to maintain an overview, we differentiate between the number of alphabet symbols used in the SCREs. Since the number of alphabet symbols in RPQs recently found in query logs is typically low [53, 55], even the results on one or two alphabet symbols are of practical interest.

One or Two Alphabet Symbols

If the SCREs just use a single alphabet symbol, that is, we have languages definable by an $\mathsf{SCRE}(a, a?, a^*)$, then USimPath is always tractable. The next theorem shows that, for a second alphabet symbol b, factor types $b, b?$ or $b?, b^*$ can be added. We will see later that allowing both b, b^* leads to NP-completeness.

Theorem 6.7.3. *USimPath(L) is in P*

(a) for every language definable by an SCRE$(a, a?, a^*, b, b?)$ and

(b) for every language definable by an SCRE$(a, a?, a^*, b?, b^*)$.

Proof. For (a), assume that $r \in \mathsf{SCRE}(a, a?, a^*, b, b?)$. Then there exists an $\ell \in \mathbb{N}$ with $r = r_1 a_1^* r_2 \cdots a_{\ell-1}^* r_\ell$, where $r_i \in \mathsf{SCRE}(a, a?, b, b?)$ for each $i \in [\ell]$. Since ℓ is a constant that depends only on r and since paths that match each such r_i have constant length, we can iterate in polynomial time over all tuples $(p_1, \dots, p_\ell)$ of node-disjoint simple (sub)paths such that each p_i matches r_i. Assume that path p_i is from u_i to v_i (with $u_1 = s$ and $v_\ell = t$). In order to complete the subpaths to a path that matches r, we need to test if there exist $\ell - 1$ paths that are a-labeled, mutually node-disjoint, node-disjoint from $p_1, \dots, p_\ell$, and respectively from v_i to u_{i+1}, for each $i \in [\ell - 1]$. Testing if these paths exist can be done by running the polynomial-time algorithm for k-node-disjoint simple paths on the graph obtained from the input multigraph by deleting all inner nodes of $p_1, \dots, p_\ell$. The proof of (b) follows from Lemma 6.7.12. We now turn to (b). As in (a), we can rewrite every regular expression of this form in a normal form. We then show in Lemma 6.7.12 that Algorithm 4 correctly decides this problem in polynomial time. $\square$

The next few pages are devoted to the proof of Lemma 6.7.12. We first give some necessary definitions and explain the outline of our proof and the idea of Algorithm 4.

Observation 6.7.4. *Every regular expression r in SCRE$(a, a?, a^*, b?, b^*)$ can be written in a normal form*

$$r = r_1 a_1^* b_1^* r_2 a_2^* b_2^* \cdots r_\ell \tag{†}$$

with $a_i \in \{a, \varepsilon\}$, $b_i \in \{b, \varepsilon\}$, and $r_i \in$ SCRE$(a, a?, b?)$. As such, each path that matches r can be seen as a path that consists of the following subpaths:

- *paths p_i matching r_i from nodes z_{i-1} to x_i, for $i \in [\ell]$,*
- *a-paths from nodes x_i to y_i, for $i \in [\ell - 1]$, and*
- *b-paths from nodes y_i to z_i, for $i \in [\ell - 1]$,*

where $z_0 = s$ and $x_\ell = t$.

Thus $\mathsf{USimPath}(L)$ is in P for every language definable by an $\mathsf{SCRE}(a, a?, a^*, b?, b^*)$ if and only if $\mathsf{USimPath}(L)$ can be decided in polynomial time for languages definable by regular expressions of the form (†).

Definition 6.7.5. We say that a multigraph H is *2-connected* if (1) it contains at least two nodes and (2) for each node x, the multigraph $H - \{x\}$ is connected. A *1-(node-)cut* between two nodes u and v is a node x such that every path from u to v uses x. A *2-connected component* of H is a subgraph C of H that is 2-connected and maximal. Maximal means here that there is no node $x \notin C$ such that the induced subgraph of H on $C \cup \{x\}$ is 2-connected.

We say a path p *hits* a 2-connected component C if an inner node of p is a node of C.

Furthermore, given a simple path p and two nodes x, y in p, we denote by $p[x, y]$ the subpath of p from x to y.

A connection between node-disjoint paths and minimum node cuts was given by Menger [161].[6]

Theorem 6.7.6 (Menger's theorem)**.** *Let u and v be distinct, non-adjacent nodes in a connected, undirected multigraph G. Then the maximum number of internally node-disjoint paths between u and v in G equals the minimum node cut for u and v, which is the number of nodes, distinct from u and v, whose removal disconnects u and v.*

A result of Menger's theorem is that 2-connected components C have two node-disjoint paths between each pair of nodes in C.

We now explain how Algorithm 4 can decide $\mathsf{USimPath}(r)$ for regular expressions of the form (†). The outer loop of the algorithm enumerates the tuples $(p_1, \ldots, p_\ell)$ of constant-length node-disjoint simple paths that match the subexpressions of r_i, for each $i \in [\ell]$. Once we have these, we can find a simple path that matches r if we can complete $(p_1, \ldots, p_\ell)$ with node-disjoint simple paths that match the subexpressions of the form a^* or b^* at the appropriate places. This problem is essentially the problem of finding disjoint paths where some of the paths need to be labeled a and others need to be labeled b. The challenge for the present proof is that this latter problem is NP-complete. Indeed, in a given undirected graph with edge labels a and b, deciding if there are two node-disjoint simple paths between (s_1, t_1) and (s_2, t_2), one labeled with a's and the other with b's, is NP-complete [105, Theorem 16]. We therefore need a different approach.

Our approach uses a structural graph-theoretic argument to reduce the problem to finding sets of node-disjoint a-paths in graphs $G_A^{X,N}$ and sets of node-disjoint b-paths in other graphs $G_B^{X,N}$. These graphs will be computed from G based on $p_1, \ldots, p_\ell$ (that is, inner nodes of $p_1, \cdots, p_\ell$ will be removed) and a second much more intricate loop, that we describe later. The crux is that, if these sets of a-paths and b-paths exist for any $(p_1, \ldots, p_\ell)$ and any $G_A^{X,N}$ and $G_B^{X,N}$, then there is a simple path that matches r in G, because $G_A^{X,N}$ and $G_B^{X,N}$ are node-disjoint (up to start/end nodes of paths that we are interested in). If these sets of paths do not exist, then we need to prove a non-trivial re-routing result that shows that no simple path that matches r exists in general. This result (Lemma 6.7.7) shows that if a simple path that matches r exists, then there exists one that satisfies a number of conditions that allow us to run the inner loop of the algorithm in polynomial time.

[6]While Menger worked on graphs, the theorem immediately holds for multigraphs.

We note that in line 4, we call the tuple of nodes $(y_1, \dots, y_{\ell-1})$ *consistent with* $(a_i, b_i)_{i \in [\ell-1]}$ if $(x_i = y_i)$ is equivalent to $a_i = \varepsilon$ and $(y_i = z_i)$ is equivalent to $b_i = \varepsilon$. Thus this line enumerates possible nodes between the a_i- and b_i-paths.

We now describe the workings of the inner loop.

Let k be the number of occurrences of a factor of the form a in r, that is, k is the length of the shortest word in $L(r)$, which is a constant for the purposes of the present decision problem.

Let G' be the multigraph G without the inner nodes of $p_1, \dots, p_\ell$ and their adjacent edges. For each $i \in [\ell - 1]$ we define the undirected multigraph G_i depending on x_i and y_i as follows:

- if $x_i = y_i$, G_i is the single node x_i.
- otherwise, G_i is the the induced subgraph of G'_a on the nodes of simple paths from x_i to y_i (including x_i and y_i, but without the nodes z_j for which $z_j \neq x_{j+1}$ and $y_j \neq z_j$).

Notice that in G_i the 1-cuts between x_i and y_i are totally ordered by their proximity to x_i. Furthermore, between every pair of such consecutive 1-cut nodes between x_i and y_i in G_i, there either is nothing, or

- a 2-connected component with no simple cycle of length at least $2k$, or
- a 2-connected component with a simple cycle of length at least $2k$.

We call a 2-connected component of G_i that has a simple cycle of length at least $2k$ a *large component*, and otherwise a *small component*.

We distinguish these components because large components have long simple paths (length at least k) between every pair of their nodes, while we can show that small components have simple paths of length at most $4k^2$ (Lemma 6.7.9).

The crux of our argument is in the following lemma, which says that we can assume that the b-subpaths of solutions of $\mathsf{USimPath}(L)$ hit no large component and at most a constant number of small components.

Lemma 6.7.7. *Let L be a language definable by a regular expression of the form (†). If there exists a solution of $\mathsf{USimPath}(L)$, then there is a solution $p_1 p_1^a p_1^b p_2 p_2^a p_2^b \cdots p_\ell$, such that each p_i matches r_i, each p_i^a matches a_i^*, each p_i^b matches b_i^* and, furthermore, (1) no p_i^b hits a large component and (2) all p_i^b together hit at most $\ell(\ell + k)$ different small components.*

Our algorithm will therefore consider sets X that contain at most $\ell(\ell + k)$ different small components and consider subgraphs $G_A^{X,N}$ of G'_a through which we will search for disjoint a-paths. First we construct a set A_X, containing all the nodes x_i, y_i, all 1-cuts between x_i and y_i, all nodes of large components between x_i and y_i, and all nodes of small components $C \notin X$. For each small component $C \in X$, observe that, while the size of C is not necessarily constant,[7] each a-labeled subpath of a simple path that matches

[7]Take C with edges $(s, i), (i, t)$ with $i \in [n]$ for arbitrarily large n.

r traverses C at most once. As the number of maximal a-labeled subpaths of a path matching an expression of the form (†) is bounded by $(\ell - 1)$, they can traverse C at most $(\ell - 1)$ times. Since simple paths in C have length at most $4k^2$, there are at most $(\ell - 1)4k^2$ nodes per $C \in X$ that can be used by a-labeled subpaths of a simple path that matches r. As such, we can iterate over sets N of nodes of $\bigcup_{C \in X} C$, where $|N| \in O(\ell^3 k^3)$. For each such subset N, we consider the graph $G_A^{X,N}$ which is the subgraph of G'_a induced by $A_X \cup N$. The graph $G_B^{X,N}$ is the subgraph of G'_b induced by all nodes y_i, z_i (that is, start/end nodes of b-subpaths) and the nodes that are not in $G_A^{X,N}$. Our problem can now be solved by finding $\ell - 1$ node-disjoint paths in $G_A^{X,N}$ and $\ell - 1$ node-disjoint paths in $G_B^{X,N}$.

This concludes the outline of the proof. We will now give useful observations and lemmas. We first observe an important property of languages of the form (†). Intuitively, these languages are special because we can replace substrings with a "long enough" sequence of the symbol a. Let k be the length of the shortest word in $L(r)$.

Observation 6.7.8. *Let r be a regular expression of the form (†). Then each word $w \in L(r)$ can be written in the form $w_1 w_1^a w_1^b \cdots w_{\ell-1} w_{\ell-1}^a w_{\ell-1}^b w_\ell$ such that w_i matches r_i, w_i^a matches a_i^*, and w_i^b matches b_i^*. Let $x, y \in \{a, b\}$. If, for some $i, j \in \{1, \ldots, m\}$, $x_i = x$ and $y_j = y$ and there is an $h \in \{i, \ldots, j\}$ with $a_h^* = a^*$, then the word obtained by replacing the substring between w_i^x and w_j^y (and arbitrary parts of w_i^x and w_j^y) with any word in $a^{\geq k_{i,j}}$ where $k_{i,j}$ is the number of factors a between x_i^* and y_j^* in r, is in $L(r)$. Let k be the number of factors a in r. Since $k_{i,j} \leq k$ for every i, j, any word in $a^{\geq k}$ can be used.*

We now prove that the length of simple paths in 2-connected components without a simple cycle of length at least $2k$ is bounded.

Lemma 6.7.9. *For each 2-connected component C without a simple cycle of length at least $2k$ it holds that the length of the longest simple path between all pairs $v_1, v_2 \in C$ is at most $(2k)^2$.*

Proof. Let us assume towards contradiction that there exist two nodes s, t with a simple path p of length at least $(2k)^2 + 1$ between them. Due to the 2-connectedness and Menger's theorem, see Theorem 6.7.6, there exist two node-disjoint paths p_1, p_2 from s to t. If $|p_1| + |p_2| \geq 2k$, we find a simple cycle of length at least $2k$ by concatenating p_1 and p_2. Thus we can assume without loss of generality that $|p_1| + |p_2| < 2k$. This implies that $|p_1| < 2k$. We now prove that there exist nodes x, y in p_1 and p such that (1) $|p[x, y]| \geq 2k$ and (2) $p_1[x, y]$ and $p[x, y]$ are node-disjoint (up to x and y).

The proof is due to the lengths of p and p_1. Since p_1 and p are paths from s to t, they are not node-disjoint. And as p_1 has at most $2k$ nodes while p has length at least $(2k)^2 + 1$, there must be a subpath p' of p of length at least $2k$ which is node-disjoint from p_1 (up to its endpoints). If we choose a maximal such subpath, we can choose its endpoints as x and y.

Thus we obtain a simple cycle of length at least $2k$ by joining $p[x, y]$ and $p_1[x, y]$, which leads to a contradiction. □

Algorithm 4: Deciding USimPath(L) for L of the form (†)

```
Input: Undirected Multigraph $G = (V, E, \mathcal{E})$, nodes $z_0, x_\ell$
Output: "Yes" if there is a simple path from $z_0$ to $x_\ell$ in $G$ that matches $L$; "no" otherwise
1  $k \leftarrow$ length of the shortest word in $L$          ▷ This word is of the form $a^k$
2  foreach tuple of simple paths $\bar{p} = (p_1, \dots, p_\ell)$ matching $(r_1, \dots, r_\ell)$ do
3      $G' \leftarrow G$ without inner nodes of $(p_1, p_2, \dots, p_\ell)$ and their adjacent edges
           ▷ $x_i$, $z_j$ with $i, j \in [\ell - 1]$ are determined by $\bar{p}$
4      foreach tuple $(y_1, \dots, y_{\ell-1})$ consistent with $(a_i, b_i)_{i \in [\ell-1]}$ do
5          $A \leftarrow \emptyset$          ▷ Set of nodes for $a$-paths
6          $S \leftarrow \emptyset$          ▷ Set of small components
7          foreach $i \in [\ell - 1]$ do
8              foreach node $v$ in a 1-cut between $x_i$ and $y_i$ in $G_i$ do
9                  add $v$ to $A$
10             foreach large component $C$ between $x_i$ and $y_i$ in $G_i$ do
11                 add each node of $C$ to $A$
12             foreach small component $C$ between $x_i$ and $y_i$ in $G_i$ do
13                 add $C$ to $S$
14         foreach small component $C \in S$ do
15             $I_C \leftarrow \{i \mid C$ is a small component between $x_i$ and $y_i$ in $G_i\}$
16         foreach $X \subseteq S$ with $|X| \leq \ell \cdot (\ell + k)$ do
               ▷ We consider groups $X$ of $O(|r|^2)$ many components. For each
                 component $C \in X$ we iterate over all simple paths in $C$.
17             $A_X \leftarrow \emptyset$
18             foreach $C \in S - X$ do
19                 add each node of $C$ to $A_X$
20             $N \leftarrow \emptyset$
21             foreach set of $|I_C|$ disjoint simple paths between $(s^i_C, t^i_C)_{i \in I_C}$ with $C \in X$
               do
22                 add each node of these paths to $N$
                   ▷ The crux is that we do not add every node of $C$ to $N$, so that we can
                     use some $C$-nodes for $b$-paths.
23                 $G^{X,N}_A \leftarrow$ induced subgraph of $G_a$ on $(A \cup A_X \cup N) \cup (x_i, y_i)_{i \in [\ell-1]}$
24                 $G^{X,N}_B \leftarrow$ induced subgraph of $G_b$ on $(V' - (A \cup A_X \cup N)) \cup (y_i, z_i)_{i \in [\ell-1]}$.
25                 if there are node-disjoint simple paths between $(x_i, y_i)_{i \in [\ell-1]}$ in $G^{X,N}_A$
                   and there are node-disjoint simple paths from between $(y_i, z_i)_{i \in [\ell-1]}$ in
                   $G^{X,N}_B$ then
26                     return "yes"
27 return "no"
```

From this we easily obtain the following lemma, which we will need to bound the running time of our algorithm.

Lemma 6.7.10. *In a 2-connected component C with n nodes and without a simple cycle of length at least $2k$ there are at most $n^{(2k)^2}$ many simple paths with different node sequences between every pair of nodes $v_1, v_2 \in C$.*

Proof. As C has no simple cycle of length at least $2k$ and is 2-connected, the length of the longest path in C is at most $(2k)^2$, see Lemma 6.7.9. As in each step there are at most n choices for the next node, this yields at most $n^{(2k)^2}$ many simple paths with different node sequences from v_1 to v_2. □

On the other hand, we can show that 2-connected components with a simple cycle of length at least $2k$ will always have a path of length at least k between each node-pair.

Lemma 6.7.11. *In a 2-connected multigraph G with a simple cycle of length at least $2k$ there is a simple paths of length at least k between each pair of nodes $v_1, v_2 \in G$.*

Proof. Let C be a simple cycle of length at least $2k$. Let v_1, v_2 be two nodes in G. We perform a case distinction on the relation of v_1 and v_2 regarding C. Case(1): If they both are on C, we clearly have a simple path of length at least k between them—we can always choose the longer arc of C.

Case (2): If both are not on C, we show that there are nodes y_1, y_2 on C and two node-disjoint paths from v_1 to y_1 and from v_2 to y_2 which do not use nodes of C other than y_1 and y_2. Then we can again route via the longer arc of C to obtain a long path from v_1 to v_2. Let z_1 and z_2 be two nodes from C. Due to the 2-connectedness we find a simple path p_1 from v_1 to z_1. We can assume without loss of generality that z_1 is the first hit of p_1 with C (otherwise rechoose z_1). Due to the 2-connectedness, there are two node-disjoint paths from v_1 to z_2—so at least one of them avoids z_1. We name one of the paths that avoids z_1 p_2. Again, let z_2 be the first hit of p_2 with C (otherwise rechoose z_2). Since v_2 and z_2 are nodes in the 2-connected multigraph, there is also a simple path p from v_2 to z_2. We choose y_1 and y_2 depending on p. We illustrate the possible behavior of p in Figure 6.7. If p is node-disjoint with p_1 or meets a node in C before it hits p_1, we set $y_1 = z_1$ and choose y_2 as the first node of p which is in C (possibly z_2). Indeed, p_1 and $p[v_2, y_2]$ are node-disjoint and do not use nodes in C except y_1 and y_2 by construction. Otherwise, p uses a node of p_1 before it uses a node in C. We will re-route p via p_1 or p_2, depending on which is intersected first. Let x be the first node of p that is in p_1 or p_2. If x is in p_1, we set $y_1 = z_2$ and $y_2 = z_1$. We note that p_2 and $p[v_2, x)p_1[x, z_1]$ are node-disjoint by construction and do not use nodes in C except y_1 and y_2. Otherwise x is in p_2, and we set $y_1 = z_1$ and $y_2 = z_2$. Again, p_1 and $p[v_2, x)p_2[x, z_2]$ are node-disjoint by construction and do not use nodes in C except y_1 and y_2. This concludes case (2).

Case (3): It remains to consider the case where one of $\{v_1, v_2\}$ is on C and the other is not. Let without loss of generality $v_1 \in C$. Due to the 2-connectedness there exists a node $y \neq v_1, y \in C$ and two node-disjoint paths from v_2 to y. Since they are disjoint, only one can use v_1, so we route over the other and possibly rechoose y to the first hit

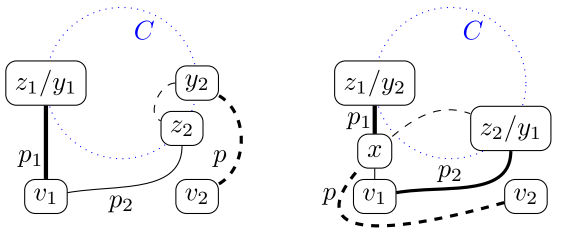


Figure 6.7: Illustration of case (2) in the proof of Lemma 6.7.11.

with C. Then we can use the path from v_2 to y and from y the long arc of C to v_1 to construct a simple path of length at least k. □

Before we continue, we need some more notation. We already defined 1-cuts and 2-connected components in Definition 6.7.5. For clarification and readability, we repeat some definitions given in the outline and add a few new ones. Let $z_0, x_1, y_1, z_1 \ldots, x_{\ell-1}, y_{\ell-1}, z_\ell$ be distinguished nodes in G. Let $(p_1, \ldots, p_\ell)$ be a tuple of constant-length node-disjoint simple paths from z_{i-1} to x_i that match the subexpressions of r_i, for each $i \in [\ell]$. Let G' be the induced multigraph obtained from G after removing the inner nodes of $(p_1, \ldots, p_\ell)$.

Let G'' be the induced multigraph obtained from G' after additionally removing the nodes z_i unless $z_i = x_{i+1}$ or $z_i = y_i$. (That is, we remove the start/end-nodes which do not belong to a_i-paths.)

We define G_i to be the induced subgraph of G''_a which contains x_i and y_i and, if $x_i \neq y_i$, all nodes which are on simple paths from x_i to y_i. If $x_i = y_i$, then G_i contains only the node x_i and no edges. We observe that G_i depends only on G, non-empty paths $(p_j)_{j\in[\ell]}$, x_i and y_i. For all empty paths, that is, if $p_j = \varepsilon$, we have $z_j = x_{j+1}$, thus empty paths p_j will not be the reason for any removed nodes.

Notice that in G_i the 1-cuts between x_i and y_i are totally ordered by their proximity to x_i. Furthermore, between every pair of such consecutive 1-cut nodes in G_i (or between x_i and the first 1-cut and between the last 1-cut and y_i), there either is nothing, or a 2-connected component. Thus, the graph G_i resembles a "string-of-beads", or a single bead if there is a 2-connected component containing x_i and y_i.

We name a 2-connected component C in G_i *large component* if it contains a simple cycle of length at least $2k$ and *small component* otherwise. On this "string of beads" from x_i to y_i, we name the leftmost node of a 2-connected component C s^i_C and the rightmost one t^i_C. All s^i_C and t^i_C are 1-cuts separating x_i and y_i (or x_i, y_i themselves) by construction. Note that the same 2-connected component C can be between different nodes x_i, y_i and x_j, y_j, therefore, there can be different nodes s^i_C,t^i_C and s^j_C,t^j_C for $i \neq j$.

We now have the ingredients to prove Lemma 6.7.7 which is restated here for readability:

Lemma 6.7.7. *Let L be a language definable by a regular expression of the form (†). If there exists a solution of $\mathsf{USimPath}(L)$, then there is a solution $p_1p^a_1p^b_1p_2p^a_2p^b_2 \cdots p_\ell$, such* *that each p_i matches r_i, each p_i^a matches a_i^*, each p_i^b matches b_i^* and, furthermore, (1) no p_i^b hits a large component and (2) all p_i^b together hit at most $\ell(\ell + k)$ different small components.*

Proof Sketch. We start with a solution p where the length of the p_i^bs is as short as possible. If p contradicts (1) or (2), we successively replace contradicting subpaths with long a-paths, which is allowed by Observation 6.7.8. We will find long a-paths in large components with Lemma 6.7.11 or, if many small components in a row are hit, by using a long a-path via the small components in the middle. Examples of reroutings are depicted in Figure 6.8. When replacing subpaths, the newly generated path π will have at least one b-edge less than p, thus contradicting the choice of p. □

Proof. Let $r_1 a_1^* b_1^* r_2 a_2^* b_2^* \cdots r_\ell$ be given and let p be a solution such that the length of the p_i^bs is as short as possible. Let $x_1, \dots, x_\ell$, $y_1, \dots, y_\ell, z_0, \dots, z_\ell$ be the respective nodes in p, such that each path from x_i to y_i is a a-path, each path from y_i to z_i is a b-path, and each path from z_{i-1} to x_i matches r_i.

We will show that restriction (1) is valid, that is, if there is solution p, then there is a solution satisfying (1). Let C be a large component between x_h and y_h, that is hit by some b-path. Let x be the first node in p that is also in C and y the last one. Lemma 6.7.11 ensures that there is an a-path of length at least k between all nodes in a large component. Let $\pi[x, y]$ be an a-path of length at least k from x to y in this large component. Then $\pi[x, y]$ uses at least one b-edge less than $p[x, y]$ since it avoids at least some part (at least the first or last edge) of the b-path that hits C. Let $\pi = p[s, x] \cdot \pi[x, y] \cdot p[y, t]$. The so constructed path π is a simple path by construction and matches L, see Observation 6.7.8. Furthermore, π contains less b-edges than p, contradicting the choice of p.

Let us now assume that (1) holds. We show that there is a solution for which (1) and (2) hold. So let us assume that all p_i^b together hit at least $\ell(\ell + k) + 1$ different small components. Then there exist x_h, y_h such that at least $\ell + k + 1$ small components between them are hit. By definition of p, x_h and y_h, in every small component C from x_h to y_h there is a path π_C^h from s_C^h to t_C^h which only uses nodes in p_h^a.

We now observe that each p_i^a with $i \neq h$ can hit at most one small component from x_h to y_h by definition of 2-connected component.[8] Thus there are at least $k + 2$ small components that are hit by some p_i^b path, but not by any p_j^a with $h \neq j$. We name the second such component, C_x, and an other such component which is at least $k - 1$ small components closer to y_h, C_y. We note that this implies that $p[t_{C_x}^h, s_{C_y}^h]$ is a subpath of p_h^a of length at least k.

Since C_x is only used by b-paths and p_h^a, in C_x there must exist either

(a) a node x in $p[s, x_h]$ and in C which is part of a b-path and such that there exists an a-path in C_x from x to $t_{C_x}^h$ not using any inner nodes in p_i^b for any i, or

[8]Let $i \in \{1, \dots, \ell - 1\}$ with $i \neq h$. Since p is a simple path, p_i^a and p_h^a are disjoint (up to one of their endpoints). If p_i^a hits two different 2-connected components between x_h and y_h, there would be two disjoint paths between those components. Thus there could not be a 1-cut between these components, and therefore be the same 2-connected component, see Definition 6.7.5.

(b) a node x' in $p[y_h, t]$ and in C which is part of a b-path and such that there exists an a-path in C_x from x' to $t^h_{C_x}$ not using any inner nodes in p^b_i for any i.

Analogously for C_y, there must exist a node y in $p[s, x_h]$ or a node y' in $p[y_h, t]$ which is part of a b-path and such that there exists an a-path from y to $s^h_{C_y}$ in C_y (or from y' to $s^h_{C_y}$, respectively) not using any inner nodes in p^b_i for any i.

We perform a case-distinction, depending on which of the nodes x, x', y, y' exist. By definition of C_x, either x or x' has to exist and by definition of C_y, either y or y' has to exist.

- If x exists, we can reroute as in Figure 6.8(top), that is, we obtain the simple path

  $$\pi = p[s, x] \cdot \pi_1 \cdot p[t^h_{C_x}, t],$$

  where π_1 is the a-path from y to $t^h_{C_x}$ which does exist by definition of y and is therefore node-disjoint from $p[s, x]$ and $p[t^h_{C_x}, t]$ (up to y to $t^h_{C_x}$).

- If y' exists, we can reroute as in Figure 6.8(mid), that is, we obtain the simple path

  $$\pi = p[s, s^h_{C_y}] \cdot \pi_1 \cdot p[y', t],$$

  where π_1 is the a-path from to $s^h_{C_y}$ to y' which exists by definition of y' therefore node-disjoint from $p[s, s^h_{C_y}]$ and $p[y', t]$ (up to $s^h_{C_y}$ and y').

- If x' and y exist, we can reroute as in Figure 6.8(bottom), that is, we obtain the simple path

  $$\pi = p[s, y] \cdot \pi_1 \cdot (p[t^h_{C_x}, s^h_{C_y}])^{\mathsf{rev}} \cdot \pi_2 \cdot p[x', t],$$

  where π_1 and π_2 exist by definition of x' and y and are node-disjoint from each other and from $p[s, y], p[t^h_{C_x}, s^h_{C_y}]$, and $p[x', t]$ (up to start/end-nodes). We note that $(p[t^h_{C_x}, s^h_{C_y}])^{\mathsf{rev}}$ is the subpath $p[t^h_{C_x}, s^h_{C_y}]$ read from right to left.

We note that in all cases, Observation 6.7.8 ensures that π matches L. Furthermore, x, x', y, y' are inner nodes of b-paths. Thus, in each rerouting, we omitted at least the first or last b-edge of at least one of these b-paths (namely one of the b-paths through x, x', y, or y'). This implies that each π is a solution and has at least one b-edge less than p, contradicting the choice of p. □

With this we can finally prove the following lemma, which implies Theorem 6.7.3(b).

Lemma 6.7.12. *Algorithm 4 works correctly and in polynomial time for fixed languages.*

Proof. We first explain why Algorithm 4 is in P:

- The paths $(p_1, \dots, p_\ell)$ have constant length and thus all possibilities can be enumerated in line 2 in polynomial time.

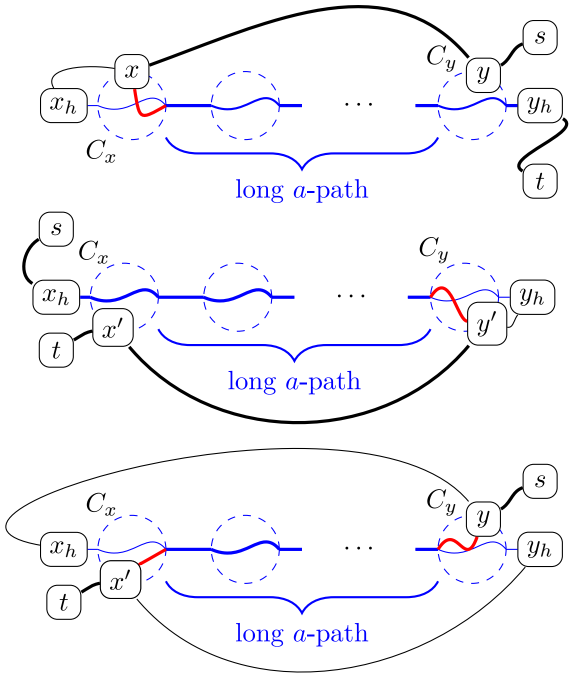


Figure 6.8: Three example reroutings in Lemma 6.7.7. The dashed circles depict small components, and the red edges exist by definition of x, x', y, or y'. The new paths follows the thick edges from s to t.

- One can iterate over all possible tuples $(y_1, \ldots, y_{\ell-1})$ consistent with $(a_i, b_i)_{i \in [\ell-1]}$ in line 4 in polynomial time. Indeed, we can enumerate all possible tuples with $x_i = y_i$ if $a_i = \varepsilon$ and with $y_i = z_i$ if $b_i = \varepsilon$ (which is exactly the definition of consistency).

- Given $(p_1, \ldots, p_\ell)$, x_i, y_i, one can construct G_i in polynomial time: If $x_i = y_i$, it is only a single node. Otherwise, if $x_i \neq y_i$, we start with G_a and remove some nodes depending on $(p_1, \ldots, p_\ell)$. More precisely, we remove all inner nodes of $(p_1, \ldots, p_\ell)$ and all end-nodes if they do not coincide with $(x_i, y_i)_{i \in [\ell-1]}$. We can then determine all nodes on simple paths from x_i to y_i by adding an edge from x_i to y_i (if it does not already exist) and using the polynomial time algorithm of Hopcroft and Tarjan [118] to determine (all) biconnected components in this multigraph.

- The test whether v is in a 1-cut between x_i and y_i in line 8 can be done by testing reachability from s to t in G_i and in $G_i - \{v\}$.

- To test in line 10 if C is a large component, it suffices to test if C has a simple cycle of length $2k$ as a minor.
- $\ell \cdot (\ell + k)$ is a constant, and one can iterate over all possible subsets of constant size in polynomial time in line 16.
- For each small component C, there are only polynomially many simple paths with different node sequences from s^i_C to t^i_C for each i, see Lemma 6.7.10. Since $|I_C| \leq \ell$, we can iterate over all polynomially many choices of paths with different node sequences in line 21.
- One can test for ℓ node-disjoint paths in line 25 in polynomial time, see Proposition 6.2.1.

We now prove correctness: If Algorithm 4 answers “yes”, we can construct a solution in an obvious way.

For the other direction, let us assume there is a solution to $\mathsf{USimPath}(L)$. Then Lemma 6.7.7 guarantees that there exists a solution $p = p_1 p^a_1 p^b_1 p_2 p^a_2 p^b_2 \cdots p_\ell$ such that p_i matches r_i, each p^a_i matches a^*_i, each p^b_i matches b^*_i and, furthermore, (1) no p^b_i hits a large component and (2) all p^b_i together hit at most $\ell(\ell + k)$ different small components, where k is the length of the shortest path in L.

Since the algorithm checks for each possible combination of paths $(p_1, \ldots, p_\ell)$ and nodes $(y_i)_{i \in [\ell-1]}$ if a solution of this form exists, it will return the correct result. □

This concludes the proof of Theorem 6.7.3(b).

A natural question is now if the algorithms in Theorem 6.7.3 can be combined to show that $\mathsf{USimPath}(L)$ is in P for every language definable by $\mathsf{SCRE}(a, a?, a^*, b, b?, b^*)$. This, however, is not the case, even for $\mathsf{SCRE}(a, a^*, b, b^*)$. The following can be obtained by using $G_{3\mathrm{SAT}}$ with $w_\mathsf{s} = \varepsilon, w_\mathsf{b} = b, w_\mathsf{o} = \varepsilon, w_\mathsf{r} = aa, w_\mathsf{m} = ab, w_\mathsf{t} = \varepsilon$.

Proposition 6.7.13. *$\mathsf{USimPath}(b^*abb^*a^*)$ is NP-complete.*

Proof. $\mathsf{USimPath}(b^*abb^*a^*)$ is trivially in NP. To prove NP hardness, we use $G_{3\mathrm{SAT}}$ from Construction 6.4.1 with $w_\mathsf{s} = \varepsilon, w_\mathsf{b} = b, w_\mathsf{o} = \varepsilon, w_\mathsf{r} = aa, w_\mathsf{m} = ab, w_\mathsf{t} = \varepsilon$. Since the “red edges” are labeled aa, and the w_m-path is the only occurrence of a single a followed by b, every simple path from s to t which matches $b^*abb^*a^*$ has to read the w_m-path before reading a red edge. Thus NP hardness follows from Theorem 6.4.2. □

On the other hand, it is also not the case that *every* language that uses all the factors a, a^*, b, b^* is NP-hard. An obvious example is a^+b^+, and a more intriguing one is summarized in the following theorem.

Theorem 6.7.14. *$\mathsf{USimPath}(a^*b^+a^+b^*)$ is in P.*

Proof Sketch. We reduce the problem to two calls to the 2 node-disjoint paths problem. We iterate over all triples (x_1, x_2, x_3) of nodes. For each such triple, we compute two sets of nodes A and B. The former set should be avoided by b-paths and the latter by

a-paths. We add each 1-cut node between s and x_1 in G_a to A and each 1-cut node between x_3 and t in G_b to B. Intuitively, if a path p labeled $a^*b^+a^+b^*$ is supposed to have label alternations at x_1, x_2, and x_3, then the nodes in A (respectively, B) need to be used by a-labeled (respectively, b-labeled) subpaths of p. If A and B intersect, we move to the next triple (x_1, x_2, x_3). The algorithm tests if, for some triple, $G_a - B$ has two node-disjoint paths between (s, x_1) and (x_2, x_3) and $G_b - A$ has two node-disjoint paths between (x_1, x_2) and (x_3, t). Correctness is non-trivial. ☐

Proof. We prove in Lemma 6.7.20 that this problem can be solved with the algorithm in the proof sketch, which is also depicted as Algorithm 5. ☐

In order to prove Theorem 6.7.14, we show that Algorithm 5 solves USimPath($a^*b^+a^+b^*$) and is in P. Since the algorithm starts with enumerating nodes x_1, x_2, x_3, we assume in the following lemma that G is an undirected multigraph, and x_1, x_2, x_3 are nodes in G.

We start with some notation. Let 1-cut and 2-connected component be as defined in Definition 6.7.5. By *2-connected components of G_a from s to x_1* we refer to the 2-connected components in the subgraph of G_a induced by the nodes of simple paths from s to x_1, and by *2-connected components of G_b from x_3 to t*, we refer to the 2-connected components in the subgraph of G_b induced by the nodes of simple paths from x_3 to t. (Note that a 2-connected component of G_a could contain t.) If $s = x_1$ there are no 2-connected components in G_a from s to x_1, and if $x_3 = t$ there are no 2-connected components in G_b from x_3 to t. Let A be the set of all nodes which are 1-cuts between s and x_1 in G_a and let B be a set disjoint from A which stores all 1-cuts between x_3 and t in G_b.

We say that a path p *touches* a 2-connected component C if a node of p is in C (note that this can also refer to the start or end-node of p).

Given x_1, x_2, x_3, by p_{a1} we will always denote an a-path from s to x_1, by p_{b1} a b-path from x_1 to x_2, by p_{a2} an a-path from x_2 to x_3, and by p_{b2} a b-path from x_3 to t.

The following observation is very important for our algorithm:

Observation 6.7.15. *If there exist x_1, x_2, x_3 such that there exist 2-disjoint a-paths from s to x_1 and from x_2 to x_3, and there exist 2-disjoint b-paths from x_1 to x_2 and from x_3 to t, and the paths s to x_1 and from x_3 to t do not intersect, then there is a simple path matching $a^*b^+a^+b^*$ from s to t.*

Therefore, we will focus mostly on the paths p_{a1} and p_{b2}.

Lemma 6.7.16. *Let C_1 be 2-connected component of G_a from s to x_1 and p be a simple b-path ending in t in $G_b - A$ which*

(1) touches C_1 in at least 3 nodes, or

(2) touches C_1 and another 2-connected component C_2 of G_a from s to x_1 both at least twice.

Then there is a simple path matching $a^*b^+a^+b^*$ from s to t.

Proof. Let the b-path p be fixed. Let C_1 be the first 2-connected component of G_a from s to x_1 that is touched at least twice. That is, we choose C_1 as close to s as possible. We perform a case distinction on whether C_1 is touched twice or more often. Case 1: C_1 is touched at least three times. If there are more touch points, we choose v_1, v_2, v_3 such that the b-path between v_1 and v_2, and the b-path from v_3 to t does not touch any other node in C_1. By definition of C_1, every component before C_1 is touched at most once, thus we will find an a-path from s to C_1 which is node-disjoint from the b-path (as the b-path does not use nodes in A and touches every component before C_1 at most once, it follows from the definition of components).

Let now s_{C_1} be the first node in C_1 which is seen by the a-path from s to x_1. (Note: s_{C_1} is unique.) We add a new node s' connected to s_{C_1} and v_3 and a new node t' connected to v_1 and v_2. Clearly, s' and t' belong to the component, so there are two node-disjoint paths from s' to t' due to Menger's theorem, see Theorem 6.7.6. Thus there are two node-disjoint paths, one from s_C to v_1 or v_2 and one from v_3 to the other node (v_2 or v_1) in G_a.

As there are b-paths from v_1 to v_2 and from v_3 to t, and those are undirected, we can combine them with the node-disjoint a-paths and the a-path from s to s_C to obtain a solution.

We now turn to case 2, in which C_1 is touched exactly twice. Then there is another 2-connected component of G_a from s to x_1 that is touched at least twice. Let C_2, with $C_1 \neq C_2$, be the next 2-connected component of G_a from s to x_1 that is touched at least twice.

Since C_1 and C_2 are the 2-connected components closest to s which are touched at least twice, we can find an a-path from s to C_1 and one from C_1 to C_2. Especially, these a-paths can be chosen such that they are node-disjoint with the b-path because the b-path does not use nodes in A and by definition of 2-connectedness. Let v be the last touch point of the b-path in C_1 or C_2, closest to t. Since v leads to t, we must use it last. If v is in C_2, we can use the b-path in C_1 (it must exist, because C_1 is touched exactly twice), and then use an a-path to v in C_2. So we can assume without loss of generality that, $v \in C_1$. Let u be the other touch point in C_1. We name the first node of C_1 in G_a s_1 and the last node of C_1 in G_a t_1. (Note that $s_1, t_1 \in A$ by definition of 2-connected component and 1-cut.) As we have a 2-connected component, we can conclude with Menger's theorem, see Theorem 6.7.6, that there are two node-disjoint paths, one from s_1 to t_1 or u, and one from v to u or t_1. More precisely, we add a new node s' connected to s_1 and v and a new node t' connected to u and t_1. Then Menger's theorem implies two node-disjoint paths from s' to t' in G_a. We explain how to construct a simple path matching $a^*b^+a^+b^*$ from s_1 to v: If there exist node-disjoint paths from s_1 to u and from t_1 to v, we can use the first one, then the b-path to the first node in C_2, and from that node in C_2 we can use a path to t_1 and from there the node-disjoint one to v. Otherwise, we have node-disjoint paths from s to t_1 and from u to v. We first use the path from s to t_1 to go to the second component. As C_2 is touched by the b-path, we go to one of these nodes and use the b-path from there to u, then the a-path from u to v.

We can then construct a solution by adding the a-path from s to s_1 and the b-path from v to t. The b-paths are node-disjoint since p was a simple b-path, and the a-paths are node-disjoint by definition of components. This concludes the proof. □

By symmetry, the next result follows immediately for components of G_b:

Lemma 6.7.17. *Let C_1 be 2-connected component of G_b from x_3 to t and p be a simple a-path starting in s in $G_a - B$ which*

(1) touches C_1 in at least 3 nodes, or

(2) touches C_1 and another 2-connected component C_2 of G_b from x_3 to t both at least twice.

Then there is a simple path matching $a^*b^+a^+b^*$ from s to t.

We now prove that if Algorithm 5 returns "yes", then we find paths $p_{a1}, p_{a2}, p_{b1}, p_{b2}$ such that if p_{a1} and p_{b2} intersect, then p_{b2} touches each component of G_a in which they intersect at least twice. For a set of nodes S, we call a path *S-avoiding* if none of the edges in the path uses a node in S.

Lemma 6.7.18. *If there are*

- *node-disjoint B-avoiding a-paths p_{a1} from s to x_1 and p_{a2} from x_2 to x_3 and*
- *node-disjoint A-avoiding b-paths p_{b1} from x_1 to x_2 and p_{b2} from x_3 to t,*

then there exist nodes y_1, y_2, y_3 and

- *node-disjoint B-avoiding a-paths p'_{a1} from s to y_1 and p'_{a2} from y_2 to y_3 and*
- *node-disjoint A-avoiding b-paths p'_{b1} from y_1 to y_2 and p'_{b2} from y_3 to t,*

such that, additionally, p'_{a1} is node-disjoint from p'_{b2} in every component in G_a from s to y_1 that p'_{b2} touches at most once.

Proof Sketch. If p_{a1} and p_{b2} intersect in a 2-connected component of G_a, then we can re-route p_{a1} because of the 2-connectedness. If p_{b2} touches this component only once, then the re-routed subpath of p_{a1} is node-disjoint from p_{b2}, but the re-routed path might not be node-disjoint with p_{b1} or p_{a2}. We explain how to reroute in these cases (possibly changing x_1, x_2, x_3). An example is shown in Figure 6.9. □

Proof. Firstly, we can assume without loss of generality that all paths but p_{a1} and p_{b2} are pairwise node-disjoint (otherwise we shortcut and update x_1, x_2, x_3, A, and B accordingly).

Let C be the component of G_a from s to x_1 that is closest to s such that p_{a1} and p_{b2} intersect and p_{b2} touches C at most once. We name the start and end of C in G_a s_C and t_C, respectively. We name the intersection point of p_{a1} and p_{b2} in C x.

Let p' be a rerouting of p_{a1} in C which avoids x. Since p_{b2} touches C at most once, p' is node-disjoint from p_{b2}. If it is node-disjoint from p_{b1} and p_{a2}, we can continue with the next component contradicting the lemma. Otherwise, we perform a case distinction depending on whether p_{b1} (case 1) or p_{a2} (case 2) intersects with p' first.

Case 1: Let v denote the first intersection of p' with p_{b1}. We can then shortcut to the simple path $p'[s, v] \cdot p_{b1}[v, x_2] \cdot p_{a2} \cdot p_{b2}$. As there is no 2-connected component after C in G_a, we are done. The result follows with $y_1 = v, y_2 = x_2, y_3 = x_3$, $p'_{a1} = p'[s, v]$, $p'_{b1} = p_{b1}[v, x_2]$, $p'_{a2} = p_{a2}$ and $p'_{b2} = p_{b2}$.

Case 2: We show that we either find an alternative a-path which does not touch p_{a2}, or such that we can reroute as in Figure 6.9. For this rerouting to work we have to show that there is an a-path p'' from s_C to the intersection with p_{a2} which is node-disjoint from $p_{a1}[x, t_C]$:

So let us assume that p' and $p[x, t_C]$ are not node-disjoint. We show how to find a better choice p''. Since p' and $p[x, t_C]$ are not node-disjoint, there exists some node u in p' and $p[x, t_C]$, such that $p'[s_C, u]$ and $p[x, t_C]$ share only the node u. If $p'[s_C, u]$ is disjoint from p_{a2}, we are done with this specific component C, since we can use the path $p'' = p'[s_C, u] \cdot p_{a1}[u, t_C]$ which is disjoint from p_{a2} to reroute in C, taking care of touches from the p_{b1} path as in case 1 if necessary.

If $p'[s_C, u]$ is not disjoint from p_{a2}, then we can reroute as depicted in Figure 6.9. Let y be the node where $p'[s_C, v]$ first touches p_{a2}. Note that $y \neq s_C$ and $y \neq u$ since p_{a1} and p_{a2} are disjoint by assumption. We can choose $y_1 = x_2, y_2 = x_1, y_3 = x$ and

$$\begin{aligned}
p'_{a1} &= p_{a1}[s, s_C] \cdot p'[s_C, y] \cdot (p_{a_2}[x_2, y])^{\mathsf{rev}}, \\
p'_{b1} &= (p_{b1})^{\mathsf{rev}}, \\
p'_{a2} &= (p_{a1}[x, x_1])^{\mathsf{rev}}, \text{ and} \\
p'_{b2} &= p_{b2}[x, t].
\end{aligned}$$

Here, p^{rev} denotes the path p read from right to left.

By construction, the paths p'_{a1}, p'_{a2}, and p'_{b1} are node-disjoint. Furthermore, since p_{a2} and p_{b2} were node-disjoint, we do not have any more intersections of p'_{a1} and p'_{b2} after C. Since C was the component closest to s violating the lemma and our new paths starting from C, that is, $p'_{a1}[s_C, y_1]$ and p_{b2} are node-disjoint, the result follows. ☐

Combining the lemmas so far, and rerouting again if necessary, we obtain:

Lemma 6.7.19. *If there are*

- *node-disjoint B-avoiding a-paths p_{a1} from s to x_1 and p_{a2} from x_2 to x_3 and*
- *node-disjoint A-avoiding b-paths p_{b1} from x_1 to x_2 and p_{b2} from x_3 to t,*

then there exist nodes y_1, y_2, y_3 and

- *node-disjoint B-avoiding a-paths p'_{a1} from s to y_1 and p'_{a2} from y_2 to y_3 and*
- *node-disjoint A-avoiding b-paths p'_{b1} from y_1 to y_2 and p'_{b2} from y_3 to t,*

Algorithm 5: Deciding USimPath($a^*b^+a^+b^*$)

```
   Input: Undirected multigraph $G = (V, E, uedge)$, nodes $s, t$
   Output: "Yes" if there is a simple path from $s$ to $t$ in $G$ that matches
           $a^*b^+a^+b^*$; "no" otherwise
1  forall nodes $x_1, x_2, x_3$ do
2      $A \leftarrow \emptyset$                                  ▷ nodes exclusively for $a$-paths
3      $B \leftarrow \emptyset$                                  ▷ nodes exclusively for $b$-paths
4      foreach 1-cut $v$ between $s$ and $x_1$ in $G_a$ do
5          $A \leftarrow A \cup \{v\}$
6      foreach 1-cut $v$ between $x_3$ and $t$ in $G_b$ do
7          if $v \in A$ then
8              continue with next triple of $x_1, x_2, x_3$
9          $B \leftarrow B \cup \{v\}$
10     if there exist two node-disjoint a-paths between $(s, x_1)$ and $(x_2, x_3)$ that do not
       use nodes in $B$, and there exist two node-disjoint b-paths between $(x_1, x_2)$ and
       $(x_3, t)$ that do not use nodes in $A$ then
11         return "yes"
12 return "no"
```

such that additionally p'_{a1} and p'_{b2} share at most one node.

Proof. Using Lemma 6.7.18 we know that we can find paths p_{a1}, p_{a2}, p_{b1}, p_{b2} such that p_{a1} and p_{b2} can only intersect in components which are touched more than once by p_{b2}. By Lemma 6.7.16 at most one such component can exist. (Otherwise there is a solution, and a solution implies node-disjoint simple paths p_{a1} and p_{b2}.) So we only need to show that if p_{a1} and p_{b2} meet in an component C of G_a from s to x_1 more than once, then we can reroute in C_1. Again by Lemma 6.7.16, if p_{a1} and p_{b2} intersect (at least) three times in C, we are done (as then p_{b2} touches C at least three times, so there is a solution and a solution implies node-disjoint simple paths p_{a1} and p_{b2}).

So let x and x' be the intersections of p_{a1} and p_{b2} in C. For the proof, we will reroute the a-path in C_1 in order to not use x'. If the rerouted path p'_{a1} touches p_{b2} in any other node than x, we are done by Lemma 6.7.16. If the rerouted path touches p_{b1} or p_{a2}, we reroute as in Lemma 6.7.18 to ensure that p'_{a1} is node-disjoint from both p_{b1} and p_{a2}. This completes the proof. □

In Lemma 6.7.20 we can thus focus on paths where p_{a1} and p_{b2} share at most a single node.

Lemma 6.7.20. *Algorithm 5 is correct.*

Proof. If a solution exists, the algorithm will clearly return "yes". So it remains to show: If the algorithm returns "yes", then there exists a solution. By Lemma 6.7.19 we can then find such paths where p_{a1} and p_{b2} intersect in at most one node x. Since p_{a1} does not use nodes in B and p_{b2} does not use nodes in A, we have $x \notin A \wedge B$. This implies that there is a 2-connected component C_1 of G_a from s to x_1 with $x \in C_1$ and a 2-connected component C_2 of G_b from x_3 to t with $x \in C_2$.

Let now p'_{a1} be a rerouting of p_{a1} in C_1 and p'_{b2} be a rerouting of p_{b2} in C_2. Let x'_a be the intersection of p'_{a1} with p_{b2} and x'_b be the intersection of p_{a1} with p'_{b2}. (These nodes x'_a and x'_b must exist, since otherwise we would be done.) Note that $x'_a \neq x'_b$ since otherwise p_{a1} and p_{b2} would both contain the node $x'_a = x'_b$, which contradicts our choice of p_{a1} and p_{b2} (by Lemma 6.7.19 they share at most one node, which is x.) Furthermore, since $x'_a \in C_1$ und $x'_b \in C_2$, the intersections must be unique—otherwise we can use Lemma 6.7.16 or 6.7.17 to ensure that there is a solution. Using the same argument, it follows that $x'_a \notin C_2$ and $x'_b \notin C_1$. So the touch points must either be “before” or “after” C_1 and C_2.

Thus, those intersections then look (up to symmetry, that is, choice of s and t) like in Figure 6.10. We note that $s_{C_1}, t_{C_1}, s_{C_2}$, and t_{C_2} depend on s and t and have therefore been renamed $u_{C_1}, v_{C_1}, u_{C_2}$, and v_{C_2} where $u_{C_1} = s_{C_1}$ if $s = s_1$, and $u_{C_1} = t_{C_1}$ otherwise. Analogously, $u_{C_2} = s_{C_2}$ if $t = t_2$, and $u_{C_2} = t_{C_2}$ otherwise. We note that reroutings in C_1 or C_2 are allowed to use nodes in A and B. Yet we can find for each choice of $t \in \{t_1, t_2\}$ and $s \in \{s_1, s_2\}$ a a simple path matching $a * b + a + b*$ from s to t in Figure 6.10 as follows:

- If $s = s_1, t = t_1$, then a possible solution in Figure 6.10 is a concatenation of an a-path from s_1 to x, a b-path from x via v_{C_2} to x'_b, an a-path from x'_b to x'_a, and the b-path from x'_a to t_1. More formally, we use

$$p[s,x] \cdot (p[s_{C_2},x])^{\mathsf{rev}} \cdot p'_{b2}[s_{C_2},x'_b] \cdot (p_{a1}[t_{C_1},x'_b])^{\mathsf{rev}} \cdot (p'_{a1}[x'_a,t_{C_1}])^{\mathsf{rev}} \cdot p_{b2}[x'_a,t]\,.$$

- If $s = s_1, t = t_2$, then a possible solution in Figure 6.10 is a concatenation of an a-path from s_2 to x'_b, a b-path from x'_b via v_{C_2} to x, an a-path from x to x'_a, and an b-path from x'_a to t_1. More formally, we use

$$p[s,x] \cdot (p[x'_a,x])^{\mathsf{rev}} \cdot p'_{a1}[x'_a,t_{C_1}] \cdot p_{a1}[t_{C_1},x'_b] \cdot p'_{b2}[x'_b,t_{C_2}] \cdot p_{b2}[t_{C_2},t]\,.$$

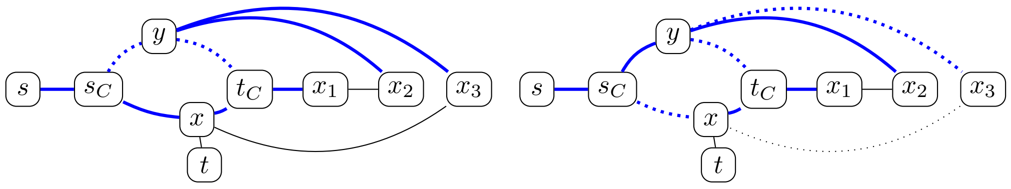
- If $s = s_2, t = t_1$, then a possible solution in Figure 6.10 is a concatenation of an a-path from s_2 to x'_b, an b-path from x'_b via v_{C_2} to x, an a-path from x to x'_a, and a b-path from x'_a to t. More formally, we use

$$p_{a1}[s,x'_b] \cdot (p_{b2}[s_{C_2},x'_b])^{\mathsf{rev}} \cdot p_{b2}[s_{C_2},x] \cdot p_{a1}[x,t_{C_1}] \cdot (p'_{a1}[x'_a,t_{C_1}])^{\mathsf{rev}} \cdot p_{b2}[x'_a,t]\,.$$

- If $s = s_2, t = t_2$, then a possible solution in Figure 6.10 is a concatenation of an a-path from s_2 to x'_b, a b-path from x'_b via u_{C_2} to x'_a, an a-path from x'_a to x, and an b-path from x to t_2. More formally, we use

$$p_{a1}[s,x'_b] \cdot (p_{b2}[s_{C_2},x'_b])^{\mathsf{rev}} \cdot (p_{b2}[x'_a,s_{C_2}]^{\mathsf{rev}} \cdot p'_{a1}[x'_a,t_{C_1}] \cdot (p_{a1}[x,t_{C_1}])^{\mathsf{rev}} \cdot p_{b2}[x,t]\,.$$

We especially note that since the paths between s_1 and u_{C_1}, t_1 and x'_a, x'_b and s_2, and v_{C_2} and t_2 are pairwise node-disjoint, the solution is independent of paths not depicted in Figure 6.10 (such as p_{b1} and p_{a2}). □



(a) Original paths (b) rerouting p_{a1} leads to a rerouting of p_{a2}.

Figure 6.9: Possible rerouting in Lemma 6.7.18. The a-paths are blue and thick. We show that if the paths p_{a1} and p_{b2} touch each other at most once in a component, then we can reroute, even if this means rerouting p_{a2}.

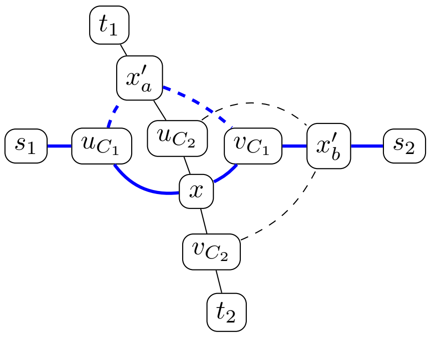


Figure 6.10: The paths p_{a1} and p_{b2} with reroutings intersecting one another. Note that $s \in \{s_1, s_2\}$, $t \in \{t_1, t_2\}$, $\{u_{C_1}, v_{C_1}\} = \{s_{C_1}, t_{C_1}\}$, and $\{u_{C_2}, v_{C_2}\} = \{s_{C_2}, t_{C_2}\}$. The a-path is blue and thick, the b-path is black. The paths only intersect in the depicted nodes, the reroutings of p_{a1} and p_{b2} are dashed.

We note that we see no "easy" way to extend Theorem 6.7.14 to $\mathsf{USimPath}(a^*b^{\geq k}a^+b^*)$, because the re-routing arguments of Lemmas 6.7.16 and 6.7.17 do not work for $k \geq 2$.

More than Two Alphabet Symbols

Theorem 6.7.3 generalizes in the following sense to larger alphabets, with only minor changes to the proof.

Theorem 6.7.21. *$\mathsf{USimPath}(L)$ is in P*

(a) for every language definable by an $\mathsf{SCRE}(a, a?, a^*, \star, \star?)$ and

(b) for every language definable by an $\mathsf{SCRE}(a, a?, a^*, b?, b^*, \star?)$.

Proof. The proof is closely related to the proof of Theorem 6.7.3: The only changes are to use in (a) $r_i \in \mathsf{SCRE}(a, a, \star, \star?)$ instead of $r_i \in \mathsf{SCRE}(a, a?, b, b?)$, and in (b) $r_i \in \mathsf{SCRE}(a, a?, b?, \star?)$ instead of $r_i \in \mathsf{SCRE}(a, a?, b?)$. □

Whereas $\mathsf{USimPath}(a^*ba^*)$ is tractable (Theorem 6.7.3), it follows from Theorem 6.5.2 that the following closely related language is intractable.

Corollary 6.7.22. *$\mathsf{USimPath}(a^*bc^*)$ is NP-complete.*

This implies that $\mathsf{SCRE}(a^*, b, c^*)$ can define languages for which $\mathsf{USimPath}(L)$ is NP-hard and therefore shows that the SCREs used in Theorem 6.7.21 cannot be further extended without introducing languages for which USimPath becomes intractable.

6.8 Parity Languages

We now discuss another interesting difference between directed and undirected multigraphs. Whereas Mod-2-path is NP-complete for directed graphs [140], the problem is in P for undirected multigraphs [140]. We generalize this tractability result to a wide class of languages involving parity tests.

Assume that $\Sigma = \{a_1, \dots, a_\ell\}$. The *Parikh vector* of a word w is defined as $p(w) = (|w|_{a_1}, \dots, |w|_{a_\ell})$, where $|w|_{a_i}$ is the number of occurrences of the label a_i in w. The *Parikh image* of a language L is the set $\{p(w) \mid w \in L\}$. A *parity set* is a semi-linear set of the form $\{v_1 + v_2 n \mid n \in \mathbb{N}, v_1 \in V_1\}$, where $V_1 \subseteq \{0,1\}^\ell$ and $v_2 = (2, \dots, 2) \in \{2\}^\ell$. A *parity language* is a language for which its Parikh image is a parity set. Every such language is regular.

Theorem 6.8.1. *$\mathsf{UTrail}(L)$ and $\mathsf{USimPath}(L)$ are in P for every parity language L.*

As the proof of this theorem relies on the minor theorem on group-labeled graphs [119], we start with some background: Let Γ be a finite abelian group. A Γ-labeled graph is a directed graph whose edges are labeled with a group-label in Γ. The group-label of an edge e is denoted by $\gamma_G(e)$. Following an edge in its direction adds the value $\gamma_G(e)$, while following it in the reverse direction adds the value $-\gamma_G(e)$. The *group-value of a path* is the sum of the values of its edges. A Γ-labeled graph $H = (V_H, E_H, \mathcal{E}_H)$ is a minor of a Γ-labeled graph $G = (V_G, E_G, \mathcal{E}_G)$ if and only if $V_H \subseteq V_G$ and for each edge e in H, there exists a simple path p_e with value $\gamma_H(e)$ from origin(e) to destination(e) in G. Furthermore, except for their first and last node, all p_e are pairwise node-disjoint.

Huynh [119] proves that for any fixed Γ-labeled graph H, there is a polynomial time algorithm which determines if an input Γ-labeled graph G contains a minor isomorphic to H.

We want to use this to solve some group-labeled variant of the kDisjointPaths problem, that is, given a group-labeled graph G, pairs of nodes $(s_i, t_i)_{i \in [k]}$ and values $(\gamma_i)_{i \in [k]}$, we want to know if there are simple paths p_i from s_i to t_i with group value γ_i in G

such that the p_i are pairwise node-disjoint. However, Kawase et al. [135, Footnote 3] observed that the reduction in [119] cannot distinguish between two paths, one from s_j to t_j and one from s_i to t_i, and two paths, one from s_i to s_j and one from t_i to t_j for any distinct i and j. Angela Bonifati and Guillaume Bagan [18] pointed out that this can be fixed by considering $\Gamma \times \mathbb{Z}_3 \times \cdots \times \mathbb{Z}_3$-labeled graphs instead. In these graphs, every edge label can be interpreted as a vector of length $k+1$. Let G' be the graph obtained from G by relabeling all edges e with $(\gamma_G(e), 0, \ldots, 0)$ and adding $2k$ new nodes $(s'_j, t'_j)_{j \in [k]}$ and edges from s'_i to s_i and from t_i to t'_i which are labeled with the e^{th}_{i+1} unit vector over $\mathbb{Z}_{k+1}$, that is, $(0, \ldots, 0, 1, 0, \ldots, 0)$ where the $i{+}1^{\text{th}}$ entry is 1 and all others are 0. Since the only simple path with group-label $(\gamma_i, 0, \ldots, 0, 2, 0, \ldots, 0)$, where the $i{+}1^{\text{th}}$ entry is 2, is from s'_i to t'_i, this implies: There are simple paths p'_i with group value $(\gamma_i, 0, \ldots, 0, 2, 0, \ldots, 0)$ which are pairwise node-disjoint in G' if and only if there are simple paths p_i from s_i to t_i with group value γ_i in G such that the p_i are pairwise node-disjoint.

In fact, the fix of Bonifati and Bagan inspired us to the following proof.

Proof of Theorem 6.8.1. We start with the proof for simple paths. The proof idea is to interpret L as a language over a finite abelian group for which group-labeled and undirected graphs are closely related. We then relate the problem of finding a simple path in a undirected multigraph to the problem of finding a minor in the group-labeled graph, which can be decided in polynomial time, see Huynh [119].

In order to make group-labeled graphs coincide with undirected graphs, we need that $\gamma_G(e) = -\gamma_G(e)$ for every edge e. This is the case if every element of the group is its own inverse, that is, when there is only a single element or the group is $\mathbb{Z}_2 = (\{0,1\}, +, 0)$ or a direct product thereof, that is, of the form $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \cdots \times \mathbb{Z}_2$.

Every parity language can be interpreted as a language over a finite abelian group of the form $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \cdots \times \mathbb{Z}_2$. For instance, if $\Sigma = \{a, b\}$, then an a-edge can be encoded as $(1,0)$, a b-edge as $(0,1)$. Conversely, if the value of a path in a so-constructed, group-labeled graph is $(0,1)$, we can interpret it as a path with an even number of a's and odd number of b's. In order to find a simple path from s to t that matches L in an undirected multigraph G (if it exists) we will:

1. We construct the underlying undirected graph G_{graph} of G, that is, if there are edges e_1, e_2 in G with $\mathcal{E}(e_1) = \mathcal{E}(e_2)$, then we remove one of them and repeat until $\mathcal{E}$ is an injective function.

2. We construct a group-labeled graph G' that is obtained from G_{graph} by relabeling the edges: We replace a_i with the i^{th} unit vector over $\mathbb{Z}_2^\ell$, which is $(0, \ldots, 0, 1, 0, \ldots, 0)$. Furthermore, we direct each edge in an arbitrary direction.

3. We define a set of group-labeled graphs $\mathcal{H}$, which are edges from s to t that are labeled with a vector in V_1. (Where V_1 comes from the Parikh image of L.)

4. We then test if there is a graph $H \in \mathcal{H}$ that is a minor of G'. If such a minor exists, then there is a simple path from s to t in G that matches L, otherwise there is none.

This procedure can be conducted in polynomial time since all changes to G are linear in $|G| + k$, the graphs in $\mathcal{H}$ depend only on L and are therefore of fixed size, and we can test for $|V_1|$ minors G' using the polynomial time algorithm proposed by Huynh [119]. Towards correctness, we first observe that a simple path in an undirected multigraph G exists if and only if that simple path exists in its underlying undirected graph G_{graph}, since a simple path can use at most one edge between each pair of nodes. By construction of G' and since L is a parity language, a simple path from s to t in G_{graph} that matches L is a simple path from s to t in G' whose group-value is in V_1. We can test for the latter using $|V_1|$ minor tests, which completes the correctness proof.

For trails, we proceed similar to the simple path case. That is, we again interpret L as a language over a finite abelian group of the form $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \cdots \times \mathbb{Z}_2$ and relabel the edges of the multigraph G accordingly, that is, we replace a_i with the i^{th} unit vector over $\mathbb{Z}_2^\ell$, which is $(0, \ldots, 0, 1, 0, \ldots, 0)$, and direct the edges arbitrarily.

We then relate the problem to the case for simple paths by using some variant of the *extended line graph*, introduced in Section 6.2. More precisely, we replace every node v by a clique in which every edge of the clique is labeled $(0, \ldots, 0)$. The size of the clique replacing v is the number of edges adjacent to v. Similar to Lemma 6.2.2, if v is the start or end-node of the trail, we add extra nodes s^* or t^* to the newly created clique.

More formally, let $G' = (V', E', \mathcal{E}')$ be the group-labeled multigraph obtained from G by relabeling the edges with unit vectors, and let nodes s and t be given. We define the group-labeled graph $G^* = (V^*, E^*, \mathcal{E}^*)$ and s^*, t^* as follows:

$$\begin{aligned}
V^* = \quad & \{(v, e) \mid v \in V', e \in E', e \text{ is incident to } v\} \cup \{s^*, t^*\} \\
E^* = \quad & \{((v, e_1), (0, \ldots, 0), (v, e_2)) \mid (v, e_1), (v, e_2) \in V^*\} \\
& \cup \{((v_1, e), \text{lab}(e), (v_2, e)) \mid (v_1, e), (v_2, e) \in V^*\} \\
& \cup \{(s^*, (0, \ldots, 0), (v, e)) \mid s = v, (v, e) \in V^*\} \\
& \cup \{(t^*, (0, \ldots, 0), (v, e)) \mid s = v, (v, e) \in V^*\},
\end{aligned}$$

and $\mathcal{E}^*((x, \sigma, y)) = \{(x, \sigma, y) \mid (x, \sigma, y) \in E^*\}$.

By construction, there is a trail from s to t in G matching L if and only if there is a simple path from s^* to t^* in G^* whose group-value is in V_1. The latter can again be tested by constructing a set of group-labeled graphs from V_1 and testing if G^* has one of them as a minor. □

Since Huynh [119] can also deal with more complex minors, (including minors which are k disjoint edges,) it follows that:

Lemma 6.8.2. *Let $L_1, \ldots, L_k$ be parity languages over Σ. Then we can find k node-disjoint simple paths (or k edge-disjoint trails) from s_i to t_i matching L_i in polynomial time.*

Proof. Since $L_1, \ldots, L_k$ are parity languages, there exist sets $V_{1,1}, \ldots, V_{1,k} \subseteq \{0, 1\}^\ell$ such that that each L_i can be written in the form $\{v_1 + v_2 n \mid n \in \mathbb{N}, v_1 \in V_{1,i}\}$ and $v_2 = (2, \cdots, 2)$ with $i \in [k]$. The only change to the proof of Theorem 6.8.1 is that the group-labeled graphs in $\mathcal{H}$ are not single edges, but k node-disjoint edges. More precisely, a group-labeled graph in $\mathcal{H}$ has for each $i \in [k]$ an edge from s_i to t_i that is labeled with a word in $V_{1,i}$. Since each possible combination is in $\mathcal{H}$, $|\mathcal{H}| = |V_{1,1}| \cdots |V_{1,k}|$. $\square$

Remark 6.8.3. For simple paths it is important that $L_1, \dots, L_k$ use the same alphabet, as the problem of finding two node-disjoint paths, one labeled a^*, the other b^* is NP-complete, see Gourves et al. [105, Theorem 16].

Corollary 6.8.4. *Let $L_1, \dots, L_k$ be parity languages over Σ and let $F_1, \dots, F_{k+1}$ be finite languages. Then $\mathsf{USimPath}(F_1 L_1 F_2 \cdots L_k F_{k+1})$ and $\mathsf{UTrail}(F_1 L_1 F_2 \cdots L_k F_{k+1})$ are in P.*

Proof. We enumerate the node-disjoint (or edge-disjoint) finite simple paths (or trails) $(p_1, \dots, p_{k+1})$ matching $F_1, \dots, F_{k+1}$. Let p_i be a path from t_{i-1} to s_i for each $i \in [k+1]$. Then we can use Lemma 6.8.2 to test in polynomial time if there exist k node-disjoint simple paths (or edge-disjoint trails) from s_i to t_i matching L_i in the multigraph without the inner nodes (or edges) of the paths $(p_1, \dots, p_{k+1})$. $\square$

Using the observation that G_a and G_b are edge-disjoint for all pair of symbols $a \neq b$ we obtain:

Corollary 6.8.5. *Let $L_1, \dots, L_k$ be parity languages over alphabets $\Sigma_1, \dots, \Sigma_n$, such that $\Sigma_i = \Sigma_j$ or $\Sigma_i \cap \Sigma_j = \emptyset$ for each pair i, j. Let $F_1, \dots, F_{k+1}$ be finite languages. Then $\mathsf{UTrail}(F_1 L_1 F_2 \cdots L_k F_{k+1})$ is in P.*

Proof. We explain how to decide $\mathsf{UTrail}(F_1 L_1 F_2 \cdots L_k F_{k+1})$ in polynomial time. Let G be an undirected multigraph. We enumerate the edge-disjoint trails $(p_1, \dots, p_{k+1})$ matching $F_1, \dots, F_{k+1}$ in G. Let p_i be a path from t_{i-1} to s_i for each $i \in [k+1]$. Let $I_1, \dots, I_m$ be sets of indices such that $\Sigma_i = \Sigma_j$ for all $i, j \in I_x$ and $\Sigma_i \cap \Sigma_j = \emptyset$ for all $i \in I_x, j \in I_y$ and $x \neq y$. Since for each $x \in [m]$ the subgraphs of G restricted to edges with labels in Σ_i with $i \in I_x$ are edge-disjoint from each subgraph of G restricted to edges with labels in Σ_j with $j \notin I_x$ per construction of $I_1, \dots, I_m$, we can perform tests on each subgraph separately. More precisely, for each $x \in [m]$ we use Lemma 6.8.2 to test in polynomial time if there exist $|I_j|$ edge-disjoint trails from s_i to t_i matching L_i with $i \in I_x$ in the multigraph restricted to edges with labels in Σ_i with $i \in I_x$, and without edges of the paths $(p_1, \dots, p_{k+1})$. $\square$

Chapter 7

Towards Fine-grained Dichotomies for STEs

Similar to Chapter 5 we now have a look at the parameterized complexity of PSimPath and PTrail on undirected multigraphs. We name the respective problems PUSimPath and PUTrail to distinguish them from the problems on directed multigraphs. We again focus on the class of STEs, since they are prominent in practice.

7.1 Tractable Fragments are Preserved for Undirected Multigraphs

We first note that the tractability results on directed multigraphs also hold on undirected multigraphs.

Theorem 7.1.1. *Let $\mathcal{R}$ be a cuttable class of STEs. Then PUSimPath($\mathcal{R}$) is in FPT.*

Proof. Let $G = (V, E, \mathcal{E})$ be an undirected multigraph. Let $G' = (V, E', \mathcal{E}')$ be the directed multigraph obtained from G by replacing every undirected edge $e \in E$ with $\mathsf{Node}(e) = \{u, v\}$ with two directed edges e_1, e_2 and $\mathcal{E}'(e_1) = (u, \mathrm{lab}(e), v)$, $\mathcal{E}'(e_2) = (v, \mathrm{lab}(e), u)$. Since every simple path can use at most one edge between any pair of nodes, a simple path in G' corresponds directly to a simple path in G and vice versa. Furthermore, since $\mathcal{R}$ is cuttable, PSimPath($\mathcal{R}$) is in FPT by Theorem 5.2.2. Thus, we can solve PSimPath($\mathcal{R}$) on G' in FPT and give the same answer. □

Theorem 7.1.2. *Let $c \in \mathbb{N}$ be a constant and let $\mathcal{R}$ be the class of STEs with at most c conflict positions, that is, $\mathcal{R}$ is almost conflict-free. Then PUTrail($\mathcal{R}$) is in FPT. More precisely, for an $r \in \mathcal{R}$, PUTrail(r) is in time $2^{O(|r|)} \cdot |E|^{c+6}$.*

Proof. Let $r = B_{\text{pre}} T^* B_{\text{suff}} \in \mathcal{R}$ and $G = (V, E, \mathcal{E})$ be an undirected multigraph and $s, t \in V$. Let $\$_1, \$_2, \#_1, \#_2$ be four new symbols which occur neither in r nor in G. From G, we construct a directed graph $G' = (V', E', \mathcal{E}')$ with $V' = V \cup \{x_e, y_e \mid e \in E\}$, $E' = \{(u, \$_1, x_e), (u, \#_1, x_e), (v, \$_2, x_e), (v, \#_2, x_e), (x_e, \mathrm{lab}(e), y_e), (y_e, \$_1, u), (y_e, \#_1, u), (y_e, \$_2, v), (y_e, \#_2, v) \mid e \in E, \mathsf{Node}(e) = (u, v)\}$, and $\mathcal{E}'((x, \sigma, y)) = (x, \sigma, y)$ with $(x, \sigma, y) \in E'$. That is, every edge is replaced with a gadget as depicted in the middle

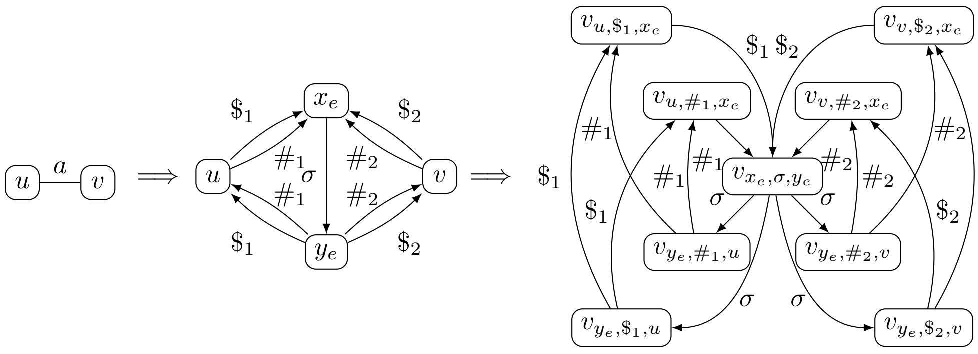


Figure 7.1: Illustration of the construction of the directed graphs G' and H in the proof of Theorem 7.1.2.

of Figure 7.1. We use different edges labeled $\$_1$, $\#_1$ to avoid introducing new conflict positions. The different edges labeled $\$_1$ and $\$_2$ are used to forbid cycles labeled $\$_1 a \$_1$ or $\$_2 a \$_2$ for any symbol a. They have been used in the same way as in Lemma 6.3.2(b). The symbols $\#_1$ and $\#_2$ are used analogously.

Let c_1 be the left cut border of r and c_2 be its right cut border. Let $f\colon \{1,\dots,c_1+c_2\} \to \{1,2\}$ be a function and $\overline{f}(i) = 3 - f(i)$. Let $A^{\$} = (A \cup \{\$_1, \$_2\})$ for any set A. We define

$$r'_f = B'_{\text{pre},f}(T^{\$})^* B'_{\text{suff},f}$$

where $B'_{\text{pre},f} =$

- $(\#_1{+}\#_2)?\, A_1?\, (\#_1{+}\#_2)?\, (\#_1{+}\#_2)?\, A_2?\, (\#_1{+}\#_2)? \cdots (\#_1{+}\#_2)?\, A_{k_1}?\, (\#_1{+}\#_2)?$ if $B_{\text{pre}} = A_1? \cdots A_{k_1}?$,
- $\#_{f(1)} A_1 \#_{\overline{f}(1)}\ \#_{f(2)} A_2 \#_{\overline{f}(2)}\ \cdots \#_{f(c_1)} A_{c_1} \#_{\overline{f}(c_1)}\ T^{\$} A^{\$}_{c_1+1} T^{\$} \cdots T^{\$} A^{\$}_{k_1} T^{\$}$ otherwise, that is, if $B_{\text{pre}} = A_1 \cdots A_{k_1}$.

and $B'_{\text{suff},f} =$

- $(\#_1{+}\#_2)?\, A'_{k_2}?\, (\#_1{+}\#_2)? \cdots (\#_1{+}\#_2)?\, A'_1?\, (\#_1{+}\#_2)?$ if $B_{\text{suff}} = A'_{k_2}? \cdots A'_1?$.
- $T^{\$}(A'_{k_2})^{\$} T^{\$} \cdots T^{\$}(A'_{c_1+c_2+1})^{\$} T^{\$} \#_{f(c_1+c_2)} A'_{c_2} \#_{\overline{f}(c_1+c_2)} \cdots \#_{f(c_1+1)} A'_1 \#_{\overline{f}(c_1+1)}$ otherwise, that is, if $B_{\text{pre}} = A_1 \cdots A_{k_1}$.

By construction, there is a trail from s to t in G matching r if and only if there is a function f and a trail from s to t in G' matching r'_f that does not contain a subpath labeled $\$_1 \sigma \$_1$ or $\$_2 \sigma \$_2$ for any $\sigma \in \Sigma$.

We now show that the number of conflict positions in r and r'_f is identical: To this end, we first recall some definitions: The left cut border of STEs with $B_{\text{pre}} = A_1 \cdots A_{k_1}$ is the largest value $i \in [k_1]$ such that T has a symbol that is not in A_i. The right cut border of STEs with $B_{\text{suff}} = A'_{k_2} \cdots A'_1$ is the largest value $i \in [k_2]$ such that T has a symbol that is not in A'_i. If $B_{\text{pre}} = A_1? \cdots A_{k_1}?$ the left cut border is 0. If $B_{\text{suff}} = A'_{k_2}? \cdots A'_1?$, the right cut border is 0. A position left of the left cut border or right of the right cut border is a conflict position if T and the set on this position have a non-empty intersection.

By construction, if $c_1 = 0$ (respectively, $c_2 = 0$), then the left (respectively, right) cut border of r' is 0. On the other hand, if $c_1 \geq 1$, the left cut border of r' is exactly on the position of $\#_{\overline{f}(c_1)}$. This is because $T \subseteq A_i$ for each $i \in \{c_1 + 1, \dots, k_1\}$ by definition of c_1 and therefore also $T^\$ \subseteq A_i^\$$. Analogously, if $c_2 \geq 1$, the right cut border of r' is on the position of $\#_{f(c_1+c_2)}$. So the left and right cut border of r' are $3c_1$ and $3c_2$, respectively.

Since $\{\#_1, \#_2\} \cap T^\$ = \emptyset$ by definition, the positions of the symbols $\#_1$ and $\#_2$ cannot be conflict positions. As $T^\$ \cap A_i = \emptyset$ for $i \in [c_1]$ if and only if $T \cap A_i = \emptyset$ for $i \in [c_1]$ and $T^\$ \cap A'_i = \emptyset$ for $i \in [c_2]$ if and only if $T \cap A'_i = \emptyset$ for $i \in [c_2]$, the number of conflict positions in r and r' is identical.

Thus the number of conflict positions of r' is bounded by a constant c. While Lemma 5.6.3 allows us to decide whether there exists a trail matching r'_f from s to t in G' in time $2^{O(|r'_f|)} \cdot |E'|^{c+6}$, it does not take into account that subpaths matching $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ are forbidden. In order to prove this, we proceed similar to the proof of Lemma 5.6.3.

That is, we use the construction from the proof of Lemma 5.5.3 to obtain a directed graph $H = (V_H, E_H, \mathcal{E}_H)$ such that there is a trail from s to t matching r'_f in G' if and only if there is a simple path from s' to t' matching $a \cdot r'$ in H where a is a new symbol not occurring in r'_f or G'. By construction, we can go a step further: there is a trail from s to t matching r'_f in G' that does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$ if and only if there is a simple path from s' to t' matching $a \cdot r'$ in H that does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$. We relabel the expression to a conflict-free expression $\tilde{r}_f$ and iterate over all possible sets S of nodes of size up to c and relabel H depending on S. Then, in each directed graph H_S, we test for a simple path from s to t matching $a \cdot \tilde{r}_f$ that does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$.

In the following, we will focus on $B'_{\text{pre},f}$ being of the form

$$\#_{f(1)}A_1\#_{\overline{f}(1)}\ \#_{f(2)}A_2\#_{\overline{f}(2)}\ \cdots \#_{f(c_1)}A_{c_1}\#_{\overline{f}(c_1)}\ T^\$ A^\$_{c_1+1}T^\$ \cdots T^\$ A^\$_{k_1} T^\$.$$

Similarly to f, we will define a function g with: $g\colon [k_1] \to \{1, 2\}$ and $\overline{g}(i) = 3 - g(i)$. Let

$$B'_{\text{pre},f,g} = \#_{f(1)}A_1\#_{\overline{f}(1)}\ \#_{f(2)}A_2\#_{\overline{f}(2)}\ \cdots \#_{f(c_1)}A_{c_1}\#_{\overline{f}(c_1)} \cdot \\ \$_{g(c_1+1)}A^\$_{c_1+1}\$_{\overline{g}(c_1+1)} \cdots \$_{g(k_1)}A^\$_{k_1}\$_{\overline{g}(k_1)},$$

and analogously for $B'_{\text{suff},f,g}$. Let $\mathcal{R}_1$ be the set containing all renamed versions $B'_{\text{pre},f,g}$. We define a set $\mathcal{R}_2$ for $B'_{\text{suff},f,g}$ analogously.

Lemma 7.1.3 (similar to Lemmas 5.3.11 and 5.6.4)**.** *Let H_S, $\tilde{r}_f$, nodes s, t, and $\mathcal{R}_1$ as defined in the proof of Theorem 7.1.2 be given. Let k'_1 be the length of the longest word in $\mathcal{R}_1$ and k'_2 be the length of the longest word in $B'_{\mathit{suff},f}$. We define $k' = k'_1 + k'_2$.*

Then, $H_S = (V_H, E_H, \mathcal{E}_H)$ has a simple path from s to t that matches $a \cdot r$ and does not contain a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$ if and only if there exists a regular expression $r_1 \in \mathcal{R}_1$, a node $v \in V_H$, and a set of nodes $X \in \hat{P}^{a\cdot r_1}_{s,v} \subseteq^{k'+2}_{rep} P^{a\cdot r_1}_{s,v}$, such that H_S has a simple path from s to t that matches $a \cdot r$, has no subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$, and with the first $k'_1 + 2$ nodes belonging to X.

Proof. The if direction is straightforward. For the only-if direction, let $p = e_1 \cdots e_n$ be a shortest simple path from s to t that matches $a \cdot \tilde{r}_f$ and has no subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$. We make the following case distinction on the length of p.

If $|p| \leq 2k'_1 + k'_2 + 3$, we define $P = e_1 \cdots e_{k'_1+1}$ and $Q = e_{k'_1+3} \cdots e_n$. By definition of $\mathcal{R}_1$, there is a $r_1 \in \mathcal{R}_1$ such that P matches $a \cdot r_1$, and $e_{k'_1+2} \cdot Q$ matches $(T^{\$})^* B'_{\text{suff},f}$. Let $v_{k'_1+1} = \text{destination}(e_{k'_1+1})$ We have that $V(P) \in P^{a\cdot r_1}_{s,v_{k'_1+1}}$, we have $|V(Q)| \leq k' + 2$, and $V(P) \cap V(Q) = \emptyset$. Let $\hat{P}^{a\cdot r_1}_{s,v_{k'_1+1}}$ be a $(k' + 2)$-representative set of $P^{a\cdot r_1}_{s,v_{k'_1+1}}$. Then there exists a set $X \in \hat{P}^{a\cdot r_1}_{s,v_{k'_1+1}}$ with $X \cap V(Q) = \emptyset$. By definition of $P^{a\cdot r_1}_{s,v_{k'_1+1}}$, there exists a simple path P' from s to $v_{k'_1+1}$ with $V(P') = X$ that matches $a \cdot r_1$. Therefore, $P' \cdot e_{k'_1+2} \cdot Q$ is a simple path from s to t that matches $a \cdot \tilde{r}_f$ and has no subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ for any $\sigma \in \Sigma$.

Otherwise, we have $|p| > 2k'_1 + k'_2 + 3$. We define $P = e \cdots e_{k'_1+1}$, $R = e_{k'_1+3} \cdots e_{n-k'-2}$, and $Q = e_{n-k'} \cdots e_n$. We thus have

$$p = P \cdot e_{k'_1+2} \cdot R \cdot e_{n-k'-1} \cdot Q.$$

Since p matches $a \cdot \tilde{r}_f$, we furthermore know that P matches $a \cdot r_1$ for some $r_1 \in \mathcal{R}_1$, R matches $(T^{\$})^*$, and Q matches $(T^{\$})^*(T^{\$})^{k'_1+1}B'_{\text{suff},f}$.[1] Since $|V(Q)| = k' + 2$, $V(P) \in P^{a\cdot r_1}_{s,v_{k'_1+1}}$, and $V(P) \cap V(Q) = \emptyset$, the definition of $\hat{P}^{a\cdot r_1}_{s,v_{k'_1+1}} \subseteq^{k'+2}_{\text{rep}} P^{a\cdot r_1}_{s,v_{k'_1+1}}$ guarantees, similar as in the previous case, the existence of a simple path P' from s to $v_{k'_1+1}$ that matches $a \cdot r_1$ with $V(P') \in P^{a\cdot r_1}_{s,v_{k'_1+1}}$ and $V(P') \cap V(Q) = \emptyset$. Let $P' = e'_1 \cdots e'_{k'_1+1}$. If P' is disjoint from R, the path

$$p' = P' \cdot e_{k'_1+2} \cdot R \cdot e_{n-k'-1} \cdot Q$$

is a simple path matching r, and we are done.

We show that P' must be disjoint from R. Let c_1 be the left cut border of $\tilde{r}_f$. Clearly, the paths P' and R cannot intersect in the first $c_1 + 1$ nodes[2] of P' since those nodes only have outgoing edges that have labels not in T by construction of H_S (the outgoing edges of each node have the same label) and since $\tilde{r}_f$ has no conflict positions.

Towards a contradiction, assume that there is an $i \in \{c_1 + 1, \ldots, k'_1\}$[3] such that $\text{destination}(e'_i) = \text{origin}(e_j)$ for some $j \in \{k'_1 + 3, \ldots, n - k' - 1\}$.

[1] Since $B'_{\text{suff},f}$ might have shorter words, we cannot simply write $(T^{\$})^{k'_1+1}B'_{\text{suff},f}$.

[2] Since P' matches $a \cdot \tilde{r}_f$, and a is a new symbol, we also have $a \notin T$.

[3] We note that s and $v_{k'_1+1}$ are in $V(P)$ and therefore not in $V(R)$.

We choose i minimal and build a new simple path

$$p' = e'_1 \cdots e'_i e_{j+1} \cdots e_n.$$

This path matches $a \cdot B_1 \cdots B_{c_1} \cdots B_i (T^\$)^* (T^\$)^{k'_1+1} B'_{\text{suff},f}$. Since c_1 is the left cut border of $\tilde{r}_f$, we have $T^\$ \subseteq B_j$ for all $c_1 + 1 \leq j \leq k'$, so the new path matches $a \cdot \tilde{r}_f$. If p' has no subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$, we are done since p' does not contain the edge $e_{k'_1+2}$, p' is shorter than p, contradicting the choice of p.

On the other hand, if p' contains a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$, then this subpath must be where P' and R intersected. An example is given in Figure 7.2. While in H_S the paths labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ are not necessarily loops, they always have the form

$$(v_{(u,\$_i,x_e)}, \$_i, v_{(x_e,\text{lab}(e),y_e)})(v_{(x_e,\text{lab}(e),y_e)}, \text{lab}(e), v_{(y_e,\$_i,u)})(v_{(y_e,\$_i,u)}, \$_i, v_{(u,\_,\_)})$$

with $i \in \{1, 2\}$ and _ are some values. By construction of H, that is, Lemma 5.5.3, every node $v_{(u,\_,\_)}$ has the same incoming edges, independent of the values of _. Thus, we can remove the path labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$ from p' and replace the incoming edge to $v_{(u,\$_i,x_e)}$ with the one to $v_{(u,\_,\_)}$ to obtain a new simple path p'' that does not have a subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$. It remains to prove that p'' matches $a \cdot \tilde{r}_f$. To this end, we observe that given H_S, P' and R, p'' is obtained from p' by replacing e'_{i-1} with a new edge e''_{i-1} that has the same label and removing $e'_i e_{j+1} e_{j+2}$, with $\text{lab}(e'_i), \text{lab}(e_{j+2}) \in \{\$_1, \$_2\}$, $\text{lab}(e_{j+1}) \in T \cap A_m$ with $m > c_1$.[4] That is, $p'' = e'_1 \cdots e_{c_1+1} \cdots e'_{i-2} e''_{i-1} e_{j+3} \cdots e_n$ and p'' matches $a \cdot B_1 \cdots B_{c_1} \cdots B_{i-1} (T^\$)^* (T^\$)^{k'_1+1} B'_{\text{suff},f}$ and therefore $a \cdot \tilde{r}_f$.

Finally, we note that p'' does not contain the edge $e_{k'_1+2}$, so p'' is shorter than p, which contradicts that p was a shortest simple path from s to t matching r and that has no subpath labeled $\$_1\sigma\$_1$ or $\$_2\sigma\$_2$. So P' must be disjoint from R. □

Similar to Lemmas 5.3.11 and 5.6.4, Lemma 7.1.3 allows us to use (a variant of) Algorithm 3 to find a path from s to t in H_S that matches the conflict-free regular expression $\tilde{r}_f$ in FPT. More precisely, we need to additionally iterate over all sets S for H_S, compute $\hat{P}^{a \cdot r_1}_{s,v}$ for each $r_1 \in \mathcal{R}_1$ in line 2, and then iterate over all sets X in $\bigcup_{r_1 \in \mathcal{R}_1} \hat{P}^{a \cdot r_1}_{s,v}$ in line 3. In line 8 we symmetrically need to iterate over all sets $X' \in \bigcup_{r_2 \in \mathcal{R}_2} \hat{P}^{r_2}_{u,t}$.

We now turn to the running time. First we have $2^{c_1+c_2}$ different expressions r_f, where c_1 is the left cut border of r and c_2 is the right cut border. We can compute H in polynomial time of its size, which is $O(|E|^2)$, since that is $V_H = O(|E|)$ and $E_H = O(|E|^2)$. If r_f has c conflict positions, then we have to iterate over all sets of up to c nodes in H to obtain H_S, which multiplies the running time with $|E|^c$. After preprocessing time of $|r||H_S|$ we can compute $\hat{P}^{a \cdot r_1}_{s,v}$ for each $r_1 \in \mathcal{R}_1$ in time $2^{O(|r|)}|E_H| \log |V_H| \cdot 2^{k_1}$. We then have to consider up to $|V_H|^2 \cdot 2^{O(|r|)}$ sets X. We can compute $\hat{P}^{r_2}_{u,t}$ for each $r_2 \in \mathcal{R}_2$ in $2^{O(|r|)}|E_H| \log |V_H| \cdot 2^{k_2}$ and finally do for each set X' a reachability test for a path from

[4] Note that $m \neq c_1$ since A_{c_1} either was a conflict position and therefore renamed, or $A_{c_1} \cap T = \emptyset$.

v to u that matches T^* in $O(|V_H| + |E_H|)$. Thus the running time of PUTrail(r) can be bounded by

$$2^{c_1+c_2} \cdot |V_H|^c \Big(|r||H_S| + O(2^{O(|r|)}|E_H| \log |V_H|) \cdot 2^{k_1} + \\ |V_H|^2 \cdot 2^{O(|r|)} \cdot \big(2^{O(|r|)}|E_H| \log |V_H| \cdot 2^{k_2} + |V_H|^2 \cdot 2^{O(|r|)} \cdot O(|V_H| + |E_H|)\big)\Big),$$

which can be simplified to $2^{O(|r|)} \cdot |E|^{c+6}$, where $|c|$ is the number of conflict positions in r. □

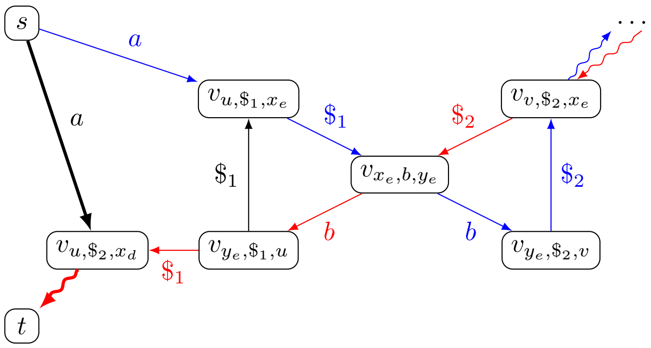


Figure 7.2: Illustration of a case in the proof of Lemma 7.1.3 on a subgraph of H_S. If the paths P' (blue) and R (red) intersect, the naive simple path $(s, a, v_{u,\$_1,x_e})(v_{u,\$_1,x_e}, \$_1, v_{x_e,b,y_e})(v_{x_e,b,y_e}, b, v_{y_e,\$_1,u})(v_{y_e,\$_1,u}, \$_1, v_{u,\$_2,x_d}) \dots$ contains a subpath labeled $\$_1 b \$_1$, even though the original paths had no subpath labeled $\$_1 \sigma \$_1$ for any $\sigma \in \Sigma$. Due to the special structure of the graph H_S, we can remove the subpath labeled $\$_1 b \$_1$ by taking the incoming a-edge to $v_{u,\$_2,x_d}$ instead of the a-edge to $v_{y_e,\$_1,u}$.

7.2 Extension of the Tractable Fragment

We now prove that the class of tractable STEs on undirected multigraphs is larger than in the directed case if we consider trail semantics. Cai and Ye [59] studied the closely related problems:

- UPDisjointTrails$(=, *)$: Given a undirected graph G, nodes s_1, t_1, s_2, and t_2, and a parameter $k \in \mathbb{N}$, are there two edge-disjoint trails, one from s_1 to t_1 of length exactly k and one from s_2 to t_2?
- UPDisjointTrails$(\leq, *)$: Given a undirected graph G, nodes s_1, t_1, s_2, and t_2, and a parameter $k \in \mathbb{N}$, are there two edge-disjoint trails, one from s_1 to t_1 of length at most k and one from s_2 to t_2?

Lemma 7.2.1 (Cai and Ye [59]). *UPDisjointTrails$(=,*)$ and UPDisjointTrails$(\leq,*)$ are in FPT.*

Re-inspecting their proof, we can see that with only slight changes[5] their FPT result also holds on multigraphs. Thus a straight-forward result is the following:

Corollary 7.2.2. *PUTrail(a^kba^*) and PUTrail$(a^{\leq k}ba^*)$ are in FPT.*

Proof. Indeed, for undirected graphs we can iterate over all b-edges and then solve UPDisjointTrails$(=,*)$ or UPDisjointTrails$(\leq,*)$ on the remaining subgraph G_a. Taking a closer look at the proof of Cai and Ye, we can see that it also works on multigraphs—we make this more concrete in the proof of Theorem 7.2.5. □

Since the class of STEs of the form a^kba^* are not almost conflict-free but tractable, this is a very interesting result. Indeed, we can lift Cai and Ye's proof of Lemma 7.2.1 to show that PUTrail is tractable for classes of STEs $B_{\text{pre}}T^*B_{\text{suff}}$ with $T = \Sigma$.

To this end, we first need the following lemma.

Lemma 7.2.3. *Let G be an undirected multigraph, $s_1, t_1, s_2,$ and t_2 nodes and A_1, A_2 NFAs. Then it is in FPT to decide whether there exist two edge-disjoint trails, one from s_1 to t_1 matching A_1 of length at most k_1 and one from s_2 to t_2 matching A_2 of length at most k_2. More precisely, this can be decided in time $2^{O(|k_1|+|k_2|)}(|A_1|+|A_2|)|G|\log|E|$.*

The proof is a combination of dynamic programming and color coding [7]. Indeed, we closely follow the proof of Bagan et al. [20, Theorem 8] who showed that one can decide in time $2^{O(k)}|A|\cdot|G|\log|G|$ whether there is a simple path from s to t of length at most k matching a given NFA A. We note that instead of coloring nodes, we will color the edges.

A k-coloring of E is a function $c\colon E \to [k]$. A set $S \subseteq E$ is colorful for c if $c(e_1) = c(e_2)$ implies $e_1 = e_2$ for every $e_1, e_2 \in S$. We use the following result by Alon et al.:

Theorem 7.2.4 ([7]). *Given $k, m \geq 0$ and a set E of m elements, one can compute in time $2^{O(k)}|E|\log|E|$ a set of $\ell \in 2^{O(k)}\log|E|$ k-coloring functions $c_1, \ldots, c_1$ such that every set S of E of size k is colorful for at least one c_i, $i \in [\ell]$.*

Proof of Lemma 7.2.3. Let $G = (V, E, \mathcal{E})$ be an undirected multigraph, and $A_1 = (Q_1, \Sigma, \delta_1, \{i_1\}, F_1)$, $A_2 = (Q, \Sigma, \delta_2, \{i_2\}, F_2)$ NFAs. Let $k = k_1 + k_2$ and $c_1, \ldots, c_\ell$ be k-coloring functions as stated in Theorem 7.2.4. We define a function $f\colon V \times Q_1 \times V \times Q_2 \times \mathcal{P}([k]) \to \{0,1\}$ such that $f(v_1, q_1, S_1, v_2, q_2, S_2) = 1$ if and only if S_1 and S_2 are disjoint and there exists a colorful path p_1 from s_1 to v_1 using only colors of S_1 and such that $\delta_1(i_1, p_1) = q_1$ and a colorful path p_2 from s_2 to v_2 using only colors of S_2 and such that $\delta_2(i_2, p_2) = q_2$. This function can be computed using dynamic programming with the following equations:

[5]That is, the additional argument that a shortest path p_2 uses at most one "nearby-edge" between any pairs of "nearby-nodes" is needed. We revisit their proof and this argument in the proof of Theorem 7.2.5.

- $f(s_1, i_1, \{\}, s_2, i_2, \{\}) = 1$.
- $f(v_1, q_1, S_1, v_2, q_2, S_2) = 1$ if $S_1 \cap S_2 = \emptyset$ and there is
  - a subset $S_1' \subsetneq S_1$ such that $f(v_1, q_1, S_1', v_2, q_2, S_2) = 1$, or
  - a subset $S_2' \subsetneq S_2$ such that $f(v_1, q_1, S_1, v_2, q_2, S_2') = 1$.
- $f(v_1, q_1, S_1, v_2, q_2, S_2) = 1$ if $S_1 \cap S_2 = \emptyset$ and there is
  - an edge e with $c(e) \in S_1$, a state q' such that $\delta_1(q', e) = q_1$, $\mathsf{Node}(e) = \{v_1, v'\}$ and $f(v', q', S_1 - \{c(e)\}, v_2, q_2, S_2) = 1$, or
  - an edge e with $c(e) \in S_2$, a state q' such that $\delta_2(q', e) = q_2$, $\mathsf{Node}(e) = \{v_2, v'\}$ and $f(v_1, q_1, S_1, v', q', S_2 - \{c(e)\}) = 1$.

This function can be computed in time $O(2^{k_1+k_2}(|A_1| + |A_2|)|G|)$ We compute f for every function c_i, $i \in [\ell]$ where $\ell \in 2^{O(k)} \cdot \log |E|$. Clearly, there are edge-disjoint trails from s_1 to t_1 and from s_2 to t_2 of length at most k_1 and k_2, respectively, if and only if there are $i \in [\ell]$, $S_1, S_2 \subseteq [k]$ with $S_1 \cap S_2 = \emptyset$, $q_1 \in F_1$, and $q_2 \in F_2$ such that $f(t_1, q_1, S_1, t_2, q_2, S_2) = 1$ for some coloring function c_i. Consequently, we can decide in time $2^{O(k_1+k_2)}(|A_1| + |A_2|)|G| \log |E|$ whether these trails exist. □

Theorem 7.2.5. *Let $\mathcal{R}$ be a class of STEs of the form $B_{pre}\Sigma^* B_{suff}$. Then $\mathsf{PUTrail}(\mathcal{R})$ is in FPT.*

Proof. We build upon the proof of Cai and Ye [59] that $\mathsf{UPDisjointTrails}(=, *)$ and $\mathsf{UPDisjointTrails}(\leq, *)$ are in FPT. The main idea of their proof is that if we fix some path from s_1 to t_1 of length k, then the shortest edge-disjoint path from s_2 to t_2 can only use $k^2 - 1$ many edges which are close to s_1 and t_1. This allows them to give a randomized algorithm which they then derandomize with the help of universal sets to obtain a deterministic FPT algorithm.

Let G be an undirected multigraph, $r \in \mathcal{R}$ be of the form $A_1 \cdots A_{k_1} \Sigma^* A'_{k_2} \cdots A'_1$ and let $x, y \in V$ be two arbitrary nodes in G.

A node is a *nearby-node* if there are paths to s and x of length at most $(k_1 + k_2)/2$ or there are paths to y and t of length at most $(k_1 + k_2)/2$. An edge e is a *nearby-edge* if all nodes in $\mathsf{Node}(e)$ are nearby-nodes.

Assume there is a trail p_1 from s to x labeled $A_1 \cdots A_{k_1}$ and a trail p_3 from y to t labeled $A'_{k_2} \cdots A'_1$ and such that p_1 and p_3 are edge-disjoint. Let p_2 be a shortest path labeled A^* from x to y. Then

1. all edges in p_1 and p_3 are nearby-edges, and
2. p_2 contains at most $(k_1 + k_2 + 2)^2$ many nearby-nodes and $(k_1 + k_2 + 2)^2 - 1$ nearby-edges.

The first statement is obvious, while for the second we assume towards contradiction that p_2 contains at least $(k_1 + k_2 + 2)^2 + 1$ many nearby-nodes. For each $x \in V(p_1) \cup V(p_3)$

we define

$$N^*_x = \{v \mid v \text{ is a nearby-node in } V(p_2) \text{ and there is a path of length at most } (k_1+k_2)/2 \text{ from } v \text{ to } x \text{ which is node-disjoint from } p_1 \text{ and } p_3 \text{ (up to } x)\}.$$

Clearly, every nearby-node of p_2 belongs to (at least one) set N^*_x with $x \in V(p_1) \cup V(p_3)$. Since $|V(p_1) \cup V(p_3)| = k_1 + k_2 + 2$, it follows by the pigeon hole principle that there is some $y \in V(p_1) \cup V(p_3)$ with $|N^*_y| \geq (k_1 + k_2 + 2) + 1$. Let x_s be the first node of p_2 in N^*_y and x_t be the last node. By definition, there exist paths of length at most $(k_1+k_2)/2$ from x_s to y and from y to x_t and these paths are node- and therefore also edge-disjoint from p_1 and p_3 (up to y). Thus there exists a path of length at most $k_1 + k_2$ from x_s via y to x_t, contradicting the choice of p_2 as shortest possible path that is edge-disjoint from p_1 and p_3. We illustrate this in Figure 7.3.

Since p_2 contains at most $(k_1 + k_2 + 2)^2$ many nearby-nodes and a shortest path will not use the same node twice (otherwise, we find a shorter path), p_2 uses at most $(k_1 + k_2 + 2)^2 - 1$ nearby-edges.[6]

Thus a straight-forward randomized algorithm proceeds as follows:

1. For all possible nodes x, y:
2. Find all nearby-nodes (for example by doing four rounds of BFS, starting from s, x, y, and t).
3. Between each pair (u, v) of nearby-nodes, randomly color one edge with $\mathsf{Node}(e) = \{u, v\}$ by color 1 or 2 with probability 1/2, and color all remaining edges with $\mathsf{Node}(e) = \{u, v\}$ with color 1.[7]
4. Color all uncolored edges of G with color 2.
5. Find two edge-disjoint trails, one from s to x matching $A_1 \cdots A_{k_1}$ and one from y to t matching $A'_{k_2} \cdots A'_1$ in the 1-colored graph, and a trail from x to y in the 2-colored graph. If these paths exist, return "yes".

Let G_i be the subgraph of G that contains only edges of color i. Let p_1, p_2, p_3 be a solution that minimizes the length of p_2. Then p_1 and p_3 are entirely in G_1 with probability $\geq 1/2^{(k_1+k_2)}$ and p_2 is entirely inside G_2 with probability $> 1/2^{(k_1+k_2+2)^2}$.

By Lemma 7.2.3 it takes $2^{O(|k_1|+|k_2|)}|r||G| \log |E|$ time to decide if there exist edge-disjoint trails matching $A_1 \cdots A_{k_1}$ and $A'_{k_2} \cdots A'_1$, respectively, in G_1. Using BFS, it takes $O(|G|)$ time find a shortest path from x to y in G_2. So, if a solution exists, this algorithm will answer "yes" with probability $> 1/2^{k_1+k_2+(k_1+k_2+2)^2}$ in $|V|^2 2^{O(|k_1|+|k_2|)}|r||G| \log |E|$ time.

[6]This shortest path argument is needed in the case of multigraphs, since between any pair of nearby-nodes there can be arbitrary many nearby-edges.

[7]Here we use again the argument that the shortest path p_2 will use at most one edge between any pair of nodes.

It remains to derandomize the algorithm. Let m' be the number of nearby-edges and $\ell = k_1 + k_2 + (k_1 + k_2 + 2)^2$. As was shown by Cai and Ye [59, Proof of Theorem 2] one can use an (m', d, ℓ)-perfect hash family for derandomization, where d is a power of 2 between $\ell(\ell - 1)/2 + 2$ and $2\ell(\ell - 1) + 4$. The running time of the derandomized algorithm is then $2^{((k_1+k_2)\cdot\log(k_1+k_2))} \log m' \cdot |V|^2 2^{O(|k_1|+|k_2|)}|r||G| \log |E|$. □

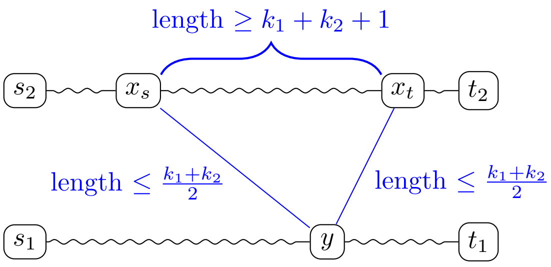

length $\geq k_1 + k_2 + 1$
s_2
x_s
x_t
t_2
length $\leq \frac{k_1+k_2}{2}$
length $\leq \frac{k_1+k_2}{2}$
s_1
y
t_1

Figure 7.3: Illustration how to find a shorter path from s_2 to t_2 in the proof of Theorem 7.2.5 if the path from s_2 to t_2 uses too many nearby-nodes.

For simple paths, the picture less clear. Indeed, Cai and Ye [59] left the complexity of the following problems open:

- UPDisjointSimPaths$(=, *)$: Given a undirected graph G, nodes s_1, t_1, s_2, and t_2, and a parameter $k \in \mathbb{N}$, are there two node-disjoint simple paths, one from s_1 to t_1 of length exactly k and one from s_2 to t_2?
- UPDisjointSimPaths$(\leq, *)$: Given a undirected graph G, nodes s_1, t_1, s_2, and t_2, and a parameter $k \in \mathbb{N}$, are there two node-disjoint simple paths, one from s_1 to t_1 of length at most k and one from s_2 to t_2?

Since it is not known if these problems are in FPT, also the complexities of PUSimPath$(a^k ba^*)$ and PUSimPath$(a^{\leq k} ba^*)$ remain open. We make the connection more concrete in the next lemma.

Lemma 7.2.6.

(a) PUSimPath$(a^k ba^*)$ is in FPT if and only if UPDisjointSimPaths$(=, *)$ is in FPT.

(b) PUSimPath$(a^{\leq k} ba^*)$ is in FPT if and only if UPDisjointSimPaths$(\leq, *)$ is in FPT.

Proof. We start with (a). We can solve UPDisjointSimPaths$(=, *)$ by (1) deleting all non-a-edges, (2) adding a simple path labeled ba^{k+1} from t_1 to s_2. Then there is are two disjoint simple path from s_1 to t_1 of length k and from s_2 to t_2 if and only if there is a path labeled $a^k ba^*$ from s_1 to t_2. Thus if PUSimPath$(a^k ba^*)$ is in FPT, then UPDisjointSimPaths$(=, *)$ is also in FPT.

On the other hand, we can solve PUSimPath$(a^k ba^*)$ by enumerating over all possible b-edges $\{u, v\}$ in E and asking for two disjoint simple path in G_a, either one from s to

u of length k and one from v to t, or one from s to v of length k and one from u to t. Since every simple path will use at most one edge between any pair of nodes, we can solve this problem on the underlying graph of G_a, that is, if there are multiple edges between a pair of nodes $\{x, y\}$ in G_a, we remove all but one edge between them. Thus, if $\mathsf{UPDisjointSimPaths}(=, *)$ is in FPT, $\mathsf{PUSimPath}(a^k ba^*)$ is also in FPT. The proof for (b) is analogous. □

Interestingly, we could not find a method to prove W[1]-hardness for any class of STEs in the undirected case.[8] Thus we leave as an open question whether there are classes of STEs for which PUTrail or PUSimPath are W[1]-hard.

[8] We cannot use an undirected version of G_{col} from Construction 5.4.2 because the direction of the b-edges depicted in Figure 5.6 ensures that at most one gadget per row can be skipped.

Part III

Enumeration

Chapter 8

Enumeration Framework

While in Parts I and II we focused on decision problems, we now want to output or enumerate witnessing paths. In this chapter we define the respective enumeration problems and give a general framework. In the next chapter, we show that the tractability results from Parts I and II carry over to the enumeration setting.

8.1 Preliminaries Enumeration

Given a path $p = e_1 \cdots e_n$ and $1 \leq i \leq j \leq n$, we denote by $p[i,j]$ the subpath $e_i \cdots e_j$. We denote the set of *edges of path* p with $E(p) = \{e_1, \ldots, e_n\}$. For convenience, we define $p[1,0] = \varepsilon$ and therefore $V(p[1,0]) = E(p[1,0]) = \emptyset$. Furthermore, let p be a path from s to t. We denote by $\text{destination}(p[i,j])$ end of the subpath $p[i,j]$ and define $\text{destination}(p[1,0]) = s$, that is, the start of p.

An *enumeration problem* $\mathcal{P}$ is a (partial) function that maps each input i to a finite or countably infinite set of *outputs for* i, denoted by $\mathcal{P}(i)$. Terminologically, we say that, given i, the task is to *enumerate* $\mathcal{P}(i)$.

An *enumeration algorithm* for $\mathcal{P}$ is an algorithm that, given input i, writes a sequence of answers to the output such that every answer in $\mathcal{P}(i)$ is written precisely once. If $\mathcal{A}$ is an enumeration algorithm for an enumeration problem $\mathcal{P}$, we say that $\mathcal{A}$ runs in *polynomial delay* if the time before writing the first answer and the time between writing every two consecutive answers is polynomial in $|i|$. By *between answers*, we mean the number of steps between writing the first symbol from an answer until writing the first symbol of the next answer. We use the term *preprocessing time* to refer to the computation time before writing the first answer.

For several enumeration problems, we will consider the *radix order* on paths. To this end, we assume that there exists an order $<$ on Σ. We extend this order to words and paths. For words w_1 and w_2, we say that $w_1 < w_2$ in radix order if $|w_1| < |w_2|$ or $|w_1| = |w_2|$ and w_1 is lexicographically before w_2. For two paths p_1 and p_2, we say that $p_1 < p_2$ in radix order if $\text{lab}(p_1) < \text{lab}(p_2)$.

To this end, a *parameterized enumeration problem* is defined analogously to an enumeration problem, but its input is of the form $(x, k) \in \Sigma^* \times \mathbb{N}$. It is in *FPT delay* if the preprocessing time (time before writing the first answer) and the time between writing every two consecutive answers is bounded by $f(k) \cdot |(x,k)|^c$ for a constant c and a computable function f.

Enumeration Problems

We now consider the following variants of SimPath and Trail:

EnumSimPaths(L)	
Given:	An directed multigraph G, nodes s, t.
Question:	Enumerate the simple paths from s to t in G that match L.

EnumTrails(L)	
Given:	An directed multigraph G, nodes s, t.
Question:	Enumerate the trails from s to t in G that match L.

Again, we will consider parameterized and undirected variants of these enumeration problems. We will add an U to the problem name when we consider undirected instead of directed multigraphs and parameterized versions will be denoted by a preceding P. More precisely, we will also consider the following variants:

- EnumUSimPaths(L): Given an undirected multigraph G, nodes s, t, enumerate the simple paths from s to t in G that match L.
- EnumUTrails(L): Given an undirected multigraph G, nodes s, t, enumerate the trails from s to t in G that match L.
- PEnumSimPaths($\mathcal{R}$): Given a directed multigraph G, nodes s, t, $r \in \mathcal{R}$, parameter $|r|$, enumerate the simple paths from s to t in G that match r.
- PEnumTrails($\mathcal{R}$): Given a directed multigraph G, nodes s, t, $r \in \mathcal{R}$, parameter $|r|$, enumerate the trails from s to t in G that match r.
- PEnumUSimPaths($\mathcal{R}$): Given an undirected multigraph G, nodes s, t, $r \in \mathcal{R}$, parameter $|r|$,enumerate the simple paths from s to t in G that match r.
- PEnumUTrails($\mathcal{R}$): Given an undirected multigraph G, nodes s, t, $r \in \mathcal{R}$, parameter $|r|$, enumerate the trails from s to t in G that match r.

8.2 Enumeration of Arbitrary Paths and Shortest Paths

We first show that enumeration for arbitrary and shortest paths can be done in polynomial delay on directed and undirected multigraphs.

Mendelzon and Wood [160] showed that for a given directed graph G, nodes s, t, and regular expression r, it can be decided in P if there is a path from s to t that matches r in G. Indeed, one only needs to construct the product of the graph (G, s, t) and an NFA N for the RPQ and test if (t, q_f) is reachable from (s, q_0), where q_0 and q_f are an initial and an accepting state of N, respectively. Since it is already allowed to use edges multiple times, this result immediately holds on directed multigraphs. This favorable complexity carries over to the enumeration setting for arbitrary paths. At the core lies the following result by Ackerman and Shallit.

Theorem 8.2.1 (Theorem 3 in [6]). *Given an NFA N and a number $\ell \in \mathbb{N}$ in unary, enumerating the words in $L(N)$ of length ℓ can be done in polynomial delay, even when the paths need to be enumerated in radix order.*

This result generalizes a result of Mäkinen [148], who proved that the words of length ℓ in $L(N)$ can be enumerated in polynomial delay if the finite automaton N is deterministic. Ackerman and Shallit genereralized this result to nondeterministic N and proved that, for a given length ℓ (which they call *cross-section*), the smallest word in radix order in $L(N)$ can be found in time $O(|Q|^2\ell^2)$, where Q is the set of states on N ([6], Theorem 1). They then prove that the set of all words of length ℓ can be enumerated in radix order in total time $O(|Q|^2\ell^2 + |\Sigma||Q|^2x)$, where x is the sum of the length of the words of length ℓ ([6], Theorem 2). A closer inspection of their algorithm shows that it has delay $O(|\Sigma||Q|^2|w|)$ where $|w|$ is the size of the next output.

We now extend the algorithm of Ackerman and Shallit to enumerate paths in a directed multigraph.

Lemma 8.2.2. *Given a directed multigraph G, nodes s, t, a number $\ell \in \mathbb{N}$, and a regular expression r, we can enumerate all paths from s to t of length ℓ that match r in polynomial delay, even when the paths need to be enumerated in radix order.*

Proof. We first construct an NFA N_r for r and take the product with G. In the directed multigraph $G \times N_r$ we change the label of every edge $(e, (q_1, q_2))$ to $(\text{lab}(e), e)$ (instead of $\text{lab}(e)$). This way, every edge between two nodes has a different label, and thus we have constructed a directed (simple) graph, which we name G'.

We interpret G' as an NFA and define its set of initial states as $\{(s, i) \mid i$ is an initial state of $N_r\}$ and its set of accepting states as $\{(t, f) \mid f$ is an accepting state of $N_r\}$. Enumerating the words of length ℓ from the resulting automaton and projecting away the first component of each label corresponds to enumerating the paths from s to t in G of length ℓ that match r.

The first component of each label is indeed used to ensure radix order. We explain this in more detail now. We extend the order $<$ on Σ to an order $<$ on (Σ, E) as follows: Let $<_E$ be an arbitrary order on E. We define $<$ on (Σ, E) as follows: $(a_1, e_1) < (a_2, e_2)$ if and only if $a_1 < a_2$ or $a_1 = a_2$ and $e_1 <_E e_2$. This order implies that, given two paths $(\text{lab}(e_1), e_1) \cdots (\text{lab}(e_\ell), e_\ell)$ and $p_2 = (\text{lab}(e'_1), e'_1) \cdots (\text{lab}(e'_\ell), e'_\ell)$, then $p_1 < p_2$ implies $\text{lab}(e_1) \cdots \text{lab}(e_\ell) < \text{lab}(e'_1) \cdots \text{lab}(e'_\ell)$ or $\text{lab}(e_1) \cdots \text{lab}(e_\ell) = \text{lab}(e'_1) \cdots \text{lab}(e'_\ell)$.

By Theorem 8.2.1 we can enumerate the words of length ℓ in G' in polynomial delay in radix order. By projecting away the first component of each label, we enumerate the paths from s to t of length ℓ in G in polynomial delay in radix order. □

Furthermore, the length of a shortest path from s to t in G that matches r is polynomial in the size of G and r. Thus it follows that we can also enumerate shortest paths in polynomial delay.

Lemma 8.2.3. *Given a directed multigraph G, nodes s, t, and a regular expression r, we can enumerate all shortest paths from s to t that match r in polynomial delay, even when the paths need to be enumerated in radix order.*

We note that the results from Lemmas 8.2.2 and 8.2.3 also hold for undirected multigraphs.

Theorem 8.2.4. *Given a directed or undirected multigraph G, nodes s, t, a number $\ell \in \mathbb{N}$, and a regular expression r, we can*

- *enumerate all paths from s to t in G of length ℓ that match r*
- *enumerate all shortest paths from s to t in G that match r*

in polynomial delay, even when the paths need to be enumerated in radix order.

Proof. For directed multigraphs, the result follows from Lemmas 8.2.2 and 8.2.3. So let $G = (V, E, \mathcal{E})$ be an undirected multigraph. The idea is to replace every undirected edge with two directed edges. More formally, we construct a directed multigraph $G' = (V', E', \mathcal{E}')$ with $V' = V, E' = \{e_1, e_2 \mid e \in E\}$, and $\mathcal{E}'$ such that if $\mathcal{E}(e) = (u, a, v)$, then $\mathcal{E}'(e_1) = (u, a, v)$ and $\mathcal{E}'(e_2) = (v, a, u)$. We then use Lemma 8.2.2 or 8.2.3 to enumerate paths in G'. If we replace in the output every directed edge e_1 or e_2 with its original edge e, then we have a path in G. $\square$

8.3 Enumerating Simple Paths: Yen's algorithm

We now turn to enumerating simple paths in polynomial delay. A starting point is Yen's algorithm [209] for enumerating simple paths from a source s to target t, without label constraints. Yen's algorithm usually takes another parameter K and returns the K shortest simple paths. Yen's algorithm without this parameter K is presented in Algorithm 6.

We give a high-level explanation. First, observe that each shortest path in a graph is also a simple one. Therefore, the first solution is obtained by finding a shortest path p. The next shortest path must differ in some edge from p. So we search (if it exists), for all i, the shortest path that shares the first i edges with p, but not the $(i+1)$th edge. One of the shortest paths found this way is the next solution, which we again store in p. The next shortest path must again differ in some edge from the paths we already found. So we search again, for all i, for a shortest path that shares the first i edges with the new p, but not the $(i+1)$th edge. To avoid rediscovering an old path, we also forbid other edges to appear in the new path (lines 9–10).

We give an sketch of the correctness proof given in [209] next.

Theorem 8.3.1 (Implicit in [209])**.** *Given a directed or undirected graph G and nodes s and t, Algorithm 6 enumerates all simple paths from s to t in polynomial delay.*

Algorithm 6: Yen's algorithm

```
   Input: Directed or undirected graph $G = (V, E, \mathcal{E})$, nodes $s, t$
   Output: The simple paths from $s$ to $t$ in $G$ in order of increasing length
1  $A \leftarrow \emptyset$                    ▷ $A$ is the set of paths already written to output
2  $B \leftarrow \emptyset$                    ▷ $B$ is a set of paths from $s$ to $t$
3  $p \leftarrow$ a shortest path from $s$ to $t$ in $G$
4  while $p \neq null$ do
5      output $p$
6      Add $p$ to $A$
7      for $i = 0$ to $|p| - 1$ do
8          $G' \leftarrow$ the induced subgraph of $G$ on $V' = V - V(p[1, i])$
9          for every path $p_1$ in $A$ with $p_1[1, i] = p[1, i]$ do
10             Delete the edge $p_1[i+1, i+1]$ in $G'$      ▷ Also deletes $p[i+1, i+1]$ since $p \in A$
           ▷ $G'$ now no longer has paths already in $A$
11         $p_2 \leftarrow$ a shortest path from destination($p[1, i]$) to $t$ in $G'$
12         Add $p[1, i] \cdot p_2$ to $B$
13     $p \leftarrow$ a shortest path in $B$               ▷ $p \leftarrow$ null if $B = \emptyset$
14     Remove $p$ from $B$
```

Proof sketch. The original algorithm of Yen [209] finds, for a given G, s, t, and $K \in \mathbb{N}$, the K shortest simple paths from s to t in G. Its difference with Algorithm 6 is that it stops after K paths have been returned.

Let $G = (V, E, \mathcal{E})$ be a directed or undirected graph. Yen does not prove that his algorithm has polynomial delay, but instead shows that the delay is $O(K|V| + |V|^3)$.[1] On lines 3 and 11, he uses an algorithm from [210] to find a shortest, and therefore simple, path in time $O(|V|^2)$. (Instead, one can also use Dijkstra's algorithm or breadth-first search.)

Unfortunately, K can be exponential in $|G|$ in general. However, the reason why the algorithm has K in the complexity is line 9, which iterates over all paths in A. If we do not store A as a linked list as in [209] but as a prefix tree of paths instead, the algorithm only needs $O(|V| + |E|)$ steps to complete the entire for-loop on line 9 (without any optimizations). Indeed, if paths p and p'' share the first i edges, they will share a path of length i from the root node in the prefix tree. So we can find all forbidden $(i+1)$th edges by forbidding all edges that start at the end of this path. We therefore obtain delay $O(|V|^3 + |V||E|)$ from Yen's analysis. □

[1] In [209], Section 5, he notes that computing path number k in the output costs, in his terminology, $O(KN)$ time in Step I(a) and $O(N^3)$ in Step I(b), with $N = |V|$.

8.4 New Variants of Yen's algorithm

Yen's Algorithm on Labeled Multigraphs

Algorithm 7: Yen's algorithm with regular expression

```
Input: Directed or undirected multigraph $G = (V, E, \mathcal{E})$, nodes $s, t$, regular
       language $L$
Output: The simple paths from $s$ to $t$ in $G$ that match $L$
1  $A \leftarrow \emptyset$                ▷ $A$ is the set of paths already written to output
2  $B \leftarrow \emptyset$                ▷ $B$ is a set of candidate paths from $s$ to $t$
3  $p \leftarrow$ a shortest simple path from $s$ to $t$ in $G$ that matches $L$
4  while $p \neq null$ do
5    output $p$
6    Add $p$ to $A$
7    for $i = 1$ to $|p| - 1$ do
8      $G' \leftarrow$ the induced subgraph of $G$ on $V' = V - V(p[1, i])$
9      for every path $p_1$ in $A$ with $p_1[1, i] = p[1, i]$ do
10       Delete the edge $p_1[i + 1, i + 1]$ in $G'$
       ▷ $G'$ now no longer has paths already in $A$
11     $p_2 \leftarrow$ a shortest simple path from destination($p[1, i]$) to $t$ in $G'$ that matches
       $(\mathrm{lab}(p[1, i]))^{-1}L$
12     Add $p[1, i] \cdot p_2$ to $B$
13   $p \leftarrow$ a shortest path in $B$                ▷ $p \leftarrow$ null if $B = \emptyset$
14   Remove $p$ from $B$
```

Since we focus on multigraphs in this work, we first note that Yen's algorithm easily extends to multigraphs since each simple path can use at most one edge between any pair of nodes. Furthermore, Yen's algorithm can easily be adapted to solve EnumSimPaths for regular languages, see Algorithm 7. Indeed, the only changes occur in lines 3 and 11, where the subroutines now need to find a shortest simple path matching a given regular language instead of only a shortest path. Yet the main idea of Yen's algorithm still holds for paths matching a regular language L, and therefore correctness follows immediately from Yen's original algorithm.

Remark 8.4.1. Algorithm 7 makes two important calls to a black box algorithm for computing a shortest simple path that matches a regular language, namely on lines 3 and 11. (There is another mention of "shortest path" on line 13, but here we only need to find a shortest path stored in B.)

Algorithm 8: Yen's algorithm changed to work with trails on multigraphs

```
Input: Directed or undirected multigraph $G = (V, E, \mathcal{E})$, nodes $s, t \in V$, a
       regular language $L$
Output: All trails from $s$ to $t$ in $G$ that match $L$ under bag semantics
1  $A \leftarrow \emptyset$                        ▷ $A$ is the set of trails already written to output
2  $B \leftarrow \emptyset$                        ▷ $B$ is a set of trails from $s$ to $t$ matching $L$
3  $p \leftarrow$ a shortest trail from $s$ to $t$ matching $L$   ▷ $p \leftarrow \perp$ if no such trail exists
4  while $p \neq \perp$ do
5      output $p$
6      Add $p$ to $A$
7      for $i = 0$ to $|p|$ do
8          $G' \leftarrow (V, E - E(p[1,i]), \mathcal{E}|_{E-E(p[1,i])})$   ▷ Delete the edges of $p[1,i]$
9          $S = \{e \in E \mid p[1,i] \cdot e$ is a prefix of a trail in $A\}$
10         $p_1 \leftarrow$ a shortest trail from destination($p[1,i]$) to $t$ in $G'$ that matches
             $((\mathrm{lab}(p[1,i]))^{-1}L) \setminus \{\varepsilon\}$ and does not start with an edge from $S$
11         Add $p[1,i] \cdot p_1$ to $B$
12     $p \leftarrow$ a shortest trail in $B$          ▷ $p \leftarrow \perp$ if $B = \emptyset$
13     Remove $p$ from $B$
```

Yen's Algorithm on Labeled Multigraphs for Trails

In Algorithm 8 we provide yet another variant of Yen's algorithm, this time enumerating trails matching a regular language.

The reason for this variant are twofold. While Lemma 5.5.3 allows us to enumerate trails from s to t matching L in a directed multigraph G by enumerating simple paths matching L in directed graphs $(H, s_1, t_1), \ldots, (H, s_n, t_n)$, there is no equivalent of this lemma on undirected graphs. Furthermore, the complexity might rise: since $\mathsf{SP}_{\mathsf{tract}}$ is a strict subset of $\mathsf{T}_{\mathsf{tract}}$, it is not clear whether this method allows a polynomial delay algorithm for all languages in $\mathsf{T}_{\mathsf{tract}}$.[2]

The changes in Algorithm 8 are straightforward: instead of deleting nodes, we only delete edges. The correctness again follows from Yen's algorithm.

Enumerating with Possibly Non-Shortest Path Subroutines

Our next variants of Yen's algorithm are minor changes to Algorithms 7 and 8 that allow us to use subroutines that return paths which are not necessarily shortest.

[2]We recall that the graphs obtained from Lemma 5.5.3 have a special structure. This structure allows us to decide $\mathsf{SimPath}(L)$ in polynomial time even for some languages not in $\mathsf{SP}_{\mathsf{tract}}$. For example, we can decide in polynomial time whether such a directed graph H has a simple path matching $(ab)^*$ from s to t because every node only has outgoing a- or outgoing b-edges, which allows us to remove loops. Indeed, it would be interesting whether $\mathsf{SimPath}(L)$ restricted to such graphs is tractable for all languages $L \in \mathsf{T}_{\mathsf{tract}}$.

These variants will be handy when there is no (known) polynomial time algorithm for finding a shortest path.

Proposition 8.4.2. *There is no known polynomial time algorithm that returns a shortest simple path (or trail) from s to t that matches a^*bca^*.*

Proof. We will reduce the problem to the *min-sum k disjoint paths problem.* This problem asks, given an undirected graph, nodes $s_1, t_1, \dots, s_k, t_k$ for k disjoint paths, one from s_i to t_i for each $i \in [k]$, such that the sum of their lengths is minimal. For each integer k, this problem comes in a node-disjoint and an edge-disjoint variant.

The complexity of min-sum k disjoint paths problem for $k \geq 2$ has been open for several decades, although it has been extensively studied, see Fenner et al. [92] for an overview of the edge-disjoint variant and Kobayashi and Sommer [139] for an overview of the node-disjoint variant.[3]

We now show that finding a shortest simple path from s to t that matches a^*bca^* is possible in polynomial time if and only if the *min-sum two node-disjoint paths problem* can be solved in polynomial time.

- To return a shortest simple path from s to t that matches a^*bca^* in a undirected multigraph G, we first iterate over all possible triples of pairwise different nodes x, y, z. For each such triple, we test for an b-edge from x to y and a c-edge from y to z. If this test succeeds, we only need to find two node-disjoint path in G_a without y, one from s to x and one from z to t, such that the sum of their lengths is minimal. Although G_a is a undirected multigraph, it suffices to consider its underlying undirected graph, since a shortest path will not use multiple edges between nodes and the node-disjointness ensures that the two different paths will not use an edge between the same pair of nodes. Thus this problem is exactly the min-sum two node-disjoint paths problem. After having iterated over all triples, we return the overall shortest path.
- On the other hand, we can find two node-disjoint paths from s_1 to t_1 and from s_2 to t_2 such that the sum of their length is minimal as follows: We relabel every edge with a and add a path labeled bc from t_1 to s_2. Then the problem reduces to finding a shortest simple path matching a^*bca^* from s_1 to t_2 and returning the respective subpaths.

We now turn to trail semantics. That is, we now show that finding a shortest trail from s to t that matches a^*bca^* is possible in polynomial time if and only if the *min-sum two edge-disjoint paths problem* can be solved in polynomial time.

- To return a shortest trail from s to t that matches a^*bca^* in a directed multigraph G, we first iterate over all possible pairs of nodes u, v and edges e_1, e_2 with

[3]We note that there exists a polynomial time Monte Carlo algorithm for the case $k = 2$, see Björklund and Husfeld [47], but no deterministic polynomial time algorithm is known. If there is only one start and one endnode, a polynomial time algorithm for min-sum k node-/edge-disjoint paths with $k = 2$ was introduced by [192, 193] and later extended to arbitrary $k \geq 2$ by Bhandari [44]. Yang and Zheng [207] gave a polynomial time algorithm for the case $k = 2$ with $s_1 = s_2$ and $t_1 \neq t_2$.

lab$(e_1) = b,$ lab$(e_2) = c$ such that e_1e_2 is a path from u to v. Since G and therefore G_a is a multigraph, while the the min-sum two edge-disjoint paths problem is defined over undirected (simple) graphs, we cannot use G_a directly. To this end, we construct a simple graph G' by (1) removing all edges of the form (z, a, z) from G_a and (2) replacing every edge with two new ones and an extra node. More formally, let $G_a = (V, E_a, \mathcal{E}_a)$ be the subgraph of G restricted to a-edges. We define $G' = (V', E', \mathcal{E}')$ with $V' = V \cup \{z_e \mid e \in E_a\}$, $E' = \{e_1, e_2 \mid e \in E_a \text{ and } |\mathsf{Node}(e)| > 1\}$, and if $\mathcal{E}_a(e) = (x, a, y)$, then $\mathcal{E}'(e_1) = (x, a, z_e)$ and $\mathcal{E}'(e_2) = (z_e, a, y)$. Now we only need to find two edge-disjoint paths in G', one from s to u and one from v to t, such that the sum of their lengths is minimal, which is exactly the min-sum two edge-disjoint paths problem.

After having iterated over all pairs of nodes and edges, we return the overall shortest trail.

- On the other hand, we can find two edge-disjoint paths from s_1 to t_1 and from s_2 to t_2 such that the sum of their length is minimal as follows: We relabel every edge with a and add a path labeled bc from t_1 to s_2. Then the problem reduces to finding a shortest trail matching a^*bca^* from s_1 to t_2 and returning the respective subpaths.

This completes the proof. □

We now revisit the correctness proof of Yen's algorithm [209], in order to show that Algorithm 7 (and therefore also Algorithm 6) also works correctly if the subroutines only return *simple* instead of shortest paths.

Lemma 8.4.3. *Algorithm 7 is also correct if the assignments on lines 3 and 11 assign simple (possibly non-shortest) paths to* p *and* p_2*, respectively.*

Proof. On line 10, Algorithm 7 considers the last path p that was added to A and compares it to every path p_1 in A that shares the first i edges with p. Since the algorithm deletes, for every such path p_1, the edges $p_1[i + 1, i + 1]$, we cannot find a simple path more than once. So we just have to show that every simple path is eventually found.

Clearly, the algorithm is correct if there exists at most one simple path from s to t. Assume, towards a contradiction, that there is more than one such simple path and the algorithm terminated, that is, B is empty, but there exists a simple path p' from s to t that matches L and is not in A. Let C be the set of paths in A that share the longest prefix with p'. Since all paths in A start in s and $A \neq \emptyset$ due to line 3, we have that $C \neq \emptyset$. Let i be the maximal integer with $p[1, i] = p'[1, i]$ for all $p \in C$. Let $p \in C$ be the last path that was added to A. After adding p to A, we must have executed the for loops again. In line 10, we then cannot have deleted the edge $p'[i + 1, i + 1]$, otherwise this would contradict the definition of C. Since p was the last path from C that we added to A and B is empty by assumption, p' must have been found during this execution of the while-loop. Thus we must have $p' \in B$ or $p' \in A$. Contradiction. □

While under simple path semantics we only needed to omit the “shortest” in Algorithm 7 from lines 3 and 11, we have to be more careful under trail semantics. More precisely, we additionally need to test before returning a path p whether it has a prefix which was not yet written to the output. The reason herefore is that, if there are trails p_1, p_2 from s to t such that p_1 is a prefix of p_2 and if p_2 is first written to the output, then the “original” algorithm might not find p_1.

Lemma 8.4.4. *Algorithm 8 is also correct if we make the following changes: (1) the assignments on lines 3 and 10 assign (possibly non-shortest) trails to* p *and* p_2*, respectively, and (2) in line 12 it is additionally tested if* p *contains a prefix* p' *which is a trail from* s *to* t *that matches* L *and is not yet in* A*. If this test succeeds, we take a shortest such prefix instead of* p*.*

Proof. On line 8, Algorithm 8 considers the last path p that was added to A and compares it to every path p_1 in A that shares the first i edges with p. Since the algorithm forbids, for every such path p_1, the next edge to be $p_1[i+1, i+1]$, we cannot find a trail more than once. So we just have to show that every trail is eventually found.

Clearly, the algorithm is correct if there exists at most one trail from s to t that matches L. Assume, towards a contradiction, that there is more than one such trail and the algorithm terminated, that is, B is empty, but there exists a trail p' from s to t that matches L and is not in A. Due to (2), p' cannot be a prefix of a path already in A. Let C be the set of paths in A that share the longest prefix with p'. Since all paths in A start in s and $A \neq \emptyset$ due to line 3, we have that $C \neq \emptyset$. Let i be the maximal integer with $p[1, i] = p'[1, i]$ for all $p \in C$. We note that $i < |p'|$ due to (2).

Let $p \in C$ be the last path that was added to A. After adding p to A, we must have executed the for loops again. In line 8, we then cannot have deleted the edge $p'[i+1, i+1]$, otherwise this would contradict the definition of C. Since p was the last path from C that we added to A and B is empty by assumption, p' must have been found during this execution of the while-loop. Thus we must have $p' \in B$ or $p' \in A$. Contradiction. ☐

8.5 The Framework

We now study the time guarantees of the enumeration algorithms. The time needed by Algorithms 7 directly depends on the two subroutines in lines 3 and 11. The next lemma implies that if there are polynomial time algorithms (FPT algorithms, respectively) for the subroutines, then the enumeration is possible in polynomial delay (FPT delay, respectively).

Lemma 8.5.1. *Let* $\mathcal{R}$ *be a class of regular expressions. If there exist algorithms* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *that, when given as input a directed (or undirected, respectively) multigraph* G*, nodes* s *and* t*, a word* w *with* $|w| \leq |G|$*, and* $r \in \mathcal{R}$*, return in time* $f(|G|, |r|)$ *(with* $f(|G|, |r|) \geq |G|$*),*

1. *a simple path from* s *to* t *in* G *that matches* $L(r)$ *if it exists and “no” otherwise and*

2. *a simple path from s to t in G that matches $w^{-1}L(r)$ if it exists and "no" otherwise*

respectively, then EnumSimPaths($\mathcal{R}$) *(or* EnumUSimPaths($\mathcal{R}$)*, respectively) is in delay $O(|V|f(|G|,|r|))$ with preprocessing time $O(f(|G|,|r|))$.*

Furthermore, if $\mathcal{A}_1$ and $\mathcal{A}_2$ always return a shortest simple path (respectively, a smallest simple path in radix order), then the enumeration can be done in order of increasing length (respectively, in radix order), with the same time guarantees.

Proof. We use Algorithm 7 with calls to algorithm $\mathcal{A}_1$ on line 3 and to algorithm $\mathcal{A}_2$ on line 11. Furthermore, we choose an arbitrary path, shortest path, or smallest path in radix order in B on line 13, depending on whether we want to enumerate in arbitrary order, order of increasing length, or radix order, respectively. The correctness for EnumSimPaths($\mathcal{R}$) and EnumUSimPaths($\mathcal{R}$) follows from Lemma 8.4.3.

Clearly, we need time $O(f(|G|,|r|))$ to output the first path (if it exists). Then, Algorithm 7 does up to $|V|$ iterations in line 7. We note that in $|A|$ we will only store paths of length at most $|V|$. If we use a prefix tree as a data structure for A, we can insert a path in $O(|V| \cdot |E|)$ time or find a path p in A in $O(|V|)$ time. Thus we can also find the right node in the prefix tree and then delete the up to $|E|$ many outgoing edges in G line 10 in $O(|V| + |E|)$. In line 11, we call algorithm $\mathcal{A}_2$.

In line 13 we need to find a minimal path among the candidates in B. If we again use a prefix tree as a data structure and start with $|p|$ instead of the first node in p, we can always output the leftmost path (without the $|p|$), which is a minimal simple path. Finding and deleting are in time $O(|V|)$. Thus, we have a delay of $O(f(|G|,|r|))$ until the first output, and afterwards time $O(|V|(|V| + |E| + f(|G|,|r|)))$. □

We can give similar time guarantees for Algorithm 8. Here, the time needed depends on the subroutines in lines 3 and 10.

Lemma 8.5.2. *Let $\mathcal{R}$ be a class of regular expressions. If there exist algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ that, when given as input a directed (or undirected, respectively) multigraph G, nodes s and t, a word w with $|w| \leq |G|$, and $r \in \mathcal{R}$, return in time $f(|G|,|r|)$ (with $f(|G|,|r|) \geq |G|$),*

1. *a trail from s to t in G that matches $L(r)$ if it exists and "no" otherwise and*
2. *a trail from s to t in G that matches $w^{-1}L(r)$ if it exists and "no" otherwise*

respectively, then EnumTrails($\mathcal{R}$) *(or* EnumUTrails($\mathcal{R}$)*, respectively) is in delay $O(|E|^2 \cdot f(|G|,|r|))$ with preprocessing time $O(f(|G|,|r|))$.*

Furthermore, if $\mathcal{A}_1$ and $\mathcal{A}_2$ always return a shortest trail (respectively, a smallest trail in radix order), then the enumeration can be done in order of increasing length (respectively, in radix order), with the same time guarantees.

Proof. We use the variant of Algorithm 8 described in the statement of Lemma 8.4.4, with calls to algorithm $\mathcal{A}_1$ on line 3, and up to $|E|$ calls of algorithm $\mathcal{A}_2$ on line 10. Furthermore, we choose an arbitrary path, shortest path, or smallest path in radix order in B on line 12, depending on whether we want to enumerate in arbitrary order, order of increasing length, or radix order, respectively. The correctness for $\mathsf{EnumTrails}(\mathcal{R})$ and $\mathsf{EnumUTrails}(\mathcal{R})$ follows from Lemma 8.4.4.

Clearly, we need time $O(f(|G|, |r|))$ to output the first path (if it exists). Then, Algorithm 8 does up to $|E|$ iterations in line 7. We note that all paths are trails and therefore have length at most $|E|$. If we use a prefix tree as a data structure for A, we can insert a path in time $O(|E|^2)$ or find a path p in A in $O(|E|)$ time. Thus we can also find the right node in the prefix tree and then compute S in line 9 in $O(|E|)$ time. In line 10, we need multiple calls to $\mathcal{A}_2$ to ensure that the path starts with an edge not in S. More precisely, let $G' = (V', E', \mathcal{E}')$ for each edge $e \in E' - S$, we compute a new multigraph $G'_e = (V'_e, E'_e, \mathcal{E}'_e)$ by adding a new node s' to G', and change $\mathcal{E}'$ such that $\mathcal{E}'_e(e) = (s', \text{lab}(e), v)$ instead of $\mathcal{E}'(e) = (\text{destination}(p[1,i]), \text{lab}(e), v)$. Then there is a trail from s' to t in G'_e that matches $((\text{lab}(p[1,i]))^{-1}L) \setminus \{\varepsilon\}$ if and only if there is such a trail from destination$(p[1,i])$) to t in G' that starts with e. Furthermore, these trails use the same edges. Thus we can call $\mathcal{A}_2$ on each G'_e with $e \in E' - S$ and take an arbitrary, an overall shortest, or an overall smallest trail in radix order as p_1.

In line 12 we need to find a minimal path among the candidates in B. If we again use a prefix tree as a data structure and start with $|p|$ instead of the first node in p, we can always output the leftmost path (without the $|p|$), which is a minimal simple path. Finding and deleting are in time $O(|E|)$. Thus, we have a delay of $O(f(|G|, |r|))$ until the first output, and afterwards delay $O(|E|(|E| + |E| \cdot f(|G|, |r|)))$. $\square$

Chapter 9

Enumeration Results

We now use the lemmas from Section 8.5 to show tractability for several enumeration problems.

9.1 With Arbitrary Order

We use the standard method of self-reducibility to query a decision algorithm multiple times in order to reconstruct the solution. That is, each polynomial time algorithms that decides whether a simple path or trail matching L exists in G can be used to obtain a polynomial time algorithm that can return such a path (if it exists).

Lemma 9.1.1. *Let $G = (V, E, \mathcal{E})$ be a directed or undirected multigraph, s and t two nodes, and $\mathcal{B}$ be an algorithm that decides in time x whether there is a simple path (respectively trail) from s to t in G that matches L. Then there exists an algorithm that returns in time $O(|E| \cdot x)$ (respectively $O(|E|^2 \cdot x)$) a simple path (respectively trail) from s to t in G that matches L if one exists, and "no" otherwise.*

Proof. For simple paths, we construct an algorithm $\mathcal{A}$ as follows: If $\mathcal{B}$ returns "no", $\mathcal{A}$ will also return "no". Otherwise, there exists a simple path from s to t in G which matches L. Let e_1 be an edge adjacent to s. Let G' be the graph obtained from G by removing all edges adjacent to s except e_1. If $\mathcal{B}$ returns "no", we remove e_1 from G and repeat the procedure with the next edge adjacent to s, otherwise, if $\mathcal{B}$ returns "yes" on graph G', we can choose e_1 as first edge and remove the other edges adjacent to s (that is, except e_1) permanently from G. After we have found the ith edge, ending in a node u with $u \neq t$, we can find the i+1th edge with a similar method: Let e_{i+1} be an edge adjacent to u. Let G' be the graph obtained from G by removing all edges adjacent to u except e_i and e_{i+1}. If $\mathcal{B}$ returns "no", we remove e_{i+1} from G and repeat the procedure with the next edge adjacent to u, otherwise, if $\mathcal{B}$ returns "yes" on graph G', we can choose e_{i+1} as i+1th edge and remove the other edges adjacent to u (that is, except e_i and e_{i+1}) permanently from G. Once we found an edge e_ℓ ending in t, we return $e_1 \cdots e_\ell$.

The so-constructed path $e_1 \cdots e_\ell$ is a solution by construction. Furthermore, since every edge is considered at most once, the running time is bounded by $O(|E| \cdot x)$.

We now turn to trail semantics. We construct algorithm $\mathcal{A}$ as follows: If $\mathcal{B}$ returns "no", $\mathcal{A}$ will also return "no". Otherwise, there exists a trail from s to t which matches L.

We enumerate over all edges e_1 adjacent to s. Let $\mathcal{E}(e_1) = (s, a, u)$. We define $G' = (V', E', \mathcal{E}')$ to be the multigraph obtained from $G = (V, E, \mathcal{E})$ by adding a new node s' and defining $\mathcal{E}'(e_1) = (s', a, u)$. We now use $\mathcal{B}$ to decide the existence of a trail from s' to t in G' that matches L. If $\mathcal{B}$ returns "no", there is no solution which uses e_1 as first edge, so we continue with the next edge adjacent to s. If $\mathcal{B}$ returns "yes", then there is a solution which uses e_1 as first edge, and we permanently delete e_i from G.

After we have found the ith edge, ending in a node u, we can find the i+1th edge with a similar method: Let e_{i+1} be an edge adjacent to u and $e_i \neq e_{i+1}$. Let G' be the graph obtained from G by removing e_{i+1}, and adding a new node s' and a path labeled $\text{lab}(e_1 \cdots e_{i+1})$ from s' to u. We use $\mathcal{B}$ to decide if there is a trail from s' to t matching L in G'. If it returns "no", we repeat the procedure with the next edge adjacent to u, otherwise, if $\mathcal{B}$ returns "yes" on graph G', we can choose e_{i+1} as i+1th edge and permanently remove e_{i+1} from G. Once we found an edge e_ℓ ending in t such that $\text{lab}(e_1 \cdots e_\ell) \in L$, we return $e_1 \cdots e_\ell$. If e_ℓ ends in t, but $\text{lab}(e_1 \cdots e_\ell) \notin L$, we continue with the procedure.

The so-constructed path $e_1 \cdots e_\ell$ is a solution by construction. Furthermore, since the returned trail has length at most $|E|$ and there are at most $|E|$ candidates for the ith edge, the running time is bounded by a polynomial in $O(|E|^2 \cdot x)$. □

This implies that we can not only find, but also return simple paths and trails that match languages in the tractable classes $\mathsf{SP}_{\mathsf{tract}}$, $\mathsf{USP}_{\mathsf{tract}}$, $\mathsf{T}_{\mathsf{tract}}$, and $\mathsf{UT}_{\mathsf{tract}}$. Since all these classes are closed under taking derivatives, see Lemma 3.5.3 and Theorem 6.3.1, the algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ required in Lemmas 8.4.3 and 8.4.4 exist for these languages. This implies the next theorem.

Theorem 9.1.2. *The following problems are in polynomial delay:*

- *EnumSimPaths*(L) *for each* $L \in \mathsf{SP}_{\mathsf{tract}}$
- *EnumUSimPaths*(L) *for each* $L \in \mathsf{USP}_{\mathsf{tract}}$*,*
- *EnumTrails*(L) *for each* $L \in \mathsf{T}_{\mathsf{tract}}$*, and*
- *EnumUTrails*(L) *for each* $L \in \mathsf{UT}_{\mathsf{tract}}$*.*

In order to extend our tractability on the parameterized versions, We first show that derivatives of STEs are unions of STEs with at most the same cut border and at most the same number of conflict positions.

Lemma 9.1.3. *Let $w \in \Sigma^*$ and r be a c-bordered STE of size n. Then $w^{-1}L(r)$ is a union of STEs $r_1, \ldots, r_m$ that can be computed in time $O(|w||r|)$ such that*

- *$m \leq n$ and*
- *each r_i is c'-bordered for some $c' \leq c$.*

Furthermore, if r is an STE with at most c conflict positions then, every STE in $w^{-1}L(r)$ also has at most c conflict positions.

Proof. Let $r = B_1 \cdots B_n$ be a c-bordered STE such that each B_i is either of the form A, $A?$, or T^* as in Definition 4.2.1. Let $w \in \Sigma^*$, $J_w = \{j \mid w \in L(B_1 \cdots B_j)\}$. Then a regular expression for $w^{-1}L(r)$ consists of the union

$$\Sigma_{j \in J_w} B_{j+1} \cdots B_n$$

and, if $w \in L(B_1 \cdots B_j)$ with $B_j = T^*$, we add $B_j \cdots B_n$ to the union.

Clearly, the union is of size at most n, and since each expression $B_{j+1} \cdots B_n$ or $B_j \cdots B_n$ is c'-bordered for some $c' \leq c$ by definition, the result follows. Since we can test $w \in L(r)$ in $O(|w||r|)$, we can compute J_w and therefore also the derivatives in $O(|w||r|)$. □

Example 9.1.4. For the regular expression $r = a^*aab$ and the word $w = aaa$, the derivative $w^{-1}L(r)$ is $\{a^*aab + aab + ab + b\}$.

Theorem 9.1.5. *Let $\mathcal{R}$ be a class of STEs.*

- *If $\mathcal{R}$ is cuttable, then* PEnumSimPaths$(\mathcal{R})$ *and* PEnumUSimPaths$(\mathcal{R})$ *are in FPT delay.*
- *If $\mathcal{R}$ is almost conflict-free, then* PEnumTrails$(\mathcal{R})$ *and* PEnumUTrails$(\mathcal{R})$ *are in FPT delay.*

Proof. By Lemmas 5.6.3 and 5.3.16, and Theorems 7.1.1 and 7.1.2 there exist FPT algorithms for the respective decision problems. By Lemma 9.1.1 we thus also have FPT algorithms that can return a solution. Lemma 9.1.3 guarantees that the derivative $w^{-1}L(r)$ of an STE r is a union of STEs, each having at most the same cut border and at most the same number of conflict positions as r. Thus, we can also find a solution matching $w^{-1}L(r)$ by using the respective FPT algorithm that returns a solution on STE in the union separately.

We then use these algorithms as $\mathcal{A}_1$ and $\mathcal{A}_2$ in Lemma 8.4.3 in the case of simple paths or in Lemma 8.4.4 in the case of trails to obtain an FPT delay algorithm for the respective enumeration problem. □

9.2 With Order of Increasing Length

We now want to enumerate the paths in order of increasing length. Such an order is helpful if the user only wants to the the most relevant answers first or if the user is only interested in the K shortest paths—just as the original version of Yen's algorithm. In this section we will only give results on directed multigraphs. The reason is that on undirected multigraphs there are languages for which we can find an arbitrary simple path or trail in polynomial time (with Lemma 9.1.1), but it is not known if there is an algorithm that returns a shortest such path in (deterministic) polynomial time, see Proposition 8.4.2.

Theorem 9.2.1. *The problems*

- *$\mathsf{EnumSimPaths}(L)$ for each $L \in \mathsf{SP}_{\mathsf{tract}}$ and*
- *$\mathsf{EnumTrails}(L)$ for each $L \in \mathsf{T}_{\mathsf{tract}}$*

are in polynomial delay, even when the paths need to be enumerated in order of increasing length.

Proof. Bagan et al. [20] observe that their algorithm for $\mathsf{SimPath}(L)$ for $L \in \mathsf{SP}_{\mathsf{tract}}$ can easily be adapted to output a shortest path for positive instances in NL.

In Corollary 3.4.15 we describe an algorithm that can output a shortest trail from s to t matching L for $L \in \mathsf{T}_{\mathsf{tract}}$ in NL.

We note that $\mathsf{SP}_{\mathsf{tract}}$ and $\mathsf{T}_{\mathsf{tract}}$ are closed under derivatives, see Lemma 3.5.3. Thus we can use these algorithms for as subprocedures in Lemmas 8.4.3 and 8.4.4. $\square$

Theorem 9.2.2. *Let $\mathcal{R}$ be a class of STEs. If $\mathcal{R}$ is cuttable, then $\mathsf{PEnumSimPaths}(\mathcal{R})$ is in FPT delay, even when the paths need to be enumerated in oder of increasing length. If $\mathcal{R}$ is almost conflict-free, then $\mathsf{PEnumTrails}(\mathcal{R})$ is in FPT delay, even when the paths need to be enumerated in order of increasing length.*

Proof. We note that radix order implies order of increasing length. Thus it follows from Theorems 9.3.2 and 9.3.7. $\square$

9.3 With Radix Order

We now turn to radix order. We first explain how to deal with downward closed languages such as $A_1? \cdots A_k?$ or T^*. To this end, we will use the algorithm of Ackerman and Shallit [6, Theorem 1] that finds a minimal word in an NFA N in $\theta(|N|^2 n^2)$ operations, where n is the length of the shortest word in $L(N)$. As a result, we can prove that, if $L(N)$ is downward closed, it is possible to output a smallest simple path in G in radix order that matches N in polynomial time. (If $L(N)$ is not downward closed, then the smallest path that matches N is not necessarily simple.)

Proposition 9.3.1. *Let N be an NFA such that $L(N)$ is downward closed. Given a directed multigraph G and two nodes s and t, a shortest simple path from s to t in G that matches N can be found in time $O(|G||N|)$ if such a path exists. A smallest such path in radix order can be found in time $O(|G|^2|N|^2|V|^2)$ if it exists.*

Proof. Let G be a directed multigraph. Concerning a shortest path, the algorithm from Lemma 5.3.1 can easily be adjusted to return a shortest path, for instance, using breadth-first search for the reachability test. This does not influence the time bound.

Concerning a smallest path in radix order, we first observe that each shortest path from s to t in G that matches N is simple—otherwise we could obtain a shorter path by making the path simple (that is, removing edges that form a loop), and obtain a path that still matches N because $L(N)$ is downward closed.

We now proceed as in the proof of Lemma 8.2.2. That is, we relabel the edges in the product $G \times N$ and view it as an NFA. We use the method of Ackerman and Shallit [6, Theorem 1] to find a smallest word in radix order from an inital state to an accepting state, and then output the corresponding path in G. We need $O(|N||G|)$ time to construct the product and $O(|N|^2|G|^2|p|^2)$ time to compute a smallest word in radix order in $G \times N$. □

Enumeration for Simple Transitive Expressions

We show that Theorem 5.2.2(a)—the FPT part—can be extended to enumeration problems. We do not need to prove any hardness results, since hardness for enumeration problems immediately follows from the hardness of their decision version, that is, Theorem 5.2.2(b). Notice that each problem in polynomial delay is also in FPT delay.

The goal of this section is to prove the following theorem.

Theorem 9.3.2. *Let $\mathcal{R}$ be a cuttable class of STEs. Then PEnumSimPaths($\mathcal{R}$) is in FPT delay, even when the paths need to be enumerated in radix order.*

This theorem immediately implies that the enumeration versions of PSimPathLength and PSimPathLength$^{\geq}$ (from Section 5.3) are in FPT delay.

Theorem 9.3.3. *PEnumSimPathLength and PEnumSimPathLength$^{\geq}$ are in FPT delay, even when the paths need to be enumerated in order of increasing length.*

We now turn to proving Theorem 9.3.2. In fact, the proofs of the enumeration results are all along the same lines and use Lemma 8.5.1. Since each derivative of an STE with cut border c is a union of STEs with cut border at most c by Lemma 9.1.3, we can strengthen Lemma 8.5.1.

Lemma 9.1.3 implies that we can strengthen Lemma 8.5.1 in the case of STEs.

Lemma 9.3.4. *Let $\mathcal{R}$ be a class of STEs. If there exists an algorithm $\mathcal{A}$ that, when given as input a directed multigraph G, nodes s and t, and $r \in \mathcal{R}$, returns in time $f(n)$ (with $f(n) \geq n$),*

a simple path from s to t in G that matches $L(r)$ if it exists and "no" otherwise,

then EnumSimPaths($\mathcal{R}$) is in delay $O(|V||r|f(n) + |r| \cdot |V|^3)$ with preprocessing time $O(f(n))$, where $n = |G| + |r|$. Furthermore, if $\mathcal{A}$ always returns a shortest simple path (respectively, a smallest simple path in radix order), then the enumeration can be done in order of increasing length (respectively, in radix order), with the same time guarantees.

Proof. Since we search for simple paths, we only need to compute derivatives for words w of length at most $|V|$. Lemma 9.1.3 implies that we can compute a single such derivative in time $O(|V||r|)$. According to Lemma 9.1.3, each derivative of an STE with cut border c is a union of at most $|r|$ many STEs with cut border at most c. Therefore, we can use algorithm $\mathcal{A}$ to also solve problem (2) in Lemma 8.5.1, by running it for each STE in the union separately. The smallest existing path in radix order can be found by taking the smallest returned path overall, for each STE in the union. To be precise, we need $O(f(n))$ time until the first output, and afterwards delay $O(|V|(|r| \cdot f(n) + |r| \cdot |V|^2))$. Since w is a prefix of lab(p), the algorithm needs to compute $w^{-1}L$ at most $|V|$ times in each of the $|V|$ iterations in line 7. □

We will now show how the decision algorithm for Theorem 5.2.2(a) can be adapted to return a smallest path in radix order. From now on, we refer to such a path as a *minimal* path. We show that Algorithm 3, for computing a simple path matching a 0-bordered STE, can be adjusted to compute a minimal path.

Lemma 9.3.5. *Let G be a directed multigraph, s and t nodes, and $r = A_1 \cdots A_{k_1} T^* A'_{k_2} \cdots A'_1$ a 0-bordered STE. If there exists a simple path from s to t matching r, then a shortest such path can be computed in time $2^{O(|r|)} \cdot |V|^3|E|$ and a minimal such path in time $2^{O(|r| \log |r|)} \cdot |V|^6|E|^2$.*

Proof. Since Algorithm 3 already solves the decision version of the problem, we only need to show that it can be adapted to compute a shortest, respectively, minimal path in the required time. We first show that Algorithm 3 can output a shortest path. If Algorithm 3 returned "yes", there exist nodes $x, y \in V$ and sets $X \in \hat{P}^{r_1}_{s,x}$ and $X' \in \hat{P}^{r_2}_{y,v}$, and a simple path p from x to y that matches T^* and is node-disjoint from X and X' except for x and y. (See Lemma 5.3.11.) By definition of $P^{r_1}_{s,x}$, the nodes in $X \in \hat{P}^{r_1}_{s,x}$ form a path from u to x that matches $r_1 = A_1 \cdots A_{k_1}$. The construction of $\hat{P}^{r_1}_{s,x}$ in Lemma 5.3.9 allows us to order the elements in the sets such that they directly correspond to such a path. (In fact, the construction is analogous to [97, Lemma 5.2], which also shows that a witnessing path can be obtained.) So we can construct a path p_1 from u to x that uses only nodes in X and matches r_1 and a path p_2 from y to v that uses only nodes in X' and matches $r_2 = A'_{k_2} \cdots A'_1$. This also holds for a shortest such path, see Corollary 5.3.12.

To output a minimal path, we need to make some small changes to Algorithm 3. That is, we enumerate in line 2 all words $w_1 \in L(r_1)$ and compute $\hat{P}^{w_1}_{s,x} \subseteq^{k_1+k_2+1}_{\text{rep}} P^{w_1}_{s,x}$ for each such word. This way we can ensure that we really considered each word and, in particular, each prefix of a minimal simple path that matches r.[1] We proceed analogously in line 7 for all words $w_2 \in L(r_2)$.

Thus, we can use Algorithm 3 and iterate, for all words w_1 and w_2 and all nodes x, y over $\hat{P}^{w_1}_{s,x} \subseteq^{k+1}_{\text{rep}} P^{w_1}_{s,x}$ in line 3 and $\hat{P}^{w_2}_{y,t} \subseteq^{w_2}_{y,t} P^{w_2}_{y,t}$ in line 8. Then we find a minimal simple path from x to y matching T^* in line 11 in time $O(|G|^2|r|^2|V|^2)$ with Proposition 9.3.1.

Concerning the time bounds, Algorithm 3 without changes has a running time of $2^{O(|r|)} \cdot |V|^{c+3}|E|$, see Lemma 5.3.14. Iterating over the words is in $O(|r|^{|r|})$ and using Proposition 9.3.1 instead of Lemma 5.3.1 and the reachability test for T^* adds a factor $O(|G||r|^2|V|^2)$. Rewriting $O(|r|^{|r|})$ into $2^{O(|r| \log |r|)}$ yields the result. □

Finally, the following result implies Theorem 9.3.2.

Lemma 9.3.6. *Let $\mathcal{R}$ be a class of STEs with cut border at most c. Then EnumSimPaths($\mathcal{R}$) is in FPT delay with radix order, to be more precise, with $2^{O(|r| \log |r|)} \cdot |V|^{c+6}|E|^2$ preprocessing time and delay $2^{O(|r| \log |r|)} \cdot |V|^{c+7}|E|^2$. If we only need order of increasing length, the preprocessing is $2^{O(|r|)} \cdot |V|^{c+3}|E|$ and the delay is $2^{O(|r|)} \cdot |V|^{c+4}|E|$.*

[1] If we start in Lemma 5.3.11 with a minimal simple path, we can replace P with a P' such that P' and R do not intersect. If additionally P and P' match the same word, the new path must also be a smallest one in radix order.

Proof. By Lemma 9.3.4, we only need to show the existence of an algorithm $\mathcal{A}$ that finds a minimal path within the required time bound. To this end, let $r \in \mathcal{R}$ and let c_1 and c_2 be the left and right cut border of r, respectively. Hence, $r = A_1 \dots A_{c_1} r' A'_{c_2} \cdots A'_1$. (If $c_i = 0$, then the respective part of r is simply missing.) We can compute, for all $u, v \in V$, all paths p_1 from s to u matching $A_1 \cdots A_{c_1}$ and all paths p_2 from v to t matching $A'_{c_2} \cdots A'_1$ in time $O(|V|^c)$.[2] We then do a loop over all pairs (p_1, p_2) of such paths that are node-disjoint. For each such pair, we will compute a *candidate path* $P_{(p_1,p_2)}$. The overall idea of the algorithm is that it first computes all such candidate paths and then, when it has iterated through all (p_1, p_2), takes the minimal one.

For the remainder of the proof, fix such a pair (p_1, p_2) and let p_{c_1} and p_{c_2} be the smallest paths (in radix order) obtained from p_1 and p_2 by considering the edge labels in G. The subexpression r' of r is of the form $r' = B'_{\text{pre}} T^* B'_{\text{suff}}$ and is 0-bordered. So we now search for a minimal simple path matching r' from u to v. We first delete in G all nodes in $(V(p_1) \setminus \{u\}) \cup (V(p_2) \setminus \{v\})$. Then, we perform a case distinction on the form of r'.

If $r' = A_1? \cdots A_{k_1}?T^* A'_{k_2}? \cdots A'_1?$, its language $L(r')$ is downward closed, so we can find a simple path p matching r' that is a minimal path using Proposition 9.3.1 and take $P_{(p_1,p_2)} = p_{c_1} p\, p_{c_2}$.

For $r' = A_{c_1+1} \cdots A_{k_1} T^* A'_{k_2} \cdots A'_{c_2+1}$, we know from Lemma 9.3.5 that we can compute a minimal path p. We then define $P_{(p_1,p_2)} = p_{c_1} p\, p_{c_2}$.

If r' has another form, that is

$$r' = A_{c_1+1} \cdots A_{k_1} T^* A'_{k_2}? \cdots A'_1? \qquad \text{or} \qquad r' = A_1? \cdots A_{k_1}?T^* A'_{k_2} \cdots A'_{c_2+1}\,,$$

we can also obtain a minimal simple path. In the first case, we again iterate over all words $w_1 \in A_{c_1+1} \cdots A_{k_1}$, compute the minimal path in $\hat{P}^{w_1}_{u,x} \subseteq^{k'_1+1}_{\text{rep}} P^{w_1}_{u,x}$, and use Proposition 9.3.1 to find a minimal path from x to t for the downward closed part. The other case is symmetric.

In each of the cases, the algorithm then iterates through all (p_1, p_2) and, for each such pair, adds a candidate path. Finally, it outputs the smallest candidate path.

Concerning the running time, we need time $O(|V|^c)$ to guess p_1 and p_2. To output a simple path (not necessarily minimal), we need $2^{O(|r|)} \cdot |V|^{c+3}|E|$ time, see Lemma 5.3.16. This is also the time we need to output shortest simple paths, since we can use the same algorithm. For minimal paths in radix order, we use Proposition 9.3.1 with running time $O(|G|^2|r|^2|V|^2)$ instead of Lemma 5.3.1 with running time $O(|G||r|)$ and instead of the reachability test for the T^* part. Furthermore, depending on r, we might need to enumerate all words $w_1 \in L(A_{c_1} \cdots A_{k_1})$ and $w_2 \in L(A'_{k_2} \cdots A'_1)$, and compute the rest of the algorithm depending on these words. Thus we need $2^{O(|r| \log |r|)} \cdot |V|^{c+6}|E|^2$ time overall in this case. The delay then follows from Lemma 9.3.4. □

We now turn to trails. Since the construction from Lemma 5.5.3 implies a bijection between paths, we can proceed similar to the decision version and reduce the problem of

[2]For the purpose of the proof, it suffices to compute the paths without the edge labels here. We can find the labels on the edges in p_1 and p_2 that are smallest words in the corresponding expressions in radix order later.

enumerating trails to enumerating simple paths in a newly constructed graph. Thus the FPT result from Theorem 5.2.4 also carries over to the enumeration setting.

Theorem 9.3.7. *Let $\mathcal{R}$ be a class of STE that is almost conflict-free. Then, $\mathsf{PEnumTrails}(\mathcal{R})$ is in FPT delay, even when the paths need to be enumerated in radix order.*

More precisely, we obtain the following delays:

Lemma 9.3.8. *Let $\mathcal{R}$ be a class of STEs with at most c conflict positions. Then, $\mathsf{PEnumTrails}(\mathcal{R})$ is in FPT delay with radix order, to be more precise, in $2^{O(|r|\log|r|)} \cdot |E|^{c+11}$ preprocessing time and delay $2^{O(|r|\log|r|)} \cdot |E|^{c+12}$. If we only need order of increasing length, the preprocessing is $2^{O(|r|)} \cdot |E|^{c+6}$ and the delay is $2^{O(|r|)} \cdot |E|^{c+7}$.*

Proof. By Corollary 5.5.4, we have a bijection between the trails matching a word w in the directed multigraph G and the simple paths matching $\sigma \cdot w$ in the directed graph H, where H is obtained from G as in Lemma 5.5.3. Here, σ is an arbitrary label from Σ. Thus, we can use Lemma 9.3.4 on $H = (V_H, E_H, \mathcal{E}_H)$ to enumerate the simple paths in the respective order and output the corresponding trails in G. We note that, due to Lemma 9.1.3, derivatives of STEs with at most c conflict positions again have at most c conflict positions. The computation time and size bounds can be found in Lemma 9.1.3.

So we need an algorithm that computes simple paths on H, matching $\sigma \cdot r$ and derivatives thereof in the respective order. Notice that the existence of such an algorithm is not immediate from our results on simple paths, since $\mathcal{R}$ is not necessarily cuttable. In fact, we need to relabel H and r as in (1)–(3) from the proof of Lemma 5.6.3. In (1), we relabeled r and some edges of H. Concerning the ordering of labels, we assume that, if $a < b$, then $a < \tilde{a} < b < \tilde{b}$. Notice that every A_i, T, or A_i' has only a or $\tilde{a}$ but not both, so this ordering does not affect the minimality of the path that we find. For every minimal path p matching $\sigma \cdot r$ in H there is a set S such that a minimal path in H_S matching $\sigma \cdot \tilde{r}$ will use the same nodes in the same order as p. We can compute, for each set S, a minimal path p_S in H_S, compare all such paths p_S, and take the minimal one. In (2), we only get rid of σ, so this will not change the minimality of a path. Finally, in (3), we use the same methods as in Lemma 5.3.15, which can be used to output simple paths in the respective order, see Lemma 9.3.6. Using the bijection between these simple paths and the trails in G, we can enumerate the trails.

So we can indeed output trails in the radix order or in order of increasing length. We now turn to the running time. Combining the blow-ups from the construction in Lemma 5.5.3 and the multiple graphs H_S we obtain from each different choice of S, we can find a shortest simple path that matches $\sigma \cdot \tilde{r}$ in time $2^{O(|r|)} \cdot |E|^{c+6}$ and a minimal simple path in time $2^{O(|r|\log|r|)} \cdot |E|^{c+11}$. Together with Lemma 9.3.4 this enables us to enumerate the simple paths and output the corresponding trails with delay $2^{O(|r|)} \cdot |E|^{c+7}$ for order of increasing length and delay $2^{O(|r|)} \cdot |E|^{c+12}$ for radix order. □

Conclusions and Future Work

Summary and Directions for Future Research

Throughout this thesis we studied several problems regarding regular simple path and trail queries. For the broad community, the take-away message is the following. The majority of RPQs that users ask in practice[3] indeed belong to the tractable classes for simple path and trail semantics, both for directed and undirected multigraphs, which is good news. Furthermore, these paths can be enumerated in polynomial delay, that is, we can also output all the results with reasonable delay between two answers.

Summary

We will now summarize our results in more detail and discuss open problems and directions for future research. In Chapter 3 we have defined the class $\mathsf{T}_{\mathsf{tract}}$ of regular languages L for which finding trails in directed multigraphs that are labeled with L is tractable iff NL $\neq$ NP. We have investigated $\mathsf{T}_{\mathsf{tract}}$ in depth in terms of closure properties, characterizations, and the recognition problem, also touching upon the closely related class $\mathsf{SP}_{\mathsf{tract}}$ (for which finding *simple paths* is tractable) when relevant.

In Chapter 4 we took a closer look at queries that are asked in practice, taking a look at the study of query logs from Bonifati et al. [53, 55]. We define the class of *simple transitive expressions (STEs)*, which are a class of regular expressions powerful enough to capture over 99.99% of the RPQs occurring in a recent practical study [53]. For this class, we find in Chapter 5 two dichotomies on the parameterized complexity of evaluating STEs under *simple path* and *trail* semantics, respectively.

Under simple path semantics, the central property that we require for a class of expressions so that evaluation is in FPT is *cuttability*, that is, having bounded *cut borders* (also see Figure 5.2). Looking at Table 4.1, we see that the cut borders for expressions in practice are indeed very small: it is one for a^*b, two for abc^*, and zero in all other cases. Under trail semantics, the central property for evaluation in FPT is *almost conflict freeness*, that is, a constant number of conflict positions. Looking again at the underlying data for Table 4.1, we discovered that all STEs had zero conflict positions. (We needed to look deeper again, because some classes in Table 4.1 aggregate others. For instance, "a^*b" also contains expressions of the form aa^*.)

Therefore, although evaluation under simple path and trail semantics of RPQs is known to be hard in general, it seems that the RPQs that users actually ask are much less

[3]We refer to [53, 55] for detailed overviews of RPQs used in query logs.

complex.[4] In fact, since the vast majority (over 99%) of expressions in Table 4.1 has cut borders of at most two and no conflict positions, our FPT results in Theorems 5.2.2 and 5.2.4 imply that evaluation for this majority of expressions is in FPT with small parameter. We show in Chapter 7 that these favourable results carry over to undirected multigraphs. Furthermore, we were able to show that some classes of STEs that are not tractable on directed multigraphs, are tractable on undirected multigraphs under trail semantics.

In Part II we studied the data complexity of trail and simple path evaluation of RPQs on undirected multigraphs. Although this sounds like a single problem, it is actually a very general class of problems, which subsumes several well-studied problems, such as disjoint-path problems and trail or simple path problems with length constraints.

Using a wide range of methods, such as the minor theorem from Robertson and Seymour [182], the minor theorem on group-labeled graphs [119], the extended line graph [130], Edmond's matching technique and extensions thereof [5, 194], and several structural arguments, we were able to pinpoint several interesting tractable cases of the problem.

One interesting finding is that all languages in $\mathsf{SP}_{\mathsf{tract}}$ are again tractable on undirected multigraphs, making this class quite robust.

On the intractable side, we provided a gadget $G_{3\text{SAT}}$ that can be used to show NP-hardness of a wide range of trail and simple path problems (for example, $\mathsf{UTrail}((abc)^*)$) and used directed two-disjoint paths techniques for languages such as $(abab)^*$.

In Part III we first extended Yen's algorithm to various variants. While the original algorithm enumerates simple paths from s to t and requires a shortest-path subroutine, our new variants can also restrict the paths returned to those that match a certain regular expression and where trails instead of simple paths are returned. We summarize these results in Section 8.5. In Chapter 9 we then used the framework together with the results from Parts I and II. More precisely, we were able to give enumeration algorithms that are in polynomial or FPT delay whenever the corresponding decision problem is in P or FPT, respectively.

Discussion and Further Work

The principled research conducted in this thesis gives us important insights into the further development of query languages. In our view, graph database manufacturers can have the following tradeoffs in mind concerning simple path ($\mathsf{SP}_{\mathsf{tract}}$) and trail semantics ($\mathsf{T}_{\mathsf{tract}}$) in database systems:

- $\mathsf{SP}_{\mathsf{tract}} \subsetneq \mathsf{T}_{\mathsf{tract}}$, that is, there are strictly more languages for which finding regular paths under trail semantics is tractable than under simple path semantics. Some of the languages in $\mathsf{T}_{\mathsf{tract}}$ but outside $\mathsf{SP}_{\mathsf{tract}}$ are of the form $(ab)^*$ or a^*bc^*, which were found to be relevant in several application scenarios involving network problems,

[4]A recent study confirmed this hypothesis on a corpus of 208M queries from Wikidata logs [55]. Here, about 39% of the unique queries used property paths.

genomic datasets, and tracking provenance information of food products [175] and appear in query logs [53, 55].

- $\mathsf{SP_{tract}} \subseteq \mathsf{USP_{tract}}$ and $\mathsf{SP_{tract}} \subseteq \mathsf{UT_{tract}}$, that is, the languages in $\mathsf{SP_{tract}}$ are still tractable under simple path and trail semantics on undirected or bidirectional graphs and multigraphs. On the other hand, $\mathsf{T_{tract}} \not\subseteq \mathsf{UT_{tract}}$, thus $\mathsf{SP_{tract}}$ seems more “robust” than $\mathsf{T_{tract}}$.

- Both $\mathsf{SP_{tract}}$ and $\mathsf{T_{tract}}$ can be syntactically characterized but, currently, the characterization for $\mathsf{SP_{tract}}$ (Section 7 in [20]) is simpler than the one for $\mathsf{T_{tract}}$. This is due to the fact that connected components for automata for languages in $\mathsf{T_{tract}}$ can be much more complex than for automata for languages in $\mathsf{SP_{tract}}$.

- On the other hand, the *single-occurrence* condition, that is, each alphabet symbol occurs at most once, is a sufficient condition for regular expressions to be in $\mathsf{T_{tract}}$. This condition is trivial to check and also captures languages outside $\mathsf{SP_{tract}}$ such as $(ab)^*$ and a^*bc^*. Moreover, the condition seems to be useful: we analyzed the 50 million RPQs found in the logs of [54] and discovered that over 99.8% of the RPQs are single-occurrence. But again, there are single-occurrence expressions like $(abc)^*$ which are NP-complete to evaluate under trail semantics on undirected graphs.

- In terms of closure properties, learnability, or complexity of testing if a given regular language belongs to $\mathsf{SP_{tract}}$ or $\mathsf{T_{tract}}$, the classes seem to behave the same.

- The tractability for the decision version of RPQ evaluation can be lifted to the enumeration problem, in which case the task is to output matching paths with only a polynomial delay between answers.

Thus our results seem to imply that choosing $\mathsf{SP_{tract}}$ instead of $\mathsf{T_{tract}}$ is a “safer” choice when taking two-way RPQs (and therefore undirected graphs) into account, regardless of the choice of semantics. On the other hand, $\mathsf{T_{tract}} \cap \mathsf{UT_{tract}}$ contains languages like $(ab)^*$ which are not part of $\mathsf{SP_{tract}}$. Thus, when focusing only on trail semantics, the class of tractable languages is slightly larger than $\mathsf{SP_{tract}}$, but we have not been able to characterize the class $\mathsf{T_{tract}} \cap \mathsf{UT_{tract}}$ yet. Indeed, complete dichotomy results for regular path queries on undirected graphs and on two-way RPQs under simple path and trail semantics remain as open problems. The most prominent problem to be solved here is probably the question if it is decidable in polynomial time whether an undirected graph has a simple path of length 0 modulo 3 between two given nodes. We believe that resolving this question will be the key to also resolving the question for larger modulo values.

We note that while we considered multigraphs in this thesis, all NP-hardness proofs required only simple graphs. While the classes $\mathsf{SP_{tract}}$, $\mathsf{T_{tract}}$, and $\mathsf{USimPath}$ are the same if we restrict the multigraphs to simple graphs, we do not know if the same can be said about UTrail. Thus the the question arises if the class $\mathsf{UT_{tract}}$ is “the same” when we do not consider multigraphs, but only simple graphs.

Another line of future research is the static analysis of regular path queries under simple path and trail semantics. Before evaluating a query, database systems often optimize their running time by rewriting the query into a smaller one. To this end, the complexity of the *minimization*, *equivalence*, and *containment* problems of the queries play a role. While these problems are EXPSPACE-complete for conjunctive regular path queries [60, 96], to the best of our knowledge, no results are known for the complexity of these problems when taking conjunctions of regular simple path and trail queries. It would also be interesting to study the complexity of these problems when only fragments of regular expressions are allowed [85, 94].

Appendix A

On the Complexity of Properly Edge Colored Disjoint Paths

The work on Gourvès et al. [105] on edge-colored graphs contains some flawed proofs, which we discuss here. The flawed proofs are Theorem 9 and Corollary 10. We first repeat their statements in the notation of this thesis, sketch how their proof is flawed, and give an idea how to fix it.

Theorem A.0.1 (Theorem 9 in [105]). *Let G^c be a c-edge-colored graph with no (almost) PEC closed trails through s or t, and a constant $L > 0$. Then, the problem of finding 2 vertex/edge disjoint PEC s-t paths, each having at most L edges in G^c is NP-complete in the strong sense, even for graphs with maximum vertex degree equal to 3.*

We now restate their theorem in the notion of our paper and neglect some restrictions posed to the graph. Please note that their length L depends on the 3SAT instance, thus it is no "constant" in the sense of this thesis.

Theorem A.0.2 (restated version of Theorem 9 in [105]). *Let G be an edge-labeled, undirected graph, s, t two nodes, and $L \in \mathbb{N}$. Then, the problem of finding 2 node-/edge-disjoint s-t-paths, labeled $(ab)^*$ and both having at most length L, is NP-complete.*

In their proof, Gourvès et al. consider clause and variable gadgets, similar to the ones presented in Figure 6.3. (They use a slightly more elaborate clause gadget to achieve node degree at most 3, but we will not go into details here.) Due to the length constraints, they can force a path through the variable gadgets. Yet, they do not see that the path through the clause gadgets still has the opportunity to skip gadgets, using the unused side of a variable gadget. We sketch this in Figure A.1. We note that the length constraints given in the original paper allow this skip, as the resulting path will only get shorter.

We note that Theorem A.0.2 is still correct: From Aboueliam et al. [5, Theorem 3.2] it follows that two disjoint $(ab)^*/(ab)^*$-paths, one from s_1 to t_1 and one from s_2 to t_2 is NP-complete. With length constraints, we can add new nodes s and t and $(ab)^*$-paths of length $L' = 2|E|$ from s to s_1 and from t_2 to t, and (ab)-paths of length 2 from s to s_2 and from t_1 to t. Then every path from s to t of length at most $3|E| + 2$ is either from s_1 to t_1 or from s_2 to t_2.

For completeness, we note that Theorem A.0.1 is also correct, which means that the problem is still NP-hard when restricted to graphs without an cycle labeled $(ab)^*$ through

s or t (and without an "almost $(ab)^*$ cycle" through s or t, which means that every cycle through s or t contains the substring aa or bb at least twice) and with node-degree at most 3. We can prove it with slight changes to $G_{3\mathrm{SAT}}$. More precisely, we need to change the clause gadgets similar to [105] to ensure that every node has degree at most 3. Furthermore, we add new nodes s and t and use the path length to ensure that every path labeled $(ab)^*$ from s to t which uses the path from s to s_1 must not use the w_{r}-labeled paths. This then implies that the path from s_1 to t must be via t_1 (not t_2).

We use $G_{3\mathrm{SAT}}$ with the clause gadget depicted in Figure A.2 and the words $w_{\mathsf{b}} = a$, $w_{\mathsf{m}} = w_{\mathsf{o}} = b$, $w_{\mathsf{r}} = a(ba)^i$. We explain i later. We then add new nodes s and t with an edge labeled $(ab)^j$ from s to s_1 and edges labeled ab from s to s_2, and labeled b from t_1 to t and t_2 to t. We choose i and j such that $i + j > L$, while the intended path from s_2 to t_2 does not exceed length L. Let m be the number of clauses of φ and n be the number of variables. The length of an "intended" path from s_1 to t_1 is $3m(2|w_{\mathsf{b}}| + 2|w_{\mathsf{o}}|) - |w_{\mathsf{o}}| = 12m - 1$, therefore the length of an indented path from s to t via s_1 is $12m + j$. The length of an "intended" path from s_2 to t_2 is $m(4|w_{\mathsf{r}}| + 4|w_{\mathsf{o}}|) + n(3|w_{\mathsf{r}} + 3|w_{\mathsf{o}}|) - |w_{\mathsf{o}}| = m(8i + 8) + n(6i + 6) - 1$. We set $i = 12m + 1$, $L = 96m^2 + 16m + 72n + 12n + 2$ and $j = L - 12m$ to complete the construction.

The length constraints imply that the path from s to t that uses the path from s to s_1 must not use w_{r}-paths. By construction, this path must then enter t via the path from t_1. Thus we have a path from s_1 to t_1 which must not use any of the w_{r}-paths. The correctness then follows similar to the proof of Theorem 6.4.2.

Corollary A.0.3 (Corollary 10 in [105]). *Let s and t be two vertices in a c-edge-colored graph G^c with maximum vertex degree equal to 3 and with no (almost) PEC closed trails through s or t. Then, it is NP-complete to decide whether there exist 2 vertex/edge disjoint s-t paths such that exactly one of them is a PEC s-t path.*

Corollary A.0.4 (restated version of Corollary 10 in [105]). *Let a graph G and nodes s, t be given. Then it is NP-complete to decide whether there exist 2 node-/edge-disjoint s-t-paths such that one of them matches $(ab)^*$.*

In their proof, Gourvès et al. use the same construction as in their Theorem 10, but color the clause gadgets blue, that is, all edges in the clause gadget get the label a. Since one path has to be PEC, that is, match $(ab)^*$, this again forces one of the paths to go through each variable gadget. But, similar to before, the path using the clause gadgets can shortcut via the unused side of variable gadgets.

We note that Corollary A.0.4 and Corollary A.0.3 are correct. Indeed, we can use the variant of $G_{3\mathrm{SAT}}$ which we used to prove Theorem A.0.1 and choose $w_{\mathsf{r}} = aa$ and relabel the path from s to s_2 with aa. Then the path from s to t that matches $(ab)^*$ must use the $(ab)^*$-labeled path from s to s_1 and must not use w_{r}-paths. This implies that the path must enter t via t_1. Since the path from s_1 to t_1 must not use w_{r}-paths, and the other path from s to t must be node-/or edge-disjoint and therefore use the disjoint path from s_2 to t_2, the correctness follows from Theorem 6.4.2.

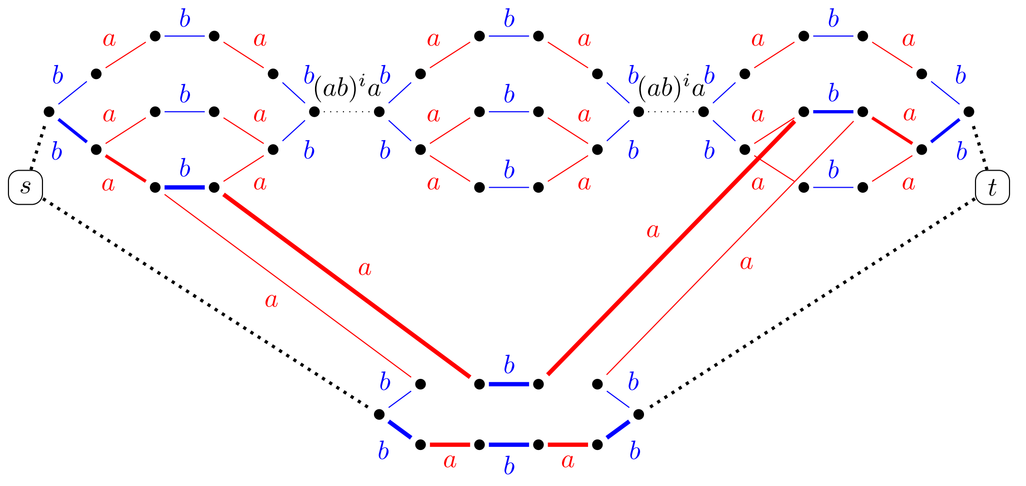


Figure A.1: Example snippet of the reduction from 3SAT to the problem of finding two properly edge-colored, edge-disjoint paths with length constraints in a two-colored graph [105, Theorem 9]. For readability, we additionally labeled blue edges with b and red edges with a. We note that, although the "variable path" (thick edges on bottom) only uses variable gadgets, the "clause path" does not need to use all clause gadgets (see thick path on top).

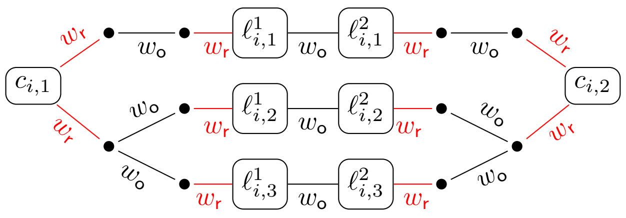


Figure A.2: Alternative clause gadget for G_{3SAT} if we want to ensure that every node has degree at most 3.

Bibliography

[1] Parosh Aziz Abdulla, Aurore Collomb-Annichini, Ahmed Bouajjani, and Bengt Jonsson. Using forward reachability analysis for verification of lossy channel systems. *Formal Methods in System Design*, 25(1):39–65, 2004.

[2] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[3] Serge Abiteboul, Luc Segoufin, and Victor Vianu. Static analysis of active XML systems. *ACM Transactions on Database Systems*, 34(4):23:1–23:44, 2009.

[4] Abdelfattah Abouelaoualim, Kinkar Chandra Das, Wenceslas Fernandez de la Vega, Marek Karpinski, Yannis Manoussakis, Carlos A. J. Martinhon, and Rachid Saad. Cycles and paths in edge-colored graphs with given degrees. *Journal of Graph Theory*, 64(1):63–86, 2010.

[5] Abdelfattah Abouelaoualim, Kinkar Chandra Das, Luérbio Faria, Yannis Manoussakis, Carlos A. J. Martinhon, and Rachid Saad. Paths and trails in edge-colored graphs. *Theoretical Computer Science (TCS)*, 409(3):497–510, 2008.

[6] Margareta Ackerman and Jeffrey Shallit. Efficient enumeration of words in regular languages. *Theoretical Computer Science (TCS)*, 410(37):3461–3470, 2009.

[7] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.

[8] Denise Amar and Yannis Manoussakis. Cycles and paths of many lengths in bipartite digraphs. *Journal of Combinatorial Theory, Series B*, 50(2):254–264, 1990.

[9] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *Symposium on Principles of Database Systems (PODS)*, pages 89–103. ACM, 2019.

[10] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1421–1432, 2018.

[11] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

[12] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. Pg-keys: Keys for property graphs. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 2423–2436. ACM, 2021.

[13] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

[14] Renzo Angles, Juan L. Reutter, and Hannes Voigt. Graph query languages. In *Encyclopedia of Big Data Technologies*. Springer, 2019.

[15] Marcelo Arenas, Pablo Barceló, Leonid Libkin, and Filip Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

[16] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *International Conference on World Wide Web (WWW)*, pages 629–638, 2012.

[17] Esther M. Arkin, Christos H. Papadimitriou, and Mihalis Yannakakis. Modularity of cycles and paths in graphs. *Journal of the ACM*, 38(2):255–274, 1991.

[18] Guillaume Bagan and Angela Bonifati. Personal communication, 2019.

[19] Guillaume Bagan, Angela Bonifati, and Benoît Groz. A trichotomy for regular simple path queries on graphs. *CoRR*, abs/1212.6857, 2012.

[20] Guillaume Bagan, Angela Bonifati, and Benoît Groz. A trichotomy for regular simple path queries on graphs. *Journal of Computer and System Sciences*, 108:29–48, 2020.

[21] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems*, 29(4):710–751, 2004.

[22] Jørgen Bang-Jensen, Thomas Bellitto, and Anders Yeo. On supereulerian 2-edge-coloured graphs. *Graphs and Combinatorics*, 37(6):2601–2620, 2021.

[23] Jørgen Bang-Jensen and Gregory Z. Gutin. Alternating cycles and paths in edge-coloured multigraphs: A survey. *Discrete Mathematics*, 165-166:39–60, 1997.

[24] Jørgen Bang-Jensen and Gregory Z. Gutin. Alternating cycles and trails in 2-edge-coloured complete multigraphs. *Discrete Mathematics*, 188(1-3):61–72, 1998.

[25] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs - Theory, Algorithms and Applications, Second Edition.* Springer Monographs in Mathematics. Springer, 2009.

[26] Pablo Barceló. Querying graph databases. In *Symposium on Principles of Database Systems (PODS)*, pages 175–188, 2013.

[27] Pablo Barceló, Diego Figueira, and Miguel Romero. Boundedness of conjunctive regular path queries. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPIcs*, pages 104:1–104:15, 2019.

[28] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems*, 37(4):31:1–31:46, 2012.

[29] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. Querying graph patterns. In *Symposium on Principles of Database Systems (PODS)*, pages 199–210. ACM, 2011.

[30] Pablo Barceló, Leonid Libkin, and Juan L. Reutter. Querying regular graph patterns. *Journal of the ACM*, 61(1):8:1–8:54, 2014.

[31] Pablo Barceló and Pablo Muñoz. Graph logics with rational relations: The role of word combinatorics. *ACM Transactions on Computational Logic*, 18(2):10:1–10:41, 2017.

[32] Pablo Barceló, Jorge Pérez, and Juan L. Reutter. Relative expressiveness of nested regular expressions. In *Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, volume 866 of *CEUR Workshop Proceedings*, pages 180–195. CEUR-WS.org, 2012.

[33] Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. Semantic acyclicity on graph databases. In *Symposium on Principles of Database Systems (PODS)*, pages 237–248. ACM, 2013.

[34] Christopher L. Barrett, Riko Jacob, and Madhav V. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[35] Thomas Bellitto and Benjamin Bergougnoux. On minimum connecting transition sets in graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 11159 of *Lecture Notes in Computer Science*, pages 40–51. Springer, 2018.

[36] Kristóf Bérczi and Yusuke Kobayashi. The directed disjoint shortest paths problem. In *European Symposium on Algorithms (ESA)*, volume 87 of *LIPIcs*, pages 13:1–13:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[37] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Symposium on Principles of Database Systems (PODS)*, pages 303–318. ACM, 2017.

[38] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *ACM Transactions on Database Systems*, 43(2):7:1–7:32, 2018.

[39] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *International Conference on World Wide Web (WWW)*, pages 712–721, 2005.

[40] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. Dtds versus XML schema: A practical study. In *Proceedings of the Seventh International Workshop on the Web and Databases (WebDB)*, pages 79–84, 2004.

[41] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. Inference of concise regular expressions and dtds. *ACM Transactions on Database Systems*, 35(2):11:1–11:47, 2010.

[42] Geert Jan Bex, Frank Neven, and Stijn Vansummeren. Inferring XML schema definitions from XML data (VLDB). In *International Conference on Very Large Data Bases (VLDB)*, pages 998–1009, 2007.

[43] Ivona Bezáková, Radu Curticapean, Holger Dell, and Fedor V. Fomin. Finding detours is fixed-parameter tractable. *SIAM Journal on Discrete Mathematics*, 33(4):2326–2345, 2019.

[44] Ramesh Bhandari. Optimal physical diversity algorithms and survivable networks. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 433–441. IEEE Computer Society, 1997.

[45] Meghyn Bienvenu, Magdalena Ortiz, and Mantas Simkus. Regular path queries in lightweight description logics: Complexity and algorithms. *Journal of Artificial Intelligence Research*, 53:315–374, 2015.

[46] Meghyn Bienvenu and Michaël Thomazo. On the complexity of evaluating regular path queries over linear existential rules. In *International Conference on Web Reasoning and Rule Systems (RR)*, volume 9898 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2016.

[47] Andreas Björklund and Thore Husfeldt. Shortest two disjoint paths in polynomial time. *SIAM Journal on Computing*, 48(6):1698–1710, 2019.

[48] Andreas Björklund, Thore Husfeldt, and Sanjeev Khanna. Approximating longest directed paths and cycles. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 222–233, 2004.

[49] Henrik Björklund, Wouter Gelade, and Wim Martens. Incremental xpath evaluation. *ACM Transactions on Database Systems*, 35(4):29:1–29:43, 2010.

[50] Henrik Björklund, Wim Martens, and Thomas Schwentick. Validity of tree pattern queries with respect to schema information. In *Mathematical Foundations of Computer Science (MFCS)*, volume 8087 of *Lecture Notes in Computer Science*, pages 171–182. Springer, 2013.

[51] Henrik Björklund, Wim Martens, and Thomas Timm. Efficient incremental evaluation of succinct regular expressions. In *International Conference on Information and Knowledge Management (CIKM)*, pages 1541–1550. ACM, 2015.

[52] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Slawek Staworko, and Dominik Tomaszuk. Threshold queries in theory and in the wild. *CoRR*, abs/2106.15703, 2021.

[53] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *Proceedings of the VLDB Endowment (PVLDB)*, 11(2):149–161, 2017.

[54] Angela Bonifati, Wim Martens, and Thomas Timm. DARQL: deep analysis of SPARQL queries. In *Companion Proceedings of the The Web Conference (WWW) (Companion Volume)*, pages 187–190. ACM, 2018.

[55] Angela Bonifati, Wim Martens, and Thomas Timm. Navigating the maze of wikidata query logs. In *The Web Conference (WWW)*, pages 127–138. ACM, 2019.

[56] Sarra Bouhenni, Saïd Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. A survey on distributed graph pattern matching in massive graphs. *ACM Computing Surveys*, 54(2):36:1–36:35, 2021.

[57] Joel Brynielsson, Johanna Högberg, Lisa Kaati, Christian Mårtenson, and Pontus Svenson. Detecting social positions using simulation. In *International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 48–55. IEEE Computer Society, 2010.

[58] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[59] Leizhen Cai and Junjie Ye. Two edge-disjoint paths with length constraints. *Theoretical Computer Science (TCS)*, 795:275–284, 2019.

[60] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 176–185. Morgan Kaufmann, 2000.

[61] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. View-based query processing for regular path queries with inverse. In *Symposium on Principles of Database Systems (PODS)*, pages 58–66. ACM, 2000.

[62] 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(3):443–465, 2002.

[63] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.

[64] Katrin Casel and Markus L. Schmid. Fine-grained complexity of regular path queries. In *International Conference on Database Theory (ICDT)*, volume 186 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[65] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *ACM Symposium on Theory of Computing (STOC)*, pages 77–90. ACM, 1977.

[66] Yijia Chen and Jörg Flum. On parameterized path and chordless path problems. In *IEEE Conference on Computational Complexity (CCC)*, pages 250–263, 2007.

[67] W. S. Chou, Y. Manoussakis, O. Megalakaki, M. Spyratos, and Zs. Tuza. Paths through fixed vertices in edge-colored graphs. *Mathématiques et Sciences humaines*, 127:49–58, 1994.

[68] Rina S. Cohen and Janusz A. Brzozowski. Dot-depth of star-free events. *Journal of Computer and System Sciences*, 5(1):1–16, 1971.

[69] Mariano P. Consens and Alberto O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Symposium on Principles of Database Systems (PODS)*, pages 404–416, 1990.

[70] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 18(3):265–298, 2004.

[71] Alejandro Contreras-Balbuena, Hortensia Galeana-Sánchez, and Ilan A. Goldfeder. A new sufficient condition for the existence of alternating hamiltonian cycles in 2-edge-colored multigraphs. *Discrete Applied Mathematics*, 229:55–63, 2017.

[72] Alejandro Contreras-Balbuena, Hortensia Galeana-Sánchez, and Ilan A. Goldfeder. Alternating hamiltonian cycles in 2-edge-colored multigraphs. *Discrete Mathematics & Theoretical Computer Science*, 21(1), 2019.

[73] Stephen A. Cook. The complexity of theorem-proving procedures. In *ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM, 1971.

[74] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 323–330, 1987.

[75] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer, 2015.

[76] Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. `https://www.w3.org/TR/rdf11-concepts/`, 2014. World Wide Web Consortium.

[77] Wojciech Czerwinski, Wim Martens, Matthias Niewerth, and Pawel Parys. Minimization of tree patterns. *Journal of the ACM*, 65(4):26:1–26:46, 2018.

[78] Wojciech Czerwinski, Wim Martens, Pawel Parys, and Marcin Przybylko. The (almost) complete guide to tree pattern containment. In *Symposium on Principles of Database Systems (PODS)*, pages 117–130. ACM, 2015.

[79] Andreas Darmann and Janosch Döcker. On simplified NP-complete variants of monotone 3-Sat. *Discrete Applied Mathematics*, 292:45–58, 03 2021.

[80] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in dynfo. *Journal of the ACM*, 65(5):33:1–33:24, 2018.

[81] Dbpedia. `wiki.dbpedia.org`.

[82] Holger Dell. Personal communication, 2017.

[83] Xiaotie Deng and Christos H. Papadimitriou. On path lengths modulo three. *Journal of Graph Theory*, 15(3):267–282, 1991.

[84] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Hannes Voigt, Oskar van Rest, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. Graph pattern matching in GQL and SQL/PGQ, 2021.

[85] Alin Deutsch and Val Tannen. Optimization properties for classes of conjunctive regular path queries. In *International Workshop on Database Programming Languages DBPL*, volume 2397 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2001.

[86] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995.

[87] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theoretical Computer Science (TCS)*, 141(1):109–131, 1995.

[88] Zdenek Dvorák. Two-factors in orientated graphs with forbidden transitions. *Discrete Mathematics*, 309(1):104–112, 2009.

[89] Tali Eilam-Tzoreff. The disjoint shortest paths problem. *Discrete Applied Mathematics*, 85(2):113–138, 1998.

[90] Shimon Even, Alon Itai, and Adi Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, 1976.

[91] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Yinghui Wu. Adding regular expressions to graph reachability and pattern queries. *Frontiers of Computer Science*, 6(3):313–338, 2012.

[92] Trevor I. Fenner, Oded Lachish, and Alexandru Popa. Min-sum 2-paths problems. *Theory of Computing Systems*, 58(1):94–110, 2016.

[93] Diego Figueira. Containment of UC2RPQ: the hard and easy cases. In *International Conference on Database Theory (ICDT)*, volume 155 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[94] Diego Figueira, Adwait Godbole, Shankara Narayanan Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. Containment of simple conjunctive regular path queries. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 371–380, 2020.

[95] Sergio Flesca, Filippo Furfaro, and Elio Masciari. On the minimization of XPath queries. *Journal of the ACM*, 55(1):2:1–2:46, 2008.

[96] Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Symposium on Principles of Database Systems (PODS)*, pages 139–148. ACM Press, 1998.

[97] Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, and Saket Saurabh. Efficient computation of representative families with applications in parameterized and exact algorithms. *Journal of the ACM*, 63(4):29:1–29:60, 2016.

[98] Steven Fortune, John Hopcroft, and James Wyllie. The directed subgraph homeomorphism problem. *Theoretical Computer Science (TCS)*, 10(2):111–121, 1980.

[99] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1433–1445. ACM, 2018.

[100] Nadime Francis, Luc Segoufin, and Cristina Sirangelo. Datalog rewritings of regular path queries using views. In *International Conference on Database Theory (ICDT)*, pages 107–118. OpenProceedings.org, 2014.

[101] Dominik D. Freydenberger and Nicole Schweikardt. Expressiveness and static analysis of extended conjunctive regular path queries. *Journal of Computer and System Sciences*, 79(6):892–909, 2013.

[102] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*, volume FS-06-02 of *AAAI Technical Report*, pages 45–53. AAAI Press, 2006.

[103] Wouter Gelade, Marc Gyssens, and Wim Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM Journal on Computing*, 41(1):160–190, 2012.

[104] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Optimizing and parallelizing ranked enumeration. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11):1028–1039, 2011.

[105] Laurent Gourvès, Adria Ramos de Lyra, Carlos A. J. Martinhon, and Jérôme Monnot. On paths, trails and closed trails in edge-colored graphs. *Discrete Mathematics & Theoretical Computer Science*, 14(2):57–74, 2012.

[106] Laurent Gourvès, Adria Lyra, Carlos A. J. Martinhon, and Jérôme Monnot. Complexity of trails, paths and circuits in arc-colored digraphs. *Discrete Applied Mathematics*, 161(6):819–828, 2013.

[107] Laurent Gourvès, Adria Lyra, Carlos A. J. Martinhon, Jérôme Monnot, and Fábio Protti. On s-t paths and trails in edge-colored graphs. *Electronic Notes in Discrete Mathematics*, 35:221–226, 2009.

[108] GQL standard website. `https://www.gqlstandards.org/`. 2021.

[109] GQL influence graph. `https://www.gqlstandards.org/existing-languages`. 2021.

[110] Martin Grohe and Magdalena Grüber. Parameterized approximability of the disjoint cycle problem. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 363–374, 2007.

[111] Martin Grohe, Ken-ichi Kawarabayashi, Dániel Marx, and Paul Wollan. Finding topological subgraphs is fixed-parameter tractable. In *ACM Symposium on Theory of Computing (STOC)*, pages 479–488. ACM, 2011.

[112] Claudio Gutiérrez, Jan Hidders, and Peter T. Wood. Graph data models. In *Encyclopedia of Big Data Technologies*. Springer, 2019.

[113] Gregory Z. Gutin, Mark Jones, Bin Sheng, Magnus Wahlström, and Anders Yeo. Chinese postman problem on edge-colored multigraphs. *Discrete Applied Mathematics*, 217:196–202, 2017.

[114] Gregory Z. Gutin and Eun Jung Kim. Properly coloured cycles and paths: Results and open problems. In *Graph Theory, Computational Intelligence and Thought*, volume 5420 of *Lecture Notes in Computer Science*, pages 200–208. Springer, 2009.

[115] Leonard H. Haines. On free monoids partially ordered by embedding. *Journal of Combinatorial Theory*, 6(1):94–98, 1969.

[116] Steve Harris and Andy Seaborne. SPARQL 1.1 query language. `https://www.w3.org/TR/sparql11-query/`, 2013. World Wide Web Consortium.

[117] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 453–462. IEEE Computer Society, 1995.

[118] John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Communications of the ACM*, 16(6):372–378, 1973.

[119] Tony Huynh. The linkage problem for group-labelled graphs. *IEEE Expert / IEEE Intelligent Systems - EXPERT*, 01 2009.

[120] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1259–1274. ACM, 2017.

[121] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. General dynamic yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal*, 29(2-3):619–653, 2020.

[122] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[123] Aubin Jarry and Stéphane Pérennes. Disjoint paths in symmetric digraphs. *Discrete Applied Mathematics*, 157(1):90–97, 2009.

[124] Pierre Jullien. *Contribution à l'étude des types d'ordres dispersés.* PhD thesis, Universite de Marseille, 1969.

[125] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. Flexible caching in trie joins. In *International Conference on Extending Database Technology (EDBT)*, pages 282–293, 2017.

[126] Mamadou Moustapha Kanté, Christian Laforest, and Benjamin Momège. Trees in graphs with conflict edges or forbidden transitions. In *Theory and Applications of Models of Computation (TAMC)*, volume 7876 of *Lecture Notes in Computer Science*, pages 343–354. Springer, 2013.

[127] Mamadou Moustapha Kanté, Fatima Zahra Moataz, Benjamin Momège, and Nicolas Nisse. Finding paths in grids with forbidden transitions. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, volume 9224 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2015.

[128] Richard M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(4):45–68, 1975.

[129] Naoki Katoh, Toshihide Ibaraki, and Hisashi Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982.

[130] Ken-ichi Kawarabayashi and Yusuke Kobayashi. Edge-disjoint odd cycles in 4-edge-connected graphs. *Journal of Combinatorial Theory, Series B*, 119:12–27, 2016.

[131] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce A. Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, 2012.

[132] Ken-ichi Kawarabayashi, Bruce A. Reed, and Paul Wollan. The graph minor algorithm with parity conditions. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 27–36. IEEE Computer Society, 2011.

[133] Ken-ichi Kawarabayashi, Robin Thomas, and Paul Wollan. A new proof of the flat wall theorem. *Journal of Combinatorial Theory, Series B*, 129:204–238, 2018.

[134] Ken-ichi Kawarabayashi and Paul Wollan. A shorter proof of the graph minor algorithm: the unique linkage theorem. In *ACM Symposium on Theory of Computing (STOC)*, pages 687–694. ACM, 2010.

[135] Yasushi Kawase, Yusuke Kobayashi, and Yutaro Yamaguchi. Finding a path with two labels forbidden in group-labeled graphs. *Journal of Combinatorial Theory, Series B*, 143:65–122, 2020.

[136] Jens Keppeler. *Answering Conjunctive Queries and FO+MOD Queries under Updates*. PhD thesis, Humboldt University of Berlin, Germany, 2020.

[137] Benny Kimelfeld and Yehoshua Sagiv. Revisiting redundancy and minimization in an XPath fragment. In *International Conference on Extending Database Technology (EDBT)*, volume 261 of *ACM International Conference Proceeding Series*, pages 61–72. ACM, 2008.

[138] Benny Kimelfeld and Yehoshua Sagiv. Extracting minimum-weight tree patterns from a schema with neighborhood constraints. In *International Conference on Database Theory (ICDT)*, pages 249–260, 2013.

[139] Yusuke Kobayashi and Christian Sommer. On shortest disjoint paths in planar graphs. *Discrete Optimization*, 7(4):234–245, 2010.

[140] Andrea S. LaPaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984.

[141] Andrea S. Lapaugh and Ronald L. Rivest. The subgraph homeomorphism problem. *Journal of Computer and System Sciences*, 20(2):133 – 149, 1980.

[142] Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[143] Ruonan Li, Hajo Broersma, Chuandong Xu, and Shenggui Zhang. Cycle extension in edge-colored complete graphs. *Discrete Mathematics*, 340(6):1235–1241, 2017.

[144] Ruonan Li, Hajo Broersma, and Shenggui Zhang. Properly edge-colored theta graphs in edge-colored complete graphs. *Graphs and Combinatorics*, 35(1):261–286, 2019.

[145] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

[146] Katja Losemann and Wim Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems*, 38(4):24:1–24:39, 2013.

[147] Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In *Joint Meeting of the Annual Conference on Computer Science Logic (CSL) and the Symposium on Logic in Computer Science (LICS) (CSL-LICS)*, pages 67:1–67:10. ACM, 2014.

[148] Erkki Mäkinen. On lexicographic enumeration of regular and context-free languages. *Acta Cybernetica*, 13(1):55–62, 1997.

[149] Yannis Manoussakis. Alternating paths in edge-colored complete graphs. *Discrete Applied Mathematics*, 56(2-3):297–309, 1995.

[150] Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. Bonxai: Combining the simplicity of DTD with the expressiveness of XML schema. *ACM Transactions on Database Systems*, 42(3):15:1–15:42, 2017.

[151] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for simple regular expressions. In *Mathematical Foundations of Computer Science (MFCS)*, pages 889–900, 2004.

[152] Wim Martens, Frank Neven, and Thomas Schwentick. Complexity of decision problems for XML schemas and chain regular expressions. *SIAM Journal on Computing*, 39(4):1486–1530, 2009.

[153] Wim Martens, Matthias Niewerth, and Tina Popp. A trichotomy for regular trail queries. *CoRR*, abs/1903.00226, 2021.

[154] Wim Martens, Matthias Niewerth, and Tina Trautner. A trichotomy for regular trail queries. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 154 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[155] Wim Martens and Tina Popp. The complexity of regular trail and simple path queries on undirected graphs. In *Symposium on Principles of Database Systems (PODS)*, pages 165–174. ACM, 2022.

[156] Wim Martens and Tina Trautner. Evaluation and enumeration problems for regular path queries. In *International Conference on Database Theory (ICDT)*, volume 98 of *LIPIcs*, pages 19:1–19:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[157] Wim Martens and Tina Trautner. Bridging theory and practice with query log analysis. *SIGMOD Record*, 48(1):6–13, 2019.

[158] Wim Martens and Tina Trautner. Dichotomies for evaluating simple regular path queries. *ACM Transactions on Database Systems*, 44(4):16:1–16:46, 2019.

[159] Robert McNaughton and Seymour Papert. *Counter-free automata.* MIT Press, 1971.

[160] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 12 1995.

[161] Karl Menger. Zur allgemeinen kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

[162] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[163] B. Monien. How to find long paths efficiently. In *Analysis and Design of Algorithms for Combinatorial Problems*, volume 109 of *North-Holland Mathematics Studies*, pages 239–254. North-Holland, 1985.

[164] Harold Marston Morse. Recurrent geodesics on a surface of negative curvature. *Transactions of the American Mathematical Society*, 22(1):84–100, Jan 1921.

[165] Katta G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[166] Zhivko Prodanov Nedev and Peter T. Wood. A polynomial-time algorithm for finding regular simple paths in outerplanar graphs. *Journal of Algorithms*, 35(2):235–259, 2000.

[167] Neo4j. `neo4j.com`.

[168] Neo4j. The neo4j developer manual v3.2. `https://neo4j.com/docs/developer-manual/3.2/`, 2017.

[169] Matthias Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *Symposium on Logic in Computer Science (LICS)*, pages 769–778. ACM, 2018.

[170] Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *Symposium on Principles of Database Systems (PODS)*, pages 179–191. ACM, 2018.

[171] Tatsuo Ohtsuki. The two disjoint path problem and wire routing design. In *Graph Theory and Algorithms*, volume 108 of *Lecture Notes in Computer Science*, pages 207–216. Springer, 1980.

[172] Cypher query language reference, version 9, mar. 2018. `https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf`.

[173] Oracle spatial and graph. `www.oracle.com/technetwork/database/options/spatialandgraph/`.

[174] Yehoshua Perl and Yossi Shiloach. Finding two disjoint paths between two pairs of vertices in a graph. *Journal of the ACM*, 25(1):1–9, 1978.

[175] Neo4J Petra Selmer. Personal communication.

[176] Property graph query language. `https://pgql-lang.org/spec/1.4/`, 2021. PGQL 1.4 Specification.

[177] Jean-Eric Pin. Syntactic semigroups. In *Handbook of Formal Languages (1)*, pages 679–746. Springer, 1997.

[178] Jean-Éric Pin. The dot-depth hierarchy, 45 years later. In *The Role of Theory in Computer Science*, pages 177–202. World Scientific, 2017.

[179] Thomas Place and Luc Segoufin. Decidable characterization of FO2$(<, +1)$ and locality of da. *CoRR*, abs/1606.03217, 2016.

[180] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. In *International Conference on Database Theory (ICDT)*, pages 177–194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[181] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theory of Computing Systems*, 61(1):31–83, 2017.

[182] Neil Robertson and Paul D. Seymour. Graph minors .xiii. the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63(1):65–110, 1995.

[183] Miguel Romero, Pablo Barceló, and Moshe Y. Vardi. The homomorphism problem for regular graph patterns. In *Symposium on Logic in Computer Science (LICS)*, pages 1–12. IEEE Computer Society, 2017.

[184] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki

Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9):62–71, 2021.

[185] Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965.

[186] Paul D. Seymour. Disjoint paths in graphs. *Discrete Mathematics*, 29(3):293–309, 1980.

[187] Yossi Shiloach. A polynomial solution to the undirected two paths problem. *Journal of the ACM*, 27(3):445–456, 1980.

[188] Aleksandrs Slivkins. Parameterized tractability of edge-disjoint paths on directed acyclic graphs. *SIAM Journal on Discrete Mathematics*, 24(1):146–157, 2010.

[189] Slawek Staworko and Piotr Wieczorek. Characterizing XML twig queries with examples. In *International Conference on Database Theory (ICDT)*, volume 31 of *LIPIcs*, pages 144–160. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[190] Howard Straubing. A generalization of the schützenberger product of finite monoids. *Theoretical Computer Science (TCS)*, 13:137–150, 1981.

[191] Dimitri Surinx, George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. Relative expressive power of navigational querying on graphs using transitive closure. *Logic Journal of the IGPL*, 23(5):759–788, 2015.

[192] J. W. Suurballe. Disjoint paths in a network. *Networks*, 4(2):125–145, 1974.

[193] J. W. Suurballe and Robert Endre Tarjan. A quick method for finding shortest pairs of disjoint paths. *Networks*, 14(2):325–336, 1984.

[194] Stefan Szeider. Finding paths in graphs avoiding forbidden transitions. *Discrete Applied Mathematics*, 126(2-3):261–273, 2003.

[195] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[196] TigerGraph Team. GSQL language reference. `https://docs-legacy.tigergraph.com/v/3.3/dev/gsql-ref`, 2021.

[197] Denis Thérien. Classification of finite monoids: The language approach. *Theoretical Computer Science (TCS)*, 14:195–208, 1981.

[198] Denis Thérien and Thomas Wilke. Over words, two variables are as powerful as one quantifier alternation. In *ACM Symposium on Theory of Computing (STOC)*, pages 234–240. ACM, 1998.

[199] Carsten Thomassen. 2-linked graphs. *European Journal of Combinatorics*, 1(4):371–378, 1980.

[200] Axel Thue. *Über unendliche Zeichenreihen*. Skrifter udg. af Videnskabs-Selskabet i Christiania : 1. Math.-Naturv. Klasse. Dybwad [in Komm.], 1906.

[201] Tigergraph. `www.tigergraph.com`.

[202] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[203] Xin Wang, Yang Wang, Yang Xu, Ji Zhang, and Xueyan Zhong. Extending graph pattern matching with regular expressions. In *International Conference on Database and Expert Systems - Part II (DEXA (2))*, volume 12392 of *Lecture Notes in Computer Science*, pages 111–129. Springer, 2020.

[204] Wikidata. `wikidata.org`.

[205] Peter T. Wood. Query languages for graph databases. *SIGMOD Record*, 41(1):50–60, 2012.

[206] Wanhong Xu and Z. Meral Özsoyoglu. Rewriting XPath queries using materialized views. In *International Conference on Very Large Data Bases (VLDB)*, pages 121–132. ACM, 2005.

[207] Bing Yang and S. Q. Zheng. Finding min-sum disjoint shortest paths from a single source to all pairs of destinations. In *International conference on Theory and Applications of Models of Computation (TAMC)*, volume 3959 of *Lecture Notes in Computer Science*, pages 206–216. Springer, 2006.

[208] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Symposium on Principles of Database Systems (PODS)*, pages 230–242, 1990.

[209] Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[210] Jin Y. Yen. Finding the lengths of all shortest paths in n-node nonnegative-distance complete networks using ½n^3 additions and n^3 comparisons. *Journal of the ACM*, 19(3):423–424, 1972.

[211] Anders Yeo. A note on alternating cycles in edge-coloured graphs. *Journal of Combinatorial Theory, Series B*, 69(2):222–225, 1997.

Index of Notations

Index

List of My Publications

Trichotomy for Regular Trail Queries

- **Preliminary version**: Wim Martens, Matthias Niewerth, and Tina Trautner. A trichotomy for regular trail queries. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 154 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020
- **Full version**: Wim Martens, Matthias Niewerth, and Tina Popp. A trichotomy for regular trail queries. *CoRR*, abs/1903.00226, 2021

Dichotomies on Simple Transitive Expressions

- **Preliminary version**: Wim Martens and Tina Trautner. Evaluation and enumeration problems for regular path queries. In *International Conference on Database Theory (ICDT)*, volume 98 of *LIPIcs*, pages 19:1–19:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018
- **Full version**: Wim Martens and Tina Trautner. Dichotomies for evaluating simple regular path queries. *ACM Transactions on Database Systems*, 44(4):16:1–16:46, 2019
- **Extended Abstract**: Wim Martens and Tina Trautner. Bridging theory and practice with query log analysis. *SIGMOD Record*, 48(1):6–13, 2019

Containment of Fragments of CRPQs

- **Preliminary version**: Diego Figueira, Adwait Godbole, Shankara Narayanan Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. Containment of simple conjunctive regular path queries. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 371–380, 2020

Complexity for Regular Trail and Simple Path Queries on Undirected Graphs

- **Preliminary version**: Wim Martens and Tina Popp. The complexity of regular trail and simple path queries on undirected graphs. In *Symposium on Principles of Database Systems (PODS)*, pages 165–174. ACM, 2022