Optimizing Tree Patterns for Querying Graph- and Tree-Structured Data[*]

Wojciech Czerwiński[†]
University of Warsaw
wczerwin@mimuw.edu.pl

Wim Martens
Universität Bayreuth
wim.martens@uni-bayreuth.de

Matthias Niewerth[‡]
Universität Bayreuth
matthias.niewerth@uni-bayreuth.de

Paweł Parys
University of Warsaw
parys@mimuw.edu.pl

ABSTRACT

Many of today's graph query languages are based on graph pattern matching. We investigate optimization for tree-shaped patterns with transitive closure. Such patterns are quite expressive, yet can be evaluated efficiently. The *minimization* problem aims at reducing the number of nodes in patterns and goes back to the early 2000's. We provide an example showing that, in contrast to earlier claims, tree patterns cannot be minimized by deleting nodes only. The example resolves the $M \overset{?}{=} NR$ problem, which asks if a tree pattern is minimal if and only if it is nonredundant. The example can be adapted to also understand the complexity of minimization, which was another question that was open since the early research on the problem. Interestingly, the latter result also shows that, unless standard complexity assumptions are false, more general approaches for minimizing tree patterns are also bound to fail in some cases.

1. INTRODUCTION

Tree patterns are a very natural and user-friendly means to query graph- and tree-structured data. This is why they can be found in the conceptual core of widely used query languages for graphs and trees.

1.1 Motivation from Graph Query Languages

Graph pattern matching is a fundamental concept in modern declarative graph query languages. Indeed, graph query languages usually take one of two main perspectives: *graph traversal* or *graph pattern matching*, the former being the imperative and the latter being the declarative variant [31]. Today's most prominent declarative graph query languages are SPARQL 1.1 [33] and Neo4J Cypher [25]. Both languages make it very clear in their specifications that they have graph pattern matching at their core. SPARQL 1.1 explicitly writes "SPARQL is based around graph pattern matching" [33, Section 5], and the introduction of Neo4J's documentation on Cypher [25, Section 3.1.1] is essentially an introduction to the principles of graph pattern matching. Gremlin [19], another popular graph query language, leans more towards the graph traversal side of the spectrum, but also supports pattern matching style querying. It performs graph pattern matching similar to SPARQL [31].

The reason why graph pattern matching is so popular is not surprising. Graph patterns are expressive, reasonably simple and intuitive to understand, and often efficient to evaluate. Consider the graph in Figure 1. It contains information on artists, their occupation, and their place of birth. The graph structure is inspired on *property graphs*, a popular model for graph databases in practice [30, 3]. In this model, each node and edge carry a label and, in addition, nodes can have a set of attributes. For instance, the node related to Jimi Hendrix has the label Person, its "name" attribute is Jimi Hendrix, and its "aka" attribute is James Marshall Hendrix.

Assume that we would like to find the artists who were born in the United States. This corresponds to finding names of Person nodes that have (1) an occupation edge to "a subclass of artist" and (2) a place of birth edge to a city that is located in the United States. For expressing these conditions, we need to reason about paths in the graph. The occupation in (1) should be connected to artist by a path of subclassof-edges and the city in (2) to United States by a path of locatedin-edges.

These conditions are expressed in the pattern in Figure 2.[1] It has two types of edges and two types of nodes. Single

[*] The original version of this article was published in PODS 2016, titled "Minimization of tree pattern queries" [12].

[†] Supported by Poland's National Science Centre grant no. UMO-2013/11/D/ST6/03075.

[‡] Supported by grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft (Emmy Noether Nachwuchsgruppe).

This is a minor revision of the work published in PODS'16, ISBN 978-1-4503-4191-2/16/06...$15.00, June 26–July 01, 2016 San Francisco, CA, USA. DOI: http://dx.doi.org/10.1145/2902251.2902295.

[1] The pattern is closely related to *graph patterns*, which were identified by Angles et al. [3] as a part of the conceptual core of many of today's graph query languages.

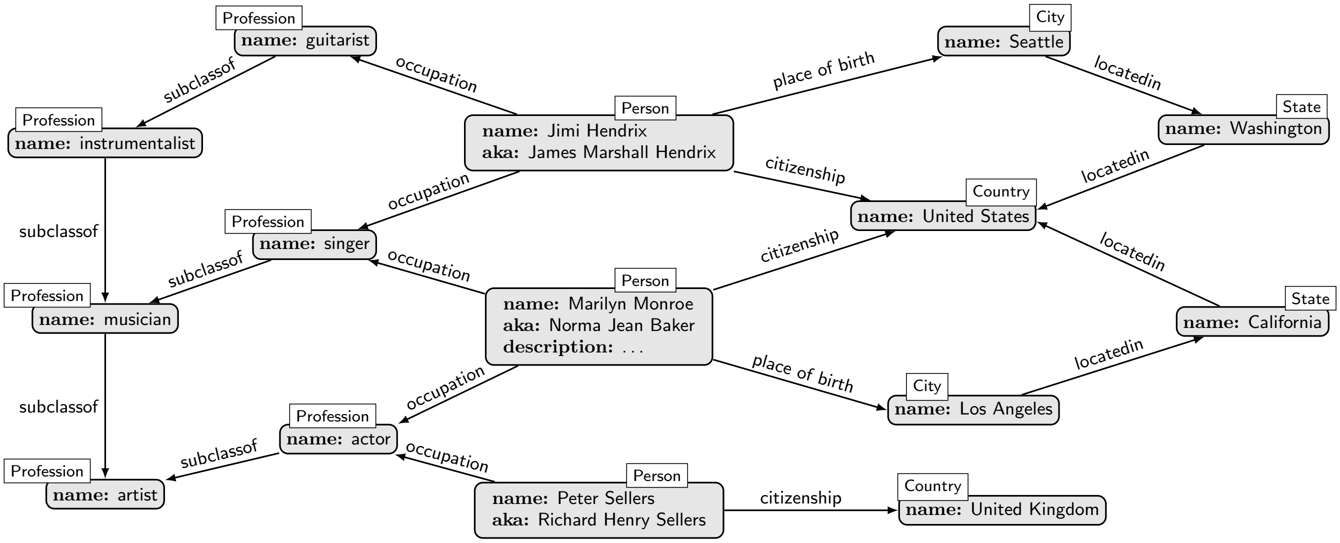


Figure 1: A graph database (as a *property graph*), inspired on a fragment of WikiData

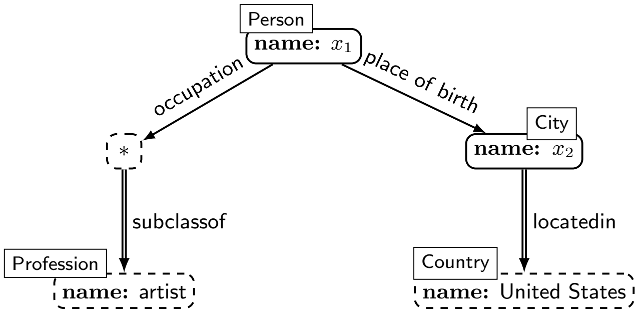


Figure 2: A tree pattern finding the artists who were born in the United States. The query returns the person names and the cities where they were born. (Fully circled nodes are return nodes.)

edges in the pattern can be matched to single edges in the graph with the same label. The double edges can be matched to *paths* in the graph on which every edge has the label given in the query. (For instance, the locatedin edge in the query can be matched on the path from the Los Angeles to United States nodes.) The solid nodes in the query are *output nodes* and the dashed nodes are ordinary nodes. The symbol $*$ is a wildcard symbol that can be matched to any label. The query has two variables: x_1 and x_2. Intuitively, computing the answers to the pattern corresponds to finding *matches* of the pattern in the graph and, for each such match, return the nodes (or values) matched by the variables in output nodes of the pattern. When evaluated on the graph in Figure 1, this pattern would return (Jimi Hendrix, Seattle) and (Marilyn Monroe, Los Angeles).

Our example query is structured as a tree. In general, the underlying structure of queries in SPARQL or Cypher can be an arbitrary graph and can therefore contain cycles. The acyclic queries form, however, an important subclass. Graph patterns closely correspond to *conjunctive queries*, which are known to be NP-complete to evaluate [10]. The tree-shaped patterns closely correspond to *acyclic conjunctive queries*, which can be evaluated in polynomial time. In fact, the quest for subclasses of conjunctive queries with a polynomial time evaluation problem is rich of beautiful results (see, e.g., [17]). In this paper, however, we focus on queries whose underlying structure is a tree and, for this reason, have a tractable (polynomial time) evaluation problem. (We note that the transitive closure operators we use make no difference in this respect.)

From a graph query language perspective, the tree patterns from this paper correspond to tree-shaped conjunctive queries (or tree-shaped graph patterns) with transitive closure. Transitive closure seems to be becoming increasingly popular in graph query languages, even though there have been challenges in the early version of the operator in SPARQL 1.1 [5, 23]. In WikiData's list of *example queries* [34], which help users getting started with the data set, 72 out of 272 queries use transitive closure of a label, which means that the feature is important.

1.2 Motivation from Tree Query Languages

Tree-structured data is among us in many forms, JSON and XML being two examples. The tree pattern queries that we consider were originally introduced to investigate query languages for tree-structured data [24]. They are an abstraction of a fragment of XPath [28] and therefore also appear in XQuery [29], XSLT [21], and languages for querying JSON, see, e.g., [20]. Indeed, patterns such as the one in Figure 2 can equally well be used for querying tree-structured data. (This is easy to see, since a tree is a special case of a graph.)

Tree pattern queries are also important for many topics in fundamental research on tree-structured data. For instance, they form a basis for conjunctive queries over trees [18, 8], for models of XML with incomplete information [6], and the closely related pattern-based XML queries [16]. They are used for specifying guards in Active XML systems [1] and for specifying schema mappings in XML data exchange [4].

1.3 The Core Problem

We report in this paper on recent progress on the minimization problem for tree patterns [12]. Optimization of queries has been a main topic of database research ever since the beginning and therefore is very natural to consider for tree patterns. Tree pattern query optimization already attracted significant attention in the form of *query containment* [24, 26, 13], *satisfiability* [7], and *minimization* [2, 11, 15, 22, 27, 35].

Almost all this former work on containment, satisfiability, and minimization exclusively considered tree patterns as a language for querying *tree-structured data*. However, as argued by Miklau and Suciu [24, Section 5.3], many of these results hold just the same if we use tree patterns to query *graph-structured data*, i.e., if we use tree patterns as in Section 1.1. The same argument holds for the minimization problem. For this reason, one can often obtain results for tree patterns on graph-structured data while only considering tree-structured data in proofs.

We note that the tree patterns that were considered in this former work (and the ones we consider in the proofs of [12]) cannot express the query in Figure 2, for the simple reason that they cannot express the transitive closure of *subclassof*. We will argue that our results extend to these more expressive queries as well.

Another difference is that we consider Boolean queries, whereas the query in Figure 2 returns tuples of answers. Again, we will argue that our results also apply for higher-arity queries. We consider the following problem.

Tree Pattern Minimization	
Given:	A tree pattern p and $k \in \mathbb{N}$
Question:	Is there a tree pattern q, equivalent to p, such that its size is at most k?

The main difficulties for this problem are already present in a very restricted set of tree patterns that

- only query *graphs that are node-labeled and are tree-shaped*; and
- over these graphs, only use *labeled node tests*, *wildcard node tests*, the *child relation*, and the *descendant relation*.

These are precisely the patterns introduced by Miklau and Suciu [24].

1.4 History of the Problem

Although the patterns we consider here have been widely studied [14, 24, 36, 15, 22, 1, 9, 4, 32], their minimization problem remained elusive for a long time. The most important previous work for their minimization was done by Kimelfeld and Sagiv [22] and by Flesca, Furfaro, and Masciari [14, 15].

The key challenge was understanding the relationship between *minimality* (M) and *nonredundancy* (NR). Here, a tree pattern is minimal if it has the smallest number of nodes among all equivalent tree patterns. It is nonredundant if none of its leaves (or branches[2]) can be deleted while remaining equivalent. The question was if minimality and nonredundancy are the same ([22, Section 7] and [15, p. 35]):

M $\overset{?}{=}$ NR Problem:

Is a tree pattern minimal
if and only if it is nonredundant?

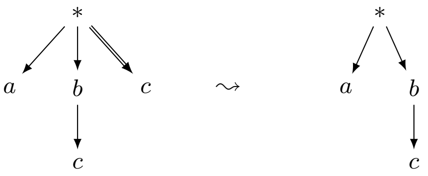


Figure 3: Minimizing a tree pattern by removing redundant nodes

Notice that a part of the M $\overset{?}{=}$ NR problem is easy to see: a minimal pattern is trivially also nonredundant (that is, M $\subseteq$ NR). The opposite direction is much less clear.

If the problem would have a positive answer, it would mean that the simple algorithmic idea summarised in Algorithm 1 correctly minimizes tree patterns. Therefore, the M $\overset{?}{=}$ NR problem is a natural question about the design of minimization algorithms for tree patterns.

Algorithm 1 Computing a nonredundant subpattern

```
Input: A tree pattern $p$
Output: A nonredundant tree pattern $q$, equivalent to $p$

  while a leaf of $p$ can be removed
                          (remaining equivalent to $p$) do
     Remove the leaf
  end while
  return the resulting pattern
```

Example 1.1. *It is easy to see that Algorithm 1 can be used for minimizing some patterns. Consider the left pattern in Figure 3. Its root (labeled with a wildcard $*$) can be matched on nodes n in a graph such that (1) n has an a-labeled successor, (2) a b-labeled successor with a c-labeled successor, and (3) a c-labeled node is reachable from n. (In this example, edge labels do not matter.) In the semantics of such patterns, it is allowed that the different c-nodes are matched on the same node in the data. Therefore, condition (3) is redundant and the pattern to the right is equivalent and smaller.*

The M $\overset{?}{=}$ NR problem is also a question about complexity. The main source of complexity of the nonredundancy algorithm lies in testing equivalence between a pattern p and a pattern p', which is generally coNP-complete [24]. If M $\overset{?}{=}$ NR has a positive answer, then Tree Pattern Minimization would also be coNP-complete.

In fact, the problem was claimed to be coNP-complete in 2003 [14, Theorem 2], but the status of the minimization- and the M $\overset{?}{=}$ NR problems were re-opened by Kimelfeld and Sagiv [22], who found errors in the proofs. Flesca et al.'s journal paper then proved that M = NR for a limited class of tree patterns, namely those where *every wildcard node has at most one child* [15]. Nevertheless, for tree patterns,

(a) the status of the M $\overset{?}{=}$ NR problem and

(b) the complexity of the minimization problem

remained open.

[2] Kimelfeld and Sagiv proved that a tree pattern has a redundant branch if and only if it has a redundant leaf [22, Proposition 3.3].

1.5 Our Contributions

We proved the following [12]:

(a) There exists a tree pattern that is nonredundant but not minimal. Therefore, M ≠ NR.

(b) Tree Pattern Minimization is Σ_2^P-complete. This implies that even the main idea in Algorithm 1 cannot work unless coNP = Σ_2^P.

Interestingly, our counterexample for (a) uses only two wildcard nodes with two children and only one transitive edge. This is only barely beyond the fragment for which it is known that minimality and nonredundancy coincide.

Outline.

In Section 2 we formally define tree patterns, their semantics, and discuss their relationship to the queries in the Introduction. We show why M ≠ NR in Section 3. In Section 4 we briefly discuss the complexity result and its consequences.

2. PRELIMINARIES

We formally define our data model and queries, recall important results about the static analysis of queries, and discuss the relationship between other data models and ours.

Data Model: Node- and Edge-Labeled Graphs.

Our data model is very simple: we use finite, node- and edge-labeled directed graphs, where the labels come from an infinite set. In the graph database world, this model is closely related to *property graphs*, the data model for Neo4J [30] (see, e.g., [3] for a formal definition of property graphs).[3]

More formally, a *(node- and edge-) labeled graph* is a triple (V, E, lab), where V is a finite nonempty set of *nodes*, E is a set of directed *edges* $(u, v) \in V \times V$ and $\text{lab} : V \cup E \to \Lambda$ is a *labeling function* assigning to every node and edge its label coming from an infinite set of labels Λ. We assume that graphs are connected. A *path* from node v_1 to v_n is a sequence of nodes $\pi = v_1 \cdots v_n$, where $(v_i, v_{i+1}) \in E$ for every $i = 1, \ldots, n-1$.

A graph is a *tree* if,

(i) for every node v, there is at most one node u (called *parent* of v) with $(u, v) \in E$ and

(ii) there is exactly one node v (called *root*) without a parent.

We assume familiarity with standard terminology on trees such as *child* and *descendant.*

The Queries: Tree Patterns.

Our formal model of graph patterns allows node- and edge label tests, wildcard tests, and transitive closures. The wildcard test (denoted by "$*$" in patterns) matches any node- or edge label in a graph. To avoid confusion, we assume that $* \notin \Lambda$.

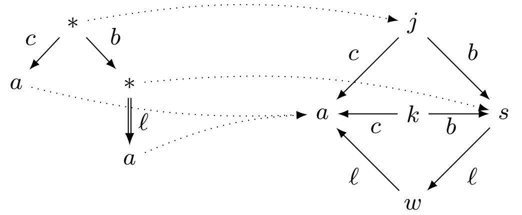


Figure 4: Example of a match from a tree pattern (left) to a labeled graph (right)

Formally, a *graph pattern* is a tuple $p = (V_p, E_p, \text{lab}_p)$ where $\text{lab}_p : V_p \cup E_p \to \Lambda \uplus \{*\}$ and V_p is partitioned in two sets: *simple edges* and *transitive closure edges.* In figures, we draw transitive closure edges using double lines. Furthermore, if we do not write a label on an edge, we implicitly assume that the edge label is the wildcard "$*$".

A *tree pattern* is a graph pattern that satisfies the conditions (i) and (ii) we required for trees. From now on in this paper, we will only consider *tree patterns* (although many definitions also apply for graph patterns). The *size* of a pattern p, denoted size(p), is the number of its nodes.

For simplicity, we will define our queries to be Boolean, that is, we will only consider whether they can be matched in a graph or not. Tree patterns with output nodes have been considered as well [24, 22] and our main results also apply to those queries. We discuss this later in the Preliminaries (see *Boolean vs. k-ary queries*).

Semantics of Queries.

We use a homomorphism-based semantics for tree patterns. For a tree pattern $p = (V_p, E_p, \text{lab}_p)$ and a graph $G = (V, E, \text{lab})$, a function $\text{m} \colon V_p \to V$ is a *match* of p in g if it fulfills all the following conditions:

(1) If $\text{lab}_p(v) \neq *$ for $v \in V_p$ then $\text{lab}_p(v) = \text{lab}(\text{m}(v))$.

(2) If $(u, v) \in E_p$ is a simple edge then $(\text{m}(u), \text{m}(v))$ is an edge in G. Furthermore, if $\text{lab}_p((u, v)) \neq *$ then $\text{lab}_p((u, v)) = \text{lab}((\text{m}(u), \text{m}(v)))$.

(3) If $(u, v) \in E_p$ is a transitive closure edge then there is a path from $\text{m}(u)$ to $\text{m}(v)$ in G that satisfies the label constraint of the edge. That is, there exists a path $\pi = u_1 \cdots u_n$ in G (with $n > 1$)) such that $\text{m}(u) = u_1$ and $\text{m}(v) = u_n$. Furthermore, if $\text{lab}_p((u, v)) \neq *$, then all edges (u_i, u_{i+1}) in π are labeled $\text{lab}_p((u, v))$.

We say that *p can be matched in G* if there exists a match from p to G. Figure 4 shows an example of a match. Notice that we do not require matches to be injective.

Definition 2.1 (Semantics of Tree Patterns). *The set of models of a tree pattern p, denoted by $M(p)$, is the set of graphs in which p can be matched.*

Containment, Equivalence, and Minimality.

A tree pattern p_1 is *contained* in a tree pattern p_2 if $M(p_1) \subseteq M(p_2)$, which we denote by $p_1 \subseteq p_2$. If $p_1 \subseteq p_2$ and $p_1 \supseteq p_2$ then we say that the patterns p_1 and p_2 are *equivalent* and we write $p_1 \equiv p_2$.

Figure 3 contains two patterns that are equivalent. (For the left pattern, the c-labeled node on the right branch can

[3] Property graphs are more refined, however, since they associate *properties* to nodes in addition to labels. From a formal perspective, we want that nodes in the graph are not uniquely determined by their label. We do not want that different occurrences of a label in a query must always be mapped to the same node in the graph. This behaviour would introduce unwanted cycles in tree pattern queries.

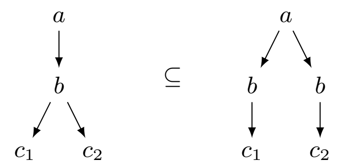


Figure 5: Example for containment of patterns. (Nonlabeled edges are implicitly assumed to have wildcard tests.)

always be matched to wherever the c-labeled node in the middle branch is matched. Therefore it is equivalent to the pattern on the right.) In Figure 5, we give an example for pattern containment. The right pattern matches a-nodes which have c_1- and c_2-nodes on distance two, such that there are b-nodes between the a and the c_i. The pattern on the left additionally requires the two b-nodes to be the same. Since the latter is more restrictive, if the left pattern can be matched in a graph, then the right one can be matched there as well.

The following problem is important in many query optimization procedures:

Tree Pattern Equivalence	
Given:	Two tree patterns p_1 and p_2
Question:	Is $p_1 \equiv p_2$?

We call a tree pattern p *redundant* if one of its nodes can be removed without changing its set of models. For a node v of p, we denote by $p \setminus v$ the pattern obtained from p by removing v and all its descendants and incident edges.

Definition 2.2 (Minimality, Nonredundancy).

- A tree pattern p is *redundant* if it is equivalent to $p \setminus v$ for a node v of p. In this case, v is a *redundant node*. If p is not redundant we say that it is *nonredundant*.
- A pattern p is said to be *minimal* if there exists no tree pattern that is equivalent to p but has strictly smaller size.

It is known that tree patterns are redundant if and only if they have a redundant leaf [22, Proposition 3.3].

Complexity.

One can obtain an almost trivial Σ_2^P upper bound for Tree Pattern Minimization (as defined in the Introduction) by using the following result.

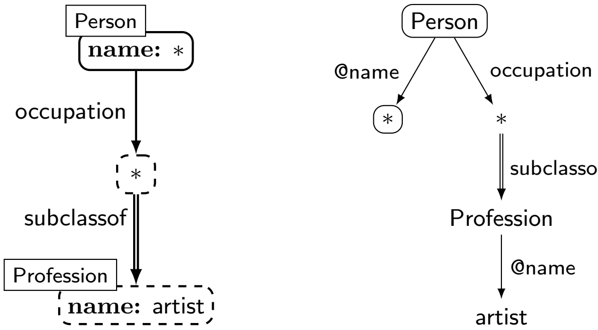
Theorem 2.3. Tree Pattern Equivalence *is* coNP-*complete.*

Proof sketch. Miklau and Suciu [24] prove this theorem for tree patterns without edge labels, but these can easily be added. Furthermore, their patterns only have tree models, whereas we consider graph models. However, they explain [24, Section 5.3] that these two variants of the problem are the same. □

From this result, a Σ_2^P upper bound for Tree Pattern Minimization is immediate.

Theorem 2.4. Tree Pattern Minimization *is in* Σ_2^P.

Proof. Given a tree pattern p and $k \in \mathbb{N}$, the Σ_2^P algorithm first guesses (existential quantification) a tree pattern p' of size at most k and then checks (universal quantification) if p' and p are equivalent. □



(a) Property Graph Query (b) Translated Query

Figure 6: Translating a subquery of Figure 2 to our simplified model

Notice that, if M = NR, then p' can be found among the subpatterns of p, which would drop the upper bound to coNP.

Boolean vs. k-ary queries.

One can easily extend tree patterns to k-ary tree patterns that return k-tuples of answers (see, e.g., [24, 22]). We argue that our results also hold for such queries. It is trivial for our M ≠ NR example, because a Boolean query is just a special case of a k-ary query. The other main result is the Σ_2^P-completeness result in Theorem 4.1. The Σ_2^P upper bound can be seen to hold for k-ary queries by using the same naive algorithm as in Theorem 2.4 and using the argument of Kimelfeld and Sagiv [22, Section 5.2] for showing that Tree Pattern Equivalence for k-ary queries polynomially reduces to the same problem for Boolean queries. The Σ_2^P lower bound follows immediately.

Relationship to the Queries in the Introduction.

The tree patterns we defined here are much simpler than the pattern we discussed in the Introduction (Figure 2). However, the two types of patterns are closely related when it comes to minimization. Again, since the patterns we have here are simpler, it is easy to see that our M ≠ NR example equally applies to the kind of patterns in the Introduction.

Moreover, the simplified patterns capture much of the expressivity of the more complex patterns modulo a simple encoding. In Figure 6, we demonstrate this translation by example, using a subquery of Figure 2. Essentially, each node of the pattern on the left becomes a node on the right labeled with the *property* (the label in the rectangular box) if present, and the "name"-attributes of nodes become children with incoming edges that identify the type of attribute. (We can make sure that the labels of these incoming edges do not appear elsewhere in the query.)

We do not claim that this translation gives a 100% correspondence between the world of tree patterns and the world of "property graph tree patterns", but we do believe that it shows a very close connection. For instance, the translation can be used for testing equivalence between certain types of property graph patterns (translate to tree patterns and test equivalence between those). Likewise, for a large class of property graph tree patterns, minimization would work very similarly to minimization of the translated tree pattern query.

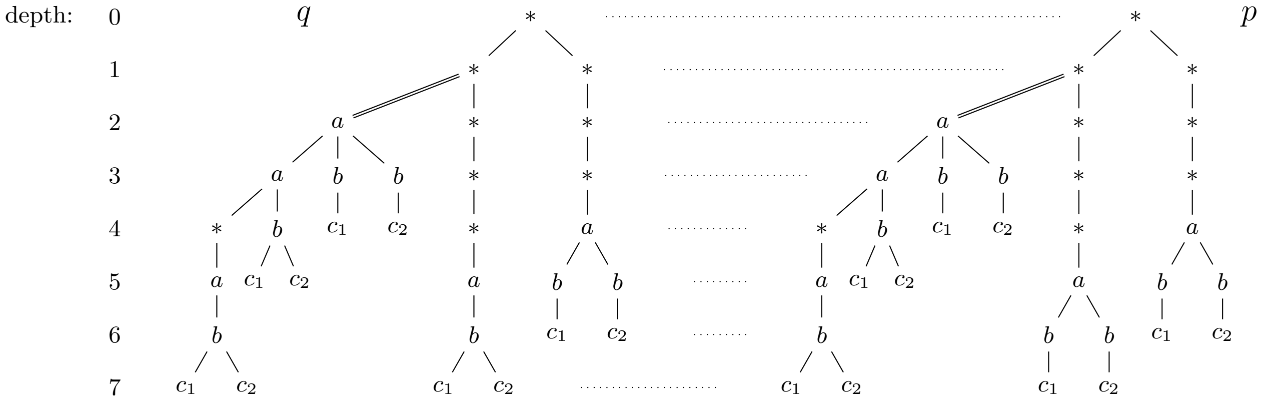


Figure 7: A non-redundant tree pattern p (right) and an equivalent tree pattern q that is smaller (left)

3. THE M $\overset{?}{=}$ NR PROBLEM

We show that M $\neq$ NR by presenting a tree pattern that is nonredundant but also not minimal.

Indeed, we will argue that the right pattern p in Figure 7 is nonredundant and not minimal. (For readability, we omitted arrows. All arrows are assumed to point downwards.) Consider the pattern q on the left of Figure 7. To convince the reader, we need to make three points: (1) p is nonredundant, (2) p is equivalent to q, and (3) q is smaller than p.

Point (3) is trivial: q can be obtained from p by merging two b-nodes on depth six. Therefore, q has one fewer node than p. Points (1) and (2) are non-trivial. Here we will only show (2) because it is the most interesting argument of the two. (Point (1) can be shown by proving that p is not equivalent to any of its subpatterns, see [12].)

We want to convince the reader of point (2) by a sequence of pictures. First of all, observe that $q \subseteq p$. The reason is the same as the one we already discussed in Figure 5. Therefore it only remains to argue why $p \subseteq q$.

In Figure 8, we depicted q (always on the left) and three patterns p_1, p_2 and p_3 on the right. If p is matched in a graph, there are three possibilities for matching the double edge connecting the $*$-node with the a-node. This double edge is matched to a path that either consists of

(a) one edge,

(b) two edges, or

(c) at least three edges.

These three possibilities are depicted on the right of Figure 8. If we have case (a), then we can also match the left pattern in Figure 8(a) (similar for (b) and (c)). (Some parts of these patterns are grey. We will get to that soon.)

The dotted edges have the following meaning. Whenever the pattern p_1, p_2, or p_3 on the right can be matched on a graph, then pattern q (on the left) can also be matched, by matching the nodes on the left to wherever the connected node on the right is matched. For instance, in case (a), the root of q can always be matched to wherever the root of p_1 was matched. The grey part of p_1 is in fact irrelevant for q in this case. All nodes of q can be matched to places where black nodes of p_1 are matched. The grey parts in (b) and (c) have the same meaning.

The dotted edges show completely how q can be matched in cases (a) and (b). In case (c), we also have a dashed edge. The dashed edge shows how the matching of q works if we have *exactly* three edges in (c), but if there are more, then the target of the edge needs to go downward accordingly. The reason for this is easy to see: the two a-nodes on the right side of q are connected to the root by paths of fixed length. So, if the target of the a-nodes move further away, the root of q needs to follow as well. Since all nodes on the path to the root are wildcards, this is possible. Therefore, q can always be matched in case (c) as well.

This gives us the following Theorem:

THEOREM 3.1 (M $\neq$ NR).
MINIMALITY $\neq$ NONREDUNDANCY

4. COMPLEXITY AND CONSEQUENCES

Leveraging the behavior of the patterns in Figure 7, we could prove the following:

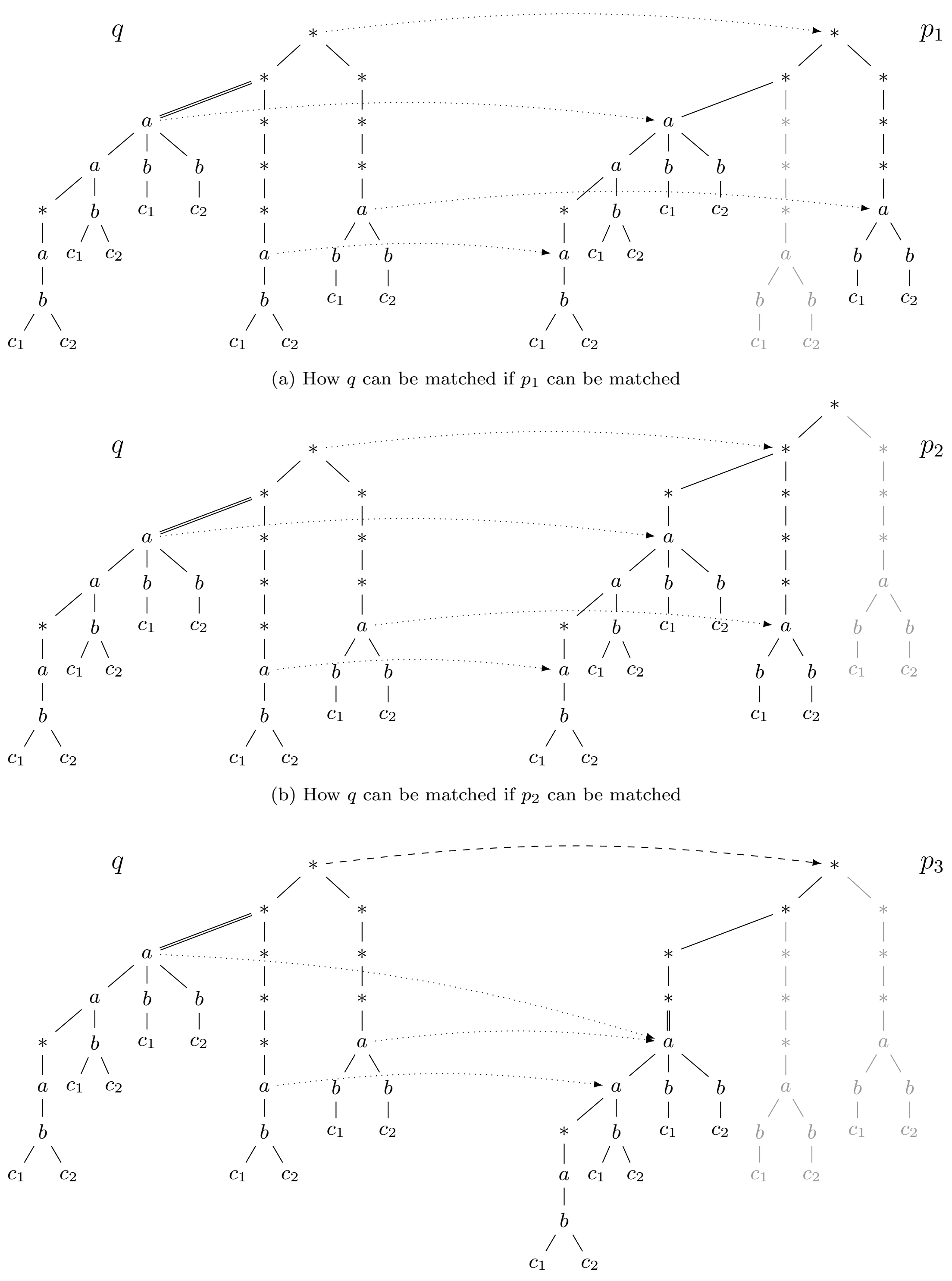
THEOREM 4.1 ([12]). TREE PATTERN MINIMIZATION *is* Σ_2^P*-complete.*

This result is even more drastic than the example in Figure 7. Observe that the query q can be obtained from p by just merging two nodes together. So, the reader may wonder if the following is true. Say that a query is in NR$'$ if none of its nodes can be *deleted or merged* while remaining equivalent. Then, M $\overset{?}{=}$ NR$'$ would be the question: *Can tree patterns always be minimized by deleting or merging nodes?*

Although Figure 7 does not show that M $\neq$ NR$'$, Theorem 4.1 shows that, if M $=$ NR$'$, then coNP $= \Sigma_2^P$. Indeed, if it *would* be possible to always minimize tree patterns by deleting or merging nodes, then Algorithm 1 (from the Introduction) can be adapted to be a coNP test for minimization. (Instead of deleting nodes, it would also merge nodes together.) For this reason, also the *search* for candidate minimal patterns is a difficult problem.

Acknowledgments

We are very grateful to Benny Kimelfeld for insightful discussions and for bringing the tree pattern minimization problem to our attention. We thank Dominik D. Freydenberger for carefully proofreading a draft of the paper.



(a) How q can be matched if p_1 can be matched

(b) How q can be matched if p_2 can be matched

(c) How q can be matched if p_3 can be matched

Figure 8: Showing that patterns p and q in Figure 7 are equivalent

5. REFERENCES

[1] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of active XML systems. *ACM Trans. Database Syst.*, 34(4), 2009.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4):315–331, 2002.

[3] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern graph query languages. *CoRR*, abs/1610.06264, 2016.

[4] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange.* Cambridge University Press, 2014.

[5] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *World Wide Web Conference (WWW)*, pages 629–638, 2012.

[6] P. Barceló, L. Libkin, A. Poggi, and C. Sirangelo. XML with incomplete information. *J. ACM*, 58(1):4, 2010.

[7] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.

[8] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. *J. Comput. Syst. Sci.*, 77(3):450–472, 2011.

[9] H. Björklund, W. Martens, and T. Schwentick. Validity of tree pattern queries with respect to schema information. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 171–182, 2013.

[10] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on Theory of Computing (STOC)*, pages 77–90, 1977.

[11] D. Chen and C. Y. Chan. Minimization of tree pattern queries with constraints. In *International Conference on Management of Data (SIGMOD)*, pages 609–622, 2008.

[12] W. Czerwiński, W. Martens, M. Niewerth, and P. Parys. Minimization of tree pattern queries. In *Symposium on Principles of Database Systems (PODS)*, pages 43–54, 2016.

[13] W. Czerwiński, W. Martens, P. Parys, and M. Przybyłko. The (almost) complete guide to tree pattern containment. In *Symposium on Principles of Database Systems (PODS)*, pages 117–130, 2015.

[14] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *International Conference on Very Large Data Bases (VLDB)*, pages 153–164, 2003.

[15] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. *J. ACM*, 55(1), 2008.

[16] A. Gheerbrant, L. Libkin, and C. Sirangelo. Reasoning about pattern-based XML queries. In *International Conference on Web Reasoning and Rule Systems (RR)*, pages 4–18, 2013.

[17] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In *Symposium on Principles of Database Systems (PODS)*, pages 57–74, 2016.

[18] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.

[19] Gremlin Language. `github.com/tinkerpop/gremlin/wiki`, 2013.

[20] `jsonpath.com/`, January 2017.

[21] M. Kay. XSL Transformations (XSLT) version 3.0. Technical report, World Wide Web Consortium, November 2015. W3C Recommendation, `www.w3.org/TR/2015/CR-xslt-30-20151119/`.

[22] B. Kimelfeld and Y. Sagiv. Revisiting redundancy and minimization in an XPath fragment. In *International Conference on Extending Database Technology (EDBT)*, pages 61–72, 2008.

[23] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24, 2013.

[24] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[25] The Cypher query language. `neo4j.com/docs/developer-manual/current/cypher/`, 2016.

[26] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[27] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *International Conference on Management of Data (SIGMOD)*, pages 299–309, 2002.

[28] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XML Path Language 3.0. Technical report, World Wide Web Consortium, April 2014. `www.w3.org/TR/2014/REC-xpath-30-20140408/`.

[29] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language. Technical report, World Wide Web Consortium, April 2014. W3C Recommendation, `www.w3.org/TR/2014/REC-xquery-30-20140408/`.

[30] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases.* O'Reilly, 2 edition, 2015.

[31] M. A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *Symposium on Database Programming Languages (DBPL)*, pages 1–10, 2015.

[32] S. Staworko and P. Wieczorek. Characterizing XML twig queries with examples. In *International Conference on Database Theory (ICDT)*, pages 144–160, 2015.

[33] SPARQL 1.1 query language. `www.w3.org/TR/sparql11-query/`. World Wide Web Consortium.

[34] Wikidata sparql query service examples. `www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`.

[35] P. T. Wood. Minimising simple XPath expressions. In *WebDB*, pages 13–18, 2001.

[36] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *International Conference on Very Large Data Bases (VLDB)*, pages 121–132, 2005.