The Complexity of XPath Query Evaluation

Georg Gottlob
Database and AI Group
Technische Universität Wien
A-1040 Vienna, Austria
gottlob@dbai.tuwien.ac.at

Christoph Koch
LFCS
University of Edinburgh
Edinburgh EH9 3JZ, UK
koch@dbai.tuwien.ac.at

Reinhard Pichler
Inst. für Computersprachen
Technische Universität Wien
A-1040 Vienna, Austria
reini@logic.tuwien.ac.at

ABSTRACT

In this paper, we study the precise complexity of XPath 1.0 query processing. Even though heavily used by its incorporation into a variety of XML-related standards, the precise cost of evaluating an XPath query is not yet well-understood. The first polynomial-time algorithm for XPath processing (with respect to combined complexity) was proposed only recently, and even to this day all major XPath engines take time exponential in the size of the input queries. From the standpoint of theory, the precise complexity of XPath query evaluation is open, and it is thus unknown whether the query evaluation problem can be parallelized.

In this work, we show that both the data complexity and the query complexity of XPath 1.0 fall into lower (highly parallelizable) complexity classes, but that the combined complexity is PTIME-hard. Subsequently, we study the sources of this hardness and identify a large and practically important fragment of XPath 1.0 for which the combined complexity is LOGCFL-complete and, therefore, in the highly parallelizable complexity class NC_2.

1. INTRODUCTION

XPath 1.0 is the node-selecting query language central to most core XML-related technologies that are under the auspices of the W3C, including XQuery, XSLT, and XML Schema. Evaluating XPath queries efficiently is essential to the effectiveness and real-world impact of these technologies.

The most natural question related to XPath query processing, its complexity, however, has received surprisingly little attention. The first polynomial-time algorithms for XPath processing (w.r.t. both the size of the data and the query, i.e., combined complexity, cf. [10]) were proposed only recently [3]. Apparently, the fact that queries can be evaluated in polynomial time with respect to combined complexity for the *full* XPath 1.0 language was not folklore. We believe that at the time of writing this, all publicly available XPath engines and systems processing languages containing XPath (such as XQuery or XSLT processors) take time exponential in the sizes of the XPath expressions in the input. This thesis is supported by experimental evidence on a number of popular systems in [3]. Moreover, immediate functional implementations of the standards documents, of the current standard, XPath 1.0 [13], as well as the now proposed XPath 2.0 language[1] (through the new XML Query 1.0 Algebra) [14], lead to exponential-time processing of XPath 1.0 queries.

The polynomial-time result of [3] was shown using a form of dynamic programming. Based on this, we presented algorithms that run in time $O(|D|^5 * |Q|^2)$ and space $O(|D|^4 * |Q|^2)$, where $|D|$ denotes the size of the data and $|Q|$ is the size of the query. We also introduced the logical core fragment of XPath, called *Core XPath*, which includes the logical and path processing features of XPath but excludes arithmetics and string manipulations. Core XPath queries can be evaluated in time $O(|D| * |Q|)$, i.e. linear in the size of the query and of the data. In a second paper [4], we improved the above upper bounds on the complexity to time $O(|D|^4 * |Q|^2)$ and space $O(|D|^2 * |Q|^2)$. Moreover, we defined a large fragment of XPath for which we provided a quadratic-time, linear-space evaluation algorithm. We also pointed out the features of XPath causing the various increases in the degrees of polynomials established when moving from a smaller XPath fragment to a larger one.

Now that the combined complexity of XPath is known to be polynomial, a natural question emerges, namely whether XPath is also **P**-hard (i.e., hard for polynomial time), or alternatively, whether it is in the complexity class **NC**, and thus effectively parallelizable. In case the problem is **P**-hard, it is interesting to understand the sources of this hardness, and to find large, effectively parallelizable fragments.

This paper thus studies the precise complexity of XPath 1.0 query processing. The contributions are as follows.

- We establish the combined complexity of XPath to be **P**-hard. This remains true even for the Core XPath fragment.

- We show that positive Core XPath, i.e. Core XPath without negation, is **LOGCFL**-complete, and thus highly parallelizable.

  Moreover, if the language is further restricted to the path expressions fragment (PF) without conditions,

[1]XPath 2.0 now includes most of XQuery and thus is Turing-complete; however, most real-world path queries will remain expressible in XPath 1.0, which is a strict fragment of XPath 2.0.

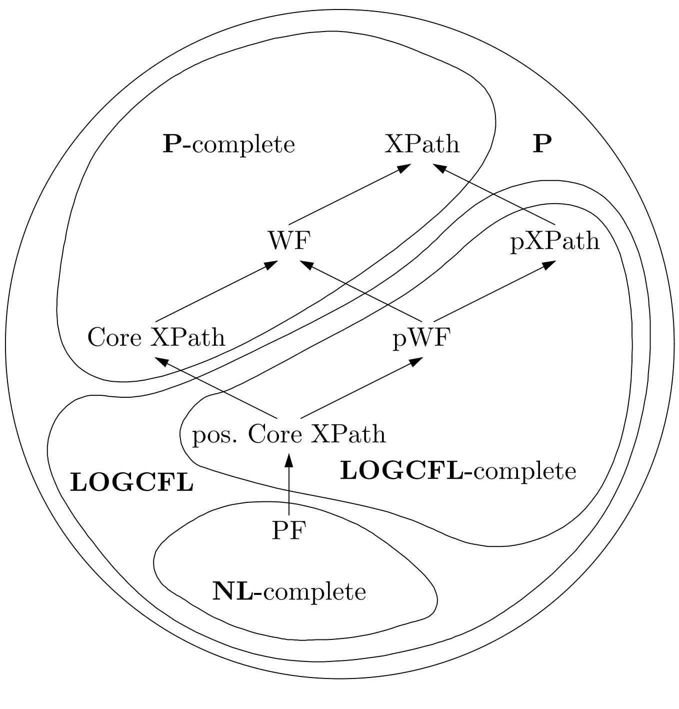


Figure 1: Combined complexity of XPath.

the complexity of evaluating queries is complete for nondeterministic logarithmic space.

- We extend Core XPath by the arithmetics features of XPath, to the so-called *Wadler Fragment* (WF), and show that a large fragment of it, which we call pWF ("positive"/"parallel" WF), is still in **LOGCFL** and can be massively parallelized. The main features excluded from WF to obtain pWF are negation and sequences of condition predicates.
- This leads us to an even larger fragment of XPath, called pXPath, which we believe contains most practical XPath queries and for which query evaluation can be massively parallelized (the combined complexity is still **LOGCFL**-complete).
- Finally, we complement our results on the combined complexity of XPath with a study of data complexity and query complexity. Both problems fall into low (highly parallelizable) complexity classes even in the presence of negation in queries.

The inclusion relationships[2] between fragments discussed and their (combined) complexities are shown in Figure 1. An arrow $\mathcal{L}_1 \rightarrow \mathcal{L}_2$ means that language $\mathcal{L}_1$ is a fragment of language $\mathcal{L}_2$.

2. PRELIMINARIES

2.1 Complexity Classes

We briefly discuss the complexity classes and some of their characterizations used throughout the paper. For more thorough surveys of the related theory see [6, 7].

[2]In the drawing, we assume that $\mathbf{NL} \subset \mathbf{LOGCFL} \subset \mathbf{P}$.

By **P**, **L**, and **NL** we denote the well-known complexity classes of problems solvable in deterministic polynomial time, deterministic logarithmic space, and nondeterministic logarithmic space, respectively, on Turing machines.

It is conjectured that problems complete for **P** are inherently sequential and cannot profit from parallel computation. A problem is instead called *highly parallelizable* if it can be solved within the complexity class **NC** of all problems solvable in polylogarithmic time on a polynomial number of processors working in parallel [5]. By $\mathbf{NC}_i$, we denote the class of problems solvable in time $O(\log^i n)$ using $O(n^i)$ processors (in terms of the size n of the input).

A simple model of parallel computation is that of boolean circuits. By a monotone circuit, we denote a circuit in which only $\wedge$-gates and $\vee$-gates (but no $\neg$-gates) are used. A family of circuits is a sequence $\mathcal{G}_0, \mathcal{G}_1, \mathcal{G}_2, \ldots$, where the n-th circuit $\mathcal{G}_n$ has n inputs. Such a family is **L**-uniform if there exists an **L**-bounded deterministic Turing machine which, on the input of n bits 1, outputs the circuit $\mathcal{G}_n$. A circuit or family of circuits has bounded fan-in if all of its gates have fan-in bounded by some constant. A semi-unbounded circuit is a monotone circuit in which all $\wedge$-gates are of bounded fan-in (w.l.o.g., we may restrict the fan-in to two) but the $\vee$-gates may have unbounded fan-in.

Definition 2.1. $\mathbf{SAC}_1$ is the class of problems solvable by **L**-uniform families of semi-unbounded circuits of depth $O(\log n)$ ($\mathbf{SAC}_1$ circuits). □

LOGCFL is usually defined as the complexity class consisting of all problems **L**-reducible to a context-free language. There are two important alternative characterizations that we are going to use.

Proposition 2.2 ([11]). $\mathbf{LOGCFL} = \mathbf{SAC}_1$. *$\mathbf{SAC}_1$ circuit value is* **LOGCFL***-complete.*

A nondeterministic auxiliary pushdown automaton (NAuxPDA) is a nondeterministic Turing machine with a distinguished input tape, a worktape, an output tape, and a stack (of which strictly only the topmost element can be accessed at any time).

Proposition 2.3 ([9]). **LOGCFL** *is the class of all decision problems solvable by an NAuxPDA with a logarithmic space-bounded worktape in polynomial time.*

Proposition 2.4 ([1]). **LOGCFL** *is closed under complement.*

Regarding the containment of these classes, we know that $\mathbf{NC}_1 \subseteq \mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{LOGCFL} \subseteq \mathbf{NC}_2 \subseteq \mathbf{NC} \subseteq \mathbf{P}$. **P**, **LOGCFL**, and **NL** are closed under **L**-reductions.

2.2 A Brief Introduction to XPath

XPath 1.0 is a language with a large number of features and therefore somewhat unwieldy for theoretical treatment. In this paper, we restrict ourselves to introducing only some of these features, and to giving an informal explanation of their semantics. For a detailed definition of the full XPath language, we refer to [13], and for a concise yet complete formal definition of the XPath semantics see [3].

In this section, we define two basic fragments of XPath. Core XPath, first defined in [3], supports the most commonly used features of XPath, path navigation and conditions with logical connectives, but excludes arithmetics, string manipulations, and some of the more esoteric aspects of the language. The second fragment, which was first discussed in [12] by Wadler, contains XPath's logical and arithmetic features, but excludes string manipulations. We refer to it as the *Wadler Fragment*, short WF.

We start by discussing Core XPath. We sketch the fragment in terms of its syntax and then informally discuss the semantics.

DEFINITION 2.5. The syntax of Core XPath is defined by the grammar

locpath ::= '/' locpath | locpath '/' locpath |
  locpath '|' locpath | locstep.
locstep ::= axis '::' ntst '[' bexpr ']' ... '[' bexpr ']'.
bexpr ::= bexpr '`and`' bexpr | bexpr '`or`' bexpr |
  '`not`(' bexpr ')' | locpath.
axis ::= '`self`' | '`child`' | '`parent`' |
  '`descendant`' | '`descendant-or-self`' |
  '`ancestor`' | '`ancestor-or-self`'
  '`following`' | '`following-sibling`'
  '`preceding`' | '`preceding-sibling`'.

where "locpath" is the start production, "axis" denotes axis relations (see below), and "ntst" denotes tags labeling document nodes or the star '*' that matches all tags ("node tests"). □

The main syntactical feature of Core XPath are *location paths*. Expressions enclosed in brackets are called *conditions* or *predicates*.

The main application of XPath is the navigation in XML document trees. This is done using the *axis relations*, natural binary relations such as "child" and "descendant" between nodes, which we do not define here (but see [13, 3]; they also have the intuitive meanings conveyed by their names). The probably most common use of XPath is to compose axis applications with selections of document nodes by their tags ("node tests"). For instance, the query /descendant::a/child::b selects all those nodes labeled "b" that are children of nodes labeled "a" that are in turn descendants of the root node (denoted by the initial slash).

Conditions enclosed in square brackets allow to impose additional constraints on node selections. For example,

/descendant::a/child::b[descendant::c and
  not(following-sibling::d)]

selects exactly those nodes v from the nodes in the result of /descendant::a/child::b that have *at least one*[3] descendant labeled "c" and do not have a right sibling in the tree that is labeled "d" (i.e., there is no child v' of the parent of v which follows v in the flow of the document and is labeled "d").

DEFINITION 2.6. The syntax of the WF-Queries is defined by the Core XPath grammar with the following extensions. "bexpr" is now

bexpr ::= bexpr '`and`' bexpr | bexpr '`or`' bexpr |
  '`not`(' bexpr ')' | locpath |
  nexpr relop nexpr.

[3]Location paths occurring in conditions – i.e., within square brackets – have an "exists"-semantics, meaning that at least one node must match the location path starting from the current node.

Moreover,

expr ::= locpath | bexpr | nexpr.
nexpr ::= '`position`()' | '`last`()' | number |
  nexpr arithop nexpr.
arithop ::= '+' | '-' | '*' | '`div`' | '`mod`'.
relop ::= '=' | '!=' | '<' | '<=' | '>' | '>='.

"expr" (rather than "locpath") is now the start production and "number" denotes constant real-valued numbers. □

Even though XPath is mainly understood as a language for selecting a subset of the nodes of an XML document tree, query results can also be of different types, namely – for the WF – numbers and booleans (as well as character strings for full XPath). XPath expressions are evaluated relative to a context, which by definition is a triple of a *context node* and two integers, the so-called *context position* and the *context size*. For details, we refer to [13, 3], but consider the example query child::a[position() + 1 = last()]. Relative to a context-triple (v, i, j), i and j are ignored when the location step child::a selects those children of v that are labeled "a". Let $\{w_1, \dots, w_m\}$ be this set of nodes, where the indices correspond to the relative order of the nodes in the document, simply speaking[4]. The application of an axis causes a change of context to which the condition [position() + 1 = last()] is applied. The condition is tried on each of the triples $(w_1, 1, m), \dots, (w_m, m, m)$. It will select all those nodes w_k for which $k + 1 = m$, i.e. the "position" k in the selection is by one smaller than the last index m, the size of the selection.

PROPOSITION 2.7 ([3]). *XPath query evaluation is in* **P** *with respect to combined complexity.*

Core XPath queries can even be evaluated in time $O(|Q| * |D|)$, where $|Q|$ denotes the size of the query and $|D|$ denotes the size of the data.

3. COMPLEXITY OF CORE XPATH

In this section, we show that XPath and even Core XPath are **P**-hard with respect to combined complexity.

REMARK 3.1. In the proof of the following theorem, we often assign several labels to one and the same node, even though each node of an XML document can have only one tag. We assume these labels to be assigned, say, using attributes or by children additionally introduced for this purpose[5]. We add condition expressions of the form $T(l)$ (where l is a label) to Core XPath. For instance, we can write child::*[T(a)] in place of child::a and now realize and verify multiple labels of one and the same node (imagine child::*[T(a) and T(b) and T(c)]).

THEOREM 3.2. *Core XPath is* **P***-complete with respect to combined complexity.*

Proof. Membership of the combined complexity even of full XPath was shown to be in **P** in [3], thus all we need to show is **P**-hardness. This is done by reduction from the *monotone boolean circuit value* problem, which is **P**-complete [7].

[4]To be precise, for some axes this order is reversed, see [13].
[5]$T(l)$ could be a shortcut for child::l.

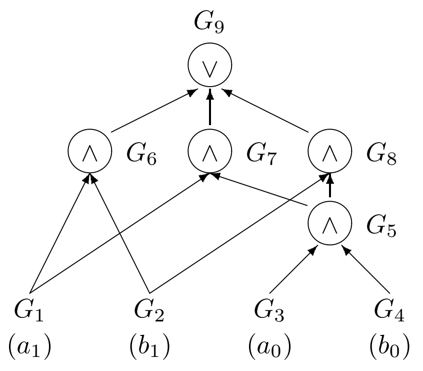
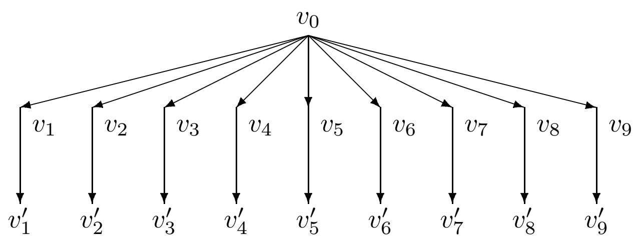


Figure 2: A 2-bit full adder carry-bit circuit.

Given an instance of this problem (a monotone boolean circuit), let M denote the number of input gates and let N denote the number of all other gates in the circuit. Let the gates be named $G_1 \dots G_{M+N}$. Without loss of generality[6], we may assume that the gates $G_1 \dots G_{M+N}$ are numbered in some order such that no gate G_i depends on the output of another gate G_j with $j > i$. In particular, the input gates are named $G_1 \dots G_M$ and the output gate is G_{M+N}.

An example of a circuit with appropriately numbered gates is shown in Figure 2. This circuit computes the carry-bit of a two-bit full-adder, i.e. it tells whether adding the two-bit numbers $a_1 a_0$ and $b_1 b_0$ leads to an overflow. The carry-bit c_1 is computed as $(a_1 \wedge b_1) \vee (a_1 \wedge c_0) \vee (b_1 \wedge c_0)$ where $c_0 = a_0 \wedge b_0$ is the carry-bit of the lower digit (a_0 and b_0).

The **document tree** is of very simple and regular structure; it consists of a root node v_0 with $M + N$ children $v_1 \dots v_{M+N}$, of which each v_i again has exactly one child v'_i (thus, the tree has depth three). For our carry-bit example of Figure 2 with $M = 4$ and $N = 5$, the tree is



Node labels are taken from the alphabet

$$\{0, 1, G, R, I_1, \dots, I_N, O_1, \dots, O_N\}$$

and each tree node is assigned a set of such labels. This is done as follows. The root node v_0 has no labels. The nodes $v_1 \dots v_{M+N}$ are assigned the label G each. (In a way described later, node v_i *represents* the value of gate G_i). Node v_{M+N} is also assigned label R (for "result"). Each node out of $v_1 \dots v_M$ is assigned the truth value at the input gate of the same index (i.e., out of $G_1 \dots G_M$), respectively. This is either the label 0 or 1. Moreover, if the output of gate G_i is an input of gate G_{M+k} (thus, by our gate ordering requirement, $i < M + k$), we add I_k to the labels of v_i and O_k to the labels of v_{M+k}. In our example, the nodes $v_1 \dots v_9$ are labeled

[6]The gates can be "sorted" to adhere to such an ordering in logarithmic space.

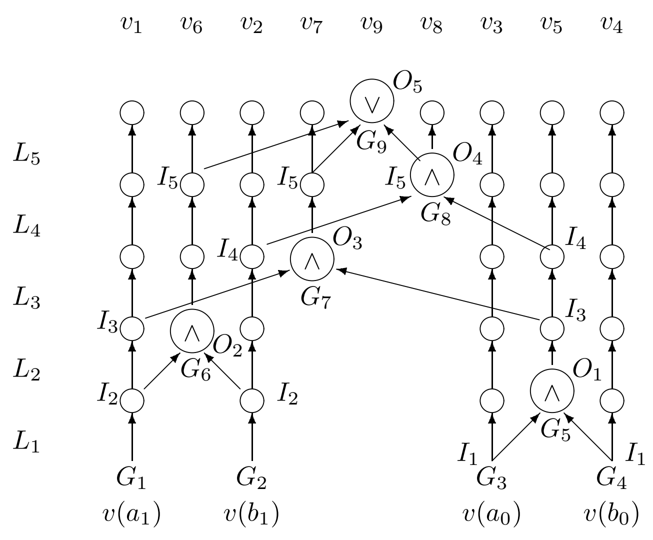


Figure 3: Circuit of Figure 2 with gates serialized.

v_1: $\{G, v(a_1), I_2, I_3\}$	v_2: $\{G, v(b_1), I_2, I_4\}$
v_3: $\{G, v(a_0), I_1\}$	v_4: $\{G, v(b_0), I_1\}$
v_5: $\{G, O_1, I_3, I_4\}$	v_6: $\{G, O_2, I_5\}$
v_7: $\{G, O_3, I_5\}$	v_8: $\{G, O_4, I_5\}$
v_9: $\{G, R, O_5\}$	

where $v(a_1), v(b_1), v(a_0), v(b_0) \in \{0, 1\}$ are the truth values a_1, b_1, a_0, and b_0, respectively, at the input gates. Figure 3 shows how the I_k and O_k labels are assigned to the nodes $v_1 \dots v_{M+N}$. (Figure 3 is explained in more detail below.)

Finally, the nodes $v'_1 \dots v'_M$ are labeled

$$\{I_1, \dots, I_N, O_1, \dots, O_N\}$$

each and the nodes v'_{M+i}, for $1 \le i \le N$, are labeled

$$\{I_k, O_k \mid i \le k \le N\}.$$

The **query** evaluating a circuit uses the intuition of processing one gate out of $G_{M+1} \dots G_{M+N}$ at a time, in the order of ascending index. It is

$$\text{/descendant-or-self::*}[T(R) \text{ and } \varphi_N]$$

with, for $1 \le k \le N$, the condition expressions

$$\varphi_k := \text{descendant-or-self::*}[T(O_k) \text{ and parent::*}[\psi_k]]$$

and

$$\psi_k := \text{not(child::*}[T(I_k) \text{ and not}(\pi_k)])$$

if the type of gate G_{M+k} is "$\wedge$" and

$$\psi_k := \text{child::*}[T(I_k) \text{ and } \pi_k]$$

otherwise, and

$$\pi_k := \text{ancestor-or-self::*}[T(G) \text{ and } \varphi_{k-1}].$$

Moreover, $\varphi_0 := T(1)$.

It is easy to see that the reduction can be effected in logarithmic space. We next argue that it is also correct.

Discussion. We use the ordering of the circuit in that we, intuitively, will evaluate the circuit in Core XPath *one gate at a time*. We treat the circuit as if layered, with all gates of a layer of the same type ("$\wedge$" or "$\vee$") and only exactly one with fan-in greater than one. (Our encoding permits *unbounded fan-in*, including one.) Figure 3 shows this alternative view of the example circuit of Figure 2. The $N = 5$ non-input gates have been aligned using five layers $L_1 \dots L_5$. The smaller empty circles denote "dummy" gates of fan-in one, which are needed to propagate the values of gates that are already available to the layers above. In our encoding, intuitively, all gates of layer L_k have to have the same type. The type of the dummy gates[7] in layer L_k is thus determined by the type of the one gate of fan-in greater than one (namely G_{M+k}). In the example, all gates of layers $L_1 \dots L_4$ are of type $\wedge$ and the gates of layer L_5 are all of type $\vee$.

The φ_k, ψ_k, and π_k all are condition expressions, and there is a natural meaning to "φ_k matches node w" or equivalently "node w satisfies φ_k", which we will denote as $w \in [\![\varphi_k]\!]$ below. Formally, $w \in [\![\varphi_k]\!]$ if and only if query /descendant-or-self::*[φ_k] selects node w. We define $w \in [\![\psi_k]\!]$ and $w \in [\![\pi_k]\!]$ analogously. This notation helps to imagine the query (tree) being processed bottom-up.

Claim. *Let $0 \leq k \leq N$ and $1 \leq i \leq M + k$. Then,*

$$v_i \in [\![\varphi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

This can be shown by an easy induction.

Induction start ($k = 0$). The gates $G_1 \dots G_{M+k}$ are precisely the input gates, which have been assigned their initial value (either 0 or 1) as label. The label 1 is not used elsewhere in the tree. By definition, φ_0 is the expression $T(1)$, so $v_i \in [\![\varphi_0]\!]$ iff the value of the input gate G_i is 1. Thus our claim holds for $k = 0$.

Induction step. Now assume that our claim holds for $k - 1 \geq 0$ (i.e., $v_i \in [\![\varphi_{k-1}]\!]$ iff gate G_i has been established to be true by step $k - 1$). We show that it also holds for k. We proceed by computing first $[\![\pi_k]\!]$, then $[\![\psi_k]\!]$, and finally $[\![\varphi_k]\!]$. One can verify by inspection of π_k that

$$[\![\pi_k]\!] = \{v_i, v'_i \mid 1 \leq i \leq M + k,\ v_i \in [\![\varphi_{k-1}]\!]\}$$

(Note that only the nodes $v_1 \dots v_{M+k}$ are labeled G.)

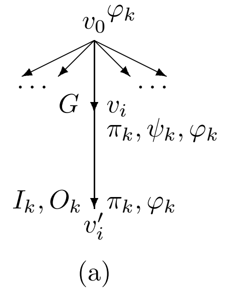
ψ_k is at the heart of our construction and performs the actual computation of the $M + k$ gates at layer k. These gates are the $M + k - 1$ "dummy" gates of fan-in one which just propagate the input and thus make sure that the truth value of gate G_i ($1 \leq i < M + k$), once computed, remains available to layers above, and the gate G_{M+k} of fan-in greater than one.

In our document, node v_i is labeled I_k iff gate G_{M+k} takes input from gate G_i. By the definition of ψ_k, $v_0 \in [\![\psi_k]\!]$ if and only if all children (for G_{M+k} a $\wedge$-gate) or at least one child (for G_{M+k} a $\vee$-gate) that is labeled I_k matches π_k. Thus,

$$v_0 \in [\![\psi_k]\!] \Leftrightarrow G_{M+k} \text{ evaluates to true.}$$

For the dummy gates, the I_k labels are one level deeper down in the tree. (The sole purpose of π_k was to push the previously computed value of G_i – as a matching on node v_i – to two *different* depths to allow for different handling of dummy gates and gate G_{M+k} by the same XPath expression ψ_k.) Node v'_i has the label I_k for each $1 \leq i < M + k$. The parent of v'_i however, v_i, has only one child (namely, v'_i), so

[7]In fact, the types of gates of fan-in one do not matter in the circuit: the conjunction as well as the disjunction of a *single* truth value is the identity. For this reason, and to save space, we do not show the types of the dummy gates in Figure 3.



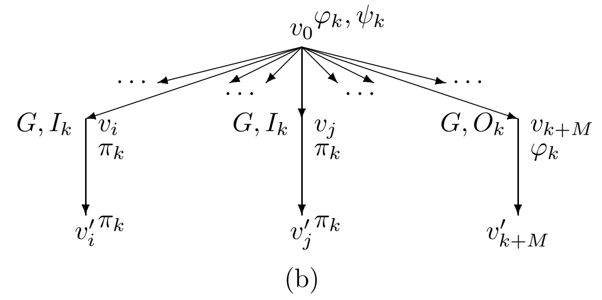


Figure 4: Schematic design of relevant tree region and $\varphi_k/\psi_k/\pi_k$-matchings made for (a) dummy gates and (b) gates of fan-in greater than one (here, two).

it does not matter whether ψ_k is of the $\wedge$- or the $\vee$-type. For $1 \leq i < M + k$,

$$v_i \in [\![\psi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

$[\![\psi_k]\!]$ computes (or preserves, in the case of dummy gates) the truth values of the gates, but the matchings that witness these truth values end up being at different depths in the tree, depending on the gate. φ_k "stores" the matchings at the same depth (the nodes $v_1 \dots v_{M+k}$) for $1 \leq i \leq M + k$:

$$v_i \in [\![\varphi_k]\!] \Leftrightarrow \text{gate } G_i \text{ evaluates to true.}$$

This proves our claim. (φ_k also matches other nodes above and below $v_1 \dots v_{M+k}$, but this does not matter because these nodes are labeled neither G nor R.)

The schematic designs of Figure 4 show the regions of the document tree that we are interested in, the relevant labels, and the matchings of condition expressions φ_k, ψ_k, and π_k at step k, for both cases of gates (dummy gates in Figure 4 (a) and gates G_{M+k} in Figure 4 (b)).

The overall query /descendant-or-self::*[$T(R)$ and φ_N] has a nonempty result exactly if the output gate G_{M+N} of the circuit evaluates to true, because v_{M+N} is the only node labeled R and $v_{M+N} \in [\![\varphi_N]\!]$ if and only if G_{M+N} evaluates to true. $\square$

COROLLARY 3.3. *Core XPath remains* **P***-hard even if*

1. *the document tree is limited to depth three and*
2. *only the axes child, parent, and descendant-or-self are allowed.*

Proof. The previous proof has the stated properties, except that it uses the ancestor-or-self axis in the definition of π_k. All we need to do is to replace ancestor-or-self::* in

π_k by descendant-or-self::*/parent::*. π_k then additionally matches the root node v_0, but this does not matter to the remainder of the construction because v_0 never carries an I_k label and thus never has an impact on ψ_k. □

We overstated the required tree depth in Corollary 3.3 to allow for multiple node labels to be encoded as additional children, as discussed in Remark 3.1. The document trees of the encoding of the proof of Theorem 3.2 are only of depth two.

Note also that the queries used in the encoding essentially do not branch out in terms of axis applications. That is, in each conjunction ("or" is not used) of expressions, there is at most one subexpression that contains an axis application.

4. INSIDE CORE XPATH

The result of the previous section is essentially negative: As Core XPath is **P**-hard, it is considered unlikely that a parallel algorithm exists for evaluating all queries of this language. It is thus natural to search for fragments of Core XPath that we can show to be in **NC** and therefore highly parallelizable.

In fact, such a fragment is obtained by removing negation ("not") from Core XPath. This fragment will be called *positive Core XPath*.

THEOREM 4.1. *The combined complexity of positive Core XPath is in* **LOGCFL**.

We postpone the proof of this theorem to the next section, where we will engineer a strictly and considerably larger **LOGCFL** fragment of XPath (see Theorem 5.5).

As we will see next, the general proof technique used to show Theorem 3.2 can be employed to prove further hardness results of XPath fragments (inside **P**) using circuits. (Recall that the proof of Theorem 3.2 does not require gates to have bounded fan-in.)

THEOREM 4.2. *Positive Core XPath is* **LOGCFL**-*hard w.r.t. combined complexity.*

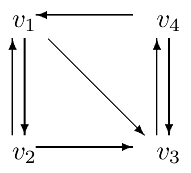
Proof (Sketch). By reduction from SAC$_1$ circuit value, which is **LOGCFL**-complete (see Proposition 2.2). Given a SAC$_1$ circuit, we use the construction of the proof of Theorem 3.2 with the following changes:

- In the document, there are exactly two I_k-labels now for each $\wedge$-layer. We call them I_k^1 and I_k^2. For "dummy"-gates propagating the value of gate G_i, the single "input line", node v_i', is assigned both I_k^1 and I_k^2.
- We construct queries as usual, but instead of negation (which allows to express an unbounded "for all"), we use the XPath language construct "and" with two inputs, one labeled I_k^1 and one I_k^2.

  That is, for gates of type "$\wedge$", ψ_k is replaced by

  $$\psi_k := \text{child::*}[T(I_k^1) \text{ and } \pi_k] \text{ and}$$
  $$\text{child::*}[T(I_k^2) \text{ and } \pi_k]$$

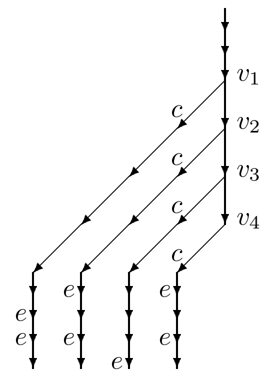
  Thus at every $\wedge$-step of the query, the subexpression of the query needs to be inserted twice. Although the query grows exponentially in the depth of the circuit, it can be computed in **L** because the depth of the circuit (and thus the size of subexpressions to be copied) is only logarithmic.



(a)

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

(b)



(c)

Figure 5: Graph (a), its (transposed) adjacency matrix (b), and tree of the encoding (c).

All else of the reduction remains the same. □

Let PF be the fragment of Core XPath containing only the location paths, without conditions (i.e., no expressions enclosed in brackets are permitted).

THEOREM 4.3. *With respect to combined complexity, PF is* **NL**-*complete under* **L**-*reductions.*

Proof (Sketch). Membership in **NL** is obvious: we can just guess the path while we verify it in **L**. **NL**-hardness follows from a **L**-reduction from the *directed graph reachability* problem, which is **NL**-complete (cf. [7]). The reduction is quite simple, so we just provide an example (see Figure 5). Let $G = (V, E)$ be the directed input graph. Assume we look for a path from node v_i to node v_j. We abbreviate the n-times repeated application of an axis χ as χ^n::*. By χ^n::c, we denote $(\chi\text{::*}/)^{n-1}\chi$::c. As is easy to verify, given a positive integer m, the query /descendant::v_i/φ_m with

$$\varphi_k := \text{child::}c/\text{descendant::}e/\text{parent}^{2*|V|}\text{::*}/\text{child}^{|V|}\text{::}c/$$
$$\text{parent::*}/\varphi_{k-1}$$

and $\varphi_0 :=$ self::v_j computes the node labeled v_j if and only if node v_j is reachable from node v_i in m steps. To extend this to reachability, we add a loop for each node of the graph (or equivalently, set the main diagonal of the adjacency matrix to ones only) and have $m := |E|$. □

5. PARALLELIZING WF

We are now going to search for restrictions on WF that push down the complexity of the query evaluation problem to the highly parallelizable complexity class **LOGCFL**.

To achieve this, we require that scalar values (i.e., values different from node sets) can be stored in logarithmic space. Moreover, we also have to exclude two important constructs from WF, namely iterated predicates of the form $\chi :: t[e_1] \dots [e_k]$ with $k \geq 2$ and the not-function. The resulting XPath fragment will be referred to as the "positive" (or "parallel") WF (short pWF). It is formally defined as follows:

DEFINITION 5.1. pWF is obtained by restricting WF in the following way:

1. Expressions of the form $\chi :: t[e_1] \dots [e_k]$ with $k \geq 2$ are not allowed, where χ denotes an axis, t is a node test and the e_i's are XPath expressions.

2. The not-function may not be used.

3. The nesting depth of arithmetic operators is bounded by some constant k. □

The first two restrictions above mean that the grammar from Definitions 2.5 and 2.6 has to be modified as follows:

locstep ::= axis '`::`' ntst '[' bexpr ']'
bexpr ::= bexpr '`and`' bexpr | bexpr '`or`' bexpr |
| locpath | nexpr relop nexpr.

REMARK 5.2. Note that positive Core XPath is strictly a fragment of pWF. This is due to the fact that the first restriction above plays no role in Core XPath. More generally, an XPath expression of the form $\chi :: t[e_1] \dots [e_k]$ is equivalent to $\chi :: t[e_1$ and ... and $e_k]$ as long as position() and last() are not used.

The classical decision problem regarding query evaluation (also known as the SUCCESS problem) is, given a database, a query, and a query result, to decide whether the given query result is correct for the given query on the input database. In our context, the XML document takes the place of the database, the XPath query in conjunction with a context-triple the place of the query, and finally, a value that is either of type boolean, number, string, or node set assumes the place of the query result to be checked.

For our purposes, it is convenient to work with the following slightly different decision problem.

DEFINITION 5.3 (SINGLETON-SUCCESS).
Input: A tuple $(D, Q, \vec{c}, v)$, where D is an XML document, Q an XPath query, $\vec{c}$ a context-triple, and v a value. If Q is of type number or string, then v is a value of this type. If Q is of type boolean, then v is the value true. Finally, if Q is of type node set, then v is a single node.
Question: Does query Q on document D and context $\vec{c}$ evaluate to v (in case of result type number, string or boolean) or does it evaluate to some node set X with $v \in X$? □

LEMMA 5.4. *The* SINGLETON-SUCCESS *problem for pWF can be decided in* **LOGCFL**.

Proof (Sketch). We describe a NAuxPDA that decides the SINGLETON-SUCCESS problem for queries from pWF. The **LOGCFL**-membership of this problem will follow immediately from the correctness and the fact that the NAuxPDA runs simultaneously with a logarithmic space-bounded worktape and in polynomial time.

Notation. Let an instance of the SINGLETON-SUCCESS problem be given through some XML document D, XPath query Q, context-triple $\vec{c}$ and value v. By $\mathcal{T}_Q$, we denote the parse tree of Q. The root node of $\mathcal{T}_Q$ will be denoted by R. Recall that every node N in $\mathcal{T}_Q$ corresponds to a subexpression of Q, which we shall denote by $expr(N)$. Finally, we write K to denote the maximum number of child nodes of the nodes in $\mathcal{T}_Q$. Actually, for pWF, $K = 2$ holds.

Basics of the NAuxPDA. The principal idea of the NAuxPDA is to traverse $\mathcal{T}_Q$ along its edges in depth-first, left-to-right order. Along this traversal, we basically pass through every node N in the query tree $\mathcal{T}_Q$ at most once in downward direction and at most K times in upward direction. If we visit a node N in downward direction, then we make some guesses (namely, a context $\vec{c}$ and the corresponding result of evaluating the subexpression $expr(N)$ of Q on the document D for this context $\vec{c}$). If we process a node N in upward direction and no further child node of N has to be processed (i.e., either, they have all been processed or the result value of $expr(N)$ is fully determined by those which have already been processed), then we carry out certain consistency checks between the guesses at the current node and at its child nodes. Similarly, if we reach a leaf node N, then we have to check whether the context and the result value guessed at N are consistent with the expression $expr(N)$.

Our traversal of the query tree $\mathcal{T}_Q$ starts at the root R of $\mathcal{T}_Q$ in downward direction. Eventually, we shall have visited all nodes of $\mathcal{T}_Q$ that are required to determine the overall result and we shall come back to the root R of the query tree. If all the consistency checks thus carried out were successful, then the overall result of our computation is "success". As soon as one such check fails, we halt with the overall result "failure".

Worktape and stack of the NAuxPDA. On our worktape, we maintain two integers *CurrN* and *AuxN* as well as a variable *Dir* which can have one of the values "down" or "up". Moreover, our worktape contains the $K + 2$ main data structures *CurrVal*, *AuxVal*, and *ChildVal*[i] with $i \in \{1, \dots, K\}$, each consisting of four components *cnode*, *cpos*, *csize*, and *res*, which stand for context-node, context-position, context-size, and result, respectively.

The variables *CurrN* and *AuxN* hold node-ids of nodes in the query tree. *CurrN* denotes the current node in the query tree and *AuxN* is an auxiliary variable. *Dir* denotes the direction of the last move (i.e., either downward or upward) of our traversal of $\mathcal{T}_Q$. *CurrVal* and *ChildVal*[i] with $i \in \{1, \dots, K\}$ contain the values of *cnode*, *cpos*, *csize*, and *res* that were guessed when processing the current node in the query tree and the i-th child of this node, respectively. *AuxVal* is used as an auxiliary variable for copying purposes. All of these data structures are initially set to the value "*undef*". The values of *CurrVal*, *ChildVal*[1], ..., *ChildVal*[K], and *CurrN* are pushed onto the stack when a node is left in downward direction. Conversely, these values are popped from the stack before a node is entered in upward direction. Consequently, whenever we start to process a node N, then the stack contains the values *CurrVal*, *ChildVal*[1], ..., *ChildVal*[K], and *CurrN* for all nodes along the path from R to the parent of N.

Initialization and main procedure. In the beginning, we select the root node R of $\mathcal{T}_Q$ as the current node *CurrN* and fill in the context and result value from the input into the variable *CurrVal* (rather than guessing these values, as we shall do for all further nodes). All the other data structures *ChildVal*[i] are initialized to "*undef*". If R is a leaf node (i.e., $\mathcal{T}_Q$ consists of the root R only), then we check the consistency of the context and result value in *CurrVal* with the XPath expression $expr(R)$. If this check is successful, then the overall result of the NAuxPDA is "success", otherwise the NAuxPDA halts with "failure". On the other hand, if R is not a leaf node, then we have to move downward in the query tree. If the XPath expression $expr(R)$ is of the form "e_1 or e_2" or of the form "$\pi_1 \mid \pi_2$", then we choose nondeter-

expressions $expr(CurrN)$ at leaf nodes of the query tree $\mathcal{T}_Q$	
$expr(CurrN)$	local consistency condition
$\chi :: t$	r can be reached from n via $\chi :: t$
position()	$r = p$
last()	$r = s$
c (= constant number)	$r = c$

expressions $expr(CurrN)$ at internal nodes of the query tree $\mathcal{T}_Q$	
$expr(CurrN)$	local consistency condition
$/\pi$	$n = \text{root} \wedge r = r_1$
$\pi_1 \mid \pi_2$	$(n = n_1 \wedge r = r_1) \vee (n = n_2 \wedge r = r_2)$
π_1/π_2	$(n = n_1 \wedge n_2 = r_1 \wedge r = r_2)$
$\chi :: t[e]$ (first child of $CurrN$ corresponds to e, the second one to $\chi :: t$)	let $Y = \{y \in \text{dom} \mid y$ can be reached from n via $\chi :: t\}$ $r \in Y \wedge$ (let p_{new} = position of r in Y, let $s_{new} = \lvert Y \rvert$ $n_1 = r \wedge p_1 = p_{new} \wedge s_1 = s_{new} \wedge r = \text{true})$
boolean(π)	$r = \text{true} \wedge (n_1 = n \wedge p_1 = p \wedge s_1 = s \wedge r_1 \in \text{dom})$
e_1 and e_2	$r = \text{true} \wedge [(n_1 = n \wedge p_1 = p \wedge s_1 = s \wedge r_1 = \text{true}) \wedge (n_2 = n \wedge p_2 = p \wedge s_2 = s \wedge r_2 = \text{true})]$
e_1 or e_2	$r = \text{true} \wedge [(n_1 = n \wedge p_1 = p \wedge s_1 = s \wedge r_1 = \text{true}) \vee (n_2 = n \wedge p_2 = p \wedge s_2 = s \wedge r_2 = \text{true})]$
e_1 RelOp e_2 (both e_1 and e_2 are numbers)	$r = \text{true} \wedge r_1 \text{ RelOp } r_2 \wedge [(n_1 = n \wedge p_1 = p \wedge s_1 = s) \wedge (n_2 = n \wedge p_2 = p \wedge s_2 = s)]$
e_1 ArithOp e_2 (both e_1 and e_2 are numbers)	$r = r_1 \text{ ArithOp } r_2 \wedge [(n_1 = n \wedge p_1 = p \wedge s_1 = s) \wedge (n_2 = n \wedge p_2 = p \wedge s_2 = s)]$

legend:				
n, p, s, r:	*CurrVal.cnode*	*CurrVal.cpos*	*CurrVal.csize*	*CurrVal.res*
n_1, p_1, s_1, r_1:	*ChildVal*[1]*.cnode*	*ChildVal*[1]*.cpos*	*ChildVal*[1]*.csize*	*ChildVal*[1]*.res*
n_2, p_2, s_2, r_2:	*ChildVal*[2]*.cnode*	*ChildVal*[2]*.cpos*	*ChildVal*[2]*.csize*	*ChildVal*[2]*.res*

Table 1: Local consistency checks for pWF.

ministically a single child of R (and ignore the whole subtree of $\mathcal{T}_Q$ rooted at the other child node of R). Otherwise we move on to the first child of R. In either case, the current values of *CurrN*, *CurrVal*, *ChildVal*[1], ..., *ChildVal*[K] are pushed onto the stack and *CurrN* is assigned the node-id of the element node to be visited next.

Processing a node in downward direction. If a node is entered in downward direction, then we guess the components of *CurrVal*. After that we basically proceed like in the main procedure explained above. In particular, the data structures *ChildVal*[i] are initialized to "*undef*". Moreover, if the current node is not a leaf node, then we make the same downward move as in the main procedure. Otherwise, if the current node is a leaf node, then we carry out the same consistency check as before. In case of a negative result of this check, we again halt with "failure". However, in case of a successful check, we are of course not yet allowed to halt with success. Instead, we move upward in the query tree. For this upward move, we save the current value of *CurrN* and *CurrVal* to the auxiliary variables *AuxN* and *AuxVal*, respectively. Then we pop *CurrN*, *CurrVal*, *ChildVal*[1], ..., *ChildVal*[K] from the stack and finally assign *AuxVal* to the appropriate variable *ChildVal*[1], ..., *ChildVal*[K], i.e., if the upward move started at the i-th child of its parent, then *AuxVal* is assigned to *ChildVal*[i].

Processing a node in upward direction. Now suppose that we have entered the current node by an upward move from its i-th child. If the current node contains a child that has to be processed yet, then we move on to this child by a downward move. Of course, prior to this move, the variables *CurrN*, *CurrVal*, *ChildVal*[1], ..., *ChildVal*[K] have to be pushed onto the stack. Otherwise, if there are no more child nodes left to be processed, then the consistency of the values *CurrVal* and *ChildVal*[1], ..., *ChildVal*[K] with the expression $expr(CurrN)$ has to be checked. If this check is successful, then we make the same kind of upward move as in case of a leaf node that is processed in downward direction. Otherwise, we halt with "failure".

Consistency checks for the current node. If a leaf node in the query tree has been reached, then we have to check whether the chosen combination of the components of *CurrVal* is indeed allowed for the subexpression $expr(CurrN)$. Similarly, if the context and result value have already been determined for a non-leaf node plus the required child nodes, then we have to check whether the nondeterministic choices were consistent with the subexpression $expr(CurrN)$. All possible kinds of checks thus required are given in Table 1, where we use the following notation: We write $\chi :: t$ for a location step consisting of an axis χ and a node test t. e stands for any XPath expression while π stands for a location path. By RelOp and ArithOp we denote relational operators ($=$, $\neq$, $\leq$, ...) and arithmetic operators ($+, -, *, \ldots$), respectively. The set of all element nodes in D is denoted by dom and root denotes the conceptual root node in the XPath data model (cf. [13]). Moreover, we assume w.l.o.g., that type conversions from node sets to boolean values are made explicit via the XPath-function boolean. Finally, we do not treat the case of undefined components separately. In general, we assume that conditions on undefined values yield the result "*undef*". But of course, we assume that true $\vee$ "*undef*" $\equiv$ true holds.

Discussion. As for the correctness of this NAuxPDA, it has to be shown that an instance $(D, Q, \langle cn, cp, cs\rangle, v)$ of the SINGLETON-SUCCESS problem of pWF yields the answer "yes" iff there exists a run of the NAuxPDA that halts with success. A detailed proof of this is provided in the full paper. It remains to be shown that the NAuxPDA works simultaneously in **L** and **P**. As for the time complexity, recall that the NAuxPDA traverses (parts of) the query tree $\mathcal{T}_Q$ in depth-first, left-to-right order. Along this traversal, every node N is processed at most once in downward direction and at most K times (with $K = 2$ in case of pWF) in upward direction. Moreover, the actions required to process a node once can be clearly done in polynomial time. As for the space complexity, note that the variables of the worktape plus a fixed number of counters and auxiliary variables clearly fit into logarithmic space. The crucial observation for the logarithmic space complexity is that we never have to explicitly compute node sets, e.g., checking $r \in Y$ and determining the position of r in Y and the size of Y can be done without explicitly computing the node set Y itself (cf. the consistency check for $\chi :: t[e]$ in Table 1). ◻

THEOREM 5.5. *pWF is in* **LOGCFL** *with respect to combined complexity.*

Proof (Sketch). The NAuxPDA of the previous proof nondeterministically guesses and verifies a result, or in the case of queries returning a node set, a single node of the result. Checking whether a given XPath query evaluates to some node set X (or equivalently, computing that node set) can be done by deciding the SINGLETON-SUCCESS problem in a loop over all elements $v \in X$ without a significant increase of the overall complexity. In our definition of the SINGLETON-SUCCESS problem, we assumed the technical restriction that results of boolean XPath queries can be only checked to be true. Checking whether a given XPath query with boolean result value evaluates to false is the co-problem of checking whether a query evaluates to true. However, by Proposition 2.4, **LOGCFL** is closed under complementation. ◻

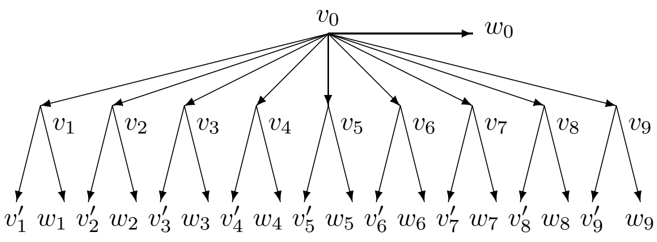
REMARK 5.6. It is well-known that the complexity class **LOGCFL** is inside the class $\mathbf{NC}_2$ of problems solvable in time $O(\log^2 n)$ with quadratically many processors working in parallel. In fact, given this intuition and the insight obtained from the reduction to NAuxPDA, it is not hard to find a highly parallel algorithm for evaluating pWF queries. The intuition for matching straight-line path queries (cf. our PF fragment from Section 4) is similar to parallel algorithms for graph reachability (cf. [7]); however, rather than connecting nodes in a graph, the goal is to connect contexts with nodes in the query result. Additional synchronization is required for branches in the query tree (e.g. "and"), which is not surprising as graph reachability is in **NL** and thus presumably simpler than a **LOGCFL**-complete problem. However, at the branches, the subexpressions below can be evaluated in parallel before finalizing the branch (i.e., proceeding bottom-up). ◻

The next result shows that pWF is in a sense a maximal **LOGCFL** fragment of WF. Of course, we are not really interested in dealing with arbitrarily big number expressions. As far as the other two restrictions in Definition 5.1 are concerned, none of them can be simply omitted. This is clear for negation, as Core XPath is strictly a fragment of pWF extended by negation. As shown next, the final restriction is essential as well. By iterated predicates, we again refer to location steps of the form $\chi :: t[e_1] \ldots [e_k]$ with $k \geq 2$.

THEOREM 5.7. *The combined complexity of pWF queries extended by iterated predicates is* **P***-complete.*

Proof (Sketch). The proof goes by an appropriate modification of the XML document D and the location paths φ, ψ, and π from the proof of Theorem 3.2.

XML document. We extend D by adding one additional child w_i to every node v_i with $i \in \{0, \ldots, M + N\}$ (as the right-most child, say). Each node w_i is labeled W. Hence, the condition $T(W)$ is fulfilled exactly by these new nodes. Moreover, for the node v_0, we introduce an additional label A ("auxiliary"). Thus, the condition $T(A)$ is only fulfilled by the root node v_0. The new document tree corresponding to the example in the proof of Theorem 3.2 looks as follows:


v_0
w_0
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9
v'_1 w_1 v'_2 w_2 v'_3 w_3 v'_4 w_4 v'_5 w_5 v'_6 w_6 v'_7 w_7 v'_8 w_8 v'_9 w_9

XPath query. The desired query Q' (encoding the value of the gate G_{M+N}) is

$$/\text{descendant-or-self::*}[T(R) \text{ and } \varphi'_N]$$

where the auxiliary location paths φ'_k, ψ'_k, and π'_k with $1 \leq k \leq N$ are defined as follows:

$$\varphi'_k := \text{descendant-or-self::*}[T(O_k) \text{ and parent::*}[\psi'_k]]$$

and

$$\psi'_k := \text{child::*}[(T(I_k) \text{ and } \pi'_k[\text{last()=1}]) \text{ or } T(W)][\text{last()=1}],$$

if the type of gate G_{M+k} is "$\wedge$" and

$$\psi'_k := \text{child::*}[T(I_k) \text{ and } \pi'_k[\text{last()} > 1]]$$

otherwise, and

$$\pi'_k := \text{ancestor-or-self::*}[(T(G) \text{ and } \varphi'_{k-1}) \text{ or } T(A)].$$

Moreover, we set $\varphi'_0 := T(1)$ as in Theorem 3.2.

In order to prove the correctness of this problem reduction, we introduce the following notion: Let ρ and σ be XPath queries that are evaluated on the document D or on D', respectively. Then we call ρ and σ equivalent on a node x (that occurs both in D and D'), iff the evaluation of boolean(ρ) on D and the evaluation of boolean(σ) on D' for the context-node x yield the same result. Then the following equivalences hold for any $k \geq 1$ (in case of ψ_k and π_k) and for any $k \geq 0$ (in case of φ_k):

(1) φ_k and φ'_k are equivalent on $v_1, \ldots, v_{M+N}$.
(2) ψ_k and ψ'_k are equivalent on $v_0, \ldots, v_{M+N}$.
(3) π_k and $\pi'_k[\text{last()} > 1]$ as well as not(π_k) and $\pi'_k[\text{last()} = 1]$ are equivalent on $v_1, \ldots, v_{M+N}, v'_1, \ldots, v'_{M+N}$.

These equivalences can be shown by an easy induction argument. We only discuss (3) here, in order to point out the central idea of "encoding" the not-function by iterated predicates: By the induction hypothesis, φ_{k-1} and φ'_{k-1} are equivalent on $v_1, \ldots, v_{M+N}$, that is, on the nodes for which the condition $T(G)$ is true. By the new disjunct $T(A)$ in the predicate of π'_k, the location path π'_k evaluates to the same set as π_k plus the node v_0. Moreover, by the condition $T(G)$, the node set resulting from π_k never contains the node v_0. The equivalence of π_k with $\pi'_k[\text{last}() > 1]$ and the equivalence of $\text{not}(\pi_k)$ with $\pi'_k[\text{last}() = 1]$ (on $v_1, \ldots, v_{M+N}, v'_1, \ldots, v'_{M+N}$) are thus obvious.

The correctness of the whole problem reduction follows immediately from the equivalence (1) for $k = N$. □

Note that in the above proof of Theorem 5.7, we only made use of predicate sequences $[e_1] \ldots [e_k]$ whose length k was bounded by 2. We thus have:

Corollary 5.8. *The combined complexity of pWF extended by iterated predicates of the form $\chi :: t[e_1][e_2]$ is* **P***-complete.*

Despite the negative results of Theorems 3.2 and 5.7, one possible direction into which pWF can be extended without leaving the complexity class **LOGCFL** is to bound the depth of negation, i.e., the maximum depth of nested occurrences of the not-function in queries.

Theorem 5.9. *The combined complexity of pWF queries augmented by negation with bounded depth is in* **LOGCFL***.*

Proof (Sketch). We modify the NAuxPDA encoding in the proof of Lemma 5.4 as follows: First, we transform the input query Q by means of de Morgan's laws in such a way that all occurrences of the not-function are either shifted immediately in front of relational operators RelOp or location paths π. Expressions of the form e_1 RelOp e_2 where both operands are numbers can be replaced by e_1 not(RelOp) e_2. In other words, $=$ is replaced by $\neq$, $<$ is replaced by $\geq$, etc. We thus get an equivalent query Q' where the "not" only occurs in the form $\text{not}(\pi)$. Then we have to modify our NAuxPDA in such a way, that it treats subexpressions of the form $\text{not}(\pi)$ by checking in a loop over all element nodes x in D whether x is in the node set resulting from the evaluation of π for the context-node *CurrVal.cnode*. This check is done by calling the NAuxPDA recursively. The correctness of this algorithm follows immediately from the correctness of the NAuxPDA in the proof of Lemma 5.4. The space complexity clearly does not significantly increase compared to the NAuxPDA in Lemma 5.4. Moreover, by the existence of a constant bound K on the nesting of negation (and, hence, by the same bound K on the nesting of the loops in the above described modified version of our NAuxPDA), the polynomial time upper bound on this computation is also preserved. □

6. PARALLELIZING XPATH

In Section 5, we proved the **LOGCFL**-membership for a fragment of XPath that was derived from WF by imposing some restrictions. In fact, we can get a much larger **LOGCFL** fragment of XPath by starting from full XPath and defining the analogous restrictions. The important fact is again that the evaluation can be done without the need to ever compute node sets explicitly and without having to deal with scalars (i.e., values different from node sets) that do not fit into **L**. Analogously to pWF, we thus define:

Definition 6.1. *Positive (or parallel) XPath* (pXPath) is obtained by imposing the following restrictions on XPath:

1. Expressions of the form $\chi :: t[e_1] \ldots [e_k]$ with $k \geq 2$ are not allowed.
2. The following functions may not be used: not, count, sum, string, and number as well as the string functions local-name, namespace-uri, name, string-length, and normalize-space.
3. Constructs of the form e_1 RelOp e_2 where at least one of the expressions e_i is of type boolean, are forbidden.
4. The depth of nesting of arithmetic operators and of the concat-function is bounded by some given constant K. Likewise, the arity of the concat-function is bounded by K. □

The above restrictions extend the ones in Definition 5.1 in the following way: The evaluation of expressions of the form count(e) and sum(e) requires the explicit computation of the node set value of e unless we again introduce loops over dom into the NAuxPDA as in Theorem 5.9. With the functions string and number as well as the string functions listed above, we would have to manipulate items of information in the document D whose size is not necessarily logarithmically bounded. Moreover, the functions string and number could also be used to "encode" negation, e.g., number(e) = 0 for a boolean expression e evaluates to true, iff e evaluates to false. Similarly, constructs of the form e_1 RelOp e_2 where at least one of the expressions e_i is of type boolean are forbidden since they can also be used to "encode" negation, e.g., by an expression of the form $e \neq \text{true}()$.

Analogously to Theorem 5.5, we have

Theorem 6.2. *The combined complexity of pXPath is in* **LOGCFL***.*

Proof (Sketch). The **LOGCFL**-membership of the problem Singleton-Success and (as a consequence, see the proof of Theorem 5.5) the combined complexity of pXPath can be established by almost the same NAuxPDA as in Section 5. The only adaptation required is an extension of Table 1. In principle, for each of the additionally allowed XPath constructs, a new line with the corresponding local consistency check has to be added. Alternatively, we can cover these new lines via the "effective semantics function" $\mathcal{F}$ of XPath operators Op that was introduced in [3] for all XPath constructs except for location paths and the functions position() and last() (whose semantics was defined separately in [3]). Then the consistency check for an expression of the form $Op(e_1, \ldots, e_l)$ comes down to the condition

$$(\bigwedge_{i=1}^{l} (n_i = n \wedge p_i = p \wedge s_i = s)) \wedge r = \mathcal{F}[\![Op]\!](r_1, \ldots, r_l)$$

Moreover, in case of a boolean expression e, we add the conjunct $r = \text{true}$. Note that the last four lines in Table 1 are (equivalent formulations of) special cases of this principle with $\mathcal{F}[\![\text{and}]\!] = \wedge$, $\mathcal{F}[\![\text{or}]\!] = \vee$, $\mathcal{F}[\![\text{RelOp}]\!] = \text{RelOp}$, and $\mathcal{F}[\![\text{ArithOp}]\!] = \text{ArithOp}$, respectively. Of course, the polynomial time and logarithmic space upper bound on the complexity also holds for the thus extended NAuxPDA. □

Finally, we mention that also pXPath can be extended by negation with bounded depth without destroying the

LOGCFL-membership. The following result is stated without proof.

THEOREM 6.3. *The combined complexity of pXPath augmented by negation with bounded depth is in* **LOGCFL**.

Conceptually, this can be shown exactly like Theorem 5.9. However, there are now quite a few new constructs which have to be considered separately since negation cannot be shifted inside them, e.g., not(e_1 RelOp e_2) where at least one of the operands e_i is a node set has to be treated in a loop over all nodes $x \in$ dom just like expressions of the form not(π) in Theorem 5.9.

7. QUERY AND DATA COMPLEXITY

In this paper, we have addressed the combined complexity of various fragments of XPath. While the general problem is **P**-hard, we have engineered large fragments that can be massively parallelized. We conclude this treatment with an outlook towards the two main other complexity measures, the complexity of queries when either the size of the query or of the data is fixed.

THEOREM 7.1. *PF is* **L**-*hard under* $\mathbf{NC}_1$-*reductions (with respect to data complexity).*

Proof. Given a tree in which all nodes have a unique label, the query /descendant-or-self::v_1/descendant::v_2 from our path expressions fragment PF (see Section 4) selects a node if and only if v_2 is reachable from v_1 in the tree. This is *directed tree reachability*, which is **L**-complete under $\mathbf{NC}_1$-reductions [2]. The query is constant and can be assumed to work on the same tree as the directed tree reachability problem. The result follows. ☐

THEOREM 7.2. *XPath is in* **L** *w.r.t. data complexity.*

Proof (Sketch). The basic idea for an XPath evaluation algorithm that runs in **L** is motivated by the bottom-up dynamic programming algorithm for full XPath of [3], which was based on the notion of so-called *context-value tables*, relations consisting of tuples containing a context and a corresponding value for (a subexpression of) the given query, one tuple for each meaningful context. We compute one such context-value table for each node of the query tree. Given the context-value tables for the direct subexpressions $e_1, \ldots, e_n$, computing the context-value table of expression $Op(e_1, \ldots, e_n)$, where Op is an atomic XPath operation (a node in the query tree), only requires a very simple computational task which can be carried out in **L**. Since we consider data complexity, the query and the number of operations in its query tree is assumed fixed. We can compose a fixed number of steps that individually run in **L** into an algorithm that runs in **L** overall. ☐

THEOREM 7.3. *XPath without multiplication or the "concat" operation is in* **L** *w.r.t. query complexity.*

Proof (Sketch). Let Q be the input query and D the (fixed) document. All operations in Q have a fixed arity not greater than $K = 3$.

Let us first assume that Q does not contain operations such as + that make strings or numbers grow (logarithmically) with the size of the query. Then, it is known from [3] that the size of each context-value table is bounded by the *constant* $|D|^4$. To compute the context-value table holding the result of Q on D, we simply have to make a bottom-up traversal of the query tree of Q, which can be performed in **L**. Regarding storage requirements, only a stack bounded by $K * \log |Q|$ context-value tables is needed, which holds context-value tables computed bottom-up but not used yet and waiting to be employed for the computation of context-value tables higher up in the query tree. (Note that this is *not* the depth of the query tree, which is not necessarily bounded by $O(\log Q)$.)

Computing context-value tables bottom-up step by step is important for handling path expressions and negation well. For string- or number-typed expressions e, these relations do not have to be materialized, but results can be generated and checked top-down when computing a node set-typed context-value table for an expression that contains e as a direct subexpression with only an additional $|D| * \log |Q|$-sized memory window. ☐

We did not provide a lower bound for the query complexity of XPath, but conjecture a considerable fragment of XPath to be **ALOGTIME**-complete with respect to query complexity.

Acknowledgments

This work was supported by the EU Research Training Network GAMES. The first author was partially funded by grant Z29-N04 of the Austrian Research Fund (FWF). The second author's visit to the University of Edinburgh's Laboratory for Foundations of Computer Science was sponsored by Erwin Schrödinger grant J2169 of the FWF.

The paper [8], also to be found in this proceedings volume, contains a number of related and overlapping results on the complexity of XPath, and is work independent from ours.

8. REFERENCES

[1] A. Borodin, S. A. Cook, P. W. Dymond, W. L. Ruzzo, and M. Tompa. "Two Applications of Inductive Counting for Complementation Problems". *SIAM Journal of Computing*, **18**:559–578, 1989.

[2] S. A. Cook and P. McKenzie. "Problems Complete for Deterministic Logarithmic Space". *J. Algorithms*, **8**:385–394, 1987.

[3] G. Gottlob, C. Koch, and R. Pichler. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, Aug. 2002.

[4] G. Gottlob, C. Koch, and R. Pichler. "XPath Query Evaluation: Improving Time and Space Efficiency". In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03)*, pages 379–390, Bangalore, India, Mar. 2003.

[5] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

[6] D. S. Johnson. "A Catalog of Complexity Classes". In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, chapter 2, pages 67–161. Elsevier Science Publishers B.V., 1990.

[7] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[8] L. Segoufin. “Typing and Querying XML Documents: Some Complexity Bounds”. In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Diego, CA, 2003.

[9] I. Sudborough. “Time and Tape Bounded Auxiliary Pushdown Automata”. In *Mathematical Foundations of Computer Science (MFCS’77)*, pages 493–503. Springer Verlag, LNCS 53, 1977.

[10] M. Y. Vardi. “The Complexity of Relational Query Languages”. In *Proc. 14th Annual ACM Symposium on Theory of Computing (STOC’82)*, pages 137–146, San Francisco, CA USA, May 1982.

[11] H. Venkateswaran. “Properties that Characterize LOGCFL”. *Journal of Computer and System Sciences*, **43**:380–404, 1991.

[12] P. Wadler. “Two Semantics for XPath”, 2000. Draft paper available at http://www.research.avayalabs.com/user/wadler/.

[13] World Wide Web Consortium. XML Path Language (XPath) Recommendation., Nov. 1999. http://www.w3c.org/TR/xpath/.

[14] World Wide Web Consortium. “XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002), 2002. http://www.w3.org/TR/query-algebra/.