Algorithmica (2015) 71:661–701
DOI 10.1007/s00453-014-9944-y

On the Space and Circuit Complexity of Parameterized Problems: Classes and Completeness

Michael Elberfeld · Christoph Stockhusen · Till Tantau

Received: 29 November 2013 / Accepted: 29 September 2014 / Published online: 16 October 2014
© Springer Science+Business Media New York 2014

Abstract The parameterized complexity of a problem is generally considered "settled" once it has been shown to be fixed-parameter tractable or to be complete for a class in a parameterized hierarchy such as the weft hierarchy. Several natural parameterized problems have, however, resisted such a classification. In the present paper we argue that, in some cases, this is due to the fact that the parameterized complexity of these problems can be better understood in terms of their *parameterized space* or *parameterized circuit* complexity. This includes well-studied, natural problems like the feedback vertex set problem, the associative generability problem, or the longest common subsequence problem. We show that these problems lie in and may even be complete for different parameterized space classes, leading to new insights into the problems' complexity. The classes we study are defined in terms of different forms of bounded nondeterminism and simultaneous time–space bounds.

Keywords Parameterized complexity · Space complexity · Completeness · Associative generability problem · Longest common subsequence problem · Feedback vertex set problem

M. Elberfeld
Chair of Computer Science 7, RWTH Aachen University, 52056 Aachen, Germany
e-mail: elberfeld@informatik.rwth-aachen.de

C. Stockhusen (✉) · T. Tantau
Institute for Theoretical Computer Science, Universität zu Lübeck, 23538 Lübeck, Germany
e-mail: stockhus@tcs.uni-luebeck.de

T. Tantau
e-mail: tantau@tcs.uni-luebeck.de

1 Introduction

Parameterization has become a powerful paradigm in complexity theory, both in theory and practice. Instead of just considering the runtime of an algorithm as a function of the input's *length*, researchers now routinely analyse runtimes as multivariate functions depending on a number of different input *parameters*, the length being just one of them. While in classical complexity theory instead of "runtime" many other resource bounds have been studied in great detail, in the parameterized world the focus has lain almost entirely on time complexity. Two notable exceptions are the early work of Cai et al. [5], where it is shown that the parameterized *space* complexity of the vertex cover problem, the planar dominating set problem, and some other problems is surprisingly low, and the more recent work of Guillemot [15], where the longest common subsequence problem is investigated.

One reason why parameterized space complexity has received limited attention is the fact that the problems commonly studied in parameterized complexity do not appear to fit into the framework of parameterized space classes that has been established in analogy to parameterized time classes [9,13]. On the one hand, only more or less artificial problems have been shown to be complete for classes like para-L and para-NL, the logarithmic space analogues of FPT, as well as for XL and XNL, the logarithmic space analogues of XP. On the other hand, Guillemot [15] *has* shown completeness of a very natural parameterized problem (the longest common subsequence problem)—but for a class defined somewhat artificially as the reduction closure of a technical problem.

The aim of the present paper is to demonstrate that "natural" parameterized space classes *can* help in understanding the complexity of "natural" parameterized problems. We introduce two kinds of classes and prove completeness of different commonly studied problems for them. The first kind of parameterized classes we introduce are defined using different amounts of *bounded nondeterminism.* The second set of classes are defined in terms of *simultaneous* restrictions on the *space and time* resources of the machines. We stress that the newly introduced classes are arguably even "more natural" than standard classes like W[2] insofar as our classes are defined in terms of simple machine models as opposed to being defined as the reduction closures of technical problems.

1.1 Our Contributions Regarding Classes of Bounded Nondeterminism

Bounded nondeterminism plays a key role in the definition of the weft hierarchy: the well-known class W[P] can be defined as "FPT plus $f(\kappa(x)) \cdot \log |x|$ nondeterministic choices" where x is the input, $\kappa(x)$ is its parameter, and f is some function. One of our key observations in this paper will be that if we replace "FPT" in this definition by parameterized space classes, we get "natural space-analogues of W[P]" like paraWL or paraWNL (rigorous definitions are given in Sect. 3).

Our claim, that parameterized space classes defined using bounded nondeterminism are the natural analogues of the class W[P], will be substantiated in two ways. First, we show that a number of natural parameterized problems lie in these classes and

some are even complete for them. For instance, the associative generability problem parameterized by the size of the generating set is complete for the class paraWNL (while the same problem, without the restriction to "associative" operators, is known to be complete for W[P]). Second, we show that the newly introduced classes do not only "happen" to be useful in classifying the complexity of previously studied problems; rather, there is a deeper connection. We introduce a general "union operation" that turns any language into a parameterized problem in such a way that (under certain conditions) completeness of the language for some complexity class C carries over to completeness of the parameterized problem for a class "paraWC." This sheds a new light on the just-mentioned results on the (associative) generability problem: It is known that the decision versions of the associative and non-associative generability problems are complete for NL and P, respectively. Our theorems on the union operation allow us to "transfer" these classical completeness results to the parameterized world (some extra proof effort is still needed, though). As another example of the broad applicability of the method, we use it to show that the weighted satisfiability problem is complete for the class paraWNC^1. From this, it follows that W[SAT] $\neq$ W[P] implies NC^1 $\neq$ P.

1.2 Our Contributions Regarding Time–Space Classes

In classical complexity theory it makes little sense to study complexity classes like "polynomial time, but using only logarithmic space" since it is well-known that L $\subseteq$ P holds, so any computation using logarithmic space is automatically polynomially time-bounded. Indeed, we even know that L $\subseteq$ NL $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE $=$ NPSPACE holds. In contrast, in the parameterized world new classes arise when we restrict time and space simultaneously: We introduce classes of problems solvable by machines that run in fixed-parameter time and need only slice-wise logarithmic space (think of this class as "FPT, but using only little, namely XL, space"). We will show that different problems are complete for the arising deterministic and nondeterministic classes, including the longest common subsequence problem parameterized by the number of strings.

We also show that the undirected feedback vertex set problem lies in the deterministic version of this class (so feedback vertex sets can be found simultaneously quickly and with little space) while the directed feedback vertex set does not, unless L $=$ NL. Observe that this is the first example of a (fairly natural) subclass of FPT that includes the undirected version of the feedback vertex set problem, while it (presumably) does not contain the directed version. We offer this as a partial explanation why it has been so much harder to prove that the directed feedback vertex set lies in FPT: Unless L $=$ NL, the directed version is a provably harder problem in FPT than the undirected version.

1.3 Related Work

Early work on parameterized space classes is due to Cai et al. [5] who introduced the classes para-L and para-NL, albeit under different names, and showed that several

important problems in FPT lie in these classes: the parameterized vertex cover problem lies in para-L and the parameterized k-leaf spanning tree problem lies in para-NL. Later, Flum and Grohe [13] showed that the parameterized model checking problem of first-order formulas on graphs of bounded degree lies in para-L. In particular, standard parameterized graph problems belong to para-L when we restrict attention to bounded-degree graphs. Recently, Guillemot [15] showed that the longest common subsequence problem (LCS) is equivalent under fpt-reductions to the short halting problem for NTMs, where the time and space bounds are part of the input and the space bound is the parameter. Our results on the longest common subsequence problem differ from Guillemot's insofar as we use weaker reductions (which will be crucial in the context of this paper since the classes we study are presumably closed only under the weaker reductions) and prove completeness for a class defined using a machine model rather than for a class defined as a reduction closure. Nevertheless, we gladly acknowledge that our proof is inspired by Guillemot's work.

1.4 Organisation of This Paper

Since this paper intends to paint a systematic picture of the world of parameterized space classes, in Sect. 2 we first revisit standard terminology from parameterized complexity theory (like "classes with precomputation" or "slicewise classes" or "parameterized reductions") that have been used to define parameterized space classes. We also revisit results from the literature concerning problems and these classes. A new aspect in this section is the idea to use the terminology to define parameterized *circuit* classes like "para-NC^1." We will see in the course of the paper that these classes play some role in the classification of parameterized problems; as a first example, in the course of the introductory Sect. 2 we improve the result of Cai et al. [5] that the vertex cover problem lies in para-L by showing that the problem lies even in para-TC^0.

The main topics of this paper, parameterized classes defined in terms of bounded nondeterminism and in terms of simultaneous time and space bounds, are studied in Sects. 3 and 4, respectively. In both cases, we motivate and then define the new classes, investigate their structural properties, and then place as many well-known problems as possible in the classes and prove completeness whenever possible.

2 An Introduction to Parameterized Space and Parameterized Circuit Depth

Parameterized space classes have been studied in the literature [5,9,13,15] for some time, albeit under different names and with different objectives. In the present section we review the definitions of these classes, which include para-L, para-NL, XL, and XNL. Of these classes, para-L and para-NL are known to be subclasses of FPT and have been used to differentiate between the complexity of problems that are known to be fixed-parameter tractable (and whose complexity is, thus, "indistinguishable" when FPT is the smallest class under consideration). We take this idea a step further by defining even smaller parameterized *circuit* classes like para-NC^1 or para-TC^0. As we will see, these classes can be used to classify problems even more tightly than

was previously possible. Nevertheless, the focus of this section is on establishing the terminology framework for the rest of the paper.

2.1 Parameterized Problems

The first main idea of parameterized complexity theory is to see a computational problem no longer as a simple language $Q \subseteq \Sigma^*$, but rather as a *parameterized problem* (Q, κ) where $\kappa\colon \Sigma^* \to \mathbb{N}$ is a *parameterization* that maps input instances to parameter values. In the classical definition, Downey and Fellows [10] require the parameterization to be computable, while Flum and Grohe [14] require it to be computable in polynomial time. Since we consider parameterized space and circuit classes that lie deep within FPT, we impose even stronger restrictions: The parameterization must be *first-order computable* or, equivalently, computable by logarithmic-time-uniform unbounded fan-in constant-depth circuits [24]. We refer readers unfamiliar with first-order computations to the textbook of Immerman [18], but remark that the details of the definition will not be important for the present paper. What *will* be important, is the observation that the parameter function κ is always first-order computable if the parameter is given explicitly in the input, as is the case in most applications.

A typical example of a parameterized problem is the parameterized vertex cover problem. Formally, it is a pair (Q, κ) with $Q = \{\text{code}(G, k) \mid G \text{ is a graph having a vertex cover of size exactly } k\}$ and $\kappa(\text{code}(G, k)) = k$, but we will use the following standard notation for defining parameterized problems:

Problem 2.1 (p-VERTEX-COVER)

Instance:	An undirected graph $G = (V, E)$ and a natural number k.
Parameter:	k.
Question:	Is there a set $C \subseteq V$ with $\lvert C\rvert = k$ such that for every edge $\{u, v\} \in E$ we have $\{u, v\} \cap C \neq \emptyset$?

In order to distinguish the parameterized problem from the classical plain language version, we prefix the problem name by "p-". This prefix is used to indicate that the parameter is the "natural" or "standard" parameter. Problems such as the following admit several natural parameterizations, however:

Problem 2.2 (LONGEST-COMMON-SUBSEQUENCE (LCS))

Instance:	A set S of strings over some alphabet Σ and a natural number l.
Question:	Is there a string $c \in \Sigma^l$ such that c is a subsequence of all strings in S, i. e., from all $s \in S$ we can obtain c by just removing symbols from s?

Natural parameters for this problem are the number $\lvert S\rvert$ of strings, the length l, the size $\lvert\Sigma\rvert$ of the alphabet Σ, or combinations thereof. We will indicate which parameterization is meant by adding an index to the "p-" prefix. For example, $p_{\lvert S\rvert}$-LCS denotes the longest common subsequence problem parameterized via the parameter $\lvert S\rvert$.

2.2 Variants of Fixed-Parameter "Tractability"

The second main idea of parameterized complexity theory is to assume that for a given problem most inputs will have a "small" or even "fixed" parameter value–even though the input may be large. Thus, we may be willing to initially invest a lot of time (or some other resource) that depends on the parameter (which will be small), but otherwise our computations should be efficient (for instance, polynomially time-bounded). This idea can be formalized as follows: Let C be a classical complexity class. A parameterized problem (Q, κ) belongs to the class para-C if there are an alphabet Π, a computable function $\pi : \mathbb{N} \to \Pi^*$, and a language $A \subseteq \Sigma^* \times \Pi^*$ with $A \in C$ such that for all $x \in \Sigma^*$ we have $x \in Q \iff \big(x, \pi\big(\kappa(x)\big)\big) \in A$. Note that the function π, which is called the *precomputation,* only depends on the parameter and not directly on the input itself.

The most well-known class of parameterized complexity theory, the class FPT of "fixed-parameter tractable problems" [10], is the same as para-P (and we will use para-P in the following to denote this class in order to keep the terminology consistent). By the above definition, a problem is in para-P if it can be "solved in polynomial time when, alongside the input, the result of an arbitrarily complex precomputation is provided that depends, however, only on the parameter." The definition can readily be applied to define the parameterized *space* classes para-L ("solvable in logarithmic space when the result of an arbitrarily complex precomputation is also provided") and para-NL (as before, only for nondeterministic machines) [14]; but it *also* allows us to define parameterized *circuit* classes just as easily: para-NC^1 contains problems "solvable using a family of logarithmic depth circuits of bounded fan-in that get the result of an arbitrarily complex precomputation alongside the input", para-AC^0 contains problems "solvable using a family of circuits of unbounded fan-in and constant depth that get the result of an arbitrarily complex precomputation alongside the input", and para-TC^0 is defined the same way, only now for constant depth threshold circuits. Observe that the "para-classes" inherit their inclusion structure from the underlying classical complexity classes. Thus, the following chain of inclusion holds:

$$\text{para-AC}^0 \subseteq \text{para-TC}^0 \subseteq \text{para-NC}^1 \subseteq \text{para-L} \subseteq \text{para-NL}$$
$$\subseteq \text{para-AC}^1 \subseteq \text{para-TC}^1 \subseteq \text{para-NC}^2 \subseteq \cdots \subseteq \text{para-P} \subseteq \text{para-NP} \subseteq \text{para-PSPACE}.$$

Figure 1 depicts an overview of the classes that we study in the present paper; we will come back to it at different times. The just-introduced para-classes form the "spine" of the diagram.

To get a better feeling for the classes, let us rephrase their definitions in terms of direct "O-bounds" on the involved machines and circuits. A parameterized problem (Q, κ) is in para-P if there is a function $f : \mathbb{N} \to \mathbb{N}$ such that the question $x \in Q$ can be decided within time $f(\kappa(x)) \cdot |x|^{O(1)}$ (this is the classical definition of "fixed-parameter tractable", see [10,14]). By comparison, (Q, κ) is in para-L if $x \in Q$ can be decided within space $f(\kappa(x)) + O(\log |x|)$; and for para-PSPACE the space requirement is $f(\kappa(x)) \cdot |x|^{O(1)}$. A problem (Q, κ) is in para-NC^1 if $x \in Q$ can be decided by bounded-fan-in circuits of depth $f(\kappa(x)) + O(\log |x|)$ and size $f(\kappa(x)) \cdot |x|^{O(1)}$; and

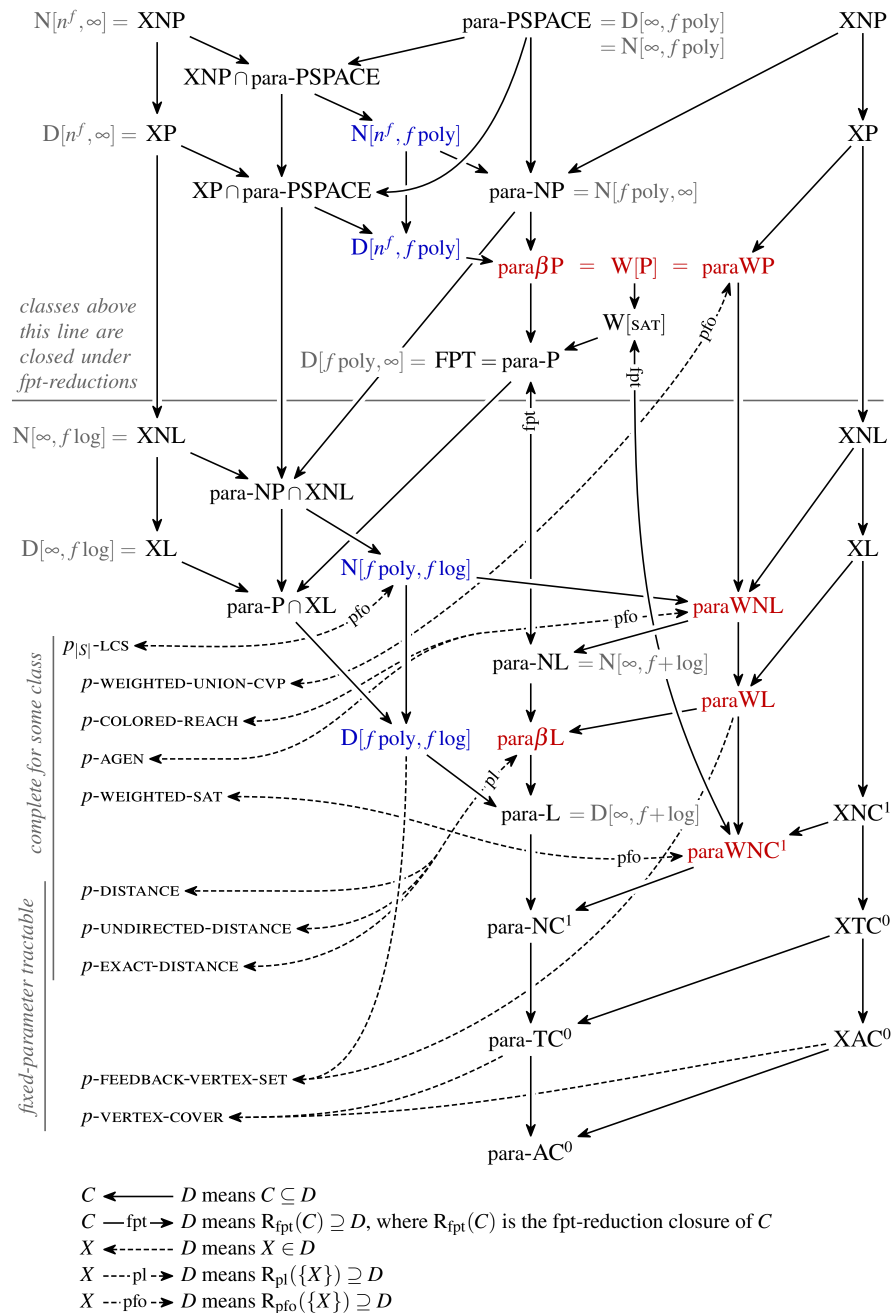


Fig. 1 The different classes studied in the present paper together with the most important completeness and membership results obtained. Bounded nondeterminism classes are shown in *red*, time–space classes in *blue*. Some of the X-classes are shown both left and right to keep the diagram readable. *Solid lines* represent class inclusions, *dashed lines* connect problems to classes. An *arrow* from a problem X to a class D with a reduction type at the tip means that X is hard for the class D under this reduction (Color figure online)

it is in para-TC^0 if $x \in Q$ can be decided by threshold circuits of size $f(\kappa(x)) \cdot |x|^{O(1)}$ and constant depth (the depth may *neither* depend on the input length *nor* on the parameter).

The most well-studied problem in parameterized complexity theory is undoubtedly the vertex cover problem, one of the first problems that has been identified to lie in para-P. Cai et al. [5] have later observed that the problem lies in an even smaller class, namely in para-L. A careful look at their proof reveals that the problem even lies in the (potentially) smaller class para-TC^0 as we will prove in Theorem 2.4 in a moment. Cai et al. also claim that a number of other problems, such as the planar dominating set problem, lie in para-L: Their argument is that the standard *search tree technique* can easily be implemented in logarithmic space, showing that this and many other problems can be placed in para-L. We do not improve these results that rely on the search tree technique; indeed, we are not convinced that the method yields para-L-algorithms as claimed and propose that the complexity of problems like the planar dominating set problem should be reinvestigated. In their proof of p-VERTEX-COVER $\in$ para-L Cai et al. do not employ the search tree technique. Rather, they use *kernelizations,* a concept which we first review and then adapt in order to prove the stronger result p-VERTEX-COVER $\in$ para-TC^0.

2.3 Kernelization

A third main ingredient of parameterized complexity theory is the idea of *kernelization.* A kernelization K of a parameterized problem (Q, κ) is a function that maps every problem instance x to a *kernel* $K(x)$ whose size depends only on $\kappa(x)$ (and not directly on x) such that $x \in Q$ if, and only if, $K(x) \in Q$. The kernel can be regarded as a much smaller "replacement" of the original problem that retains all of its computational complexity.

The importance of the notion of kernelization lies in the well-known fact that a (decidable) problem (Q, κ) lies in para-P if, and only if, it has a kernelization K that is computable in polynomial time: First, suppose (Q, κ) has a kernelization K. On an input x, first compute the kernel $K(x)$ and, then, decide $K(x) \in Q$ using "brute force." Although this last step may take a lot of time, the time is still bounded in terms of $\kappa(x)$ since the size of $K(x)$ is. Second, suppose $(Q, \kappa) \in$ para-P via some machine needing time $f(\kappa(x)) \cdot |x|^c$. On an input x, a kernelization K first tests whether $f(\kappa(x)) < |x|$ holds. If so, it solves the question $x \in Q$ directly in time $|x|^{c+1}$ and outputs a constant-size instance $K(x)$ that is in Q if, and only if, x is. Otherwise, when $f(\kappa(x)) \geq |x|$, the kernelization can simply map x to itself.

It is an easy, but very useful observation that the above arguments all work when "polynomial time" is replaced by "logarithmic space" or by some circuit class. We spell this out in the following lemma:

Lemma 2.3 *Let (Q, κ) be a decidable parameterized problem. Then*

1. $(Q, \kappa) \in$ Para-L *if, and only if, (Q, κ) has a kernelization K computable in logarithmic space; and*
2. $(Q, \kappa) \in$ Para-AC^i *if, and only if, (Q, κ) has a kernelization K computable by a family of AC^i-circuits. The same holds for TC^i- and NC^i-circuits.*

We are now ready to improve on the result of Cai et al. [5] that the vertex cover problem lies in para-L. The key observation of Cai et al. was that the well-known *Buss kernelization* can be computed in logarithmic space. We argue that the kernelization can even be done using a TC^0-circuit family.

Theorem 2.4 *p*-VERTEX-COVER $\in$ *para-TC^0*.

Proof The Buss kernelization is based on three reduction rules: Suppose an undirected graph G is given and we wish to determine whether G contains a vertex cover of size k. First, if G contains a vertex of degree at least $k+1$, then this vertex *must* be present in every vertex cover of size k and, thus, we can remove the vertex and ask whether the remaining graph has a vertex cover of size $k-1$. Second, if G contains an isolated vertex, we can remove this vertex. Third, when the first two rules can no longer be applied, if the graph has more than $k(k+1)$ vertices, it cannot have a vertex cover of size k since a set of k vertices can cover at most $k(k+1)$ edges when each vertex has maximum degree k. Observe that the first reduction rule can actually be applied in parallel to all vertices, that is, we can remove all vertices of degree $k+1$ or more at the same time from the graph. Similarly, we can also remove all isolated vertices in parallel.

Cai et al. have observed that finding out whether a vertex has degree larger than $k+1$ or is isolated can be done in logarithmic space and, thus, we can compute the reduced graph resulting from applying the first two reduction rules in logarithmic space. Since we can also test in logarithmic space whether the reduced graph has size at most $k(k+1)$, the whole kernelization is computable in logarithmic space. To prove the stronger claim of the theorem, just observe that constant-depth threshold circuits where the thresholds in the gates are at most $k(k+1)$ actually suffice to (a) determine whether a vertex has degree larger than $k+1$ or is isolated, to (b) compute the reduced graph resulting from first removing all high-degree vertices and then all isolated vertices, and (c) to test whether the reduced graph has size at most $k(k+1)$. Thus, *p*-VERTEX-COVER has a TC^0-kernelization. □

2.4 Slicewise Parameterized Classes

The core idea behind the para-classes was to capture problems that can be solved by algorithms whose resource consumption is arbitrarily bounded in the parameter, but otherwise have a "fixed degree of efficiency" with respect to the overall input size. Less "ambitious" classes result when we only require that the "degree of efficiency" with respect to the overall input size is defined by the parameter. This can be formalized in the following way: A parameterized problem (Q,κ) is in the *X-class* XC if for every number $w \in \mathbb{N}$ the slice $Q_w := \{x \mid x \in Q \text{ and } \kappa(x) = w\}$ lies in C. It is immediate from the definition that para-$C \subseteq$ XC holds. For the underlying class P, a parameterized problem (Q,κ) is in XP if it is solvable in time $O\big(|x|^{f(\kappa(x))}\big)$. For logarithmic space we have that a problem is in XL or XNL if it can be solved in deterministic or nondeterministic space $f(\kappa(x)) \cdot O(\log |x|)$, respectively. The class XP is in wide use in parameterized complexity theory [10,14]; the logarithmic space classes XL and XNL have been studied before [9,13].

When one tries to determine the smallest para-class and the smallest X-class that contain a given problem, quite different complexities may result. For instance, the problem p-CLIQUE, where we ask whether a graph has a clique of size k and k is the parameter, is not believed to lie in para-P (it is W[1]-complete), but clearly lies in XAC^0: For any *fixed* k we can easily build a circuit family that contains $\binom{|V|}{k}$ constant-depth subcircuits, each of which tests whether a certain selection of k vertices from the vertex set V forms a clique. A similar argument shows that p-VERTEX- COVER $\in \mathrm{XAC}^0$ holds.

2.5 Parameterized Reductions

Reductions are the core tool of complexity theory for comparing the complexity of problems. They are, of course, also available in the context of parameterized complexity theory: A *parameterized reduction* from a parameterized problem (Q_1, κ_1) to (Q_2, κ_2) is a mapping $r\colon \Sigma_1^* \to \Sigma_2^*$ such that

1. for all $x \in \Sigma_1^*$ we have $x \in Q_1$ if, and only if, $r(x) \in Q_2$ and
2. $\kappa_2\bigl(r(x)\bigr) \le g\bigl(\kappa_1(x)\bigr)$ for some computable function g.

Naturally, we still need to impose some restrictions on the amount of resources that may be used to compute r. For *fixed parameter tractable reductions (fpt-reductions)*—the reductions commonly used in parameterized complexity theory—r must be computable in time $f(\kappa_1(x)) \cdot |x|^{O(1)}$, that is, by a "para-P-machine." Since we study classes deep inside para-P, we need two weaker kinds of reductions: Ideally, we would always like to use *parameterized first-order reductions (pfo-reductions),* where r must be computable by a logarithmic-time-uniform para-AC^0-circuit family. (Again, we refer the interested reader to [18] for the relationship between first-order queries and constant-depth circuits, which also explains the name "parameterized *first-order* reductions.") Sometimes, however, these reductions will be too weak, such as in Theorem 3.14. There, we use *parameterized logspace reductions (pl-reductions)* where we require r to be computable in space $f(\kappa(x)) + O(\log|x|)$, that is, by a para-L-machine.

Using standard arguments one can show that all classes in this paper are closed with respect to pfo-reductions, all classes above para-L are closed under pl-reductions, and all classes above para-P are closed under fpt-reductions.

Concerning completeness for the classes we have introduced up to now, Flum and Grohe [13] have observed that completeness for classes like para-P, para-NL, and para-L is "uninteresting" in the sense that complete problems for the underlying classical complexity classes are always complete for the parameterized versions when parameterized trivially. For instance, the standard distance problem for directed graphs is pfo-complete for para-NL with the trivial parameterization $\kappa(x) = 1$ for all x. (The distance problem becomes much more interesting when parameterized by its natural parameter, namely the distance. We address this problem is Theorem 3.14.) Concerning the X-classes, only few, typically fairly artificial problems have been identified to be complete for XL and XNL. As a by-product of our study of time–space classes in Sect. 4, we will identify a number of natural new such problems, including the acceptance problem for finite multihead automata, parameterized by the number of

heads, and the acceptance problem for cellular automata, parameterized by the number of cells.

3 Bounded Nondeterminism

The interplay of nondeterminism and parameterized space seems simple at first sight: NL is closed under complement and NPSPACE is even equal to PSPACE, so only XNL and para-NL appear to be of interest. A closer look reveals, however, that useful and interesting new classes arise when we bound the *amount of nondeterminism* used by machines in dependence on the parameter. For the definition of these classes it is helpful to view nondeterministic machines as deterministic machines augmented by "choice tapes" or "tapes filled with nondeterministic bits." These are extra tapes for a deterministic Turing machine, and an input word is accepted if there is at least one bitstring that we can place on this extra tape at the beginning of the computation such that the Turing machine accepts. It is well-known that NP and NL can be defined in this way using deterministic polynomial-time or logarithmic-space machines, respectively, that have *one-way* access to a choice tape. (For NP it makes no difference whether we have one- or two-way access, but logarithmic-space machines with access to a two-way choice tape can accept all of PSPACE since logarithmic space suffices to verify that the choice tape contains a valid computation consisting of a sequence of configurations, each of which has polynomial size.)

Classes of *bounded* nondeterminism arise when we restrict the length of the bitstrings on the choice tape. For instance, the classes β^h for $h \geq 1$, see [21] and also [2] for variants, are defined in the same way as NP above, only the length of the bitstring on the choice tape may be at most $O(\log^h n)$. Classes of *parameterized* bounded nondeterminism arise when we restrict the length of the bitstring on the choice tape in dependence not only on the input length, but also on the parameter. Furthermore, in the context of bounded-space computations, it also makes a difference whether we have one-way or two-way access to the choice tapes.

In the following, we will first define the arising classes rigorously and then prove the crucial Lemma 3.5 that "links" the complexity of classical problems to the parameterized complexity of what we call the "union versions" of the classical problems. This lemma will be the cornerstone of our proofs that a number of fairly natural problems are complete for different parameterized classes of bounded nondeterminism—such as the associative generability problem. One problem for which the lemma does not help is the distance problem in graphs parameterized by the distance; we do, however, identify a fairly natural class for which this problem is complete.

3.1 Classes and Structural Results

Definition 3.1 Let C be a complexity class defined in terms of a deterministic Turing machine model (like L or P). We define para$\exists^{\leftrightarrow}C$ as the class of parameterized problems (Q, κ) for which there exists a C-machine M, an alphabet Π, and a computable function $\pi : \mathbb{N} \to \Pi^*$ such that: For every $x \in \Sigma^*$ we have $x \in Q$ if, and only if, there exists a bitstring $b \in \{0, 1\}^*$ such that M accepts with $(x, \pi(\kappa(x)))$ on its input

tape and b on the two-way choice tape. We define $\text{para}\exists^{\rightarrow} C$ similarly, only access to the choice tape is now one-way. For the classes $\text{para}\exists^{\leftrightarrow}_{f\log} C$ and $\text{para}\exists^{\rightarrow}_{f\log} C$ the length of b may be at most $f(\kappa(x)) \cdot O(\log |x|)$ for some computable function f.

Observe that, as argued earlier, $\text{para}\exists^{\leftrightarrow}\text{P} = \text{para}\exists^{\rightarrow}\text{P} = \text{para-NP}$ and $\text{para}\exists^{\rightarrow}\text{L} = \text{para-NL}$. Also observe that $\text{para}\exists^{\leftrightarrow}_{f\log}\text{P} = \text{para}\exists^{\rightarrow}_{f\log}\text{P} = \text{W[P]}$ by one of the many definitions of W[P], see also [14].

The above definition can easily be extended to the case where a universal quantifier is used instead of an existential one and where *sequences* of quantifiers are used. This is interpreted in the usual way as having a choice tape for each quantifier and the different "exists ... for all"-conditions must be met in the order the quantifiers appear. For instance, for problems in $\text{para}\exists^{\leftrightarrow}_{f\log}\exists^{\rightarrow}\text{L}$ we have $x \in Q$ if, and only if, there exists a bitstring of length $f(\kappa(x)) \cdot O(\log |x|)$ for the first, two-way-readable choice tape for which an NL-machine accepts. The classes para-NL[f log], para-L-cert, and para-NL-cert introduced in an ad hoc manner by us in [11] can now be represented systematically: They are $\text{para}\exists^{\rightarrow}_{f\log}\text{L}$, $\text{para}\exists^{\leftrightarrow}_{f\log}\text{L}$, and $\text{para}\exists^{\leftrightarrow}_{f\log}\exists^{\rightarrow}\text{L}$, respectively.

In order to make the notation more useful in practice, instead of "$\exists^{\rightarrow}$" let us write "N" and instead of "$\exists^{\rightarrow}_{f\log}$" we write "$\beta$" as is customary. As a new notation, instead of "$\exists^{\leftrightarrow}_{f\log}$" and "$\forall^{\leftrightarrow}_{f\log}$" we write "W" and "$\text{W}_{\forall}$," respectively. The three classes of [11] now become paraβL, paraWL, and paraWNL. Also observe that W[P] = paraWP holds.

To get a better intuition on the W-operator, note that it provides machines with "$f(\kappa(x)) \cdot O(\log |x|)$ bits of nondeterministic information." Equivalently, one could require that we are provided with "$f(\kappa(x))$ many nondeterministically chosen positions in the input, given in the form of a bit vector of the same length as the input with exactly $f(\kappa(x))$ many 1's." This allows us to also apply the W-operator to classes like NC^1 that are not defined in terms of Turing machines. An example of a parameterized problem in paraWNC^1 is p-WEIGHTED-SAT: For a given propositional formula ϕ and parameter k, the k nondeterministically chosen positions define the weight-k satisfying assignment of the variables, and we only need NC^1-circuits to evaluate ϕ on such an assignment, see [3]. We will have a more detailed look on this problem later.

The right half of Fig. 1 shows how the classes of bounded nondeterminism are related to the X-classes and para-classes introduced in the previous section.

3.1.1 Union Problems

Before we prove completeness of different natural problems for the just-introduced classes, we first present a new technical notion, the "union operation," and prove a technical lemma that will greatly simplify our later completeness proofs. The union operation "turns" P-, NL-, L-, and NC^1-complete problems into paraWP-, paraWNL-, paraWL-, and paraWNL-complete problems, respectively.

For numerous problems studied in complexity theory the input consists of a string in which some positions can be "selected" and the objective is to select a "good" subset of these positions. For instance, for the satisfiability problem we must select some variables such that setting them to true makes a formula true; for the circuit satisfiability problem we must select some input gates such that setting them to 1

makes the circuit evaluate to 1; and for the exact cover problem we must select some sets from a family of sets such that they form a partition of the union of the family. The *union operation* will provide us with a terminology for formulating all of these problems in a uniform way and to link them to the W-operator.

Let Σ be an alphabet that contains none of the three special symbols ?, 0, and 1. We call a word $t \in (\Sigma \cup \{?\})^*$ a *template*. We call a word $s \in (\Sigma \cup \{0, 1\})^*$ an *instantiation of* t if s is obtained from t by replacing exactly the ?-symbols arbitrarily by 0- or 1-symbols. Given instantiations $s_1, \dots, s_k$ of the same template t, their *union* is the instantiation of t that has a 1 exactly at those positions i where at least one s_j has a 1 at position i (the union is the "bitwise or" of the instantiated positions and is otherwise equal to the template).

Given a language $A \subseteq (\Sigma \cup \{0, 1\})^*$, we define three different kinds of union problems for A. As we will see in a moment, the first kind is linked to the W-operator while the last kind links several well-known languages from classical complexity theory to well-known parameterized problems. We will also see that the three kinds of union problems for a language A often all have the same complexity.

Problem 3.2 (p-FAMILY-UNION-A for a language $A \subseteq (\Sigma \cup \{0, 1\})^*$)

Instance: A template $t \in (\Sigma \cup \{?\})^*$ and a family $(S_1, \dots, S_k)$ of k sets of instantiations of t.
Parameter: k.
Question: Are there $s_i \in S_i$ for $i \in \{1, \dots, k\}$ such that their union lies in A?

Problem 3.3 (p-SUBSET-UNION-A for a language $A \subseteq (\Sigma \cup \{0, 1\})^*$)

Instance: A template $t \in (\Sigma \cup \{?\})^*$, a set S of instantiations of t, and a number k.
Parameter: k.
Question: Is there a subset $R \subseteq S$ of size $|R| = k$ such that the union of R's elements lies in A?

Problem 3.4 (p-WEIGHTED-UNION-A for a language $A \subseteq (\Sigma \cup \{0, 1\})^*$)

Instance: A template $t \in (\Sigma \cup \{?\})^*$ and a number k.
Parameter: k.
Question: Is there an instantiation $s \in A$ of t containing exactly k many 1-symbols?

To get an intuition for these definitions, think of instantiations as words written on transparencies with 0 rendered as an empty box and 1 as a checked box. Then for the family union problem we are given k heaps of transparencies and the task is to pick one transparency from each heap such that "stacking them on top of each other" and "looking at the result" yields an element of A. For the subset union problem, we are only given one stack and must pick k elements from it. We call the weighted union problem a "union" problem partly in order to avoid a clash with existing terminology and partly because the weighted union problem is the same as the subset union problem for the special set S containing all instantiations of the template of weight 1.

Concerning the promised link between well-known languages and parameterized problems, let A be the CIRCUIT-VALUE-PROBLEM (CVP) where we use Σ to encode a circuit and use 0's and 1's solely to describe an assignment to the input gates.

Then the input for p-WEIGHTED-UNION-CVP is a circuit with ?-symbols instead of a concrete assignment, together with a number k, and the question is whether we can replace exactly k of the ?-symbols by 1's (and the others by 0's) so that the resulting instantiation lies in CVP. Clearly, p-WEIGHTED-UNION-CVP is exactly the W[P]-complete problem p-WEIGHTED-CIRCUIT-SAT, which asks whether there is a satisfying assignment for a given circuit that sets exactly k input gates to 1. For a similar example, let A be the propositional formula evaluation problem BF, where we are given a propositional formula ϕ with n variables, encoded sensibly over some alphabet Σ, followed by a bitstring of length n that encodes an assignment of truth values to the variables. The problem p-WEIGHTED-UNION-BF is then the W[SAT]-complete problem p-WEIGHTED-SAT.

Concerning the promised link between union problems and the W-operator, recall that the operator provides machines with $f(\kappa(x))$ nondeterministic indices as part of the input. In particular, a W-machine can nondeterministically mark $f(\kappa(x))$ different "parts" of the input—like one element from each of $f(\kappa(x))$ many sets in a family, like the elements of a size-$f(\kappa(x))$ subset of some set, or like $f(\kappa(x))$ many positions in a template. With this observation it is not difficult to see that if $A \in C$, then all union versions of A lie in para$\mathrm{W}C$. A much deeper observation is that the union versions are also often *complete* for these classes. In the next theorem, which states this claim precisely, the following definition of a *format-respecting first-order projection* p from a language $A \subseteq \Gamma^*$ to a language $B \subseteq (\Sigma \cup \{0, 1\})^*$ is used: First, p must be a *first-order* reduction from A to B, that is, $x \in A \iff p(x) \in B$ and p is computable using a first-order query or, equivalently, a logarithmic-time-uniform constant-depth circuit family. Second, p must be a *projection,* meaning that each symbol of $p(x)$ depends on at most one symbol of x. Third, p must be *format-respecting*, meaning that for each word length n there must be a single template $t_n \in (\Sigma \cup \{?\})^*$ such for all $x \in \Gamma^n$ the word $p(x)$ is an instantiation of t_n. In other words, the symbols of Σ are used for the fixed parts of the target string that do not change for input strings x of the same length, while in $p(x)$ the positions with the symbol ? vary with x.

Lemma 3.5 *Let* $C \in \{L, NL, P\}$ *or* $C \in \{AC^i, TC^i, NC^{i+1}\}$ *for some* $i \geq 0$ *and let* $A \subseteq (\Sigma \cup \{0, 1\})^*$ *be complete for* C *via format-respecting first-order projections. Then the problem* p-FAMILY-UNION-A *is pfo-complete for paraWC.*

Proof For membership, on input of a template t and a family $(S_1, \ldots, S_k)$ of sets of instantiations of t, a para$\mathrm{W}C$-machine interprets its nondeterministic bits as k indices, one for each S_i. Let $s_i \in S_i$ be the elements selected in this way. We run a simulation of the C-machine that decides A on the union s of $s_1, \ldots, s_k$. For logspace machines, we may not have enough space to write s on a tape, so whenever the machine would like to know the jth bit of u, we simply (re)compute the bitwise or of the jth positions of the s_i. For C-machines, recall that instead of nondeterministic bits, we get a bitstring with exactly k bits set to 1 alongside the input. We interpret such a string to select a string $s_i \in S_i$ if a 1-bit is exactly at the beginning of s_i in the input (and we can easily ensure, even using an AC^0-circuit, that only one s_i is selected for each S_i).

For hardness, consider any problem $(Q, \kappa) \in$ para$\mathrm{W}C$ with $Q \subseteq \Lambda^*$. By definition, this means the following: There are a language $X \subseteq \Gamma^*$ in C and computable functions

$\pi : \mathbb{N} \to \Pi^*$ and $f : \mathbb{N} \to \mathbb{N}$ such that for all $x \in \Lambda^*$ we have $x \in Q$ if, and only if, there is a string $b \in \{0, 1\}^{f_x \lceil \log_2 n \rceil}$ with $(x, \pi(\kappa(x)), b) \in X$. (To simplify the notation, we set $n = |x|$ and $f_x = f(\kappa(x))$.) Furthermore, since A is complete for C via format-respecting first-order projections, we can reduce X to A via some p.

For the pfo-reduction of (Q, κ) to p-FAMILY-UNION-A, let an input $x \in \Lambda^*$ be given. We wish to determine whether there is a bitstring $b \in \{0, 1\}^{f_x \lceil \log_2 n \rceil}$ with $(x, \pi(\kappa(x)), b) \in X$ or, equivalently, with $p(x, \pi(\kappa(x)), b) \in A$. Since p is format-respecting, for all possible b the string $p(x, \pi(\kappa(x)), b)$ will have the special 0-symbols and 1-symbols at the same positions and all other positions, which will be elements of Σ, will not vary with b. Let t be the single template underlying the different $p(x, \pi(\kappa(x)), b)$.

We now define sets $S_1, \ldots, S_{f_x}$, where each S_i is a set of instantiations of t. Our objective is to set up the S_i in such a way that (a) when we pick one s_i from each S_i, their union is $p(x, \pi(\kappa(x)), b)$ for some b, and (b) every $p(x, \pi(\kappa(x)), b)$ *is* the union of appropriately chosen $s_i \in S_i$. To achieve this, let us think of the strings $b \in \{0, 1\}^{f_x \lceil \log_2 n \rceil}$ as sequences of f_x symbols from the alphabet $\Delta = \{0, 1\}^{\lceil \log_2 n \rceil}$, whose elements we call *blocks*. For $i \in \{1, \ldots, f_x\}$ let $S_i = \{m_i^\delta \mid \delta \in \Delta\}$ where the m_i^δ are the following instantiations of t (we only need to explain how the ?-symbols get replaced): Consider a ?-symbol at position r in t, which we must replace by $c \in \{0, 1\}$. The rth position of t will depend on the symbol at (at most) one specific position r' in $(x, \pi(\kappa(x)), b)$ and this position is independent of b. If r' does not lie in the "b-part," let c be whatever the reduction outputs based on the symbol at r' (this will be independent of b). Next, if r' *does* lie in one of the blocks of b, but not in the ith block, let $c = 0$. Otherwise, let c be whatever symbol (0 or 1) the reduction outputs when the ith block of b is δ. This concludes the construction.

As an example for the construction, suppose the reduction p simply doubles its input, but replaces 0's by 1's and vice versa in the second copy. Let π just return the empty string and let $\Sigma = \{u, v, w\}$. Consider, say, $x = uvwu$ and assume $f_x = 2$. We then have $\Delta = \{00, 01, 10, 11\}$ and a string $b = b_1 b_2 b_3 b_4$ is also a string $\delta_1 \delta_2$ of two blocks $\delta_1, \delta_2 \in \Delta$. The reduction would produce two sets S_1 and S_2. For S_1, we have a look at what p does on input of a string like $(x, \pi(\kappa(x)), b)$. For simplicity let us ignore parentheses and commas, so this string would just be $uvwub_1b_2b_3b_4$. The reduction maps this to $uvwub_1b_2b_3b_4uvwu\bar{b}_1\bar{b}_2\bar{b}_3\bar{b}_4$ with $\bar{b}_i = 1 - b_i$. In this string, the fifth, sixth, thirteenth, and fourteenth bits actually depend on the first block of $uvwub_1b_2b_3b_4$, so the reduction would produce the following set S_1 and, in a similar fashion, S_2:

$$\begin{aligned} S_1 &= \{uvwu0000uvwu1100, uvwu0100uvwu1000, \\ &\quad\; uvwu1000uvwu0100, uvwu1100uvwu0000\}, \\ S_2 &= \{uvwu0000uvwu0011, uvwu0001uvwu0010, \\ &\quad\; uvwu0010uvwu0001, uvwu0011uvwu0000\}. \end{aligned}$$

The crucial observation in this example is that we get every string $p(x, \pi(\kappa(x)), b)$ for the different possible values of b by taking the union of one string from S_1 and one string from S_2.

To see that the reduction is correct in general, consider the union of the elements of any set $\{m_1^{\delta_1}, \dots, m_{f_x}^{\delta_{f_x}}\}$ where the $m_i^{\delta_i}$ are chosen from the different S_i. By construction, their union will be exactly the image of $(x, \pi(\kappa(x)), \delta_1 \dots \delta_{f_x})$ under p. In particular, $x \in Q$ holds if, and only if, we can choose one instantiation from each S_i such that their union is in A. □

The above lemma is the main reason why we propose to use "W" to denote the quantifier "$\exists^{\leftrightarrow}_{f \log}$": There is a pattern that completeness of classical problems for L, NL, or P tends to carry over to completeness of closely related problems for paraWL, paraWNL, or paraWP—and the lemma provides precise conditions under which this always happens. We remark, however, that Guillemot, in a paper [15] on parameterized *time* complexity, uses "WNL" to denote a class different from our class paraWNL. Guillemot chose the name because his definition of the class is derived from one possible definition of W[1] by replacing a time by a space constraint. Nevertheless, we believe that our definition of a "W-operator" yields the "right analogue" of W[P]: First, there is the above lemma and, second, in Sect. 4.2.2 we show that the class WNL defined and studied by Guillemot is exactly the fpt-reduction closure of the time–space class N[f poly, f log].

3.2 Natural Problems Complete for the W-Classes

3.2.1 Parameterized Satisfiability Problems

The circuit value problem CVP is well-known to be complete for P and completeness can easily by achieved via format-respecting first-order projections. Thus, by Lemma 3.5, we get that p-FAMILY-UNION-CVP is pfo-complete for paraWP. Since one can reduce p-FAMILY-UNION-CVP to p-WEIGHTED-UNION-CVP via essentially the same reduction as that used in the proof of Theorem 3.6 below, we conclude that p-WEIGHTED-UNION-CVP is also complete for paraWP – meaning that we can reprove the well-known fact that p-WEIGHTED-CIRCUIT-SAT is complete for W[P] using mostly structural arguments and using very weak reductions (namely pfo-reductions).

We get an even more interesting result when we apply the lemma to BF, the propositional formula evaluation problem.

Theorem 3.6 *p-WEIGHTED-UNION-BF is pfo-complete for paraWNC^1.*

Proof The language BF is complete for NC^1, see [3,4], and completeness can be achieved by format-respecting projections: Indeed, for input words of the same length, the reduction will map them to the same formula, only the assignment to the variables will differ (the input word is encoded solely in this assignment). Thus, by Lemma 3.5 we get that p-FAMILY-UNION-BF is complete for paraWNC^1 under pfo-reductions.

We now show that p-FAMILY-UNION-BF reduces to p-SUBSET-UNION-BF, which then in turn reduces to p-WEIGHTED-UNION-BF. For the first reduction, let the sets S_1 to S_k be given as input. All elements s_{ij} of the S_i represent assignments to the variables of the same formula ϕ. Our aim is to construct a set S and a new formula $\phi' = \phi \wedge \psi$, where the job of ψ is to ensure that any selection of k elements from S can only lead

to ϕ' being true if the selection corresponds to picking "exactly one element from each S_i." In detail, for each s_{ij} we introduce a new variable v_{ij}. The assignment s'_{ij} for ϕ' is the same as s_{ij} for the "old" variables and is 1 only for v_{ij} among the new variables (v_{ij} "tags" s_{ij}). As an example, suppose there are three variables x, y, and z in ϕ and suppose $S_1 = \{\phi000, \phi001\}$ (meaning that one assignment sets all variables to false and the other sets only z to true) and $S_2 = \{\phi001\}$. Then there would be three additional new variables and $S = \{\phi'000\,100, \phi'001\,010, \phi'001\,001\}$. Now, setting $\psi = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{|I_i|} v_{ij}$ ensures that ψ will only be true for the union of k assignments taken from S if exactly one assignment was taken from each S_i.

Next, we reduce p-SUBSET-UNION-BF to p-WEIGHTED-UNION-BF. Towards this end, let $S = \{s_1, \ldots, s_n\}$ be given as input and let $\phi?^m$ be the template underlying the s_i. Our new formula ϕ' has exactly n variables v_1 to v_n and is obtained from ϕ by leaving the structure of ϕ intact, but substituting each occurrence of a variable x as follows: let $X \subseteq \{1, \ldots, n\}$ be the set of indices i such that in s_i the variable x is set to 1. Then we substitute x by $\bigvee_{i \in X} v_i$, yielding ϕ' and output the template $\phi'?^n$. As an example, let $\phi = x \wedge (y \to x) \wedge z$ and let $S = \{\phi000, \phi101, \phi010, \phi011\}$. Then there would be four variables v_1 to v_4 and the formula ϕ' would be $v_2 \wedge ((v_3 \vee v_4) \to v_2) \wedge (v_2 \vee v_4)$.

To see that this reduction is correct, assume that $(\phi?^m, S, k) \in p$-SUBSET-UNION-BF via a selection $\{s_1, \ldots, s_k\} \subseteq S$. Then also $(\phi'?^n, k) \in p$-WEIGHTED-UNION-BF via the instantiation where exactly the variables corresponding to s_1 to s_k are set to true: in ϕ' the expression $\bigvee_{i \in X} v_i$ that was substituted for a variable x will be true exactly if one of the s_i has set x to 1. Thus, ϕ' will evaluate to 1 for the assignment in which exactly the selected v_i are true if, and only if, ϕ evaluates to true for the "bitwise or" of the assignments $s_1, \ldots, s_k$—which it does by assumption. For the other direction, assume that $(\phi'?^n, k) \in p$-WEIGHTED-UNION-BF. Then, by essentially the same argument, we obtain a subset of S whose union is an instance of BF. □

By definition, W[SAT] is the fpt-reduction closure of p-WEIGHTED-SAT, which is the same as p-WEIGHTED-UNION-BF. Thus, by the theorem, W[SAT] is also the fpt-reduction closure of paraWNC^1 – a result that may be of independent interest. In particular, we get the following corollaries:

Corollary 3.7 *$NC^1 = P$ implies W[SAT] = W[P].*

Corollary 3.8 *paraWNC^1 $\subseteq$ para-P if, and only if, W[SAT] = para-P.*

Note that we do not claim W[SAT] = paraWNC^1 since paraWNC^1 is presumably not closed under fpt-reductions.

3.2.2 Graph Problems

In order to apply Lemma 3.5 to graph problems like REACH (the directed reachability problem) or CYCLE (the question whether a directed graph contains a cycle), we encode graphs using adjacency matrices consisting of 0- and 1-symbols. Then a template is always a string of n^2 many ?-symbols for n vertex graphs. A problem like p-SUBSET-UNION-TREE now asks whether we can pick exactly k graphs from a set of graphs such that their union is a tree (like cycles, we regard trees and forests as *directed* graphs

with edges pointing away from the roots; but one gets the same complexity results for undirected trees and forests). Note that any reduction to a union problem for this encoding is automatically format-respecting as long as the number of vertices in the reduction's output depends only on the length of its input.

Applying Lemma 3.5 to standard L- or NL-complete problems yields that their family union versions are complete for paraWL and paraWNL, respectively. By reducing the family versions further to the subset union version, we get the following:

Theorem 3.9 *1. For each $A \in \{$*REACH, DAG-REACH, CYCLE*$\}$. Then the problem-p-*SUBSET-UNION-*A is pfo-complete for the class paraWNL.*

*2. For each $B \in \{$*UNDIRECTED-REACH, TREE, FOREST, UNDIRECTED-CYCLE*$\}$. Then the problem-p-*SUBSET-UNION-*B is pfo-complete for the class paraWL.*

Proof For each of the problems, we reduce its family union version to it. This suffices: By Lemma 3.5 and the fact that the underlying problems like REACH are complete for NL and L under format-respecting first-order projections, the family versions are complete for the respective classes.

Recall that the difference between the problems p-FAMILY-UNION-A and p-SUBSET-UNION-A is that in the first we are given k sets S_i from each of which we must choose one element, while for the latter we can pick k elements from a single set S arbitrarily. We cannot just set S to the (somehow disjoint) union of the S_i since many choices of k sets from S will correspond to taking multiple elements from a single S_i. In such cases, their union should *not* be an element of A.

To achieve the effect that the union of a subset of S with multiple elements from the same S_i is not in A, we use the same construction for all A, except for $A =$ FOREST. The construction works as follows: Since the S_i are format-respecting, they are defined over the same set V of vertices. Each $s \in S_i$ encodes an edge set $E_s \subseteq V^2$. We construct a new vertex set $V' \supseteq V$ as follows: For each pair $(a, b) \in V^2$ we introduce k new vertices $v^1_{ab}, \dots, v^k_{ab}$ and add them to V'. For each $s \in S_i$ we define a new edge set $E'_s \subseteq V' \times V'$ as follows: First, for each $(a, b) \in V^2$ let $(v^{i-1}_{ab}, v^i_{ab}) \in E'_s$, where $v^0_{ab} = a$. Second, for each $(a, b) \in E_s$, let $(v^k_{ab}, b) \in E'_s$. Let s' be the bitstring encoding the adjacency matrix of E'_s. We set $S = \{ s' \mid s \in S_i \text{ for some } i \}$. An example of how this reduction works is depicted in Fig. 2.

In order to argue that the reduction works for all problems, we make two observations. Given any subset $\{s'_1, \dots, s'_k\} \subseteq S$, for each s'_i there is a unique corresponding s_i, lying in (some) S_j. Let $G' = (V', E')$ denote the graph whose adjacency matrix is the union of $\{s'_1, \dots, s'_k\}$; and let $G = (V, E)$ correspond to the union of the s_i. Now, first assume that, indeed, we have $s_i \in S_i$ for all $i \in \{1, \dots, k\}$. Then for every pair $(a, b) \in V$ the new vertices v^1_{ab} to v^k_{ab} will form a path in G' attached to a. Furthermore, for every edge $(a, b) \in E$ there is a path from a to b in E'. On the other hand, for $(a, b) \notin E$, we cannot get from a to b in G' using only new vertices: the edge (v^k_{ab}, b) will be missing. This proves our first observation: for vertices $a, b \in V$ there is a path from a to b in G' if, and only if, there is such a path in G. Our second observation concerns the case that there are two strings s'_i and s'_j such that s_i and s_j lie in the same set S_x. In this case, for every two vertices $a, b \in V$ at least one edge is missing along the path v^0_{ab} to v^k_{ab}. Thus, we observe that there is no path from any $a \in V$ to any other $b \in V$ in G'.

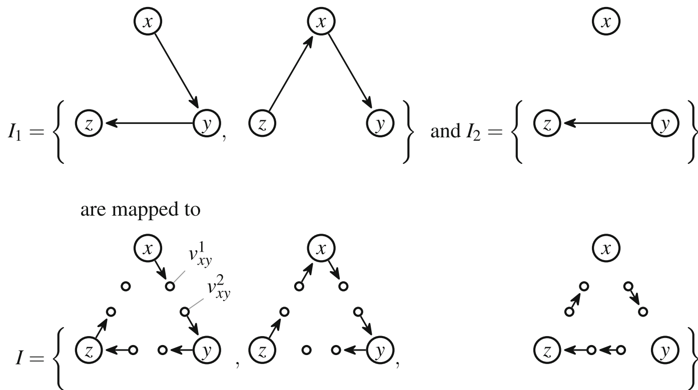


Fig. 2 An example of the reduction from a family union graph problem to a subset union graph problem. In the example, $V = \{x, y, z\}$. The s_i are indicated as edge sets even though, in reality, they are bitstrings encoding adjacency matrices. The small vertices are the v^1_{ab} and v^2_{ab}, but only those for $(a, b) \in \{(x, y), (y, z), (z, x)\}$ are shown

Let us now argue that the reduction is correct: For REACH, by the first observation reachability is correctly transferred from G to G' and by the second observation no "wrong" choice of s'_i will induce reachability. The exact same argument holds for UNDIRECTED-REACH and DAG-REACH. For the problems TREE, CYCLE, and UNDIRECTED − CYCLE the argument also works since trees and cycles remain trees and cycles for "correct" choices of the s'_i and they get destroyed for any "wrong" choice.

For FOREST, the reduction described above does not work since in case several s'_i are picked such that their s_i stem from the same S_j, the graph G' becomes a collection of small trees: a forest—and this is exactly what should *not* happen. Here we use a different reduction: For every pair $s_i, s_j \in S_x$ for $x \in \{1, \dots, k\}$ we add three new vertices to the graph: a, b, and c. In s'_i we add the edges (a, b) and (b, c), in s'_j we add the edge (c, a). Now, clearly, whenever s'_i and s'_j are picked stemming from the same S_x, a cycle will ensue; and if only one s_i is picked from each S_i, paths of length 1 or 2 will result in the new vertices that do not influence whether the graph is a forest or not. □

Readers may wonder whether a problem like "p-SUBSET-UNION-REACH" is really all that "natural." However, we point out that one can also view this problem in a slightly different light, making it more "accessible" and easier to remember:

Problem 3.10 (p-EDGE-COLORED-REACH)

Instance: A directed graph $G = (V, E)$ in which each edge has a (typically not unique) color, two vertices $s, t \in V$, and a number k.
Parameter: k.
Question: Is there a path from s to t using at most k colors?

Corollary 3.11 *p-EDGE-COLORED-REACH is pfo-complete for paraWNL.*

Proof Clearly, p-EDGE-COLORED-REACH is almost the same as p-SUBSET-UNION-REACH if we interpret each set of edges having a certain color as an element of a

set S. The "almost" refers to the fact that an edge can have only a single color, while an edge can be present in multiple elements of S. This is, however, easy to fix by replacing each edge by a path of length 2 via a fresh vertex. □

Instead of colored edges, one can also consider p-COLORED-REACH, where the vertices are colored. A simple reduction (just place a vertex "on each edge" having the edge's color) shows that this problem is also paraWNL-complete.

To conclude this section on union graph problems, we would like to point out that one can also ask which problems are complete for the "co-W-classes" $\mathrm{paraW}_\forall\mathrm{NL}$ and $\mathrm{paraW}_\forall\mathrm{L}$. It is straightforward to see that an analogue of Lemma 3.5 holds if we define problems p-FAMILY-UNION$_\forall$-A as a "universal version" of the family union problem (we ask whether *for all* choices of s_i their union is in A). For instance, the universal cycle problem p-FAMILY-UNION$_\forall$-CYCLE is complete for $\mathrm{paraW}_\forall\mathrm{NL}$. It is also relatively easy to employ the same ideas as those from the proof of Theorem 3.9 to show that the universal union versions of all problems mentioned in Theorem 3.9 are complete for $\mathrm{paraW}_\forall\mathrm{NL}$ and $\mathrm{paraW}_\forall\mathrm{L}$ *except* for p-SUBSET-UNION$_\forall$-TREE, whose complexity remains open.

3.2.3 Associative Generability

The last union problem we study is based on the GENERATORS problem, which contains tuples $(U, \circ, x, G)$ where U is a set, $\circ\colon U^2 \to U$ is (the table of) a binary operation, $x \in U$, and $G \subseteq U$ is a set. The question is whether the closure of G under $\circ$ (the smallest superset of G closed under $\circ$) contains x. A restriction of this problem is ASSOCIATIVE-GENERATORS, where $\circ$ must be associative. By two classical results, GENERATORS is P-complete [19] and ASSOCIATIVE-GENERATORS is NL-complete [20].

In order to apply the union operation to generator problems, we encode $(U, \circ, x, G)$ as follows: U, $\circ$, and x are encoded in some sensible way using the alphabet Σ. To encode G, we add a 1 after the elements of U that are in G and we add a 0 after *some* elements of U that are not in G. This means that in the underlying templates we get the freedom to specify that only some elements of U may be chosen for G. Now, p-WEIGHTED-UNION-GENERATORS equals the problem known as p-GENERATORS in the literature: Given $\circ$, a subset $C \subseteq U$ of generator candidates, a parameter k, and a target element x, the question is whether there exists a set $G \subseteq C$ of size $|G| = k$ such that the closure of G under $\circ$ contains x. Flum and Grohe [14] have shown that p-GENERATORS is complete for W[P] = paraWP (using a slightly different problem definition that has the same complexity, however). Similarly, p-WEIGHTED-UNION-ASSOCIATIVE-GENERATORS is also known as p-AGEN. We show:

Theorem 3.12 *p-AGEN is pfo-complete for paraWNL.*

Proof The language AGEN is complete for NL, see [20], and completeness can be achieved by format-respecting first-order projections. This implies p-AGEN $\in$ paraWNL since all union problems of problems in NL lie in paraWNL. To prove hardness, we first have, by Lemma 3.5, that p-FAMILY-UNION-AGEN is pfo-complete for paraWNL. We show that this problem reduces to p-SUBSET-UNION-AGEN, which in turn reduces to p-WEIGHTED-UNION-AGEN = p-AGEN.

For the first reduction let sets $S_1, \dots, S_k$ be given as input. Their template encodes a universe U, a set of generator candidates $C \subseteq U$, a target element $x \in U$, and an operation $\circ\colon U^2 \to U$. The instantiations encode subsets of the generator candidates. Our aim is to construct a new instance of p-SUBSET-UNION-AGEN, i.e., a single set S' encoding a universe U', a set of generator candidates $C' \subseteq U'$, a target element x', and an operation $\circ'$ such that there are k elements of S' that induce a generating set for x' if, and only if, there are k elements s_i, one from every S_i, such that they induce a generating set of x. To achieve this, we first set $U' = U \cup \{e_1, \dots, e_k\}$ for new elements e_i, one for each S_i, and also add them to the new set of generator candidates $C' = C \cup \{e_1, \dots, e_k\}$. We now wish to augment the operation $\circ$ to $\circ'$ with respect to the new elements, so that no e_i can be generated by any combination of two other elements from the universe and no e_i can be used to generate elements from the universe other than itself (we achieve this by actually using whole string as elements of our universe, as will be discussed later in this proof). Furthermore, we insert a new target element x' into the universe. Our aim is to enforce that x' can only be generated via the expression $x \circ' e_1 \circ' e_2 \circ' \cdots \circ' e_k$. Finally, we add an *error element* `error` to the universe that we will use to create dead ends in the evaluation of expressions: Any expression that does not make sense or contains the error element is evaluated to `error`.

The set S' contains a string s'_{ij} for every $s_{ij} \in S_i$ that is essentially s_{ij} adjusted to U', C', x', and $\circ'$, where we require that the binary string that selects a set of generators from C' also selects e_i and no other of the introduced elements e_j. From this we have that there is a selection of k elements of S' that induces a set of generators whose closure contains $x, e_1, \dots, e_k$ and therefore also x' if, and only if, there is a set of k strings $s_i \in S_i$ describing a set of generators whose closure contains x.

Unfortunately, our operator $\circ'$ is binary and, therefore, we cannot evaluate expressions like $x \circ' e_1 \circ' e_2 \circ' \cdots \circ' e_k$ in a single step. Moreover, because of the required associativity of $\circ'$, it has to be possible to completely evaluate any subexpression of a larger expression. To achieve this, we actually use strings as elements of our universe, instead of single symbols, that are evaluated "as far as possible." For instance, the expression $a \circ' b \circ' c \circ' e_1 \circ' e_2$ evaluates to $d \circ' e_1 \circ' e_2$ if the expression $a \circ' b \circ' c$ evaluates to d. Since $d \circ' e_1 \circ' e_2$ cannot be evaluated further, we want the string de_1e_2 to be part of our universe.

To formalize the idea of "strings evaluated as far as possible," we need some definitions. Given an alphabet Γ, let us call a set R of rules of the form $w \to w'$ with $w, w' \in \Gamma^*$ a *replacement system*. An *application* of a rule $w \to w'$ takes a word uwv and yields the word $uw'v$; we write $uwv \Rightarrow_R uw'v$ in this case. A word is *irreducible* if no rule can be applied to it. Let $\equiv_R$ be the reflexive, symmetric, transitive closure of $\Rightarrow_R$. Given a word u, let $[u]_R = \{\, v \mid u \equiv_R v\}$ be the equivalence class of u. We use $\Gamma^*/_{\equiv_R} = \{\, [v]_R \mid v \in \Gamma^*\}$ to denote the set of all equivalence classes of Γ^*. Observe that we can define a natural concatenation operation $\circ_R$ on the elements of $\Gamma^*/_{\equiv_R}$: Let $[u]_R \circ_R [v]_R = [u \circ v]_R$. Clearly, this operation is well-defined and associative. An *irreducible representative system* of R is a set of irreducible words that contains exactly one word from each equivalence class in $\Gamma^*/_{\equiv_R}$.

In the context of our reduction, Γ will be U' and R contains the following rules: First, for elements $a, b, c \in U$ of the original universe U with $a \circ b = c$, we have

the rule $ab \to c$. Second, we have the rule $xe_1 \dots e_k \to x'$. Third, we have the rules $\texttt{error}u \to \texttt{error}$ and $u\texttt{error} \to \texttt{error}$ for all $u \in U'$. Fourth, we have the rules $e_i u \to \texttt{error}$ for all $u \in U' \setminus \{e_{i+1}\}$ and $x'u \to \texttt{error}$ for all $u \in U'$.

We can now, finally, describe the sets to which the reduction actually maps an input $(U, \circ, x, C)$: The universe U'' is an appropriately chosen irreducible representative system of R, the operation $\circ''$ maps a pair (u, v) to the representative of $[u \circ v]_R$, the target element x'' is the representative of $[x']_R$, and C'' contains all representatives of $[c]_R$ for $c \in C'$.

Our first observation is that $(U')^*/_{\equiv_R}$ (and hence also U'') has polynomial size: Consider any $[w]_R$ and let w be irreducible. If w does not happen to be the error symbol itself, it cannot contain the error symbol (by the third rule). Furthermore, in w there cannot be any element from U to the right of any e_i or of x' (by the fourth rule). Thus, it must be of the form $w_1 w_2$ with $w_1 \in U^*$ and $w_2 \in \{e_1, \dots, e_k, x'\}^*$. Then w_1 must actually be a single letter (by the first rule) and w_2 must be x' or a sequence $e_i e_{i+1} \dots e_j$ for some $i \leq j$ (by the fourth rule). This shows that the total number of different equivalence classes is at most $1 + |U'|(k^2 + 1)$.

The second observation concerns the equivalence class of $[x']_R$, which contains the string $xe_1 \dots e_k$. We can only generate this class from elements $[c]_R$ with $c \in C'$ if these elements include all $[e_i]_R$ and also the equivalence classes $[c]_R$ of elements $c \in C$ that suffice to generate x. This shows the correctness of the reduction.

We continue our chain of reductions by reducing to p-WEIGHTED-UNION-AGEN. Given a format-respecting set $S = \{s_1, \dots, s_k\}$ whose strings encode a universe U, a set of generator candidates $C \subseteq U$, an associative operation $\circ \colon U^2 \to U$, and a target element $x \in U$, together with a selection of generator candidates, we have to construct an instance S' such that every string only selects a single generator candidate. To achieve this, we construct a new universe U' that contains the elements of the old universe U with new elements described below. As in the previous reduction, we define reduction rules alongside these new elements and then use an appropriately chosen irreducible representative system of the rules as our universe.

1. We have an error element $\texttt{error}$ with similar rules as above.
2. We have an *end element* $\lhd$. No rule generates $\lhd$ on its right-hand side if it was not already present on the left-hand side and, thus, $\lhd$ has to be an element of any generating set G. We require this element for technical reasons that we will discuss later. There are rules $\lhd u \to \texttt{error}$ for all $u \in U'$.
3. We have a *counter element* $|$. Like the end symbol, this symbol cannot be newly generated by expressions and has to be an element of any generating set.
4. We have elements σ_i for each $s_i \in S$, which we call *selector elements*. The idea behind these elements is that we will use them together with the counter element to enumerate all the elements $u_1, \dots, u_l$ of the generator candidates selected by a string $s_i \in S$. The objective is that strings like $\sigma_i||||$ can be replaced by u_4. We will give rules for this in a moment.

In our new template, the candidates are (the representatives of the equivalence classes of) the σ_i as well as $\lhd$, $|$, and $\texttt{error}$. Now, there is a selection of $k + 3$ elements of S' that forms a generating set for the target element if, and only if, there is a selection of k elements of S that forms a generating set.

It remains to explain how rules can be set up such that for instance $\sigma_i||||$ gets replaced by u_4. Consider the expression $\sigma_1||\sigma_3|||\sigma_2||$. Here, $\sigma_1||$ can be replaced by some u and $\sigma_3|||$ by some u', but $\sigma_2||$ cannot yet be replaced since it is not clear what element will be appended to the expression (if there is another element). To fix this, we use the end symbol $\triangleleft$ that has to be appended to every expression. It marks the right end of the expression and enforces the unambiguous evaluation of the very last subexpression and, therefore, the whole expression. Translated into rules, this means that if, for instance, $\sigma_i||||$ should select u_4, then we have rules like $\sigma_i||||u \to u_4u$ if $u \neq |$, but do not have the rule $\sigma_i|||| \to u_4$, so $\sigma_i||||$ is an irreducible string. (In order to ensure that we do not get overly long sequences of counter symbols, we have the rule $|^n \to \texttt{error}$ where n is the size of the universe.)

The target is the irreducible string $x\triangleleft$.

It is now easy to see that, once more, the number of equivalence classes is polynomially in the size of the universe: Basically, irreducible strings start with a sequence of counter elements of length at most n and end with another such sequence, again of length at most n and, in the middle, there is single element of the original universe possibly followed by a selector. This means that the representative system also has polynomial size. We mentioned earlier that the system should be "appropriately chosen," by which we mean that it must be chosen in such a way that our pfo-reduction can compute the resulting concatenation table using its very limited resources. While this is not overly difficult to achieve, we skip the technical details of how, exactly, this is done. □

3.3 Natural Problems Complete for the β-Class

We conclude our exploration of classes of bounded nondeterminism with a deceptively simple problem, namely the distance problem with the distance as the parameter: (We could also have started with this problem, but we will not use Lemma 3.5 in the proofs here and we preferred to keep the lemma together with its applications.)

Problem 3.13 (p-DISTANCE)

Instance: A directed graph G, two vertices s and t of G, and a natural number k.
Parameter: k.
Question: Is there a path from s to t in G of length at most k?

Theorem 3.14 *p-DISTANCE is complete for paraβL with respect to pl-reductions.*

Proof Clearly, p-DISTANCE $\in$ paraβL since we can just "guess" a path of length k from s to t and for this we need to guess $O(k \log n)$ bits, where n is the number of vertices, and we need $O(\log k + \log n)$ bits to keep track of the number of steps we have guessed and the current position.

For hardness, let (Q, κ) be a parameterized problem that is contained in paraβL via a machine M. Using standard techniques, we may assume that M has exactly one accepting and one rejecting configuration, respectively. Furthermore, we can assume that the configuration graph is a directed acyclic graph. The configuration graph of M on an input x has the maximum size $\exp(f(\kappa(x)) + O(\log n)) = f'(\kappa(x)) \cdot n^c$ for

some function f' and some constant c. In order to transform the configuration graph into an instance for p-DISTANCE, we must tackle the problem that the length of the path from the initial configuration to the accepting configuration is at most the size of the configuration graph and, therefore, not exclusively bounded by the parameter, but also by the length of the input x. However, the number of nondeterministic steps of M is bounded by $f(\kappa(x)) \cdot O(\log n)$ for every path from the initial configuration, so there are at most this many nodes with out-degree greater than 1. Let us call the nodes with out-degree greater than 1, as well as the nodes corresponding to the initial and the accepting configuration, the *red nodes* and the other nodes the *black nodes* of the graph.

In a first processing step, we iterate over the nodes of the configuration graph and replace every outgoing edge of every red node u_r that points to a black node u_b by a new edge from u_r to the first red node v_r that is reachable from u_b. Now we drop the black nodes. In the resulting graph, every path from the initial to the final configuration has length at most $f(\kappa(x)) \cdot O(\log n)$—which is still not bounded exclusively by the parameter. In a second step, we shorten the paths in the resulting graph by augmenting the graph with edges from node u to a node v if there is a path from u to v of length at most $O(\log n)$. Since the out-degree of the nodes is bounded by some constant that only depends on M (namely on the maximum number of different states that can be reached in one nondeterministic step), every path of length $O(\log n)$ can be described by $O(\log n)$ bits. The augmentation can then be done by an iteration over all the nodes v of the graph and enumerating all paths of length $O(\log n)$ starting from v. In the resulting graph, there is a path of length $f(\kappa(x))$ from the initial configuration to the accepting configuration if, and only if, there is a path from the initial configuration to the accepting configuration in the original configuration graph. Hence, we obtain the desired parametric logspace reduction, as the processing steps can be done by para-L Turing machines. □

The above theorem has an interesting corollary: The distance problem for *undirected* graphs is *also* complete for paraβL. This is in analogy to the fact that the distance problem is NL-complete both for directed and for undirected graphs.

Corollary 3.15 p-UNDIRECTED-DISTANCE *is complete for paraβL with respect to pl-reductions.*

Proof We reduce p-DISTANCE to p-UNDIRECTED-DISTANCE using a *layering trick*: For a given directed graph $G = (V, E)$ and a given number k, we output $k + 1$ copies $V_0, \dots, V_k$ of the vertex set V with $V_i = \{(v, i) \mid v \in V\}$. In the edge set we output, there is an (undirected) edge from $(u, i - 1)$ to (v, i) if, and only if, $(u, v) \in E$ or if $u = v$. Now, since a path in the new graph that "goes back a layer" is longer by at least 2 than a path that does not, there is a path of length k from $(s, 0)$ to (t, k) in the new graph if, and only if, there is a path of length k from s to t in G. □

The class paraβL has been studied independently by Chen and Müller [7] in the context of homomorphism and embedding problems together with "*jump machines*". Their results can be rephrased in our terminology as follows:

Fact 3.16 ([7]) Both the homomorphism problem and the embedding problem of a structure A into a structure B where A is a directed path, a directed cycle, or an undirected cycle, parameterized via the size of A, are complete for paraβL under pl-reductions.

Another interesting observation made by Chen and Müller is that the *exact* distance problem (where we ask whether the distance from s to t in a graph is exactly k for a parameter k and not "at most k") also lies in paraβL. (Note that just "guessing" a path from s to t of length k does not suffice.) Since it is easily seen that the hardness argument in the proof of Theorem 3.14 also works for the exact distance problem, we get the following:

Fact 3.17 ([7]) p-EXACT-DISTANCE is complete for paraβL with respect to pl-reductions.

4 Problems Complete for Time–Space Classes

In classical complexity theory, the major complexity classes are either defined in terms of time complexity (P, NP, EXP) or in terms of space complexity (L, NL, PSPACE), but not both at the same time: by the well-known inclusion chain L $\subseteq$ NL $\subseteq$ P $\subseteq$ NP $\subseteq$ PSPACE $=$ NPSPACE $\subseteq$ EXP, space and time are intertwined in such a way that bounding either automatically bounds the other in a specific way (at least for the major complexity classes).

In parameterized complexity theory, the classes para-P and XL appear to be incomparable: Machines for the first class may use $f(\kappa(x)) \cdot |x|^{O(1)}$ time and as much space as they want (which will be at most $f(\kappa(x)) \cdot |x|^{O(1)}$), while machines for the second class may use $f(\kappa(x)) \cdot O(\log |x|)$ space and as much time as they want (which will be at most $|x|^{f(\kappa(x))}$). A natural question arises: Which problems are in the intersection para-P $\cap$ XL or – even better—in the class D[f poly, f log], defined rigorously in a moment, of problems for which there is a *single* machine using only fixed-parameter time and slice-wise logarithmic space simultaneously?

As in the previous section, our exposition starts with definitions of the classes. It is easy to find artificial problems that are complete for them and we present some of them. We then move on to automata problems, but still some ad hoc restrictions are needed to make the problems complete for time–space classes. The real challenge lies in finding problems together with *natural* parameterizations that are complete. We present one such problem: the longest common subsequence problem parameterized by the number of strings. We show that LCS parameterized by the number of input strings is complete for N[$f$poly, f log], the nondeterministic version of the class D[f poly, f log].

Finally, we also show that the feedback vertex set problem lies in D[f poly, f log], while the directed feedback vertex set does not, unless L $=$ NL.

4.1 Classes and Structural Results

Definition 4.1 For a space bound s and a time bound t, both of which may depend on a parameter k and the input length n, let D$[t, s]$ denote the class of all parameterized problems that can be accepted by a deterministic Turing machine in time $t(\kappa(x), |x|)$ and space $s(\kappa(x), |x|)$. Let N$[t, s]$ denote the corresponding nondeterministic class.

Four cases are of interest: First, D[$f$poly, f log], meaning that $t(k, n) = f(k) \cdot n^{O(1)}$ and $s(k, n) = f(k) \cdot O(\log n)$, contains all problems that are "fixed-parameter tractable via a machine needing only slice-wise logarithmic space," and, second, the nondeterministic counterpart N[$f$poly, f log]. The other cases are D[n^f, $f$poly] and N[n^f, $f$poly], which contain problems that are in "slice-wise polynomial time via machines that need only fixed-parameter polynomial space." See Fig. 1 for the inclusions between the classes.

Our only structural result on the above classes is that they do, indeed, have complete problems. To find such problems, a good starting point is typically some variant of Turing machine acceptance (or halting):

Problem 4.2 (DETERMINISTIC-SPACE-BOUNDED-COMPUTATION (DSC))

Instance: (The code of) a single-tape machine M and a number s in unary.
Question: Does M accept on an initially empty tape using at most s tape cells?

Problem 4.3 (TIMED-DSC)

Instance: (The code of) a single-tape machine M, two numbers s and t in unary.
Question: Does M accept on an initially empty tape using at most s tape cells and making at most t steps?

Naturally, there are also nondeterministic variants NSC and TIMED-NSC.

Cai et al. [6] have observed that the fpt-reduction closure of p_t-TIMED-NSC (that is, the problem parameterized by t) is exactly W[1]. In analogy, Guillemot [15] proposed the name "WNL" for the fpt-reduction closure of p_s-TIMED-NSC (now parameterized by s rather than t). As pointed out in Sect. 3, we believe that this name should be reserved for the class resulting from applying the operator $\exists^{\leftrightarrow}_{f \log}$ to the class NL. Furthermore, the following theorem shows that p_s-TIMED-NSC is better understood in terms of time–space classes:

Theorem 4.4 *1.* p_s-DSC *is pfo-complete for* $D[\infty, f \log] = XL$.
2. p_s-NSC *is pfo-complete for* $N[\infty, f \log] = XNL$.
3. p_s-TIMED-DSC *is pfo-complete for* $D[f \text{ poly}, f \log]$.
4. p_s-TIMED-NSC *is pfo-complete for* $N[f \text{ poly}, f \log]$.

Proof For membership, for the first item, on input of a machine M and a space bound s in unary, an XL-machine can simulate M, making sure that no more than s tape cells are used. Clearly, the space needed to store the s tape cells is $O(s \log n)$ since $O(\log n)$ bits suffice to store the contents of a tape cell (the amount needed is not $O(1)$ since the tape alphabet is part of the input). For the third item, we have an additional time bound in the input and we only simulate the machine for t steps. Clearly, this

simulation only takes polynomial time. The nondeterministic cases work in the same way.

For hardness of the first item, first consider a problem $(Q, \kappa) \in \text{XL}$ via some machine M. Let $s_M(k, n)$ be the space bound of M. The reduction must now map inputs x to a pair $(M', 1^s)$. The reduction faces two problems: First, while M has an input tape and a work tape, M' has no input tape and starts with the empty string. Second, s cannot be set to $s_M(\kappa(x), |x|)$ since this only lies in $O\big(f(\kappa(x)) \cdot \log |x|\big)$ for some function f—while in a parameterized reduction the new parameter may only depend on the old one ($\kappa(x)$) and not on the input length. The first problem can be overcome using a standard trick: M' simulates M and uses its tape to store the contents of the work tape of M. Concerning the input tape (which M' does not have), when M accesses an input symbol, M' has this symbol "stored in its state," which means that there are $|x|$ many copies of M's state set inside M', one for each possible position of the head on the input tape. A movement of the head corresponds to switching between these copies. In each copy, the behaviour of the machine M for the specific input symbol represented by this copy is hard-wired. The second problem can also be tackled using standard tricks. Instead of mapping to M', we actually map to a new machine M'' that implements a space compression trick: For each $\lceil \log_2 |x| \rceil$ many tape cells of M', the machine M'' uses only one tape cell that also stores the position of the head. This can be achieved by enlarging the tape alphabet of M': If the old alphabet was Γ, we now use $\Gamma^{\lceil \log_2 |x| \rceil} \times \{1, 2, 3, \dots, \lceil \log_2 |x| \rceil\}$, which is still polynomial in $|x|$. Naturally, we now have to adjust the transitions and states of M' so that a step of M' for its old tape is now appropriately simulated by one step of M'' for its compressed tape. Taking it all together, we map x to (M'', s) where $s = s_M(\kappa(x), x)/\lceil \log_2 |x| \rceil$, which is bounded by a function depending only on $\kappa(x)$. Clearly, the reduction is correct.

For the hardness of the third item, where a time bound is given, we modify the above argument only very slightly: Now, we have $(Q, \kappa) \in \text{D}[f\text{poly}, f\log]$ via some machine M, a time bound $t_M(k, n)$ and a space bound $s_M(k, n)$. The reduction can now map inputs x to triples $(M'', 1^t, 1^s)$ in exactly the same way as before, only we now also set a time bound t to $t_M(k, n)$. Note that the new time bound is not a parameter and, therefore, may depend on the input length.

The hardness for the nondeterministic cases works in the same way. □

4.2 Natural Problems Complete for Parameterized Time–Space Classes

Our main objective for the present section is to show that the longest common subsequence problem parameterized by the number of strings is complete for $\text{N}[f\text{poly}, f\log]$. For the rather complex proof we first need to establish the completeness of other problems such as the acceptance problem for cellular automata parameterized in the number of cells.

4.2.1 Automata

A classical result of Hartmanis [16] states that L contains exactly the languages accepted by finite multihead two-way automata. A natural question to ask in light

of this result is how, exactly, the number of heads of an automaton influences its power. Phrased in terms of parameterized complexity theory, we are interested in the complexity of the following problem:

Problem 4.5 (p-MULTIHEAD-DFA-ACCEPTANCE (p-MDFA))

Instance: (The code of) a deterministic multihead two-way automaton M and a word x.
Parameter: The number h of heads of M.
Question: Does M accept x?

To prepare the field for arguments in later proofs, we also have a look at the "timed" variant p-TIMED-MDFA, where a time bound 1^t is given as part of the input and the question is whether M accepts x making at most t steps. Also, we can consider the nondeterministic variants p-MNFA and p-TIMED-MNFA.

Theorem 4.6 *1. p-MDFA is pfo-complete for $D[\infty, f \log] = XL$.*
2. p-MNFA is pfo-complete for $N[\infty, f \log] = XNL$.
3. p-TIMED-MDFA is pfo-complete for $D[f \text{ poly}, f \log]$.
4. p-TIMED-MNFA is pfo-complete for $N[f \text{ poly}, f \log]$.

Proof We only consider the deterministic cases since the proofs for the nondeterministic versions work in the same way.

Let us first quickly review how Hartmanis in [16] showed that $A \in L$ holds if, and only if, there is a deterministic multihead automaton that decides A. The basic idea is that the position of an automaton's head on the input tape can be used to store a number between 0 and n, where n is the input tape length, and, thus, the position of a head can store up to $\lfloor \log_2 n \rfloor$ bits of information. A fixed number of heads hence suffice to store the information of the work tape of a machine M using $c \cdot \lfloor \log_2 n \rfloor$ space. Modifications of this work tape correspond to an elaborate "dance" of auxiliary heads to compute the right number of steps to be made by one of the c heads. For instance, changing the ith bit of a block of $\lfloor \log_2 n \rfloor$ bits from 0 to 1 involves advancing the head corresponding to this block by 2^i many symbols. The exact details of the construction will not be important for proving the theorem; it suffices to note that for each step of the logspace Turing machine M the automaton will make a polynomial number of steps.

Now, to prove membership of the first item, just observe that p-MDFA can clearly be decided in space $O(h \log n)$, where h is the number of heads, by just simulating the input automaton. Similarly, for the third item, p-TIMED-MDFA can be decided in the same space and in time polynomial in the input since we only need to simulate the automaton for t steps and t is given in unary.

To prove hardness for the first claim, let $(Q, \kappa) \in XL$ via a machine needing space $f(\kappa(x)) \cdot O(\log |x|)$. Hartmanis's argument now shows that there is a multihead two-way automaton deciding Q whose number of heads depends only on $f(\kappa(x))$. Thus, a pfo-reduction can simply map the input x to this automaton together with x. To prove hardness for the third claim, let $(Q, \kappa) \in D[f \text{ poly}, f \log]$ via some M. We construct the same automaton as before; only this time we also impose a time constraint on it. Since, as we pointed out earlier, each step of the machine M results in a polynomial

number of steps of the automaton and since, with some extra effort, we can ensure that it is always the same number of steps, we can compute the exact number t of steps that the automaton may make in order to simulate $t_M(h, n)$ steps of the machine M. □

Another, rather natural kind of automata are *cellular automata,* where there is one instance of the automaton (called a *cell*) for each input symbol. The cells perform individual synchronous computations, but "see" the states of the two neighbouring cells (we only consider one-dimensional automata, but the results hold for any fixed number of dimensions). Formally, the transition function of such an automaton is a function $\delta\colon Q^3 \to Q$ (for the cells at the left and right end this has to be modified appropriately). The "input" is just a string $q_1 \dots q_k \in Q^*$ of states and the question is whether k cells started in the states q_1 to q_k will arrive at a situation where one of them is in an accepting state (one can also require all to be in an accepting state, this makes no difference).

Let DCA be the language $\{(C, q_1 \dots q_k) \mid C$ is a deterministic cellular automaton that accepts $q_1 \dots q_k\}$. Let NCA denote the nondeterministic version and let TIMED-DCA and TIMED-NCA be the versions where a time bound is given as part of the input. The following theorem states the complexity of the resulting problems when we parameterize by the number of cells:

Theorem 4.7 *1.* p_{cells}-DCA *is pfo-complete for* $D[\infty, f \log] = XL$.
2. p_{cells}-NCA *is pfo-complete for* $N[\infty, f \log] = XNL$.
3. p_{cells}-TIMED-DCA *is pfo-complete for* $D[f \text{ poly}, f \log]$.
4. p_{cells}-TIMED-NCA *is pfo-complete for* $N[f \text{ poly}, f \log]$.

Proof We start with membership. Clearly, p_{cells}-DCA lies in XL since we can keep track of the k states of the k cells in space $O(k \log n)$ and just as clearly p_{cells}-TIMED-DCA $\in$ D$[f \text{ poly}, f \log]$ since we only need to simulate the given number of steps. The arguments for the nondeterministic versions are the same.

For hardness in the first case, we reduce p_s-DSC to p_{cells}-DCA, which proves the first claim by Theorem 4.4. The input for the reduction is a pair $(M, 1^s)$. We must map this to some cellular automaton C and an initial string of states. The obvious idea is to have one automaton cell for each tape cell that can be used by M. In detail, let Q be the set of states of M and let Γ be the tape alphabet of M. The state set of C will be $R = (Q \cup \{\bot\}) \times \Gamma$, where $\bot$ is used to indicate that the head is elsewhere. Clearly, a state string from R^s allows us to encode a configuration of M. Furthermore, we can now set up the transition relation of C in such a way that one parallel step of the s automata cells corresponds exactly to one computational step of M: as an example, suppose in state q for the symbol a the machine M will not move its head, write b, and switch to state q'. Then in C for every $x, y \in \{\bot\} \times \Gamma$ there would be a transition mapping $(x, (q, a), y)$ to (q', b) and also transitions mapping $((q, a), x, y)$ to x and $(x, y, (q, a))$ to y. For triples corresponding to situations that "cannot arise" like the head being in two places at the same time, the transition function can be set up arbitrarily. The initial string of states for the cellular automaton is of course $(q_0, \Box)(\bot, \Box) \dots (\bot, \Box)$, where $\Box$ is the blank symbol and q_0 is the initial state of M. Now, with this setup the strings of states of the cellular automaton are in one-to-one correspondence with the configurations of M. In particular, we will reach a state string

containing an accepting state if, and only if, M accepts when started with an empty tape. Clearly, the reduction is a pfo-reduction.

One might expect that one can use the exact same argument for nondeterministic automata and simply use the same reduction, but starting from p_s-NSC. However, there is a complication: The cells work independently of one another. In particular, there is no guarantee that a nondeterministic decision taken by one cell is also taken by a neighboring cell. To illustrate this point, consider the situation where the machine M can nondeterministically step "left or right" in some state q. Now assume that some cell c is in state (q, x) and consider the left and right neighboring cells c_{left} and c_{right}, respectively. For both of them, there would now be a transition allowing them to "take over the head" and both could nondeterministically decide to do so—which is wrong, of course; only one of them may receive the head.

To solve this problem, we must ensure that a nondeterministic decision is taken "by only one cell." Towards this aim, we first modify M, if necessary, so that every nondeterministic decision is a binary decision. Next, we change the state set of C: Instead of $(Q \cup \{\bot\}) \times \Gamma$ we use $(Q \times \{0, 1\} \cup Q \cup \{\bot\}) \times \Gamma$. In other words, in addition to the normal states from Q we add two copies of the state set, one tagged with 0 and one tagged with 1. The idea is that when a cell is in state $(q, x) \in Q \times \Gamma$, it can nondeterministically reach $((q, 0), x)$ or $((q, 1), x)$ if the transition function of M admits a nondeterministic choice in state q reading symbol x. However, from those states, we can *deterministically* make the next step: if the state is tagged by 0, both the cell and the neighboring cells continue according to what happens for the first of the two possible nondeterministic choices, if the state is tagged by 1, the other choice is used. Note that as long as the state is not yet tagged, the neighboring cells do not change their state. With these modifications, we arrive at a new cellular automaton with the property that after every two computational steps of the automaton its string of states encodes one of the two possible next computational steps of the machine M. This shows that the reduction is correct.

It remains to consider the timed automata. Here, the reduction is now from the timed versions p_s-TIMED-DSC and p_s-TIMED-NSC. In the first case, we can simply copy the time bound and in the second case we double it (since we doubled the state set). Otherwise, the reductions are the same. □

In a moment, we will connect cellular automata and the above theorem with the longest common subsequence problem. Another application that we would first like to mention is Fact 4.8 below on *pebble games* since its proof is also based on a reduction from the different versions of cellular automata acceptance. Pebble games are played on graphs on whose vertices we place pebbles (a *pebbling* is thus a subset of the set of vertices) and, over time, we (re)move and add pebbles according to different rules. Depending on the rules and the kind of graphs, the resulting computational problems are complete for different complexity classes, which is why pebble games have received a lot of attention in the literature.

For our purposes, the following single player pebble game is of interest: A *threshold pebble game* (TPG) consists of a directed graph $G = (V, E)$ together with a threshold function $t\colon V \to \mathbb{N}$. Given a pebbling $X \subseteq V$, a vertex v *can be pebbled after* X if the number of v's pebbled predecessors is at least v's threshold, that is, $\bigl|\{\, p \mid (p, v) \in$

$E\} \cap X\big| \geq t(v)$. Given a pebbling X, *a next pebbling* is any set Y of vertices that can be pebbled after X. The *maximum next pebbling* is the inclusion-maximal such Y. The language TPG contains all threshold pebble games together with two pebblings S and T such that we can reach T when we start with S and apply the next pebbling operation repeatedly (always replacing the current pebbling X completely by Y). For the TPG-MAX problem, Y is always chosen as the maximum next pebbling (which makes the game deterministic). For ACYCLIC-TPG and ACYCLIC-TPG-MAX, the graph must be acyclic. We parameterize by the maximum number of pebbles that may be present in any step.

Fact 4.8 ([22,23])

1. p_{pebbles}-TPG-MAX is pfo-complete for $\text{D}[\infty, f \log] = \text{XL}$.
2. p_{pebbles}-TPG is pfo-complete for $\text{N}[\infty, f \log] = \text{XNL}$.
3. p_{pebbles}-ACYCLIC-TPG-MAX is pfo-complete for $\text{D}[f\text{ poly}, f \log]$.
4. p_{pebbles}-ACYCLIC-TPG is pfo-complete for $\text{N}[f\text{ poly}, f \log]$.

4.2.2 Longest Common Subsequence

The input for the LCS problem is a set S of strings over some alphabet Σ together with a number l. The question is whether there is a string $c \in \Sigma^l$ that is a *subsequence* of all strings in S, meaning that for all $s \in S$ just by removing symbols from s we arrive at c.

There are several natural parameterization of LCS: We can parameterize by the number of strings, by the size of the alphabet, by the length of the sought common subsequence, or any combination thereof. Guillemot has shown [15] that $p_{|S|,l}$LCS is fpt-complete for W[1], while $p_{|S|}$-LCS is fpt-equivalent to p_s-TIMED-NSC. Hence, by Theorem 4.4, both problems are complete under fpt-reductions for the fpt-reduction closure of $\text{N}[f\text{ poly}, f \log]$. We tighten this in Theorem 4.11 below using pfo-reductions. (Using weaker reductions is more than a technicality: $\text{N}[f\text{ poly}, f \log]$ is presumably not even closed under fpt-reduction, while it *is* closed under pfo-reductions). We remark that Guillemot's fpt-reductions appear to be "upgradeable" to pfo-reductions, but one would have to walk through the proof in detail to verify this—and his arguments deviate quite a bit from ours. Nevertheless, we believe that Guillemot's chain of reductions can be used to give a different proof of Theorem 4.11.

As a preparation for the proof of Theorem 4.11, we first present a simpler-to-prove result. Let LCS-INJECTIVE denote the restriction of LCS where all input words must be *p-sequences* [12], which are words containing any kind of symbol at most once (the function mapping word indices to word symbols is injective). We give the following proof to illustrate an idea used in the proof of Theorem 4.11.

Theorem 4.9 LCS-INJECTIVE *is NL-complete and this holds already under the restriction* $|S| \leq 4$.

Proof The problem LCS-INJECTIVE lies in NL via the following algorithm: We guess the common subsequence c nondeterministically and use a counter to ensure that c has length at least l. The problem is that we cannot remember more than a fixed number

of letters of c without running out of space. Fortunately, we do not need to: We always only keep track of the last two guessed symbols. For each such pair (a, b), we check whether a appears before b in all strings in S. If so, we move on to the next pair, and so on. Clearly, this algorithm needs only logarithmic space and correctly decides LCS-INJECTIVE.

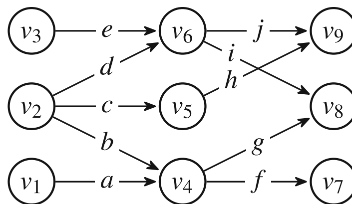
To prove hardness for $|S| = 4$, we reduce from the NL-complete language LAYERED-REACH, where the input is a layered graph G (each vertex is assigned a layer number and all edges only go from one layer to the next), the source vertex s is the (only) vertex on layer 1, and the target t is the (only) vertex on the last layer m. The question is whether there is a path from s to t.

For the reduction to LCS-INJECTIVE we introduce a symbol for each edge of G. The common subsequence will then be exactly the sequence of edges along a path from s to t. We consider the layers L_1, L_2, ..., L_m in order and, for each of them, append edge symbols to the four strings as described in the following.

Consider a layer L_i, containing vertices $\{v_1, \dots, v_n\}$. Assume i is odd. We go over the vertices v_1 to v_n in that order. For v_1, first consider all edges that end at v_1. They must come from layer $i - 1$. We add these edges in some order to the first string (for instance, in the order of the index of the start vertex of these edges). Still considering v_1, we then consider all outgoing edges and append them in some fixed order. Then we move on to v_2 and add edge symbols in the same way for it, and so on. If i is even rather than odd, we add the same edge symbols to the third rather than to the first string.

For the second (or, for even i, the fourth string), we go over the vertices in decreasing order. We start with v_n. We consider the incoming edges for v_n and add them to the second string, but in reverse order compared to the order we used for the first string. Next, we append the outgoing edges, again in reverse order. Then we consider v_{n-1} and proceed in the same way.

As an example, consider the following layered graph:



This would result in the following strings, where spaces have been added for clarity and also the symbols v_i, which are not part of the strings (so the second string is actually $edcbajhigf$):

$$\begin{array}{llllll} v_1a & v_2bcd & v_3e & fv_7 & giv_8 & hjv_9 \\ v_3e & v_2dcb & v_1a & jhv_9 & igv_8 & fv_7 \end{array}$$

$$abv_4fg \quad cv_5h \quad dev_6ij$$
$$edv_6ji \quad cv_5h \quad bav_4gf$$

We make two crucial observations. First, if an edge is included in the common subsequence, no other edge starting at the same layer can be included also: The edge symbols of one layer come in one order in the first (or third) string and in the reverse order in the second (or fourth) string. Thus, there cannot be two of them in the common subsequence. For the same reason, there can only be one edge arriving at a layer in the common subsequence. The second crucial observation is that if the sequence contains an edge e arriving at a vertex v, it can only contain edges leaving from vertex v, if it contains any edge leaving from v's layer: Only the edges leaving v will come after e in both the first and second (or third and fourth) string.

Putting it all together, we get the following: There is a path from s to t in G if, and only if, there is a common subsequence of length $m - 1$ in the constructed strings: If there is a path, the sequence of the edges on it form a subsequence; and if there is such a subsequence, because of its length, it must contain exactly one edge leaving from each layer except the last—and these edges must form a path as we just argued. □

Corollary 4.10 $p_{|S|}$-LCS-INJECTIVE *is pfo-complete for para-NL* $= N[f\text{poly}, f + \log]$.

Proof The problem lies in para-NL since by Theorem 4.9 we can solve any instance in NL without even using the parameter. On the other hand, the theorem also shows that a slice of the parameterized problem (namely for 4 strings) is already hard for NL. It is a well-known fact that in this case the parameterized problem is hard for the corresponding para-class, which happens to be para-NL. □

Theorem 4.11 $p_{|S|}$-LCS *is pfo-complete for* $N[f\text{poly}, f\log]$.

Proof Clearly, $p_{|S|}$-LCS $\in$ N$[f\text{poly}, f\log]$ since a nondeterministic machine can guess the common subsequence on the fly and only needs to keep track of k pointers into the strings, which can be done in space $O(|S|\log n)$. To prove hardness, we reduce from the N$[f\text{poly}, f\log]$-complete problem p_{cells}-TIMED-NCA, the acceptance problem for timed nondeterministic cellular automata, which we treated in Theorem 4.7.

Our first step is to tackle the problem that in an LCS instance we choose "one symbol after the other" whereas in a cellular automaton all cells make one step in parallel. To address this, we introduce a new intermediate problem p_{cells}-TIMED-NCA-SEQUENTIAL where the model of computation of the cellular automaton is modified as follows: Instead of all k cells making one parallel step, initially only the first cell makes a transition, then the second cell makes a transition (seeing already the new state reached at the first cell, but still the initial state of the third cell), then the third cell (seeing the new state of cell two and the old of cell four), and so on up to the kth cell. Then, we begin again with the first cell, followed by the second cell, and so on.

Claim p_{cells}-TIMED-NCA pfo-reduces to p_{cells}-TIMED-NCA-SEQUENTIAL.

Proof (of the claim) The trick is to have cells "remember" the states they were in: On input of $(C, q_1 \dots q_k)$, we construct a "sequential" cellular automaton C' as follows.

If Q is the state set of C, the state set of C' is $Q \times Q$. Each state $q \in Q'$ is now a pair $(q^{\text{previous}}, q^{\text{current}})$. The transition relation is adjusted as follows: If there used to be a transition $(q_{\text{left}}, q_{\text{old}}, q_{\text{right}}, q_{\text{new}}) \in Q^4$, meaning that a cell of the parallel automaton C can switch to state q_{new} if it is in state q_{old}, its left neighbor is in state q_{left}, and its right neighbor is in state q_{right}, the we now have the following transitions in C': $((q_{\text{left}}, x), (y, q_{\text{old}}), (z, q_{\text{right}}), (q_{\text{old}}, q_{\text{new}}))$ where $x, y, z \in Q$ are arbitrary. Indeed, this transition will switch a cell's state based on the *previous* state of the cell before it and on the *current* state of the cell following it and will store that previous state. For the first and last cells, this construction is adapted in the obvious manner. Clearly, the resulting sequential automaton will arrive in a sequence $(x_1, q_1) \dots (x_k, q_k)$ of states for some $x_i \in Q$ after $t \cdot k$ steps if, and only if, the original automaton arrives in states $q_1 \dots q_k$ after t steps. This proves the reduction. □

We now show how p_{cells}-TIMED-NCA-SEQUENTIAL can be reduced to $p_{|S|}$-LCS. Before we plunge into the details, let us first outline the basic idea: Each cell of a cellular automaton "behaves somewhat like a layered reachability problem" in the sense that if we create t many copies of the state set, we must now find out whether the automaton will arrive in the accepting state starting from the initial state. Thus, as in the proof of Theorem 4.9, we use four strings to represent a cell of the automaton, giving a total of $4k$ strings, where k is the number of cells. However, the cells do not act independently; rather each step of a cell depends on the states of the two neighboring cells. Fortunately, this "control" effect can be modelled by adding an "edge's" symbol (actually, a transition's symbol) not only to the four strings of the cell, but also to the four strings of predecessor and successor cells at the right position (namely "before the required state symbol"). In the following, we explain the idea just outlined in detail.

Let $(C, q_1 \dots q_k)$ and a time bound be given as input for the reduction. We may assume that the time bound is of the form $t \cdot k$ (because our reduction in the proof of the above claim yielded such a bound) and, thus, the steps of C can be grouped into t many "major steps," each consisting of k sequential steps ("minor steps") taken by cells 1 to k in that order. By modifying C, if necessary, we may assume that C makes exactly $t \cdot k$ sequential steps when it accepts the input and, otherwise, makes strictly less steps. We use s to denote a major step number.

Construction of the Strings. We map $(C, q_1 \dots q_k)$ to $4k$ strings $s_1^1, s_2^1, s_3^1, s_4^1, \dots, s_1^k, s_2^k, s_3^k, s_4^k$ and ask whether they have a common subsequence of length $t \cdot k$. Each group of four strings is set up similarly to the four strings from the proof of Theorem 4.9: s_1^i and s_2^i model the states (vertices) the ith cell has just before odd major steps s; and s_3^i and s_4^i model the states the cell has before even major steps s.

Consider cell i and its four strings s_1^i to s_4^i. Recall that in Theorem 4.9 we conceptually added the vertices of the first layer in opposite orders to s_1^i and s_2^i, although in reality these vertices were not part of the final strings and were added to make it easier to explain where the actual symbols (the edges) were placed in the strings. In our setting, the role of the vertices on the first layer is taken by the states $Q = \{q_1, \dots, q_n\}$ of the automaton C tagged by the major step number 1. Thus, s_1^i starts (conceptually) with $(q_1, 1) \dots (q_n, 1)$ and s_2^i starts with $(q_n, 1) \dots (q_1, 1)$. Next come tagged versions of the states just before the third major step, so s_1^i continues $(q_1, 3) \dots (q_n, 3)$ and s_2^i

with $(q_n, 3) \ldots (q_1, 3)$. We continue in this way for all odd major steps. For even major steps, we add analogous strings to s_3^i and s_4^i.

Continuing the idea from Theorem 4.9, we now add "edges" to the strings. However, instead of an edge from one vertex so another, the transition relation of a cellular automaton contain 4-tuples $f = (f_{\text{left}}, f_{\text{old}}, f_{\text{right}}, f_{\text{new}}) \in Q^4$ of states, which allows a cell to switch to state f_{new} when it was in state f_{old} and its left neighbor was in state f_{left} and the right neighbor was in state f_{right}. Recall that in Theorem 4.9, for each e from some vertex a on an odd layer to a vertex b, we added the symbol e *after* a in the first two strings and *before* b in the last two strings. In a similar way, for the cellular automaton for each 4-tuple f we add new "symbols" (f, s, i) consisting of a transition, a major step number, and a cell index i to the strings. This symbol is added at several places to the strings (we assume that s is odd; for even s exchange the roles of the first two and the last two strings everywhere); sometimes even more than once. The rules are as follows:

1. Iterate over all (f, s, i) in some order and insert (f, s, i) directly after (f_{old}, s) in s_1^i.
2. Next, again iterate over all (f, s, i), but now in reverse order, and insert (f, s, i) after (f_{old}, s) in s_2^i.

Note that using the two opposite orderings, as in Theorem 4.9, for each (f_{old}, s) at most one (f, s, i) can be part of a common subsequence.

3. Next, iterate over all (f, s, i) in some order and insert (f, s, i) directly before $(f_{\text{new}}, s + 1)$ in s_3^i.
4. Next, iterate over all (f, s, i) in reverse order and insert (f, s, i) directly before $(f_{\text{new}}, s + 1)$ in s_4^i.

The effect of the above is to make the automaton switch to f_{new} in cell i after major step s. Now, we still need to ensure that this switch is only possible when the preceding cell has already switched to state f_{left} after step s and the next cell is in state f_{right} before step s.

5. Next, iterate over all (f, s, i) and insert (f, s, i) directly after $(f_{\text{left}}, s + 1)$ in s_3^{i-1} and s_4^{i-1}. For $i = 1$, no symbols are added.
6. Next, iterate over all (f, s, i) and insert (f, s, i) directly after (f_{right}, s) in s_1^{i+1} and s_2^{i+1}. For $i = k$, no symbols are added.

Note that since the last two steps are applied later, the added symbols are "nearer" to the state symbols than the symbols added in the first two steps. In particular, a common subsequence can contain first a symbol added in step 6 added after some $(q, s + 1)$, then a symbol added after (q, s) in step 5, and then symbols added before or after (q, s) in one of the first four steps.

The last rule ensures that when a tuple (f, s, i) is not mentioned for a string by one of the first six rules, we can always make it part of a common subsequence:

7. Finally, iterate over the $4k$ strings. For each such string s_j^i, consider the set X of all (f, s, i) that are not present in s_j^i. Add all symbols of X in some fixed order tk times after each letter of s_j^i.

As the last step of the construction of the strings, in order to model the initial configuration $q_1 \dots q_k$ of the automaton, for each $i \in \{1, \dots, k\}$ in s_1^i to s_4^i we remove all symbols before $(q_i, 1)$.

Correctness: First Direction. Having finished the description of the reduction, we now argue that it is correct. For this, first assume that the automaton, does, indeed, accept the input sequence $q_1 \dots q_k$. By assumption, this means that the automaton will make $t \cdot k$ sequential steps. Assume that in major step s and minor step i the automaton makes transition $f^{s,i}$, meaning that the ith cell switches its state from $f_{\text{old}}^{s,i}$ to $f_{\text{new}}^{s,i}$.

We claim that $(f^{1,1}, 1, 1)(f^{1,2}, 1, 2) \dots (f^{1,k}, 1, k)(f^{2,1}, 2, 1) \dots (f^{t,k}, t, k)$ is a common subsequence of all s_j^i. To see this, consider the first symbol $(f^{1,1}, 1, 1)$. It will be present both in s_1^1 and s_2^1 since for the first transition the first cell was exactly in state $q_1 = f_{\text{old}}^{1,1}$ and, thus, this symbol *followed* $(q_1, 1)$ in the construction and was not removed in the last construction step. The symbol is also present in s_3^1 and s_4^1, namely right before the ("virtual") pair $(f_{\text{new}}, 2)$. The symbol will also be present in s_1^2 and s_2^2 since $q_2 = f_{\text{right}}^{1,1}$ and we added $(f^{1,1}, 1, 1)$ to both s_1^2 and s_2^2 in step 6. Finally, the symbol will be present in all other strings near the beginning because of step 7.

Next, consider the second symbol $(f^{1,2}, 1, 2)$, which corresponds to the second step the automaton has taken. Here, the second cell switches from $f_{\text{old}}^{1,2}$ to $f_{\text{new}}^{1,2}$ because the first cell has already switched to $f_{\text{left}}^{1,2} = f_{\text{new}}^{1,1}$ during the first transition and the third cell is still in $f_{\text{right}}^{1,2} = q_3$. Now, observe that in all strings $(f^{1,2}, 1, 2)$ does, indeed, come after $(f^{1,1}, 1, 1)$: For s_1^2 to s_4^2 this is because of steps 1 to 4. For s_3^1 to s_4^1, we have, indeed, $(f^{1,2}, 1, 2)$ following $(f^{1,1}, 1, 1)$ by step 5. For s_1^3 to s_2^3, the symbol $(f^{1,2}, 1, 2)$ is present by step 6. All other strings contain the symbol by step 7 near the beginning.

Continuing in a similar fashion with the other symbols, we see that the sequence $(f^{1,1}, 1, 1) \dots (f^{t,k}, t, k)$ is a common subsequence of all strings and it clearly has length $t \cdot k$.

Correctness: Second Direction. It remains to argue that if there is a common subsequence of the strings of length $t \cdot k$, then the automaton accepts the input. First observe that the common subsequence must be of the form $(f^{1,1}, 1, 1)(f^{1,2}, 1, 2) \dots (f^{1,k}, 1, k)\,(f^{2,1}, 2, 1) \dots (f^{t,k}, t, k)$. The reason is that for any two symbols (f, s, i) and (f', s', i') if $s < s'$ then the first of these symbols always comes before the second in all strings. The same is true if $s = s'$ and $i < i'$. Finally, for $s = s'$ and $i = i'$, the opposite orderings for the symbols in steps 1 and 2 (and, also, in steps 3 and 4) ensure that at most one of the two symbols can be present in a common subsequence. Thus, the indices stored in the symbols of the common subsequence must strictly increase and, since the length of the sequence is $t \cdot k$, all possible indices must be present.

We must now argue that the $f^{s,i}$ form a sequence of transitions that make the automaton accept. For this, we perform an induction on the length of an initial segment up to some symbol (f^{s_0,i_0}, s_0, i_0) of the common sequence. For each cell index i, let $f^i = (f^{s,i}, s, i)$ be the last symbol in the segment whose last component is i. Let $q^i = f_{\text{new}}^i$ or, if the segment is so short that there is no f^i, let q^i be the initial state q_i. The inductive claim is that after $(s_0 - 1) \cdot k + i_0$ steps of the automaton, the cells

will have reached exactly states $q^1, \ldots, q^k$. Clearly, this is correct at the start. For the inductive step, the crucial observation is that steps 1 to 6 guarantee that for $i_0 < k$ the only symbol $(f^{s_0,i_0+1}, s_0, i_0+1)$ that can follow (f^{s_0,i_0}, s_0, i_0) in a common sequence is one that makes that cell $i_0 + 1$ change its state according to the transition f^{s_0,i_0+1}. For $i_0 = k$, we similarly have that only symbols $(f^{s_0+1,1}, s_0 + 1, 1)$ can follow that make cell 1 change its state according to the transition $f^{s_0+1,1}$. □

4.3 The Complexity of the Feedback Vertex Set Problem

We conclude this section on parameterized time–space classes with new insights into the feedback vertex set problem:

Problem 4.12 (p-FVS **and** p-DFVS)

Instance:	An undirected (p-FVS) or directed (p-DFVS) graph $G = (V, E)$ and a natural number k.
Parameter:	k.
Question:	Is there a set $F \subseteq V$ with $\lvert F\rvert = k$ such that $G[V - F]$ is acyclic?

The undirected version was shown to be fixed-parameter tractable (lie in para-P) in 1993 by Bodlaender [1]. The fixed-parameter tractability of the directed version remained an open problem until Chen et al. [8] presented a para-P-algorithm in 2008. So far, these apparently different complexities could only be "felt" by looking at the complexity of the proofs.

Using the machinery of the present paper, we can actually draw a dividing line between the two problems. A first, easy observation is that p-FVS $\in$ paraWL holds: A W-machine can guess a feedback set F using $O(k \log n)$ bits and then test whether the remaining graph is acyclic—which is easily decidable in logarithmic space, see [17]. A somewhat deeper result is the below theorem that p-FVS $\in$ D[$f$poly, f log] holds. Note that we do not prove completeness of p-FVS for this class; in fact, in view of p-FVS $\in$ paraWL, completeness seems unlikely since it would imply a class collapse.

Apart from being a "nice" observation that p-FVS lies in a small subclass of para-P, we crucially also show that p-DFVS does not lie in this class, unless L = NL.

Theorem 4.13 p-FVS $\in$ $D[f\text{poly}, f\log]$.

Proof We adapt the algorithm of [10] for showing the fixed-parameter tractability of p-FVS in such a way that it uses little space, while still being "quick enough." One way that does *not* work is to cycle through all possible $\binom{|V|}{k}$ possible ways of choosing a feedback vertex set and then checking whether the remainder of the graph is acyclic. We have just enough space to store each choice and to perform the test, but would need $\Omega(n^k)$ time, which we are not allowed.

Let $G = (V, E)$ be an input graph and let k be the parameter. Let $n = |V|$. We first test whether G is acyclic (this can be done in logarithmic space) and accept, if so. Otherwise, we test whether G has a feedback vertex set of size 1 (again, this can be done in logarithmic space) and accept if $k \geq 1$. In the following, we may assume that the smallest feedback vertex set has size 2 and $k \geq 2$.

Our objective will be to identify vertices $v_1, v_2, \ldots$ that we can safely assume to be present in some minimal feedback vertex set. We will store these vertices in $O(k \log n)$ bits, which is permissible for the kind of machine we are looking for. In the following, whenever we refer to "G" we actually mean "G with the already chosen vertices removed." Note that for a logspace machine, "removing" the vertices actually means that we just "ignore" them whenever we iterate over the vertices of G during the later computations.

The first step is to make sure that all vertices of the graph have degree at least 3, so that we can apply a useful lemma, as we will discuss later. For this, we first need to get rid of all vertices of degree less than 2 and we need to do this "recursively" without using extra memory. The trick is to remove all vertices whose connected component is either a tree or becomes a tree when a single edge is removed somewhere in the graph. Clearly, this can be tested for all vertices in logarithmic space and we can (conceptually) remove all these vertices simultaneously, resulting in a graph in which all vertices have degree at least 2. Next, we (conceptually) "compress" all paths along vertices of degree 2 into single edges; that is, we remove all vertices of degree 2 from the graph and connect all vertices by an edge that were connected by such paths. Note that, as always in the context of logspace computations, we not actually "write down" the reduced graph, but rather recompute its bits whenever needed.

The result of "contracting" the degree-2-vertex paths of G will be a new graph that may have self-loops and also multiple edges between vertices. If there is a self-loop at a vertex v, we *know* that there is some minimal feedback vertex set containing v. We add v to the list of already chosen vertices and restart. Similarly, if there are two edges between two vertices u and v, we know that at least one of them must be in some minimal feedback vertex set. At this point, the computation branches in two subcomputations, one in which we choose u and one in which we choose v. This results in a standard search tree as one often encounters in fixed-parameter algorithms with a branching width of 2 and a depth of at most k. In particular, these branchings will multiply the total runtime by at most a factor of 2^k.

The tricky case arises when the graph has a minimum degree of 3, but does not contain self-loops or multi-edges. In this case, we can use the following fact [10]: If an undirected graph has minimum degree 3 and a feedback vertex set of size k, then it has girth at most $2k$. (The girth of a graph is the length of the smallest cycle.) Now, suppose we had a way of identifying such a smallest cycle. Then at least one of the $2k$ vertices would have to be present in some minimum feedback vertex set. This would allow us to branch into $2k$ subcomputations, in each of which we try out one of the vertices as the next vertex v_i that we store in our list, resulting in a search tree of size $(2k)^k$. This would result in a total runtime of $(2k)^k n^{O(1)}$ since all other computations only need polynomial time (and logarithmic space).

In the classical proof that p-FVS $\in$ para-P the argument now continues as follows: In order to find the cycle of length $2k$ (which we know must exists), we iterate over all vertices v. Starting form v, we do a breadth-first search for a distance up to k and mark the reached vertices. We stop the search whenever we rereach an already marked vertex because, then, we have found two different paths of length at most k from v to another vertex and, thus, a cycle of length at most $2k$. Note that for every vertex v on

a cycle of length at most $2k$ this search must succeed, so we will find a near-minimal cycle.

It remains to argue that the search can be implemented also by our machine, which may only use space $O(k \log n)$. The machine can easily iterate over all vertices v, but it can *not* each time mark the visited vertices (since the degree of the vertices is not bounded, the number of to-be-marked vertices may be arbitrarily large). Instead, we do the following:

We use two nested depth-first search loops. In the outer loop, for increasing depth $d = 1, 2, \dots, k$, we visit all vertices u at distance at most d using a stack of size $O(d \log n)$ to keep track of the path from v to the currently visited vertex. In an inner loop, which we run for each vertex u at distance d visited by the outer loop, we also do a depth-first search up to depth d using *another* stack of size $O(d \log n)$. We stop both loops when the path of inner loop and the path of the outer loop both lead to the same vertex v, but the paths differ. In this case, we have found a cycle of length at most $2k$. Also note that the algorithm is guaranteed to find such a cycle since when v lies on a cycle of length at most $2d$, the outer loop will, at some point, go to a vertex u using a different path from the one used in the inner loop.

The algorithm just sketched may look suspiciously like taking time $O(n^k)$ because iterating in a depth-first manner over a graph with n vertices up to a depth k may normally take time up to $O(n^k)$ (just think of a clique). However, in our case the time bound is actually $O(n^5)$: Since we break the loops whenever we have identified a vertex that can be reached in two different ways, we know that *until* the loops break, there is always a *unique* path from v to all visited vertices u. In particular, before the break, for each v up to $d - 1$, the outer loop (and hence also the inner loop) will make at most n steps and for the last d it will hence make at most n^2 steps. Since there are n possible start vertices and since we have two nested loops, we get a runtime of $O(n \cdot n^2 \cdot n^2)$. □

In contrast to the above, we have the following:

Theorem 4.14 *1.* p-DFVS $\in D[f\text{poly}, f\log]$ *implies L = NL.*
2. p-DFVS $\in$ *paraWL if, and only if, L = NL.*

Proof We first show that the parameterized problem p_0-UNREACH, the complement of the reachability problem where we ask whether there does *not* exist a path from a given vertex s to another vertex t in a directed graph parameterized by the trivial parameterization, reduces to p-DFVS. The reduction works in two steps: The first step takes the n-vertex input graph and makes it acyclic by producing n copies if it, conceptually arranging them as n layers from left to right, and drawing edges between consecutive layers from left to right in the same way as they are present in the original graph. The second step inserts an edge from t's copy in the last layer to s's copy in the first layer. The resulting directed graph remains acyclic (has a feedback vertex set of size 0) exactly if there is no path from s to t in the input graph.

Assume that p-DFVS $\in$ D[$f$poly, f log] holds. Then we also have p_0-UNREACH $\in$ D[$f$poly, f log]. However, since the latter problem is parameterized in the “trivial way,” an algorithm needing $f(\kappa(x)) \cdot O(\log|x|)$ space actually needs only $O(\log|x|)$ space, which proves that the NL-complete unreachability problem lies in L.

Next, assume that p-DFVS $\in$ paraWL. Similar to the argument just given, observe that the $f(\kappa(x)) \cdot O(\log |x|)$ bits "guessed" by a W-machine are actually just $O(\log |x|)$ bits and, thus, again, unreachability can be decided in deterministic logarithmic space. For the other direction, if L = NL, we have p-DFVS $\in$ paraWL since a W-machine can just guess a feedback vertex set of size k and then verify the acyclicity of the remaining graph in NL = L. □

5 Conclusion

Our purpose in the present paper was to show that the study of parameterized space and circuit classes helps in better understanding the complexity of natural parameterized problems. For a number of natural problems, whose exact complexity was previously unclear, we have presented classes for which they are complete under weak reductions.

Bounded nondeterminism plays a key role in parameterized complexity theory, lying at the heart of the definition of important classes like W[P], but also of W[1]. In the present paper we introduced a "W-operator" that cannot only be applied to P, yielding paraWP, but also to classes like NL or NC^1. We showed that "union versions" of problems complete for P, NL, and L tend to be complete for paraWP, paraWNL, and paraWL. Several important problems studied in parameterized complexity turn out to be union problems, including p-WEIGHTED-CIRCUIT-SAT and p-WEIGHTED-SAT, and we observed that the latter problem is complete for paraWNC^1. For the associative generability problem p-AGEN, which is also a union problem, we established its paraWNL-completeness. An interesting open problem is determining the complexity of the "universal" version of AGEN, where the question is whether *all* size-k subsets of the universe are generators. Possibly, this problem is complete for $\text{paraW}_\forall\text{NL}$.

We presented a number of problems that are complete for the time–space classes D[f poly, f log] and N[f poly, f log], the most prominent being the longest common subsequence problem. We shied away from presenting complete problems for the classes D[n^f, f poly] and N[n^f, f poly] because in their definition we need restrictions like "the machine may make at most n^k steps where k is the parameter." Such artificial parameterizations have been studied, though: In [14, Theorem 2.25] Flum and Grohe show that "p-EXP-DTM-HALT" is complete for XP. Adding a unary upper bound on the number of tape cells that the machine may use to the problem definition yields a problem easily seen to be complete for D[n^f, f poly]. Finding a *natural* problem complete for the latter class is, however, an open problem.

We demonstrated how the introduced classes can also shed a light on the complexity of problems that are not necessarily complete for them. We showed that the feedback vertex set problem lies both in D[f poly, f log] and paraWL while the directed version lies in neither, unless L = NL. It remains an open problem to identify a parameterized space class for which the feedback vertex set problem is actually complete.

References

1. Bodlaender, H.L.: On linear time minor tests with depth-first search. J. Algorithms **14**(1), 1–23 (1993). doi:10.1006/jagm.1993.1001

2. Buss, J., Goldsmith, J.: Nondeterminism within P*. SIAM J. Comput. **22**, 560–572 (1993)
3. Buss, S., Cook, S., Gupta, A., Ramachandran, V.: An optimal parallel algorithm for formula evaluation. SIAM J. Comput. **21**(4), 755–780 (1992). doi:10.1137/0221046
4. Buss, S.R.: The Boolean formula value problem is in alogtime. In: Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC 1987), ACM, pp. 123–131. (1987). doi:10.1145/28395.28409
5. Cai, L., Chen, J., Downey, R.G., Fellows, M.R.: Advice classes of parameterized tractability. Ann. Pure Appl. Log. **84**(1), 119–138 (1997a). doi:10.1016/S0168-0072(95)00020-8
6. Cai, L., Chen, J., Downey, R.G., Fellows, M.R.: On the parameterized complexity of short computation and factorization. Arch. Math. Log. **36**(4–5), 321–337 (1997b)
7. Chen, H., Müller, M.: The fine classification of conjunctive queries and parameterized logarithmic space complexity. In: PODS '13 Proceedings of the 32nd Symposium on Principles of Database Systems, ACM, pp. 309–320. (2013). doi:10.1145/2463664.2463669
8. Chen, J., Liu, Y., Lu, S., O'Sullivan, B., Razgon, I.: A fixed-parameter algorithm for the directed feedback vertex set problem. J. ACM **55**(5), 21:2–21:19 (2008). doi:10.1145/1411509.1411511
9. Chen, Y., Flum, J., Grohe, M.: Bounded nondeterminism and alternation in parameterized complexity theory. In: Proceedings of the 18th IEEE Annual Conference on Computational Complexity (CCCC 2003), pp. 13–29. (2003). doi:10.1109/ccc.2003.1214407
10. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Berlin (1999)
11. Elberfeld, M., Stockhusen, C., Tantau, T.: On the space complexity of parameterized problems. In: Thilikos, D.M., Woeginger, G.J. (eds.) Parameterized and Exact Computation, Lecture Notes in Computer Science, vol. 7535, pp. 206–217. Springer, Berlin. (2012).doi:10.1007/978-3-642-33293-7_20
12. Fellows, M., Hallett, M., Stege, U.: Analogs & duals of the mast problem for squences & trees. J. Algorithms **49**(1), 192–216 (2003)
13. Flum, J., Grohe, M.: Describing parameterized complexity classes. Inf. Comput. **187**, 291–319 (2003). doi:10.1016/S0890-5401(03)00161-5
14. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Berlin (2006). doi:10.1007/3-540-29953-X
15. Guillemot, S.: Parameterized complexity and approximability of the longest compatible sequence problem. Discret. Optim. **8**(1), 50–60 (2011). doi:10.1016/j.disopt.2010.08.003
16. Hartmanis, J.: On non-determinancy in simple computing devices. Acta Inform. **1**, 336–344 (1972). doi:10.1007/BF00289513
17. Hong, J.W.: On some deterministic space complexity problems. SIAM J. Comput. **11**(3), 591–601 (1982). doi:10.1137/0211050
18. Immerman, N.: Descriptive Complexity. Springer, New York (1998)
19. Jones, N.D., Laaser, W.T.: Complete problems for deterministic polynomial time. Theoret. Comput. Sci. **3**(1), 105–117 (1976). doi:10.1016/0304-3975(76)90068-2
20. Jones, N.D., Lien, Y.E., Laaser, W.T.: New problems complete for nondeterministic log space. Math. Syst. Theory **10**(1), 1–17 (1976). doi:10.1007/BF01683259
21. Kintala, C.M.R., Fischer, P.C.: Refining nondeterminism in relativized polynomial-time bounded computations. SIAM J. Comput. **1**(9), 46–53 (1980). doi:10.1137/0209003
22. Stockhusen, C., Tantau, T.: Completeness results for parameterized space classes. Tech. Rep. arXiv:1308.2892, ArXiv e-prints (2013a)
23. Stockhusen, C., Tantau, T.: Completeness results for parameterized space classes. In: Gutin, G., Szeider, S. (eds.) Parameterized and Exact Computation, Lecture Notes in Computer Science, vol. 8246, pp. 335–347. Springer, Berlin (2013b). doi:10.1007/978-3-319-03898-8_28
24. Vollmer, H.: Introduction to Circuit Complexity: A Uniform Approach. Springer, Berlin (1999)