Dynamic Complexity Meets Parameterised Algorithms

Jonas Schmidt
TU Dortmund University, Dortmund, Germany

Thomas Schwentick
TU Dortmund University, Dortmund, Germany

Nils Vortmeier
TU Dortmund University, Dortmund, Germany

Thomas Zeume
TU Dortmund University, Dortmund, Germany

Ioannis Kokkinis
TU Dortmund University, Dortmund, Germany

Abstract

Dynamic Complexity studies the maintainability of queries with logical formulas in a setting where the underlying structure or database changes over time. Most often, these formulas are from first-order logic, giving rise to the dynamic complexity class DynFO. This paper investigates extensions of DynFO in the spirit of parameterised algorithms. In this setting structures come with a parameter k and the extensions allow additional "space" of size $f(k)$ (in the form of an additional structure of this size) or additional time $f(k)$ (in the form of iterations of formulas) or both. The resulting classes are compared with their non-dynamic counterparts and other classes. The main part of the paper explores the applicability of methods for parameterised algorithms to this setting through case studies for various well-known parameterised problems.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Logic and databases; Theory of computation → Complexity theory and logic

Keywords and phrases Dynamic complexity, parameterised complexity

Digital Object Identifier 10.4230/LIPIcs.CSL.2020.36

Related Version A full version of this paper is available at `https://arxiv.org/abs/1910.06281`.

Funding The authors acknowledge the financial support by DFG grant SCHW 678/6-2.

Acknowledgements We are grateful to Till Tantau for some valuable discussions.

1 Introduction

Parameterised complexity studies aspects of problems that make them computationally hard. The main interest has been in the class FPT which subsumes all problems that can be solved in time $f(k)\mathsf{poly}(|x|)$ for an input x with a *parameter* $k \in \mathbb{N}$ and a computable function f. In recent work, much smaller parameterised classes have been studied, derived from classical classes in a uniform way by replacing the requirement of a polynomial bound of e.g. the circuit size (time, space, ..., respectively) by a bound of the form $f(k)\mathsf{poly}(|x|)$. In this fashion classical circuit classes AC^i and NC^i naturally translate to parameterised classes para-AC^i and para-NC^i. The lowest of these classes, para-AC^0 corresponds to the class AC^0 of problems computable by uniform families of constant-depth, polynomial size circuits with $\wedge$-, $\vee$- and $\neg$-gates of unbounded fan-in [19, 3].

This paper adds the aspect of changing inputs and dynamic maintenance of results to the exploration of the landscape between para-AC^0 and FPT.

The study of low-level complexity classes under dynamic aspects was started in [30, 15] in the context of dynamically maintaining the result of database queries. Similarly, as for *dynamic algorithms*, in this setting a dynamic program can make use of auxiliary relations that can store knowledge about the current input data (database). After a small change of the database (most often: insertion or deletion of a tuple), the program needs to compute the query result for the modified database in very short parallel time. To capture the problems/queries, for which this is possible, Patnaik and Immerman introduced the class DynFO [30]. Here, "FO" stands for first-order logic, which is equivalent to AC^0, in the presence of arithmetic [7, 24].

In this paper, we study dynamic programs that have additional resources in a "parameterised sense". We explore two such resources, which can be described as *parameterised space* and *parameterised time*, respectively. For ease of exposition, we discuss these two resources in the context of AC^0 first.

One way to strengthen AC^0 circuit families is to allow circuits of *size* $f(k)\mathsf{poly}(|x|)$. We denote the class thus obtained as para-S-AC^0 (even though it corresponds to the class para-AC^0). A second dimension is to let the *depth* of circuits depend on the parameter. As the depth of circuits corresponds to the (parallel) time the circuits need for a computation, we denote the class of problems captured by such circuits by para-T-AC^0. Of course, both dimensions can also be combined, yielding the parameterised class para-ST-AC^0.

Surprisingly, several parameterised versions of NP-complete problems can even be solved in para-S-AC^0. Examples are the vertex cover problem and the hitting set problem parameterised by the size of the vertex cover and the hitting set, respectively [4]. However, classical circuit lower bounds unconditionally imply that this is not possible for all FPT-problems. For instance, in [3] it was observed that the existence of simple paths of length k (the parameter) cannot be tested in para-S-AC^0. Likewise, the feedback vertex set problem with the size of the feedback vertex set as parameter cannot be solved in para-ST-AC^0.

When translated from circuits to logical formulas, depth roughly translates into iteration of formulas [24, Theorem 5.22], whereas size translates into the size of an additional structure by which the database is extended before formulas are evaluated. Slightly more formally, para-T-AC^0 corresponds to the class para-T-FO consisting of problems that can be defined by iterating a formula $f(k)$ many times. The class para-S-AC^0 corresponds to the class para-S-FO where formulas are evaluated on structures $\mathcal{D}$ extended by an *advice structure* whose size depends on the parameter only. In the class para-ST-FO both dimensions are combined. The parameterised *dynamic* classes that we study in this paper are obtained from DynFO just like the above classes are obtained from FO: para-S-DynFO, para-T-DynFO and para-ST-DynFO extend DynFO by an additional structure of parameterised size, $f(k)$ iterations of formulas, or both, respectively.

As our first main contribution, we introduce a uniform framework for small dynamic, parameterised complexity classes (Section 3) based on advice structures (corresponding to additional space) or iterations of formulas (corresponding to additional time) and investigate how the resulting classes relate to each other and to other non-dynamic (and even non-parameterised) complexity classes (Section 4).

As our second main contribution, we explore how methods for parameterised algorithms can be applied in this framework through case studies for various parameterised problems (Section 5). Due to space limitations, many proofs are omitted and can be found in the full version of this paper.

Related work. There is a rich literature on parameterised dynamic algorithms, e.g. [23, 16, 28, 8, 1]. Closer to our work is the investigation of (static) parameterised small (parallel) complexity classes that was initiated 20 years ago in [9]. Later, in [19], parameterised versions of space and circuit classes were defined and several known parameterised problems were shown to be complete for these classes. Also in [3] it was shown, by applying the colour-coding technique, that several parameterised problems belong in para-AC^0. Furthermore Chen and Flum [10] presented some unconditional proofs showing that some parameterised problems do not belong in para-AC^0.

The descriptive complexity of parameterised classes has also been investigated in the past. For example Flum and Grohe [20] and Bannach and Tantau [6] presented syntactic descriptions of parameterised complexity classes using logical formulas. Additionally Chen, Flum and Huang [11] showed that the k-slices of several problems can be defined using FO-formulas of quantifier rank independent of k and explored the connection between the quantifier rank of FO-sentences and the depth of AC^0-circuits.

2 Preliminaries

By $[n]$ we denote the set $\{1, \dots, n\}$. We assume familiarity with first-order logic FO and refer to [27] for basics of finite model theory. A *(relational) schema* τ consists of a set of relation symbols with a corresponding arity. A *structure* $\mathcal{D}$ over schema τ with domain D has, for every relation symbol $R \in \tau$, a relation over D with the same arity as R. Throughout this work domains are finite. A *k-ary query* Q on τ-structures is a mapping that assigns a subset of D^k to every τ-structure over domain D and commutes with isomorphisms. Each first-order formula $\varphi(\bar{x})$ over schema τ defines a query Q whose result on a τ-structure $\mathcal{D}$ is $\{\bar{a} \mid \mathcal{D} \models \varphi(\bar{a})\}$. Queries of arity 0 are also called *Boolean queries* or *problems*.

We mainly consider first-order formulas that have access to arithmetic, that is to a linear order $<$ on the domain as well as suitable, compatible addition $+$ and multiplication $\times$. We require that the result of the formulas is invariant[1] under the choice of the linear order $<$. This logic is referred to as *order-invariant first-order logic with arithmetic* and denoted by $\mathsf{FO}(+,\times)$. In linearly ordered domains, we often identify domain elements with natural numbers, the smallest element representing 1.

Dynamic Complexity. We work in the dynamic complexity framework as introduced by Patnaik and Immerman [30], and refer to [32] for details. In a nutshell, dynamic programs answer a query for an input structure that is subjected to a sequence of changes. To this end they maintain an auxiliary structure using logical formulas.

By Δ_τ we denote the set of *single-tuple change operations* for a schema τ, which consists of the insertion operations $\textsc{ins}_R$ and the deletion operations $\textsc{del}_R$ for each relation $R \in \tau$. For example, $\textsc{ins}_E(a, b)$ could add edge (a, b) to a graph. A *dynamic query* (Q, Δ) consists of a query Q over some input schema τ_{in} and a set $\Delta \subseteq \Delta_{\tau_{\text{in}}}$. Later on we will sometimes consider slightly more general change operations.

A *dynamic program* $\mathcal{P}$ for a dynamic query (Q, Δ) continuously answers Q on an *input structure* $\mathcal{I}$ over some *input schema* τ_{in} under changes of the input structure from Δ. The domain D of $\mathcal{I}$ is fixed and in particular changes cannot introduce new elements.[2] The program $\mathcal{P}$ maintains an *auxiliary structure* $\mathcal{A}$ over some *auxiliary schema* τ_{aux} with the

[1] In our scenario it is not relevant that invariance is undecidable for first-order formulas.
[2] We note that this is not a severe restriction, see e.g. [12, Theorem 17].

same domain as $\mathcal{I}$. We call $(\mathcal{I}, \mathcal{A})$ a *state* of $\mathcal{P}$ and consider it as one relational structure. The auxiliary structure includes one particular *query relation* ANS that is supposed to contain the answer of Q over $\mathcal{I}$. For each auxiliary relation $S \in \tau_{\text{aux}}$ and each change operation $\delta \in \Delta$, $\mathcal{P}$ has an update rule that specifies how S is updated after a change. It is of the form **on change** $\delta(\bar{p})$ **update** $S(\bar{x})$ **as** $\phi^S_\delta(\bar{p}; \bar{x})$ where the *update formula* $\phi^S_\delta(\bar{p}; \bar{x})$ is a formula over $\tau_{\text{in}} \cup \tau_{\text{aux}}$. For example, if the tuple $\bar{a}$ is inserted into an input relation R, each auxiliary relation S is replaced by the relation $\{\bar{b} \mid (\mathcal{I}, \mathcal{A}) \models \phi^S_{\text{INS}_R}(\bar{a}; \bar{b})\}$. By $\alpha(\mathcal{I})$ we denote the input structure that results from $\mathcal{I}$ by applying a sequence α of changes, and by $\mathcal{P}_\alpha(\mathcal{I}, \mathcal{A})$ the state $(\alpha(\mathcal{I}), \mathcal{A}')$ of $\mathcal{P}$ that results from $(\mathcal{I}, \mathcal{A})$ after processing α. The dynamic program $\mathcal{P}$ *maintains* (Q, Δ) if the relation ANS in $\mathcal{P}_\alpha(\mathcal{I}_0, \mathcal{A}_0)$ equals the query result $Q(\alpha(\mathcal{I}_0))$, for each sequence α of changes over Δ, each initial input structure $\mathcal{I}_0$ with arbitrary (finite) domain and empty relations, and the auxiliary structure $\mathcal{A}_0$ with empty relations.

The class DynFO is the set of dynamic queries that can be maintained by a dynamic program with first-order update formulas. The class DynFO$(+, \times)$ is defined analogously via FO$(+,\times)$ update formulas. We note that in the case of DynFO$(+, \times)$, we consider the arithmetic relations to be part of the input structure $\mathcal{I}$, but they can not be modified. Technically, an additional schema τ_{arith} contains the arithmetic predicates and the update formulas are over $\tau_{\text{in}} \cup \tau_{\text{aux}} \cup \tau_{\text{arith}}$. Note that τ_{arith} cannot be used for defining a query.

Parameterised Complexity. A *parameterised query* is a pair (Q, κ), where Q is a query over some schema τ and κ is a function, called the *parameterisation*, that assigns a parameter from $\mathbb{N}$ to every τ-structure. The well-known parameterised complexity class FPT contains all Boolean parameterised queries (Q, κ) having an algorithm that decides for each τ-structure $\mathcal{D}$ whether $\mathcal{D} \in Q$ in time $f(\kappa(\mathcal{D}))|\mathcal{D}|^c$, for some constant c and computable function $f\colon \mathbb{N} \to \mathbb{N}$ [17]. Like [5], we demand that κ is first-order definable, which is always the case if the parameter is explicitly given in the input.

▶ **Example 1.** p-VertexCover is a well-studied parameterised query. Formally it is the set Q of pairs (G, k), where G is an undirected graph that has a vertex cover of size k, together with the parameterisation $\kappa\colon (G, k) \mapsto k$. In more accessible notation:

Problem: p-VertexCover
Input: An undirected graph $G = (V, E)$ and $k \in \mathbb{N}$, **Parameter:** k
Question: Is there a set $S \subseteq V$ such that $|S| = k$ and $u \in S$ or $v \in S$ for every $(u, v) \in E$?

The search-tree based algorithm for p-VertexCover is a classical parameterised algorithm. It is based on the simple observation that, for each edge (u, v) of a graph, each vertex cover needs to contain u or v (or both). On input (G, k) the algorithm recursively constructs the search tree as follows, starting from the root of an otherwise empty tree. If E is empty it accepts, otherwise it rejects if $k = 0$. If $k > 0$ it chooses some edge $(u, v) \in E$, labels the current node with (u, v), and constructs two new tree nodes below the current node. It then continues recursively, from both children starting from the instance $(G - u, k - 1)$ in the first child, and from $(G - v, k - 1)$ in the second child. The algorithm accepts if any of its branches accepts. Since the inner nodes of the tree have two children and its depth is bounded by k, it can have at most $2^{k+1} - 1$ tree nodes. The overall running time can be bounded by $\mathcal{O}(2^k n^2)$. Thus p-VertexCover $\in$ FPT.

3 A Framework for Parameterised, Dynamic Complexity

We first present a uniform point of view on parameterised first-order logic. As explained in the introduction, formulas can be parameterised with respect to (at least) two dimensions: additional time by iterating formulas with the number of iterations depending on the parameter; additional space by advice structures whose size depends on the parameter.

A *first-order program* $\mathcal{F}$ over schema τ is a tuple (Ψ, φ) where Ψ is a set of $\mathsf{FO}(+,\times)$-formulas over schema $\tau \uplus \tau_\Psi$ and $\varphi \in \Psi$ is supposed to compute the final result of the program. Here, τ_Ψ is a schema that contains a fresh relation symbol R_ψ for each formula $\psi \in \Psi$ of the same arity as ψ. The semantics of $\mathcal{F}$ on a τ-structure $\mathcal{D}$ is based on inductively defined τ_Ψ-structures $\mathcal{D}_\Psi^{(\ell)}$. Initially, in $\mathcal{D}_\Psi^{(0)}$, all relations $R_\psi^{(0)}$ are empty. The *ℓ-step result* $\mathcal{D}_\Psi^{(\ell)}$ of $\mathcal{F}$, for $\ell > 0$, is defined via $R_\psi^\ell \stackrel{\text{def}}{=} \{\bar{a} \mid (\mathcal{D}, \mathcal{D}_\Psi^{(\ell-1)}) \models \psi(\bar{a})\}$. Finally, the *result* $\mathcal{F}(\mathcal{D})$ is $R_\varphi^{(\ell)}$ if $\mathcal{D}_\Psi^{(\ell-1)} = \mathcal{D}_\Psi^{(\ell)}$, for some ℓ. In this case, we say that the program reaches a fixed point after ℓ steps. Otherwise, $\mathcal{F}(\mathcal{D})$ is the empty set.

We now define how first-order programs can use advice. An *τ_{adv}-advice* π is a computable mapping from $\mathbb{N}$ to τ_{adv}-structures for some fixed advice schema τ_{adv}. Suppose that $\mathcal{F}$ is a first-order program over schema $\tau \uplus \tau_{\text{adv}}$. The result of $\mathcal{F}$ for a τ-structure $\mathcal{D}$ with advice π and parameter $k \in \mathbb{N}$ is simply the result of $\mathcal{F}$ on the structure $\mathcal{D} \uplus \pi(k)$.

For two computable functions $f, g : \mathbb{N} \to \mathbb{R}$ and a parameterised query (Q, κ) over a schema τ, an *(f,g)-parameterised first-order program for* (Q,κ) is a tuple $(\mathcal{F}, \pi)$ where $\mathcal{F}$ is a first-order program over schema $\tau \uplus \tau_{\text{adv}}$ and π is an τ_{adv}-advice such that

(a) the result of $\mathcal{F}$ with advice π is $Q(\mathcal{D})$, for all τ-structures $\mathcal{D}$;
(b) $|\pi(\kappa(\mathcal{D}))| \leq f(\kappa(\mathcal{D}))$ for all τ-structures $\mathcal{D}$; and
(c) $\mathcal{F}$ always reaches a fixed point and does so after at most $g(\kappa(\mathcal{D}))$ steps.

For computable functions f and g let para-ST-FO(f,g) be the class of parameterised queries definable by an (f,g)-parameterised first-order program. We note that these programs use $\mathsf{FO}(+,\times)$ formulas, and thus have access to arithmetic[3] over the domain of $\mathcal{D} \uplus \pi(k)$. We do not make this explicit in our naming scheme. We use the following abbreviations:

- para-ST-FO $\stackrel{\text{def}}{=} \bigcup_{f,g}$ para-ST-FO(f,g),
- para-S-FO $\stackrel{\text{def}}{=} \bigcup_{f}$ para-ST-FO$(f,1)$,
- para-T-FO $\stackrel{\text{def}}{=} \bigcup_{g}$ para-ST-FO$(0,g)$.

The class para-S-FO is in fact the same as para-AC^0, and para-ST-FO corresponds to the class para-AC$^{0\uparrow}$ in [3]. To the best of our knowledge, para-T-FO has not been studied in the context of first-order logic before.

▶ **Example 2.** We sketch a first-order program $\mathcal{F} = (\Psi, \varphi)$ that witnesses p-VertexCover $\in$ para-T-FO. Recall the search-tree based parameterised algorithm for p-VertexCover from Example 1. Intuitively, the formulas $\psi \in \Psi$ are used to traverse the search tree in a depth-first manner. At any moment, the auxiliary relations contain information about the path from the root to the current node. In particular, the *candidate set* of the current node, i.e., the set of vertices selected along its path is available. Each application of these formulas simulates one elementary step of the search: either a new child is added to the current path, or, if the current node has maximal depth or if all possible children were already added, the current node is discarded and a backtrack step to its parent is performed. If the candidate set is a vertex cover, the search ends. Since each edge of the search tree needs to be traversed at most twice, 2^{k+2} iterative steps suffice. More detail is given in the full version.

[3] In particular, "$+|\mathcal{D}|$" induces a correspondence between $\mathcal{D}$ and $\pi(k)$.

The following lemma basically states that every boolean parameterised query can be answered in para-S-FO on instances whose domain size is bounded by a function in the parameter.

▶ **Lemma 3.** *Let $f\colon \mathbb{N} \to \mathbb{N}$ be a computable function and (Q, κ) a boolean parameterised query with decidable Q. There is a computable function g and a $(g, 1)$-parameterised first-order program (φ, π) that answers Q correctly on instances $\mathcal{D}$ of size at most $f(\kappa(\mathcal{D}))$.*

Proof idea. We explain the proof idea for input structures consisting of a graph G of size n and a parameter value k with $n \leq f(k)$. The advice π produces an advice structure with domain $[2^{f(k)^2}]$. It has a ternary relation E' that contains, for every $i \in [2^{f(k)^2}]$ all tuples (i, j_1, j_2), for which the i-th graph over $[f(k)]$ in some canonical enumeration has an edge (j_1, j_2). It further contains a unary relation F that contains all numbers i, for which the i-th graph is a yes-instance of Q. The formula φ simply determines with the help of E' and built-in arithmetic the number i of G (as a graph over $[n]$) and tests whether $F(i)$ holds. ◀

Parameterised Dynamic Complexity. We study parameterised queries in a dynamic context. Formally, a *dynamic parameterised query* (Q, κ, Δ) consists of a parameterised query (Q, κ) and a set Δ of change operations. We say that a parameterised query (Q, κ) has an *explicit parameter*, if Q consists of pairs $\mathcal{I} = (\mathcal{I}', k)$, where $\mathcal{I}'$ is a structure, k is a suitably encoded number, and $\kappa(\mathcal{I}) = k$. All concrete parameterised queries we consider in this paper have an explicit parameter. For example, we often consider the dynamic variant $(p\text{-VertexCover}, \Delta_E \cup \pm 1)$ of the parameterised vertex cover query, where $\Delta_E \stackrel{\text{def}}{=} \{\text{ins}_E, \text{del}_E\}$ and $\pm 1 \stackrel{\text{def}}{=} \{+1, -1\}$ denotes the set of change operations that increment or decrement the given number k by one, as long as k stays in the admissible range. So, given some graph G with n vertices, $+1(G, k) \stackrel{\text{def}}{=} (G, k+1)$ if $k < n$, and $-1(G, k) \stackrel{\text{def}}{=} (G, k-1)$ if $k > 1$, and otherwise the changes have no effect.

For most queries[4] in this paper only parameter values in $\{1, \ldots, n\}$ are meaningful and we only allow such values. They can be represented by elements of the domain.

Similarly as parameterised first-order programs generalise first-order formulas, parameterised dynamic programs extend conventional dynamic programs in two directions: (1) they may use an advice structure whose size depends on the parameter, and (2) they may use first-order programs of parameterised iteration depth.

A *dynamic program with iteration and advice* is a tuple $(\mathcal{P}, \pi)$ where $\mathcal{P}$ is a dynamic program where auxiliary relations are updated with first-order programs and π is an τ_{adv}-advice for an advice schema τ_{adv}. For a dynamic parameterised query (Q, κ, Δ), the program $\mathcal{P}$ has update rules of the form **on change** $\delta(\bar{p})$ **update** $S(\bar{x})$ **as** (Ψ_S, φ_S) for every $\delta \in \Delta$, where (Ψ_S, φ_S) is a first-order program over schema $\tau_{\text{in}} \cup \tau_{\text{aux}} \cup \tau_{\text{adv}}$ such that φ_S has the same arity as S. States of the program $\mathcal{P}$ are of the form $(D \uplus D_{\text{adv}}, \mathcal{I}, \mathcal{A}, \mathcal{A}_{\text{adv}})$ where $\mathcal{I}$ is the input structure, $\mathcal{A}$ the auxiliary structure, and $\mathcal{A}_{\text{adv}}$ is an advice structure over a schema τ_{adv}. Tuples of the auxiliary structure $\mathcal{A}$ may range over the domain $D \uplus D_{\text{adv}}$.

For two computable functions $f, g : \mathbb{N} \to \mathbb{R}$, an (f, g)-*parameterised dynamic program* is a dynamic program $(\mathcal{P}, \pi)$ with iteration and advice such that $|\pi(k)| \leq f(k)$ for all $k \in \mathbb{N}$ and all first-order programs of $\mathcal{P}$ always reach a fixed point after at most $g(\kappa(\mathcal{I}))$ steps. The initial state of such a program depends on an initial input structure $\mathcal{I}_0$ and a number $k \in \mathbb{N}$. It is given as $(D \cup D_{\text{adv}}, \mathcal{I}_0, \mathcal{A}_0, \mathcal{A}^k_{\text{adv}})$ where $\mathcal{A}^k_{\text{adv}} \stackrel{\text{def}}{=} \pi(k)$, D and D_{adv} are the domains of $\mathcal{I}_0$ and $\pi(k)$, respectively, and $\mathcal{A}_0$ is an empty τ_{aux}-structure.

[4] The only exception is p-Knapsack in Section 5.4.

A dynamic parameterised query (Q, κ, Δ) is maintained by $(\mathcal{P}, \pi)$ if a distinguished relation ANS in $\mathcal{P}_\alpha(D \cup D_{\text{adv}}, \mathcal{I}_0, \mathcal{A}_0, \mathcal{A}^k_{\text{adv}})$ equals $Q(\alpha(\mathcal{I}_0))$, for all empty[5] input structures $\mathcal{I}_0$, all $k \in \mathbb{N}$, and all sequences α of changes over Δ such that $\kappa(\alpha'(\mathcal{I}_0)) \leq k$ for all prefixes α' of α. So, the dynamic program $(\mathcal{P}, \pi)$ only needs to maintain (Q, κ, Δ) as long as the parameter value is bounded by the initially given number k; nevertheless the program needs to work for arbitrary values of k. We denote this number k in the following as $k_{\max}$.

For computable functions $f, g : \mathbb{N} \to \mathbb{R}$ we define para-ST-DynFO(f, g) as the class of dynamic parameterised queries that can be maintained by an (f, g)-parameterised dynamic program. We define:

- para-ST-DynFO $\stackrel{\text{def}}{=} \bigcup_{f,g}$ para-ST-DynFO(f, g),
- para-S-DynFO $\stackrel{\text{def}}{=} \bigcup_{f}$ para-ST-DynFO$(f, 1)$,
- para-T-DynFO $\stackrel{\text{def}}{=} \bigcup_{g}$ para-ST-DynFO$(0, g)$,

Since the purpose of this article is to explore the basic principles of parameterised, dynamic complexity, we keep the setting simple, in particular with respect to the following two aspects. First, dynamic programs get a bound $k_{\max}$ for the parameter values at initialisation time and the program then only needs to deal with changes that obey this parameter bound. This ensures that the advice structure does not change throughout the dynamic process. Second, we assume the presence of arithmetic throughout. In non-parameterised dynamic complexity, it is known that under mild assumptions on the query, arithmetic relations can be constructed by a dynamic program on the fly [12]. Similar techniques can be applied for the parameterised setting, yet we ignore this aspect here and assume that $\mathcal{I}_0 \uplus \pi(k)$ comes with relations $<$, $+$, and $\times$ over $D \uplus D_{\text{adv}}$.

For some first intuition we provide a parameterised dynamic program that shows that $(p$-VertexCover$, \{\text{INS}_E\} \cup \pm 1)$ is in para-S-DynFO via the search-tree based approach. This result is not surprising, as it is known that p-VertexCover $\in$ para-S-FO [11, 4]. However, the dynamic program for maintaining search trees is conceptually very simple.

▶ **Example 4.** We recall the search-tree based parameterised algorithm for p-VertexCover from Example 1. The first-order program of Example 2 witnesses p-VertexCover $\in$ para-T-FO (and thus also in para-T-DynFO) by constructing a search tree from scratch. In contrast, a dynamic program witnessing $(p$-VertexCover$, \{\text{INS}_E\} \cup \pm 1) \in$ para-S-DynFO can *maintain* a search tree. To this end, for a given bound $k_{\max}$, its advice structure $\mathcal{A}^{k_{\max}}_{\text{adv}}$ stores a full binary "background" tree T of depth $k_{\max}$. Its auxiliary structure represents the actual search tree T' by maintaining an upward closed set of nodes and the candidate sets of each of those nodes. As in the search tree algorithm from Example 1, in every inner node x of T' a branching on the endpoints of some edge e of G is being simulated and in each of x's two children one vertex of e is added to the candidate set. A node x of T is a leaf of T', if the assigned candidate set of x is an actual vertex cover of G or if x is in level $k_{\max}$ of T. The program then only needs to check whether there is a leaf representing a valid vertex cover at a level below the current value of k. Maintenance under changes from ± 1 is therefore easy.

Maintaining T' under insertion of an edge (u, v) is easy as well: for each leaf of T' that is *not* at level $k_{\max}$, and whose candidate set does not cover (u, v), the program adds u to the left child and v to the right child (assuming $u < v$). Leaves at level $k_{\max}$ are not modified, but it might happen that a former vertex cover attached to such a leaf becomes invalid by not covering (u, v). Maintaining T' under edge *deletions* is slightly more subtle and will be considered in the proof of Proposition 10.

[5] For queries with explicit parameter, we require only that in $\mathcal{I}_0 = (\mathcal{I}'_0, k)$, $\mathcal{I}'_0$ is empty, but k can be non-zero.

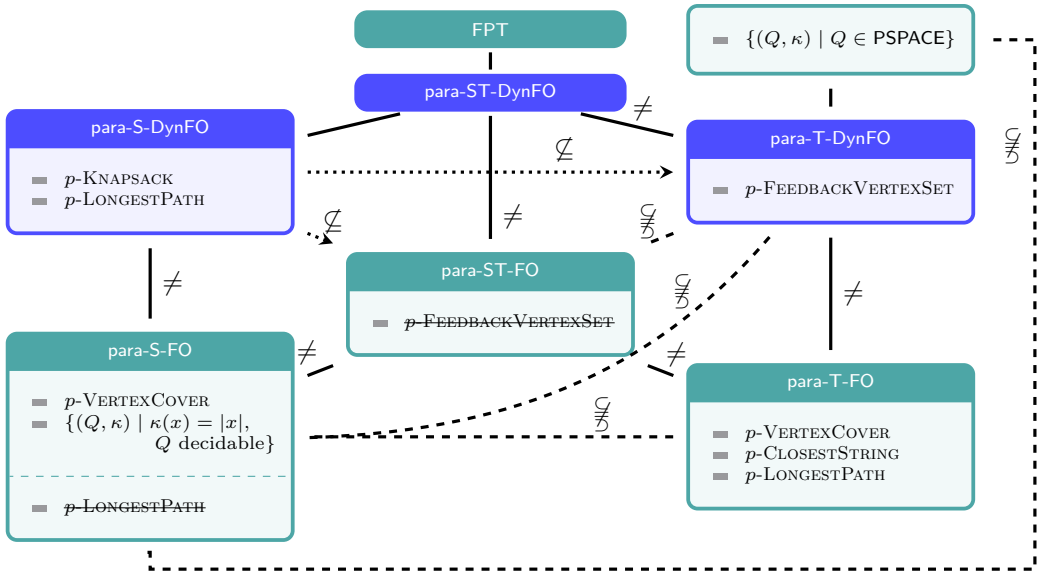


Figure 1 Inclusion diagram of the main classes. Solid lines indicate inclusions. Dashed lines marked with $\subsetneq\!\!\!\!/\;\;\supsetneq\!\!\!\!/$ indicate that the two classes are incomparable. A directed, dotted edge marked with $\not\subseteq$ from $\mathcal{C}$ to $\mathcal{C}'$ indicates $\mathcal{C} \setminus \mathcal{C}' \neq \emptyset$. If $\mathcal{C}$ is a dynamic class and $\mathcal{C}'$ a static class, $\mathcal{C} \subseteq \mathcal{C}'$ means that for each $(Q, \kappa, \Delta) \in \mathcal{C}$ with exhaustive Δ it holds that $(Q, \kappa) \in \mathcal{C}'$, and $\mathcal{C}' \subseteq \mathcal{C}$ means that for each $(Q, \kappa) \in \mathcal{C}'$ it holds that $(Q, \kappa, \Delta) \in \mathcal{C}$, for arbitrary Δ.

4 Relationships between Parameterised Classes

In this section we examine how parameterised dynamic and static complexity classes relate to each other. These relationships are summarised in Figure 1.

As a sanity check, we show first that every parameterised query (Q, κ) with $(Q, \kappa, \Delta) \in$ para-ST-DynFO is in FPT. For queries in para-T-DynFO the respective algorithm only needs polynomial space. Both statements require that Δ is *exhaustive*, i.e., that it contains the single-tuple insertion operation INS_R for every input relations R. This ensures that every possible input structure for Q can be obtained by a change sequence.[6]

▶ **Proposition 5.**

(a) *For every* $(Q, \kappa, \Delta) \in$ para-ST-DynFO *with exhaustive* Δ *it holds that* $(Q, \kappa) \in$ FPT.

(b) *For every* $(Q, \kappa, \Delta) \in$ para-T-DynFO *with exhaustive* Δ*, the parameterised query* (Q, κ) *can be solved by an* FPT*-algorithm that uses at most polynomial space with respect to the input size. In particular,* $Q \in$ PSPACE.

Statement (b) does not hold for parameterised classes with advice, as we formalise with the next proposition, which is an immediate consequence of Lemma 3.

▶ **Proposition 6.** *Every parameterised query* (Q, κ) *with decidable* Q *and* $\kappa(x) = |x|$ *is in* para-S-FO.

▶ **Proposition 7.** *For any* $(Q, \kappa) \in$ para-S-FO *and any* $\Delta \subseteq \Delta_{\tau_{\text{in}}}$ *(or* $\Delta \subseteq \Delta_{\tau_{\text{in}}} \cup \pm 1$*) it holds that* $(Q, \kappa, \Delta) \in$ para-S-DynFO.

[6] Clearly, a more general definition would be possible here, but we avoid that in the interest of simplicity.

Proof sketch. Let $(Q, \kappa) \in$ para-S-FO by some $(f, 1)$-parameterised FO program $\mathcal{F}$. In principle, a parameterised dynamic program can simulate $\mathcal{F}$ from scratch after each change. However, since the parameter of $\mathcal{I}$ might change, it might need different advice structures from $\mathcal{F}$. However, there is an easy solution for this. For the given $k_{\max}$, the dynamic program gets as its advice *all* advice structures $\pi(1), \ldots, \pi(k_{\max})$ of $\mathcal{F}$. ◀

The same argument can be applied for para-ST-FO and para-ST-DynFO.

In addition to the above inclusions and those that are immediate from the definitions, we observe the following separations between parameterised classes (also see Figure 1). Some proofs are deferred to the next section.

▶ **Proposition 8.**

(a) *There is a* $(Q, \kappa) \in$ para-S-FO *such that* $(Q, \kappa, \Delta) \notin$ para-T-DynFO*, for any exhaustive* Δ*.*

(b) *There is a* $(Q, \kappa) \in$ para-T-FO *such that* $(Q, \kappa) \notin$ para-S-FO*.*

(c) *There is a* $(Q, \kappa, \Delta) \in$ para-T-DynFO *with exhaustive* Δ *such that* $(Q, \kappa) \notin$ para-ST-FO*.*

(d) *There is a* $(Q, \kappa, \Delta) \in$ para-S-DynFO *with exhaustive* Δ *such that* $(Q, \kappa) \notin$ para-ST-FO*.*

Proof sketch. Part (a) is a consequence of Proposition 5 and Proposition 6, and witnessed by any parameterised problem (Q, κ) with decidable $Q \notin$ PSPACE and $\kappa(x) = |x|$. Part (b) is witnessed by the problem p-LongestPath which is not in para-S-FO [3], but in para-T-FO as we will see in Proposition 9. For (c) we observe that p-FeedbackVertexSet is not in para-ST-FO, as otherwise the restriction to inputs with parameter $k = 0$ would yield a first-order formula that expresses acyclicity of undirected graphs. In Proposition 12 we will show that (p-FeedbackVertexSet, $\Delta_E \cup \pm 1$) is in para-T-DynFO. The separation for (d) can be shown with the help of connectivity of undirected graphs. To this end, we consider the parameterisation by the maximal node degree. It is well-known that even for fixed $k = 2$ this property is not expressible in FO(+,×), see [21], and thus it is not in para-ST-FO. On the other hand, towards (d), the unparameterised version is in DynFO and thus the parameterised version is in para-S-DynFO.[7] ◀

5 Methods for Parameterised Complexity

The goal of this section is to explore the transferability of known methods from the realm of parameterised algorithms to dynamic parameterised complexity. We are thus not always interested in "best algorithms" but rather want to exemplify how sequential algorithmic methods for static problems translate into the dynamic (highly parallel) setting.

We start by describing colour-coding, since it turns out as particularly useful in the dynamic context and we use it in many other subsections. Then we consider three classical methods for parameterised algorithms, bounded search trees, kernelisation and dynamic programming. Afterwards we give an example for the iterated compression method, which uses an adaption of a technique from dynamic complexity.

5.1 Colour-Coding

In this subsection, we establish the usefulness of the colour-coding technique, as presented in [2], in our setting by a concrete example, p-LongestPath.

[7] Of course, this argument could have been used for (c) as well, but there we prefer a more "natural" parameterisation.

Problem: p-LongestPath
Input: An undirected graph $G = (V, E)$, $s, t \in V$ and $\ell \in \mathbb{N}$, **Parameter:** ℓ
Question: Is there a (simple) path from s to t of length ℓ?

This problem can be solved with the help of *universal colouring families*. Such a family is a small set of functions that map nodes to colours such that if a path of length ℓ exists, one of these functions colours the nodes of the path with a fixed sequence of $\ell+1$ colours. A parallel algorithm for p-LongestPath therefore only needs to test in parallel, for each function of a universal colouring family, whether it produces such a coloured path from s to t.

More precisely, a (n, k, c)-*universal colouring family* Λ has, for every subset $S \subseteq [n]$ of size k and for every mapping $\mu : S \to [c]$, at least one function $\lambda \in \Lambda$ with $\lambda(s) = \mu(s)$, for every $s \in S$. In [3, Theorem 3.2] a family $\Lambda_{n,k,c}$ of such functions is defined. The definition can be found in the full version. In the presence of arithmetic, these functions are easily first-order definable and can be enumerated in a first-order fashion.

▶ **Proposition 9.**
(a) p-LongestPath $\in$ para-S-DynFO.
(b) p-LongestPath $\in$ para-T-FO.

Proof sketch. In both parts of the proof, we use the colour-coding approach as sketched above. For a graph G, a colouring function λ, and a set C of colours, a *C-coloured path under λ* is a path whose nodes are mapped to C in a one-one fashion by λ.

For solving the p-LongestPath problem with parameter ℓ, we consider the (n, k, k)-universal colouring family $\Lambda \stackrel{\text{def}}{=} \Lambda_{n,k,k}$ with $k \stackrel{\text{def}}{=} \ell + 1$. Then a graph has a simple path of length ℓ from s to t if and only if there is a $[k]$-coloured path from s to t under some $\lambda \in \Lambda$.

We first show p-LongestPath $\in$ para-S-DynFO. The dynamic program uses a dynamic programming approach (in the classical sense of this term). It stores, for each $\lambda \in \Lambda$ and each pair (u, v) of nodes, the set $\mathcal{C}$ of color sets C, for which there is a C-coloured path from u to v under λ.

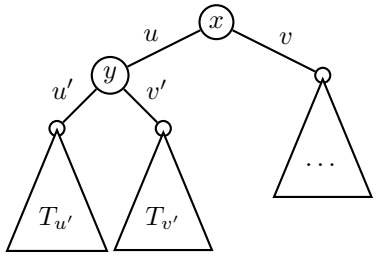
That p-LongestPath $\in$ para-T-FO can be shown with the help of the same universal colouring family Λ as above, which consists of $f(k)\mathsf{poly}(n)$ colourings. The idea for the program is to test, in $f(k)$ iterations and in each iteration for $\mathsf{poly}(n)$ colourings in parallel, whether there is a $[k]$-coloured path from s to t under the current colouring. A suitably coloured path can be found in k iterations. ◀

5.2 Bounded-depth search trees

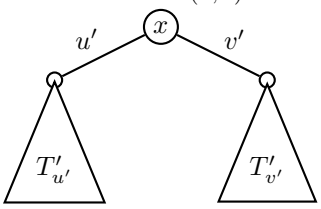
Bounded-depth search trees are a classical technique in parameterised complexity. Already in Example 4 we outlined that search trees are a viable tool also in the dynamic context by showing how a search tree for p-VertexCover can be maintained under edge insertions. Here we provide more examples. First we extend Example 4 towards edge deletions. Afterwards we consider two further problems, for which the known search-tree based algorithms can be adapted to place them in para-T-FO or para-T-DynFO, respectively: p-ClosestString and p-FeedbackVertexSet. Although we conjecture that these problems are also in para-S-DynFO, we were not able to prove it.

▶ **Proposition 10.** (p-VertexCover, $\Delta_E \cup \pm 1$) $\in$ para-S-DynFO *by a search-tree-based dynamic program.*

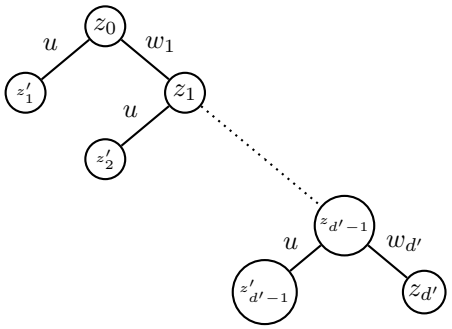
Proof sketch. Let T and T' be defined as in Example 4. It remains to explain how edge deletions can be handled. If an edge (u, v) is deleted, and a node x of T' used (u, v) for its branching step, the induced subtree of x can be replaced by the induced subtree of its left



(a) Before deletion of (u, v)



(b) After deletion of (u, v)



(c) New subtree of node $z = z_0$ at depth $d < k_{\max} - 1$. In each node z_i the edge (u, w_i) for some $w_i \neq v$ is covered. Since all uncovered edges contain u, all z'_i represent vertex covers. The node $z_{d'}$ represents a vertex cover if and only if $d' = \ell$, since then all ℓ edges that are not covered in z are covered by $w_1, \ldots, w_\ell$.

Figure 2 Modification of the search tree for p-VertexCover after deletion of an edge (u, v). The new sub-trees $T'_{u'}$, $T'_{v'}$ of x are obtained from $T_{u'}$, $T_{v'}$ respectively, by adding two new children to leaves that do not represent a vertex cover.

child y, see Figure 2.[8] More precisely, the children u' and v' of y become the new children of x, and in all candidate sets below u' and v' the vertex u is removed.

The subtree of x might now (1) have leaves of depth $k_{\max} - 1$ that do not represent an actual vertex cover, since the modification reduces the depth of all nodes in the subtree of x, and (2) have leaves at a smaller depth $d < k_{\max}-1$ which do not represent a vertex cover, since u is removed from the candidate sets and thus edges adjacent to u may not be covered any more. These defects can be corrected successively.

First, for each of the leaves from (1), two new children are added, with the help of the lexicographically smallest uncovered edge (u'', v'').

Regarding a leave z with property (2), observe that its candidate set can miss only edges of the form (u, w), where $w \neq v$. It is easy to see that the subtree rooted at z can be chosen in the following shape. Let $W = \{w_1, \ldots, w_\ell\}$ be the set of vertices with an uncovered edge $(u, w_i), i \in [\ell]$. The new subtree having depth $d' = \min\{\ell, k_{\max} - d\}$ consists of a path with nodes $z_0, \ldots, z_{d'}$ such that $z_0 = z$ and for each $i \geq 0$, the left child of z_i is a leaf obtained by adding u to the candidate set and for the right child z_{i+1}, w_{i+1} is added to the candidate set.

This new subtree can be defined in a first-order fashion with the help of colour coding. Let U be the candidate set of z. Then W consists of all neighbours of u that are not in U, so W is easily FO-definable. To define the subtree, d' vertices have to be chosen from W. To this end, we consider colourings of W that map W to $[\ell]$. With the help of an $(n, k_{\max}, k_{\max})$-universal colouring family, one can quantify over such colourings and by picking (a canonical) one, the new subtree can be defined by choosing each w_i as the node coloured with i, for every $i \in [d']$. All these updates can be expressed by first-order formulas. ◀

For the closest string problem, we fix an alphabet Σ, and let $d_H(s_1, s_2)$ denote the Hamming distance of s_1 and s_2, i.e. the number of positions where s_1 and s_2 differ.

[8] Of course, the right child would work equally well.

Problem: p-ClosestString
Input: Strings $s_1, \ldots, s_n \in \Sigma^n$ for some $n \in \mathbb{N}$, and $d \in \mathbb{N}$, **Parameter:** d
Question: Is there a string $s \in \Sigma^n$ such that $d_H(s, s_i) \leq d$?

An input to p-ClosestString with strings of length n is represented by a structure with domain $[n]$. It has the natural linear order on $[n]$ and, for every $\sigma \in \Sigma$ a relation $R_\sigma(i, j)$ with the meaning $s_i[j] = \sigma$, i.e. string s_i has symbol σ at position j.

A search tree (see [29, Section 8.5]) of depth at most d and degree at most $d+1$ gradually adapts a candidate string s, which is initially set to s_1. If an input string s_i is "far apart" from s, the tree branches on the first $d+1$ differences and changes s towards s_i.

▶ **Proposition 11.** p-ClosestString $\in$ para-T-FO.

The construction is quite straightforward and can be found in the full version.

Next, we explore the parameterised problem p-FeedbackVertexSet. Given a graph $G = (V, E)$, a feedback vertex set (FVS) for G is a set $S \subseteq V$ such that for every cycle C in G, $S \cap C \neq \emptyset$ holds, i.e. $G - S$ is a forest.

Problem: p-FeedbackVertexSet
Input: An undirected graph G, **Parameter:** k
Question: Does G have a feedback vertex set of size k?

▶ **Proposition 12.** $(p\text{-FeedbackVertexSet}, \Delta_E \cup \pm 1) \in$ para-T-DynFO.

Proof idea. We show that p-FeedbackVertexSet can be maintained in para-T-DynFO using a depth-bounded search tree, similarly as for p-VertexCover. The result uses a well-known approach relying on the fact that if a graph of minimum degree 3 has a FVS of size k then the length of its minimal cycle is bounded by $2k$ (e.g. [18]). A branching step consists of two phases: removing vertices of degree 1 or 2, and finding a small cycle. Then, each branch selects one of these cycle vertices for the FVS candidate. At the leaves of the search tree it has to be checked if the graph obtained by deleting the chosen vertices of the current branch is acyclic. A cycle exists, if there exists an edge (u, v) and u is reachable from v in $G - (u, v)$, thus this can be decided with the transitive closure of the edge relation. The latter can be maintained in DynFO under edge insertions and deletions [12] and, as we show in the full version of this paper, also under vertex deletions (simulated by removing all edges of a vertex). ◀

5.3 Kernelisation

Bannach and Tantau [5, Theorem 2.3] show that the famous meta-theorem "a problem is fixed parameter tractable if and only if a kernel for it can be computed in polynomial time" can be adapted to connect the AC-hierarchy with its parameterised counterpart. In this section we (partially) translate this relationship to the parameterised, dynamic setting.

A *kernelisation* of a Boolean parameterised query (Q, κ) over schema τ is a self-reduction K from τ-structures to τ-structures such that (1) $\mathcal{I} \in Q$ if and only if $K(\mathcal{I}) \in Q$, and (2) $|K(\mathcal{I})| \leq h(\kappa(\mathcal{I}))$, for all τ-structures $\mathcal{I}$ and some fixed computable function $h : \mathbb{N} \to \mathbb{N}$. The images of a kernelisation K are called *kernels*. We say that a kernel of (Q, κ) can be maintained in some class $\mathcal{C}$ under some set Δ of change operations, if the kernels with respect to some kernelisation K can be maintained in $\mathcal{C}$ under changes from Δ.

▶ **Theorem 13.** *Let (Q, κ, Δ) be a Boolean parameterised dynamic query of τ-structures.*

(a) *If a kernel for (Q, κ) can be maintained under Δ in* DynFO$(+, \times)$ *then (Q, κ, Δ) is in* para-S-DynFO*. In addition, if (Q, κ) has an explicit parameter and $\Delta = \Delta_\tau \cup \pm 1$ then also the converse holds.*

(b) *If $Q \in$ PSPACE and a kernel for (Q, κ) can be maintained under Δ in* DynFO$(+, \times)$ *then (Q, κ, Δ) is in* para-T-DynFO*.*

Proof sketch. Towards proving (a), suppose that a kernel of (Q, κ) with respect to a kernelisation K can be maintained under Δ by a DynFO$(+, \times)$-program $\mathcal{P}$. A para-S-DynFO-program $\mathcal{P}'$ for (Q, κ, Δ) maintains a kernel for the current input structure by simulating $\mathcal{P}$. The kernel $K(\mathcal{I})$ of an input structure $\mathcal{I}$ is represented by at most $h(\kappa(\mathcal{I}))$ elements, where h is the function from the second condition of the definition of the kernelisation K. Therefore $\mathcal{P}'$ can check whether $K(\mathcal{I}) \in Q$ by Lemma 3 and Proposition 7.

For proving the converse of (a) under the stated assumptions, suppose that (Q, κ) has an explicit parameter and that $\Delta = \Delta_\tau \cup \pm 1$. We construct, from a para-S-DynFO-program $\mathcal{P}$ with advice π that maintains (Q, κ, Δ), a DynFO$(+, \times)$-program $\mathcal{P}'$ that maintains a kernel for (Q, κ). The idea is to use a standard trick from parameterised complexity, a case distinction between small and large parameters. If the parameter is small enough in comparison to the domain size, $\mathcal{P}'$ can compute the advice structure of $\mathcal{P}$ at initialisation time and can simulate $\mathcal{P}$ from then on. If the parameter is large, $\mathcal{P}'$ uses the "small" input instance as a trivial kernel.

Towards proving (b), suppose that a kernel of (Q, κ) with respect to a kernelisation K can be maintained under Δ by a DynFO$(+, \times)$-program $\mathcal{P}$, and that $Q \in$ PSPACE. Recall that unlimited (or equivalently exponential) iteration of FO-formulas captures PSPACE over ordered structures (see, e.g., [24, Theorem 10.13]). A para-T-DynFO-program can maintain the current kernel $K(\mathcal{I})$ by simulating $\mathcal{P}$. After updating the kernel after a change, it computes the result of Q for $K(\mathcal{I})$ by iterating the first-order formulas of the PSPACE algorithm with a parameterised first-order program. Since at most $2^{|K(\mathcal{I})|^{O(1)}}$ iterations are necessary, it follows that the first-order program only needs a parameterised number of iterations. ◀

The assumptions for the proof of the second part of (a) are chosen because they are easy to state and satisfied by many natural parameterised dynamic queries. They can be relaxed though and, as an example, the result also holds for the standard change operations and the non-explicit parameter "maximal node degree" for graphs.

We now give an example of an algorithm whose underlying kernelisation can be simulated in DynFO$(+, \times)$. For a set of points in $\mathbb{N}^d$, for some $d \geq 2$, a *cover* is a set of lines such that each of the points is on at least one line. For a fixed dimension $d \geq 2$, the problem p-d-PointLineCover ("PointLineCover") is defined as follows:

Problem: p-d-PointLineCover
Input: Distinct points $\bar{p}_1, \ldots, \bar{p}_n \in \mathbb{N}^d$, **Parameter:** k
Question: Is there a cover of the points of size k?

Each point $\bar{p}_i$ with $i \in [n]$ is given by d coordinates $p_i^1, \ldots, p_i^d$ of n bits each. To encode these numbers, we identify the domain of size n with the set $[n]$ and use d binary relations $X^1, \ldots, X^d$. We let $(i, j) \in X^\ell$ if the j-th bit of p_i^ℓ is 1.

A classical kernel (see e.g. [25] or [26]) for p-d-PointLineCover can be obtained by realising that if a line contains at least $k+1$ points then it has to be used in a cover. Otherwise the points on this line can only be covered by using at least $k+1$ distinct lines. A kernel

for an instance can now be constructed by iteratively applying the following rule as long as possible: remove all points that belong to a simple line that contains at least $k+1$ points and reduce k by 1. If, in the end, more than k^2 points remain, there is no line cover with k lines.

In [5] it was observed that the above reduction can be performed in parallel, since removing all points of a line removes at most one point from any other line. This immediately yields that p-d-PointLineCover is in para-TC^0, since lines with at least $k+1$ points can be identified in TC^0. The problem, however, is not in para-AC^0 = para-S-FO [5] due to the bottleneck that collinearity of n-bit points cannot be tested in AC^0.

We show that with an oracle for testing whether three points are collinear, a kernel of p-d-PointLineCover can be actually expressed in $\mathsf{FO}(+,\times)$. Since collinearity of three points can be maintained in $\mathsf{DynFO}(+,\times)$ under bit changes of points, a kernel can be maintained in $\mathsf{DynFO}(+,\times)$. Here the allowed changes are to modify single bits of the points $\bar{p}_1,\dots,\bar{p}_n$, to enable or disable a point, and to change the number k. To allow that points can be enabled or disabled, we add an additional unary relation P to structures that contains i if $\bar{p}_i$ is part of the current instance, that is, if it is *enabled*.

▶ **Lemma 14.** *Collinearity of three d-dimensional points with n-bit coordinates can be maintained in* $\mathsf{DynFO}(+,\times)$ *under changes of single bits, for each fixed $d \in \mathbb{N}$.*

▶ **Theorem 15.** *Let* $\Delta \stackrel{def}{=} \Delta_{\{X_1,\dots,X_d,P\}} \cup \{\pm 1\}$.

(a) $(p\text{-}d\text{-PointLineCover}, \Delta) \in$ para-S-DynFO

(b) $(p\text{-}d\text{-PointLineCover}, \Delta) \in$ para-T-DynFO

Proof idea. By the previous lemma, a dynamic program can maintain a relation C that contains a triple (i_1,i_2,i_3) if the points $\bar{p}_{i_1},\bar{p}_{i_2},\bar{p}_{i_3}$ are collinear, using Lemma 14. The statement now follows from Theorem 13 and the observation that a kernel can be defined in $\mathsf{FO}(+,\times)$ from C.

If $k \geq \log n$, the input structure $\mathcal{I}$ itself is a kernel of size at most $f(k)$. Otherwise, the counting abilities of $\mathsf{FO}(+,\times)$ (see for example [14]) can be used to define a kernel. Since $k < \log n$, the set L of lines with at least $k+1$ enabled points can be defined in $\mathsf{FO}(+,\times)$, as well as the number $|L|$ of such lines. Additionally, the set P of enabled points that are not on any line from L is definable, and it can be determined in $\mathsf{FO}(+,\times)$ whether there are more than k^2 of these points. Then the current kernel is defined as follows. If $|L| > k$, or $|L| \leq k$ and $|P| > k^2$, then it outputs a constant no-instance. Otherwise the kernel is the set P with the parameter $k - |L|$. ◀

5.4 Dynamic programming

Dynamic programming is a fundamental technique in algorithm design and as such it has been applied in the field of parameterised algorithms many times (e.g., [29, Section 9]). A classical parameterised algorithm with dynamic programming shows p-Knapsack $\in \mathsf{FPT}$.

Problem: p-Knapsack
Input: A set of n items with profits $p_1,\dots,p_n$ and weights $w_1,\dots,w_n$, a capacity bound B and a profit threshold T, **Parameter:** B
Question: Is there a subset $S \subseteq [n]$ such that $\sum_{i\in S} p_i \geq T$ and $\sum_{i\in S} w_i \leq B$?

All numbers are from $\mathbb{N}$ and given as n-bit numbers. We choose a similar input encoding as for p-d-PointLineCover in Subsection 5.3: we identify the domain of size n with the set $[n]$, encode the profits p_i using a binary relation P such that $(i,j) \in P$ if the j-th bit of

p_i is 1, and analogously encode the weights w_i and the numbers B, T by a binary relation W and unary relations B, T, respectively.[9]

▶ **Proposition 16.** $(p\text{-}\textsc{Knapsack}, \Delta_{\text{KS}}) \in$ para-S-DynFO.

Here, Δ_{KS} denotes the set of changes that can arbitrarily replace the profit and the weight of one item, and set a number B or T to any value.

Proof sketch. The program combines the usual static algorithm with an idea that was used to capture regular languages in DynFO [22]. Intuitively, it maintains a three-dimensional table A such that $A(i, j, b)$ gives the maximum profit one can achieve by picking items with overall weight exactly b from $\{i, \ldots, j\}$. This table is encoded by a relation A_{BIT} of arity four in a straightforward manner. ◀

5.5 Iterative compression

The iterative compression method (introduced in [31], see also [29, Section 11.3]) is used to obtain fixed parameter tractable algorithms for minimisation problems which are parameterised by the solution size. It can roughly be described as follows: First, a trivial solution is computed for a very small fraction of the input instance. Afterwards, the fraction is continuously increased and each time a straightforwardly updated (but maybe too big) solution is constructed and improved ("compressed") afterwards (if necessary), until the input instance is completed and a valid solution is constructed. We illustrate the transfer of this technique to the dynamic setting with p-VertexCover. First we describe intuitively, how the static algorithm described in [29, Subsection 11.3.2] can be adapted to the dynamic setting.

Let $G = (V, E)$ and $G' = (V, E')$ be two input graphs, where G' results from G by inserting one edge $e = (u, v)$. Let us assume that C_0 is an optimal vertex cover for G of size k. The set $C = C_0 \cup \{u\}$ of size $k + 1$ is trivially a vertex cover for G', but the optimal one C' might have size k. The crucial observation is that if $C' = Z \cup Z'$ has size k, for a subset Z of C and a set Z' disjoint from C, then Z' must consist of all neighbours of vertices in $C - Z$ that are not in Z. By a combination of colour coding with an adaptation of a technique from [13] for the parameterised setting, a dynamic program with advice (for the universal colouring family) can basically try out all subsets of C for Z.

▶ **Proposition 17.** $(p\text{-}\textsc{VertexCover}, \Delta_{E \cup \pm 1}) \in$ para-S-FO *by a compression-based dynamic program.*

6 Conclusion

In this work we started to investigate dynamic complexity from a parameterised algorithms point of view. Besides the definition of the framework, we explored how well-known techniques from parameterised algorithms translate to our setting. Kernelisation and colour-coding worked quite well for both settings. Search-tree based techniques translated well to the setting with parameterised time and were more challenging for parameterised space. On the other hand, dynamic programming (with superpolynomial parameter values) seems better suited for parameterised space. The compression-based program for p-VertexCover translates,

[9] We note that this restricts the possible weights and profits to numbers bounded by $2^n - 1$. Larger values can be achieved by a larger domain, where additionally represented items have profit and weight 0.

in principle, also to para-T-DynFO but the handling of instances with large minimal vertex cover basically requires an additional implementation of some other method and therefore makes this approach a bit pointless. We also considered *greedy localisation* and algorithms for structures with bounded tree-width, but did not find any meaningful applications in the dynamic setting, as discussed in the full version of this paper.

Particular open questions are whether p-ClosestString or p-FeedbackVertexSet can be maintained with parameterised space and whether para-ST-DynFO is more expressive than para-S-DynFO.

References

1 Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic Parameterized Problems and Algorithms. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017*, volume 80 of *LIPIcs*, pages 41:1–41:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.ICALP.2017.41.

2 Noga Alon, Raphael Yuster, and Uri Zwick. Color-Coding. *J. ACM*, 42(4):844–856, 1995. doi:10.1145/210332.210337.

3 Max Bannach, Christoph Stockhusen, and Till Tantau. Fast Parallel Fixed-Parameter Algorithms via Color Coding. In *10th International Symposium on Parameterized and Exact Computation, IPEC 2015*, pages 224–235. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.IPEC.2015.224.

4 Max Bannach and Till Tantau. Computing Hitting Set Kernels by AC^0-Circuits. In Rolf Niedermeier and Brigitte Vallée, editors, *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018*, volume 96 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.STACS.2018.9.

5 Max Bannach and Till Tantau. Computing Kernels in Parallel: Lower and Upper Bounds. In Christophe Paul and Michal Pilipczuk, editors, *13th International Symposium on Parameterized and Exact Computation,IPEC 2018*, pages 13:1–13:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.IPEC.2018.13.

6 Max Bannach and Till Tantau. On the Descriptive Complexity of Color Coding. In Rolf Niedermeier and Christophe Paul, editors, *36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019*, volume 126 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.STACS.2019.11.

7 David A. Mix Barrington, Neil Immerman, and Howard Straubing. On Uniformity within NC^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990. doi:10.1016/0022-0000(90)90022-D.

8 Hans-Joachim Böckenhauer, Elisabet Burjons, Martin Raszyk, and Peter Rossmanith. Reoptimization of Parameterized Problems, 2018. arXiv:1809.10578.

9 Marco Cesati and Miriam Di Ianni. Parameterized Parallel Complexity. In David J. Pritchard and Jeff Reeve, editors, *Euro-Par '98 Parallel Processing, 4th International Euro-Par Conference, Proceedings*, volume 1470 of *Lecture Notes in Computer Science*, pages 892–896. Springer, 1998. doi:10.1007/BFb0057945.

10 Yijia Chen and Jörg Flum. Some Lower Bounds in Parameterized AC^0. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016*, volume 58 of *LIPIcs*, pages 27:1–27:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.27.

11 Yijia Chen, Jörg Flum, and Xuangui Huang. Slicewise Definability in First-Order Logic with Bounded Quantifier Rank. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic, CSL 2017*, volume 82 of *LIPIcs*, pages 19:1–19:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CSL.2017.19.

12 Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability Is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018. doi:10.1145/3212685.

13 Samir Datta, Anish Mukherjee, Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. A Strategy for Dynamic Programs: Start over and Muddle through. *Logical Methods in Computer Science*, 15(2), 2019. doi:10.23638/LMCS-15(2:12)2019.

14 Larry Denenberg, Yuri Gurevich, and Saharon Shelah. Definability by Constant-Depth Polynomial-Size Circuits. *Information and Control*, 70(2/3):216–240, 1986. doi:10.1016/S0019-9958(86)80006-7.

15 Guozhu Dong, Jianwen Su, and Rodney W. Topor. Nonrecursive Incremental Evaluation of Datalog Queries. *Ann. Math. Artif. Intell.*, 14(2-4):187–223, 1995. doi:10.1007/BF01530820.

16 Rodney G. Downey, Judith Egan, Michael R Fellows, Frances A Rosamond, and Peter Shaw. Dynamic Dominating set and Turbo-Charging Greedy Heuristics. *Tsinghua Science and Technology*, 19(4):329–337, 2014. doi:10.1109/TST.2014.6867515.

17 Rodney G. Downey and Michael R. Fellows. Fixed-Parameter Tractability and Completeness I: Basic Results. *SIAM J. Comput.*, 24(4):873–921, 1995. doi:10.1137/S0097539792228228.

18 Rodney G. Downey and Michael R. Fellows. Parameterized Computational Feasibility. In *Feasible mathematics II*, pages 219–244. Springer, 1995. doi:10.1007/978-1-4612-2566-9_7.

19 Michael Elberfeld, Christoph Stockhusen, and Till Tantau. On the Space and Circuit Complexity of Parameterized Problems: Classes and Completeness. *Algorithmica*, 71(3):661–701, 2015. doi:10.1007/s00453-014-9944-y.

20 Jörg Flum and Martin Grohe. Describing Parameterized Complexity Classes. *Inf. Comput.*, 187(2):291–319, 2003. doi:10.1016/S0890-5401(03)00161-5.

21 Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984. doi:10.1007/BF01744431.

22 Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. The dynamic complexity of formal languages. *ACM Trans. Comput. Log.*, 13(3):19, 2012. doi:10.1145/2287718.2287719.

23 Sepp Hartung and Rolf Niedermeier. Incremental List Coloring of Graphs, Parameterized by Conservation. *Theor. Comput. Sci.*, 494:86–98, 2013. doi:10.1016/j.tcs.2012.12.049.

24 Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999. doi:10.1007/978-1-4612-0539-5.

25 Stefan Kratsch, Geevarghese Philip, and Saurabh Ray. Point Line Cover: The Easy Kernel is Essentially Tight. *ACM Trans. Algorithms*, 12(3):40:1–40:16, 2016. doi:10.1145/2832912.

26 Stefan Langerman and Pat Morin. Covering Things with Things. *Discrete & Computational Geometry*, 33(4):717–729, 2005. doi:10.1007/s00454-004-1108-4.

27 Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004. doi:10.1007/978-3-662-07003-1.

28 Bernard Mans and Luke Mathieson. Incremental Problems in the Parameterized Complexity Setting. *Theory Comput. Syst.*, 60(1):3–19, 2017. doi:10.1007/s00224-016-9729-6.

29 Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Number 31 in Oxford Lecture Series in Mathematics and its Applications. Oxford University Press, 2006. doi:10.1093/acprof:oso/9780198566076.001.0001.

30 Sushant Patnaik and Neil Immerman. Dyn-FO: A Parallel, Dynamic Complexity Class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997. doi:10.1006/jcss.1997.1520.

31 Bruce A. Reed, Kaleigh Smith, and Adrian Vetta. Finding Odd Cycle Transversals. *Oper. Res. Lett.*, 32(4):299–301, 2004. doi:10.1016/j.orl.2003.10.009.

32 Thomas Schwentick and Thomas Zeume. Dynamic Complexity: Recent Updates. *SIGLOG News*, 3(2):30–52, 2016. doi:10.1145/2948896.2948899.