# Dynamic Parameterized Problems and Algorithms*†

**Josh Alman[1], Matthias Mnich[2], and Virginia Vassilevska Williams[3]**

1 **MIT CSAIL, Cambridge, MA, USA**
jalman@mit.edu
2 **Universität Bonn, Institut für Informatik, Bonn, Germany; and Maastricht University, Department of Quantitative Economics, Maastricht, The Netherlands**
mmnich@uni-bonn.de
m.mnich@maastrichtuniversity.nl
3 **MIT CSAIL, Cambridge, MA, USA**
virgi@mit.edu

## Abstract

Fixed-parameter algorithms and kernelization are two powerful methods to solve NP-hard problems. Yet, so far those algorithms have been largely restricted to *static* inputs.

In this paper we provide fixed-parameter algorithms and kernelizations for fundamental NP-hard problems with *dynamic* inputs. We consider a variety of parameterized graph and hitting set problems which are known to have $f(k)n^{1+o(1)}$ time algorithms on inputs of size $n$, and we consider the question of whether there is a data structure that supports small updates (such as edge/vertex/set/element insertions and deletions) with an update time of $g(k)n^{o(1)}$; such an update time would be essentially optimal. Update and query times independent of $n$ are particularly desirable. Among many other results, we show that Feedback Vertex Set and $k$-Path admit dynamic algorithms with $f(k)\log^{O(1)} n$ update and query times for some function $f$ depending on the solution size $k$ only.

We complement our positive results by several conditional and unconditional lower bounds. For example, we show that unlike their undirected counterparts, Directed Feedback Vertex Set and Directed $k$-Path do not admit dynamic algorithms with $n^{o(1)}$ update and query times even for constant solution sizes $k \leq 3$, assuming popular hardness hypotheses. We also show that unconditionally, in the cell probe model, Directed Feedback Vertex Set cannot be solved with update time that is purely a function of $k$.

**1998 ACM Subject Classification** F2.2 Nonnumerical Algorithms and Problems

**Keywords and phrases** Dynamic algorithms, fixed-parameter algorithms

**Digital Object Identifier** 10.4230/LIPIcs.ICALP.2017.41

## 1 Introduction

The area of dynamic algorithms studies data structures that store a dynamically changing instance of a problem, can answer queries about the current instance and can perform small changes on it. The major question in this area is, how fast can updates and queries be?

* A full version of the paper is available at https://arxiv.org/abs/1707.00362.

† J.A. is supported by NSF Grant DGE-114747. M.M. is supported by ERC Starting Grant 306465 (BeyondWorstCase). V.V.W. is supported by NSF Grants CCF-141-7238, CCF-1528078 and CCF-1514339, and BSF Grant BSF:2012338. This work was initiated while J.A. and V.V.W. were at Stanford University.

© Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams;
licensed under Creative Commons License CC-BY



The most studied dynamic problems are dynamic graph problems such as connectivity (e.g., [45, 47, 48, 67]), reachability [41], shortest paths (e.g., [7, 26, 42]), and maximum matching [8, 39, 66]. For a dynamic graph algorithm, the updates are usually edge or vertex insertions and deletions. Any dynamic graph algorithm that can perform edge insertions can be used for a static algorithm by starting with an empty graph, and using $m$ insertions to insert the $m$-edge input graph. That is, if the update time of the dynamic algorithm is $u(m)$ then the static problem can be solved in $O(m \cdot u(m))$ time, plus the time to query for the output. Hence, if a problem requires $\Omega(f(m))$ time to be solved statically, then any dynamic algorithm that can insert edges, and can be queried for the problem solution in $o(f(m))$ time, must need $\Omega(f(m)/m)$ (amortized) time to perform updates. This is not limited to edge updates; similar statements are true for vertex insertions and other update types. A fundamental question is which problems can be fully dynamized, i.e., have dynamic algorithms supporting updates in $O(f(m)/m)$ time where $f(m)$ is the static runtime?

This question is particularly interesting for static problems that can be solved in near-linear time. For them, we are interested in near-constant time updates—the holy grail of dynamic algorithms. The field of dynamic algorithms has achieved such full dynamization for many problems. A prime example of the successes of this vibrant research area is the *dynamic connectivity* problem: maintaining the connected components of a graph under edge updates, to answer queries about whether a pair of vertices is connected. This problem can be solved with amortized expected update time $O(\log n \log\log^2 n)$ [48, 67] and query time $O(\log n/\log\log\log n)$; polylogarithmic deterministic amortized bounds are also known, the current best by Wulff-Nielsen [71]. After much intense research on the topic [44, 46, 47], the first polylogarithmic *worst case expected* update times were obtained by Kapron et al. [53], who were the first to break through what seemed like an $\Omega(\sqrt{n})$ barrier; the bounds of Kapron et al. [53] were recently improved by Gibb et al. [37]. Similar $\tilde{O}(1)$ update and query time bounds[1] are known for many problems solvable in linear time such as dynamic minimum spanning tree, biconnectivity and 2-edge connectivity [45, 47], and maximal matching [5, 66].

Barriers for dynamization have also been studied extensively. Many unconditional, cell probe lower bounds are known. For instance, for connectivity and related problems it is known [64, 65] that either the query time or the update time needs to be $\Omega(\log n)$. However, current cell probe lower bound techniques seem to be limited to proving polylogarithmic lower bounds. In contrast, conditional lower bounds based on popular hardness hypotheses have been successful at giving tight bounds for problems such as dynamic reachability, dynamic strongly connected components and many more [1, 43, 54, 62].

While the field of dynamic algorithms is very developed, practically all the problems which have been studied are polynomial-time solvable problems. What about NP-hard problems? Do they have fast dynamic algorithms? By the discussion above, it seems clear that (unless $\mathsf{P} = \mathsf{NP}$), superpolynomial query/update times are necessary, and surely this is not as interesting as achieving near-constant time updates. If the problem is relaxed, and instead of exact solutions, approximation algorithms are sufficient, then efficient dynamic algorithms have been obtained for some polynomial time approximable problems such as dynamic approximate vertex cover [5, 8, 61]. What if we insist on exact solutions?

The efficient dynamization question does make sense for *parameterized* NP-hard problems. For such problems, each instance is measured by its size $n$ as well as a *parameter* $k$ that measures the optimal solution size, the treewidth or genus of the input graph, or any similar structural property. If $\mathsf{P} \neq \mathsf{NP}$, then the runtime of any algorithm for such a problem needs

[1] Throughout this paper, we write $\tilde{O}(f(n,k))$ to hide polylog$(n)$ factors.

to be superpolynomial, but it is desirable that the superpolynomiality is only in terms of $k$. That is, one searches for so-called *fixed-parameter algorithms* with runtime $f(k) \cdot n^c$ for some computable function $f$ and some fixed constant $c$ independent of $k$ and $n$. The *holy grail* here is an algorithm with runtime $f(k) \cdot n$ where $f$ is a modestly growing function. Such linear-time fixed-parameter algorithms can be very practical for small $k$. The very active area of fixed-parameter algorithms has produced a plethora of such algorithms for many different parameterized problems. Some examples include (1) many branching tree algorithms such as those for Vertex Cover and $d$-Hitting Set, (2) many algorithms based on color-coding [4] such as for $k$-Path, (3) all algorithms that follow from Courcelle's theorem[2] [18], and (4) many more [11, 28, 32, 52, 58, 68, 70].

We study whether NP-hard problems with (near-)linear time fixed-parameter algorithms can be made efficiently dynamic. The main questions we address are:

- Which problems solvable in $f(k) \cdot n^{1+o(1)}$ time have dynamic algorithms with update and query times at most $f(k) \cdot n^{o(1)}$?
- Which problems solvable in $f(k) \cdot n$ time have dynamic algorithms with update and query times that depend solely on $k$ and not on $n$?
- Can one show that (under plausible conjectures) a problem requires $\Omega(f(k) \cdot n^{\delta})$ (for constant $\delta > 0$) update time to maintain dynamically even though statically it can be solved in $f(k) \cdot n^{1+o(1)}$ time?

## 1.1 Prior Work

We are aware of only a handful papers related to the question that we study. Bodlaender [9] showed how to maintain a tree decomposition of constant treewidth under edge and vertex insertions and deletions with $O(\log n)$ update time, as long as the underlying graph always has treewidth at most 2. Dvořák et al. [29] obtained a dynamic algorithm maintaining a tree-depth decomposition of a graph under the promise that the tree-depth never exceeds $D$; edge and vertex insertions and deletions are supported in $f(D)$ time for some function $f$. Dvořák and Tůma [30] obtained a dynamic data structure that can count the number of induced copies of a given $h$-vertex graph, under edge insertions and deletions, and if the maintained graph has bounded expansion, the update time is bounded by $O(\log^{h^2} n)$.

A more recent paper by Iwata and Oka [51] gives several dynamic algorithms for the following problems, under the promise that the solution size never grows above $k$: (1) an algorithm that maintains a Vertex Cover in a graph under $O(k^2)$ time edge insertions and deletions and $f(k)$ time queries[3], (2) an algorithm for Cluster Vertex Deletion under $O(k^8 + k^4 \log n)$ time edge updates and $f(k)$ time queries[4], and (3) an algorithm for Feedback Vertex Set in graphs with maximum degree $\Delta$ where edge insertions and deletions are supported in amortized time $2^{O(k)}\Delta^3 \log n$. Notably, when discussing Feedback Vertex Set, the paper concludes: "It seems an interesting open question whether it is possible to construct an efficient dynamic graph without the degree restriction."

The final related papers are by Abu-Khzam et al. [2, 3]. Although these papers talk about parameterized problems and dynamic problems, the setting is very different. Their

[2] Courcelle's theorem states that every problem definable in monadic second-order logic of graphs can be decided in linear time on graphs of bounded treewidth.

[3] Here $f(k)$ denotes the runtime of the fastest fixed-parameter algorithm for Vertex Cover when run on $k$-vertex graphs.

[4] Here $f(k)$ denotes the runtime of the fastest fixed-parameter algorithm for Cluster Vertex Deletion when run on $k^5$-vertex graphs.

problem is, given two instances $I_1$ and $I_2$ of a problem that only differ in $k$ "edits", and a solution $S_1$ of $I_1$, to find a feasible solution $S_2$ of $I_2$ that is at Hamming distance at most $d$ from $S_1$. The question of study is whether such problems admit fixed-parameter algorithms for parameters $k$ and $\ell$. That question though is not about data structures but about a single update. Moreover, their algorithm is given $S_1$ as input, which—unlike a dynamic data structure—cannot force the initial solution to have any useful properties. Thus, the hardness results in their setting do not translate to our data structure setting. Furthermore, the runtimes in their setting, unlike ours, must have at least a linear dependence on the size of the input, as one has to at least read the entire input.

Besides the work on parameterized dynamic algorithms, there has been some work on parameterized streaming algorithms by Chitnis et al. [17]. This work focused on Maximal Matching and Vertex Cover. The difference between streaming and dynamic algorithms is that (a) the space usage of the algorithm is the most important aspect for streaming, and (b) in streaming, a solution is only required at the end of the stream, whereas a dynamic algorithm can be queried at any point and needs to be efficient throughout, but can use a lot of space. For Vertex Cover instances whose solution size never exceeds $k$, Chitnis et al. [17] give a one-pass randomized streaming algorithm that uses $O(k^2)$ space and answers the final query in $2^{O(k)}$ time; when the vertex cover size can exceed $k$ at any point, there is a one-pass randomized streaming algorithm using $O(\min\{m, nk\})$ space and answering the query in $O(\min\{m, nk\}) + 2^{O(k)}$ time.

The relevant prior work on (static) fixed-parameter algorithms [4, 6, 10, 12, 13, 14, 15, 16, 19, 20, 22, 23, 24, 27, 31, 34, 38, 40, 49, 50, 56, 57, 59, 68] is described in the appendix.

## 1.2 Our Contributions

**Algorithmic Results.** We first define the notion of a fixed parameter dynamic problem as a parameterized problem with parameter $k$ that has a data structure supporting updates and queries to an instance of size $n$ in time $f(k)n^{o(1)}$. The class FPD contains all such parameterized problems. By a formalization of our earlier discussion, FPD is contained in the class of parameterized problems admitting algorithms running in time $f(k)n^{1+o(1)}$. After this, we introduce two techniques for making fixed-parameter algorithms dynamic, and then use them to develop dynamic fixed-parameter algorithms for a multitude of fundamental optimization problems. Our algorithmic contributions are stated in Theorem 1 below. In the runtimes, $DC(n)$ refers to the time per update to a dynamic connectivity data structure on $n$ vertices, which from prior work can be:

- *expected amortized* update time $O(\log n(\log\log n)^2)$, or
- *expected worst case* time $O(\log^4 n)$, or
- *deterministic amortized* time $O(\log^2 n/\log\log n)$.

Which of these bounds we pick determines the type of guarantees (expected vs. deterministic, worst case vs. amortized) that the algorithm gives.

▶ **Theorem 1.** *The following problems admit dynamic fixed-parameter algorithms:*

- Vertex Cover *parameterized by solution size under edge insertions and deletions, with* $O(1)$ *amortized or* $O(k)$ *worst case update time and* $O(1.2738^k)$ *query time,*
- Connected Vertex Cover *parameterized by solution size under edge insertions and deletions, with* $O(k2^k)$ *update time and* $O(4^k)$ *query time,*
- $d$-Hitting Set *for all values of* $d$ *parameterized by solution size under set insertions and deletions, either with* $O(kd^k)$ expected *update time and* $O(k)$ *query time, or with*

$O(f(k,d))$ *(worst-case, deterministic) update time and* $O(d^k d!(k+1)^d)$ *query time, for some function* $f$ *loosely bounded by* $(d!)^d k^{O(d^2)}$.

- Edge Dominating Set *parameterized by solution size under edge insertions and deletions, with* $O(1)$ *update time and* $O(2.2351^k)$ *query time,*
- Feedback Vertex Set *parameterized by solution size under edge insertions and deletions, with* $2^{O(k \log k)} \log^{O(1)} n$ *amortized update time and* $O(k)$ *query time,*
- Max Leaf Spanning Tree *parameterized by solution size under edge insertions and deletions, with* $O(3.72^k + k^5 \log n + DC(n))$ *amortized update time, and maintains the current max leaf spanning tree explicitly in memory,*
- Dense Subgraph in Graphs with Degree Bounded by $\Delta$ *parameterized by the number of vertices in the subgraph under edge insertions and deletions, with* $2^{O(k\Delta)} \cdot DC(n)$ *update time and* $2^{O(k\Delta)} \log n$ *query time.*
- Undirected $k$-Path *parameterized by the number of vertices on the path, with* $k!2^{O(k)} \cdot DC(n)$ *update time and* $k!2^{O(k)} \log n$ *query time.*
- Edge Clique Cover *parameterized by the number of cliques and under the promise that the solution never grows bigger than* $g(k)$*, with* $O(4^{g(k)})$ *update time and* $2^{2^{O(k)}} + O(2^{4g(k)})$ *query time.*
- Point Line Cover *and* Line Point Cover *parameterized by the size of the solution and under point and line insertions and deletions, respectively, with* $O(g(k)^3)$ *update time and* $O(g(k)^{2g(k)+2})$ *query time, under the promise that the solution never grows to more than* $g(k)$.

**Discussion of the Algorithmic Results.** Our dynamic algorithm for Vertex Cover and that of Iwata and Oka [51] both have query time $O(1.2738^k)$, by using the best known fixed parameter algorithm for Vertex Cover on the maintained kernel. However, our algorithm improves upon theirs in two ways. First, our update time is amortized *constant* or $O(k)$ worst case, whereas the Iwata-Oka algorithm has update time $O(k^2)$. Second, their update time bound of $O(k^2)$ only holds if the vertex cover is guaranteed to never grow larger than $k$ throughout the sequence of updates. Namely, their update time depends on the size of their maintained kernel, which may become unbounded in terms of $k$. Our algorithm does not need any such promise—it will always have fast (amortized $O(1)$ or $O(k)$ worst case) update time and return a vertex cover of size $k$ if it exists, or determine that one does not. This is a much stronger guarantee.

Our dynamic algorithm and Chitnis et al.'s streaming algorithm for Vertex Cover are both based on Buss' kernel, but our algorithm is markedly different from theirs. In particular, we actually work with a modified kernel that allows us to achieve constant amortized update time. Because our algorithm is completely deterministic, it necessarily needs $\Omega(m)$ space, and our algorithm does indeed take linear space.

We give two algorithms for $d$-Hitting Set. The first is based on a randomized branching tree method, while the second is deterministic and maintains a small kernel for the problem. For every constant $d$, any $d$-Hitting Set instance on $m$ sets and $n$ elements has a kernel constructible in time $O(dn+2^d m)$ that has $O(d^{d+1}d!(k+1)^d)$ sets, due to van Bevern [68], and a kernel constructible in time $O(m)$ that has $O((k+1)^d)$ sets, due to Fafianie and Kratsch [33]. Unfortunately, it seems difficult to efficiently dynamize these kernel constructions. Because of this, we present a *novel kernel* for the problem. Our kernel can be constructed in $O(dn+3^d m)$ time and has size $O((d-1)!(k+1)^d)$. It also has nice properties that make it possible to maintain it dynamically with update time that is a function of only $k$ and $d$. In fact, for any fixed $d$, the update time is polynomial in $k$.

Our algorithm for Feedback Vertex Set is a nice combination of kernelization and a branching tree. Aside from our dynamic kernel for $d$-Hitting Set, this is probably the most involved of our algorithms. Iwata and Oka [51] had also presented a dynamic fixed-parameter algorithm for Feedback Vertex Set. However, their update time depends linearly on the maximum degree of the graph, and is hence efficient only for bounded degree graphs. Their paper asks whether one can remove this costly dependence on the degree. Our algorithm answers their question in the affirmative—it has fast updates regardless of the graph density.

All of our algorithms, except for the last two in the theorem, meet their update and query time guarantees regardless of whether the currently stored instance has a solution of size $k$ or not. The two exceptions, Edge Clique Cover and Point Line Cover, only work under the promise that the solution never grows bigger than a function of $k$. In this sense they are similar to most of the algorithms from prior work [29, 51]. There does seem to be an inherent difficulty to removing the promise requirement, however. In fact, in the parameterized complexity literature, these two problems are also exceptional, in the sense that their fastest fixed parameter algorithms run by computing a kernel and then running a brute force algorithm on it [23, 55], rather than anything more clever.

**Hardness Results.** In addition to the above algorithms, we also prove conditional lower bounds for several parameterized problems, showing that they are likely not in FPD. To our knowledge, ours are the first lower bounds for any dynamic parameterized problems.

The hardness hypothesis we assume concerns Reachability Oracles (ROs) for DAGs: an RO is a data structure that stores a directed acyclic graph and for any queried pair of vertices $s, t$, can efficiently answer whether $s$ can reach $t$. (An RO does not perform updates.) Our main hypothesis is as follows:

▶ **Hypothesis 2** (RO Hypothesis). *On a word-RAM with $O(\log m)$ bit words, any Reachability Oracle for directed acyclic graphs on $m$ edges must either use $m^{1+\varepsilon}$ preprocessing time for some $\varepsilon > 0$, or must use $\Omega(m^{\delta})$ time to answer reachability queries for some constant $\delta > 0$.*

The only known ROs either work by computing the transitive closure of the DAG during preprocessing, thus spending $\Theta(\min\{mn, n^{\omega}\})$ time (where $n$ is the number of vertices and $2 \leq \omega < 2.373$ [36, 69]), or by running a BFS/DFS procedure after each query, thus spending $O(m)$ time. Both of these runtimes are much larger than our assumed hardness; hence the RO Hypothesis is very conservative.

We also use a slightly weaker version of the RO Hypothesis, asserting that its statement holds true even restricted to DAGs that consist of $\ell$ layers of vertices (for some fixed constant $\ell$), so that the edges go only between adjacent layers in a fixed direction, from layer $i$ to layer $i + 1$. While this new *LRO Hypothesis* is certainly weaker, we show that it is implied by either of two popular hardness hypotheses: the 3SUM Conjecture and the Triangle Conjecture. The former asserts that when given $n$ integers within $\{-n^c, \ldots, n^c\}$ for some constant $c$, deciding whether three of them sum to 0 requires $n^{2-o(1)}$ time on a word-RAM with $O(\log n)$ bit words. The latter asserts that detecting a triangle in an $m$-edge graph requires $\Omega(m^{1+\varepsilon})$ time for some $\varepsilon > 0$. These two conjectures have been used for many conditional lower bounds [1, 35, 54].

Pătraşcu studied the RO Hypothesis, and while he was not able to prove it, the following strong cell probe lower bound follows from his work [63]: there are directed acyclic graphs on $m$ edges for which any RO that uses $m^{1+o(1)}$ preprocessing time (and hence space) in the word-RAM with $O(\log n)$ bit words, must have $\omega(1)$ query time. Using this statement, unconditional, albeit weaker lower bounds can be proven as well. This is what we prove:

▶ **Theorem 3.** *Fix the word-RAM model of computation with $w$-bit words for $w = O(\log n)$ for inputs of size $n$. Assuming the LRO Hypothesis, there is some $\delta > 0$ for which the following dynamic parameterized graph problems on $m$-edge graphs require either $\Omega(m^{1+\delta})$ preprocessing or $\Omega(m^{\delta})$ update or query time:*

- Directed $k$-Path *under edge insertions and deletions,*
- Steiner Tree *under terminal activation and deactivation, and*
- Vertex Cover Above LP *under edge insertions and deletions.*

*Under the RO Hypothesis (and hence also under the LRO Hypothesis), there is a $\delta > 0$ so that* Directed Feedback Vertex Set *under edge insertions and deletions requires $\Omega(m^{\delta})$ update time or query time.*

*Unconditionally, there is no computable function $f$ for which a dynamic data structure for* Directed Feedback Vertex Set *performs updates and answers queries in $O(f(k))$ time.*

Our lower bounds show that, although $k$-Path and Feedback Vertex Set have fixed parameter dynamic algorithms for undirected graphs, they probably do not for directed graphs. Interestingly, the fixed-parameter algorithms for $k$-Path in the static setting work similarly on both undirected and directed graphs, so there only seems to be a gap in the dynamic setting.

All problems for which we prove lower bounds have $f(k)n^{1+o(1)}$ time static algorithms, except for Vertex Cover above LP. However, it seems that the reason why the current algorithms are slower is largely due to the fact that near-linear time algorithms for maximum matching are not known. Recent impressive progress on the matching problem [60] gives hope that an $f(k)n^{1+o(1)}$ time algorithm for Vertex Cover above LP might be possible.

A common feature of most of the problems above is that they are either not known to have a polynomial kernel (like Directed Feedback Vertex Set), or do not have one unless $\mathsf{NP} \subseteq \mathsf{coNP}/poly$ (like $k$-Path [10] and Steiner Tree parameterized by the number of terminal pairs [27]). One might therefore conjecture that problems which cannot be made fixed parameter dynamic do not have polynomial kernels, or vice versa. Tempting as it is, this intuition turns out to be false. Vertex Cover Above LP does not have a dynamic fixed-parameter algorithm, yet it is known to admit a polynomial kernel [56]. On the other hand, the $k$-Path problem on undirected graphs also does not admit a polynomial kernel unless $\mathsf{NP} \subseteq \mathsf{coNP}/poly$ [10], yet we give a dynamic fixed-parameter algorithm for it. Hence, the existence of a polynomial kernel for a parameterized problem is not related to the existence of a dynamic fixed-parameter algorithm for it.

**Preliminaries.** We assume familiarity with basic combinatorial algorithms, especially graph algorithms and hitting set algorithms. When referring to a graph $G$, we will write $V(G)$ to denote its vertex set and $E(G)$ to denote its edge set. Unless otherwise specified, $n$ and $m$ will refer to the number of nodes and edges in $G$, respectively. We use the terms nodes and vertices interchangeably. By $\tilde{O}(f(n))$ we denote $f(n)\log^{O(1)} n$. We also assume familiarity with dynamic problems and parameterized problems.

## 2 Overview of the Algorithmic Techniques

**Promise model and Full model.** There are two different models of dynamic parameterized problems in which we design algorithms: the *promise model* and the *full model*. When solving a problem with parameter $k$ in the promise model, there is a computable function $g : \mathbb{N} \to \mathbb{N}$ such that one is promised that throughout the sequence of updates, there always

exists a solution with parameter at most $g(k)$. Hence, one only needs to maintain a solution under updates with good guarantees on both query and update times as long as the promise continues to hold. If at any point during the execution no solution to the parameterized problem with parameter $g(k)$ exists, then the algorithm is not required to provide any guarantees.

In the full model, there is no such promise. One needs to efficiently maintain a solution with parameter at most $k$, or the fact that no such solution exists, under any sequence of updates. When possible, it is desirable to have an algorithm with guarantees in the full model instead of only the promise model, and all but two of our algorithms (Point Line Cover and Edge Clique Cover) do work in the full model.

## 2.1 Techniques for designing dynamic fixed-parameter algorithms

We present two main techniques for obtaining dynamic fixed-parameter algorithms: dynamic kernels and dynamic branching trees.

**Dynamization via kernelization.** Using the notation of Cygan et al. [21], a *kernelization algorithm* for a parameterized problem $\Pi$ is an algorithm $\mathcal{A}$ that, given an instance $(I, k)$ of $\Pi$, runs in polynomial time and returns an instance $(I', k')$ of $\Pi$ such that the size of the new instance is bounded by a computable function of $k$ and so that $(I', k')$ is a 'yes' instance of $\Pi$ if and only if $(I, k)$ is. Frequently, when the problem asks us to output more than just a Boolean answer, then an answer for $(I', k')$ must be valid for $(I, k)$ as well. We will refer to the output of $\mathcal{A}$ as a *kernel*. For example, a kernelization algorithm for Vertex Cover might take as input a graph $G$, and return a subgraph $G'$ such that any vertex cover of $G'$ is also a vertex cover of $G$.

In the first approach, we compute a kernel for the problem, and maintain that this is a valid kernel as we receive updates. In other words, as we receive updates, we will maintain what the output of a kernelization algorithm $\mathcal{A}$ would be, without actually rerunning $\mathcal{A}$ each time. Similar to kernelizations for static fixed-parameter algorithms, if we can prove that the size of our kernel is only a function of $k$ whenever a solution with parameter $k$ exists, then we can answer queries in time independent of $n$ by running the fastest known static algorithms on the kernel.

The difficult part, then, is to efficiently dynamically maintain the kernel. The details of how efficiently we can handle updates to the kernel also determines which model of dynamic fixed-parameter algorithm our algorithm works for. If the kernel is defined by sufficiently simple or local rules such that updates can take place in time independent of the current kernel size, then the algorithm should work in the full model. If updates might take time linear in the kernel size, then the algorithm only works in the promise model.

As we will see, there are many problems for which we can efficiently maintain a kernel. In some instances we will be able to maintain the classical kernels known for the corresponding static problem, while in others, we will design new kernels which are easier to maintain.

**Dynamization via branching tree.** In the second approach, we consider so-called *set selection problems*. In these problems, the instance consists of a set of objects $U$ (e.g., vertices of a graph), the parameter is $k$, and one needs to select a subset $S \subseteq U$ of size $k$ (at least $k$/at most $k$) so that a certain predicate $P(S)$ is satisfied. Many parameterized problems are of this nature, such as $k$-Path, Vertex Cover, and (Directed) Feedback Vertex Set.

Consider a (static) set selection problem which admits a branching solution. By this we mean, for every instance $U$ of the problem, there is an 'easy to find' subset $T \subseteq U$ of size

$|T| \leq f(k)$ (for some function $f$) so that any solution $S$ of size at most $k$ must intersect $T$. Furthermore, for any choice of $t \in T$ to be placed in the solution, one can efficiently obtain a reduced instance of the problem with parameter $k-1$, which corresponds to picking $t$ to be in the solution. For instance, for Vertex Cover, every edge $\{u, v\}$ can be viewed as such a subset $T$ since at least one of $u$ and $v$ is in any vertex cover, and if we pick $u$, then we can remove it and all its incident edges from the graph to get a reduced instance.

For such problems, there is a simple fixed-parameter algorithm called the branching tree algorithm: The algorithm can be represented as a tree $\mathcal{T}$ rooted at a node $r$. Each node $v$ of the tree corresponds to a reduced instance of the original one, and in this instance, $v$ has a subset $T$ of size $f(k)$, and a child $v_i$ for every $i \in T$, where $v_i$ corresponds to selecting $i$ to be placed in the solution, and $v_i$ carries the reduced instance where $i$ is selected. The height of the tree $X$ is bounded by $k$ since at most $k$ elements need to be selected, and the branching factor is $f(k)$. Each leaf $\ell$ of the tree $\mathcal{T}$ is either a "yes"-leaf (when the predicate is satisfied on the set of elements selected on the path from $r$ to $\ell$) or a "no"-leaf (when the predicate is not satisfied). The runtime of the algorithm bounded by $f(k)^k \cdot t(N)$, where $t(N)$ is the time to find a subset $T$ that must contain an element of the solution in instances of size $N$, together with the time to find a reduced instance, once an element is selected.

What we have described so far is a static algorithmic technique, but we investigate when this algorithmic technique can be made dynamic. In other words, given an update, we would like to quickly update $\mathcal{T}$ so that it becomes a valid branching tree for the updated instance. Since the number of nodes in the branching tree is only a function of $k$, one can afford to look at every tree node. Ideally, one would like the time spent per node to only depend on $k$. However, for most problems that we consider, the branching tree needs to be rebuilt every so often, since the subset $T$ to branch on may become invalid after an update, and the time to rebuild can have a dependence on the instance size. We use two methods to avoid this. The first is to randomize the decisions made in the branching tree (e.g., which set $T$ to pick) so that, assuming an oblivious adversary that must provide the update sequence in advance, it is relatively unlikely that we need to rebuild the tree $\mathcal{T}$ (or its subtrees) after each update, and in particular, so that the expected cost of an update is only a function of $k$. The second is to make 'robust' choices of $T$, so that many updates are requires before the choice of $T$ becomes incalid, and then amortize the cost of rebuilding the tree over all the updates required to force such a rebuilding.

## 2.2 Algorithm Examples

We give overviews of the techniques used in some of our algorithms, to demonstrate the dynamic kernel and dynamic branching tree approaches, and different ways in which they can be used. We emphasize that these descriptions are substantial simplifications which hide many non-trivial details and ideas.

**Vertex Cover.** We give both a dynamic kernel algorithm and a dynamic branching tree algorithm for Vertex Cover.

Our first algorithm maintains a kernel obtained as follows: Every node of degree $\geq k+1$ 'selects' $k+1$ incident edges arbitrarily and adds them to the edge set $E'$ of the kernel, independently of other nodes. Next, every edge incident to two nodes of degree $\leq k$ is also added to $E'$. Finally, the node set of the kernel consists of all nodes that are not isolated in $E'$. This is a valid kernel, since any vertex cover of size at most $k$ needs to include every vertex of degree strictly greater than $k$. Every edge in $E'$ either has both its end points of degree $\leq k$, or is selected by one of its end points of degree at least $k+1$. Any node of a

vertex cover of the kernel either has degree $\leq k$ or selects $k+1$ edges. Thus the kernel must have size $O(k^2)$ when a $k$-vertex cover exists. To insert an edge we simply add the edge to the kernel unless one of its incident vertices has degree greater than $k$. If one of the end points $x$ used to be of degree $\leq k$ and is now of degree $k+1$, we have $x$ select all its incident edges and add them to the kernel. To delete an edge, we simply remove it from the kernel. If it was incident to a vertex $v$ of degree higher than $k+2$, then we need to find another edge incident to $v$ which is in the graph but not selected by $v$ to put into the kernel. If one of the end points now has degree $k$, we need to go through the incident edges and remove them from the kernel if their other end point has high degree and did not select them. All these operations can be performed by storing appropriate pointers so that the updates run in $O(k)$ time. With a little bit more work one can make them run in $O(1)$ amortized time. To answer queries, we answer "no" in constant time if the kernel has more than $2k(k+1)$ edges, and otherwise we run the fastest static $k$-Vertex Cover fixed-parameter algorithm on the kernel of size $O(k^2)$. This results in a $O(1.2738^k)$ update time.

Our second algorithm maintains a branching tree of depth at most $k$, which corresponds to using following randomized branching strategy: pick a uniformly random edge, and branch on adding each of its endpoints into the vertex cover. For a static branching algorithm, there is no need to pick a uniformly random edge to branch on, since at least one endpoint of every single edge must be in the vertex cover. However, a deterministic branching strategy like this in a dynamic algorithm would be susceptible to an adversarial edge update sequence, in which the adversary frequently removes edges which have been chosen to branch on. By ensuring that each edge we branch on is a uniformly random edge, we make the probability that we need to recompute any subtree of the branching tree $\mathcal{T}$ low. We compute the expected update time to be only $O(k2^k)$. Queries can be answered in only $O(k)$ time by following a path in the branching tree to an accepting leaf, if one exists.

These algorithms demonstrate some subtleties of the two techniques. In the branching tree algorithm, we use a randomized branching rule to deal with adversarial updates. In some of our other branching tree algorithms, like for Feedback Vertex Set, we are able to find a deterministic branching rule to yield a deterministic algorithm instead. In the kernelization algorithm, we manage to find a kernel which can be updated quickly even when the answer becomes larger than $k$ and the kernel size becomes large. In other problems, it will be harder to do this, and we may need to restrict ourselves to the promise model where we are guaranteed that the kernel will not grow too big in order to have efficient update times. Dynamic kernelization techniques typically lead to faster update times and query times, like in this case, because we can apply the fastest known static algorithm for the problem to the kernel to answer queries. In a branching tree algorithm, we may be using a branching rule which does not lead to the fastest algorithm because it is easier to dynamically maintain.

Interestingly, we are able to generalize both of these algorithms to the $d$-Hitting Set problem. The $d$-Hitting Set branching tree algorithm is similar to that of Vertex Cover, but the $d$-Hitting Set dynamic kernelization algorithm is much more complicated, and involves a tricky recursive rule for determining which sets to put in the kernel. We include an overview of the static kernel construction in Sect. 3.

**Max Leaf Spanning Tree.** Our algorithm for Max Leaf Spanning Tree uses the dynamic kernel approach. The kernel we maintain is simply the given graph, where we contract vertices of degree two whose neighbors both also have degree two. We can maintain this kernel by storing paths of contracted vertices in lists corresponding to edges they have been contracted into. As long as this kernel has $\Omega(k^2)$ nodes, it must always have a spanning tree with at least $k$ leaves.

Unlike in other dynamic kernel algorithms, where we maintain that the kernel does not get too large, this kernel may grow to have $\Omega(n^2)$ edges. We can nonetheless find a tree with at least $k$ leaves in time independent of $n$, by breadth-first searching from an arbitrary node in the kernel until we have $\Omega(k^2)$ kernel vertices, and just finding a tree within those vertices.

This method finds a subtree $T_S$ with at least $k$ leaves, but we need to find a tree which spans the whole graph. In the static problem, this could be accomplished by a linear-time breadth-first search away from $T_S$, but in the dynamic problem, this is too slow. To overcome this, we also maintain a spanning tree $T$ of the entire graph, which does not necessarily have $k$ leaves, using a known dynamic tree data structure. When queried for a spanning tree, we find $T_S$, and then perform a 'merge' operation to combine $T$ and $T_S$ into a spanning tree with at least $k$ leaves. This merge operation makes only $O(k^4)$ changes to $T$, so we are able to maintain a desired spanning tree in time independent of $n$.

We are able to maintain a linear size answer in only logarithmic time per update because the output is not very 'sensitive' to updates: we can always output an answer very close to $T$, which itself only changes in one edge per update. In other problems where the output can be more sensitive to updates, like Edge Clique Cover, we need to maintain a small intermediate representation of the answer instead of the answer itself.

**Feedback Vertex Set in undirected graphs.** Our algorithm for Feedback Vertex Set combines the dynamic kernel approach with the dynamic branching tree approach. We will maintain a branching tree, where we branch off of which node to include in our feedback vertex set. Then, at each node in the branching tree, we will maintain a kernel to help decide what nodes to branch on. Similar to the situation with Max Leaf Spanning Tree, our kernel can possibly have $\Omega(n^2)$ edges. Here we will deal with this by branching off of only $O(k)$ nodes in the kernel to add to our feedback vertex set, so that we can answer queries in sublinear time in the kernel size.

The kernel we maintain at each node of the branching tree is the given graph, in which vertices of degree one are deleted, and vertices of degree two are contracted. This involves many details for maintaining contracted trees, and dealing with resulting self-loops. Since the resulting graph has average degree at least three, whereas forests have much lower average degree, we show that a feedback vertex set of size at most $k$ must contain a vertex of high degree, whose degree is at least $1/(3k)$ of the total number of edges in the kernel. Since there are at most $6k$ such vertices, we can branch on which to include in our feedback vertex set.

This branching strategy works well for the static problem, but it is hard to maintain dynamically. Each edge update might change the set of vertices with high enough degree to branch on, and changing which vertex we branch on, and recomputing an entire subtree of the branching tree, can be expensive. We alleviate this issue using amortization. Instead of branching only on the $6k$ highest degree vertices, we instead branch on the $12k$ highest degree vertices. If our kernel has $m$ edges, then we prove that $\Omega(m)$ edge updates need to happen before there might be a small feedback vertex set containing none of the vertices we branched on. After these updates we need to recompute the branching tree, but this is inexpensive when amortized over the required $\Omega(m)$ updates.

## 3 A dynamic kernel for $d$-Hitting Set

In this section we present our dynamic kernel for the $d$-Hitting Set problem; we describe how to efficiently compute and maintain it in the full version of the paper. The $d$-Hitting Set problem asks to find a set $X \subseteq U$ of at most $k$ elements of a given a universe $U$ which

intersects all sets from a family $\mathcal{F}$ of subsets of $U$, each of cardinality exactly $d$. Here we present a kernel for $d$-Hitting Set, with $(d-1)!k(k+1)^{d-1}$ sets and $d!k(k+1)^{d-1}$ elements. It is known [25] that a kernel of size $O(k^{d-\varepsilon})$ for any constant $\varepsilon > 0$ would imply that $\mathsf{NP} \subseteq \mathsf{coNP}/poly$; thus, the $k^d$ dependence on the number of sets in the kernel is optimal.

Let us describe the kernel. We will recursively define the notion of a *good* set.

▶ **Definition 4** (good set)**.** Let $r \in \mathbb{N}$ and $\nu_r = r!(k+1)^r$. Let $d \in \mathbb{N}$ and let $(U, \mathcal{F})$ be an instance of $d$-Hitting Set. We define the notion of an "$(\ell, r)$-good" set inductively, in decreasing order of $\ell$ from $d$ to 1, and for fixed $\ell$, for increasing $r$ from 1 to $d - \ell$.

- Any set $S \in \mathcal{F}$ is $(d, r)$-*good* for all $r$.
- A set $S \subseteq U$ is $\ell$-*good* if $S$ is $(\ell, r)$-good for some $r$.
- A set $S \subseteq U$ is $(\ell', r)$-*strong* if $S$ is $\ell'$-good and does not contain any $(\ell' - j, j)$-good subsets for any $j \in \{1, \ldots, r-1\}$.
- A set $S \subseteq U$ is $(\ell, r)$-*good* if $|S| = \ell$ and $S$ is a subset of at least $\nu_r$ $(\ell + r, r)$-strong sets.
- A set $S \subseteq U$ is *good* if $S$ is $\ell$-good for $\ell = |S|$.

Notice that if a set is $(\ell', r)$-strong, then it is also $(\ell', r')$-strong for all $r' < r$. Also, any $\ell$-good set is $(\ell, 1)$-strong. Further, note that since the notion of $(\ell + r, r)$-strong only depends on $(\ell + a, r - a)$-good sets for $a \geq 1$, the definition of $(\ell, r)$-good is sound.

Let $\mathcal{F}'$ consist of those $S \subseteq U$ that are good and none of their subsets are good. Let $U'$ consist of all $u \in U$ that are contained in some set of $\mathcal{F}'$. Let $K = (U', \mathcal{F}')$. In the full version we prove the following lemma that shows that $(K, k)$ is a kernel for the instance $(U, \mathcal{F})$.

▶ **Lemma 5.** *Let $(U, \mathcal{F})$ be an instance of $d$-Hitting Set. If $(U, \mathcal{F})$ admits a hitting set $X$ of size at most $k$, then any good set $S \subseteq U$ intersects $X$ non-trivially.*

The lemma implies that $(K, k)$ is a kernel: first, if $X'$ is a hitting set of $K$, it is a hitting set for $\mathcal{F}$ as well since for every $F \in \mathcal{F}$, either $F \in \mathcal{F}'$ or some subset of $F$ is in $\mathcal{F}'$. Now let $X$ be a hitting set of $\mathcal{F}$ with size at most $k$. By the lemma, if some $S$ is in $\mathcal{F}'$, then it intersects $X$ non-trivially and so $X$ is a hitting set of $\mathcal{F}'$ as well. Now we argue about the size of $K$.

▶ **Lemma 6.** *If $(U, \mathcal{F})$ (and hence also $(U', \mathcal{F}')$) admits a hitting set of size at most $k$, then $|U'| \leq d|\mathcal{F}'|$ and $|\mathcal{F}'| \leq \left(1 + \frac{2}{(k+1)(d-1)}\right) \cdot d!(k+1)^d$.*

**Proof.** If $\{u\} \in \mathcal{F}'$, then no other set containing $u$ can be in $\mathcal{F}'$. Otherwise, consider some $u$ such that $\{u\} \notin \mathcal{F}'$. Consider all sets of size $r+1$ in $\mathcal{F}'$ that contain $u$, for any choice of $r \in \{1, \ldots, d-1\}$.

Since $\{u\} \notin \mathcal{F}'$, we know that $u$ cannot be $(1, r)$-good, and thus $u$ is contained in fewer than $\nu_r$ $(r+1)$-good sets that do not contain any $(j+1, r-j)$-good subsets for any $j \in \{2, \ldots, r-1\}$. Now since for every $F \in \mathcal{F}'$ we have that it contains no good subsets, this means that $u$ is contained in fewer than $\nu_r$ sets in $\mathcal{F}'$ of size $r+1$.

Thus, the number of sets of $\mathcal{F}'$ containing $u$ is at most

$$\sum_{r=1}^{d-1} \nu_r = \sum_{r=1}^{d-1} r!(k+1)^r \leq \left(1 + \frac{2}{(k+1)(d-1)}\right)(d-1)!(k+1)^{d-1},$$

where the last inequality can be proven inductively. Thus, if there is a hitting set of size at most $k$ for $\mathcal{F}'$, then the size of $\mathcal{F}'$ is at most $(1 + \frac{2}{(k+1)(d-1)})d!(k+1)^d$. ◀

In the full version we additionally show that the kernel can be computed in $O(3^d n + m)$ time and can be dynamically maintained with very fast updates.

**Acknowledgments.** The authors would like to thank Nicole Wein, Daniel Stubbs, Hubert Teo, and Ryan Williams for fruitful conversations.

## References


1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Proc. FOCS 2014*, pages 434–443, 2014.

2 Faisal N. Abu-Khzam, Judith Egan, Michael R. Fellows, Frances A. Rosamond, and Peter Shaw. On the parameterized complexity of dynamic problems with connectivity constraints. In *Proc. COCOA 2014*, volume 8881 of *Lecture Notes Comput. Sci.*, pages 625–636, 2014.

3 Faisal N. Abu-Khzam, Judith Egan, Michael R. Fellows, Frances A. Rosamond, and Peter Shaw. On the parameterized complexity of dynamic problems. *Theor. Comput. Sci.*, 607:426–434, 2015.

4 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4), 1995.

5 Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM J. Comput.*, 44(1):88–113, 2015.

6 Ann Becker, Reuven Bar-Yehuda, and Dan Geiger. Randomized algorithms for the loop cutset problem. *J. Artif. Intelligence Res.*, 12:219–234, 2000.

7 Aaron Bernstein and Liam Roditty. Improved dynamic algorithms for maintaining approximate shortest paths under deletions. In *Proc. SODA 2011*, pages 1355–1365, 2011.

8 Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. In *Proc. SODA 2015*, pages 785–804, 2015.

9 Hans L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *Proc. WG 1993*, volume 790 of *Lecture Notes Comput. Sci.*, pages 112–124, 1993.

10 Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On problems without polynomial kernels. *J. Comput. Syst. Sci.*, 75(8):423–434, 2009.

11 Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $c^k n$ 5-approximation algorithm for treewidth. *SIAM J. Comput.*, 45(2):317–378, 2016.

12 S. Buss. private communication.

13 Leizhen Cai, Siu Man Chan, and Siu On Chan. Random separation: A new method for solving fixed-cardinality optimization problems. In *Proc. IPEC 2006*, volume 4169 of *Lecture Notes Comput. Sci.*, pages 239–250, 2006.

14 Liming Cai, Jianer Chen, Rodney G. Downey, and Michael R. Fellows. Advice classes of parameterized tractability. *Ann. Pure Applied Logic*, 84(1):119–138, 1997.

15 Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoret. Comput. Sci.*, 411(40):3736–3756, 2010.

16 Jianer Chen, Yang Liu, Songjian Lu, Barry O'Sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *J. ACM*, 55(5), 2008.

17 Rajesh Hemant Chitnis, Graham Cormode, Mohammad Taghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In *Proc. SODA 2015*, pages 1234–1251, 2015.

18 Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

19 Marek Cygan. Deterministic parameterized connected vertex cover. In *Proc. SWAT 2012*, volume 7357 of *Lecture Notes Comput. Sci.*, pages 95–106, 2012.

20 Marek Cygan, Fedor Fomin, Bart M.P. Jansen, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. Open problems for FPT school 2014. `http://fptschool.mimuw.edu.pl/opl.pdf`.

21 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, Cham, 2015.

22 Marek Cygan, Stefan Kratsch, Marcin Pilipczuk, Michał Pilipczuk, and Magnus Wahlström. Clique cover and graph separation: New incompressibility results. *ACM Trans. Comput. Theory*, 6(2):6:1–6:19, 2014.

23 Marek Cygan, Marcin Pilipczuk, and Michał Pilipczuk. Known algorithms for edge clique cover are probably optimal. *SIAM J. Comput.*, 45(1):67–83, 2016.

24 Jean Daligault, Gregory Gutin, Eun Jung Kim, and Anders Yeo. FPT algorithms and kernels for the directed $k$-leaf problem. *J. Comput. Syst. Sci.*, 76(2):144–152, 2010.

25 Holger Dell and Dieter van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. *J. ACM*, 61(4):23:1–23:27, 2014.

26 Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.

27 Michael Dom, Daniel Lokshtanov, and Saket Saurabh. Incompressibility through colors and IDs. In *Proc. ICALP 2009*, volume 5555 of *Lecture Notes Comput. Sci.*, pages 378–389, 2009.

28 Frederic Dorn. Planar subgraph isomorphism revisited. In *Proc. STACS 2010*, volume 5 of *Leibniz Int. Proc. Informatics*, pages 263–274, 2010.

29 Zdeněk Dvořák, Martin Kupec, and Vojtěch Tůma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *Proc. ESA 2014*, volume 8737 of *Lecture Notes Comput. Sci.*, pages 334–345, 2014.

30 Zdeněk Dvořák and Vojtěch Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *Proc. WADS 2013*, volume 8037 of *Lecture Notes Comput. Sci.*, pages 304–315, 2013.

31 Vladimir Estivill-Castro, Michael R. Fellows, Michael A. Langston, and Frances A. Rosamond. FPT is P-time extremal structure I. In *Proc. ACiD 2005*, pages 1–41, 2005.

32 Michael Etscheid and Matthias Mnich. Linear kernels and linear time algorithms for finding large cuts. In *Proc. ISAAC 2016*, volume 64 of *Leibniz Int. Proc. Informatics*, pages 31:1–31:13, 2016.

33 Stefan Fafianie and Stefan Kratsch. A shortcut to (sun)flowers: Kernels in logarithmic space or linear time. In *Proc. MFCS 2015*, volume 9235 of *Lecture Notes Comput. Sci.*, pages 299–310, 2015.

34 Henning Fernau. Edge dominating set: Efficient enumeration-based exact algorithms. In *Proc. IPEC 2006*, volume 4169 of *Lecture Notes Comput. Sci.*, pages 142–153, 2006.

35 Anka Gajentaan and Mark H. Overmars. On a class of problems in computational geometry. *Comput. Geom.*, 45(4):140–152, 2012.

36 François Le Gall. Powers of tensors and fast matrix multiplication. In *Proc. ISSAC 2014*, pages 296–303, 2014.

37 David Gibb, Bruce Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space, 2015. URL: `https://arxiv.org/abs/1509.06464`.

38 Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Data reduction and exact algorithms for clique cover. *J. Exp. Algorithmics*, 13:2:2.2–2:2.15, 2009.

39 Manoj Gupta and Richard Peng. Fully dynamic $(1+\varepsilon)$-approximate matchings. In *Proc. FOCS 2013*, pages 548–557, 2013.

40 Andras Gyárfás. A simple lower bound on edge coverings by cliques. *Discrete Math.*, 85(1):103–104, 1990.

**41** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Improved algorithms for decremental single-source reachability on directed graphs. In *Proc. ICALP 2015*, volume 9134 of *Lecture Notes Comput. Sci.*, pages 725–736, 2015.

**42** Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Dynamic approximate all-pairs shortest paths: Breaking the $O(mn)$ barrier and derandomization. *SIAM J. Comput.*, 45(3):947–1006, 2016.

**43** Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proc. FOCS 2015*, pages 21–30, 2015.

**44** Monika Rauch Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

**45** Monika Rauch Henzinger and Valerie King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Comput.*, 31(2):364–374, 2001.

**46** Monika Rauch Henzinger and Mikkel Thorup. Sampling to provide or to bound: With applications to fully dynamic graph algorithms. *Random Struct. Algorithms*, 11(4):369–379, 1997.

**47** Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

**48** Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in $O(\log n(\log\log n)^2)$ amortized expected time, 2016. URL: `http://arxiv.org/abs/1609.05867`.

**49** Ken Iwaide and Hiroshi Nagamochi. An improved algorithm for parameterized edge dominating set problem. *J. Graph Algorithms Appl.*, 20(1):23–58, 2016.

**50** Yoichi Iwata. A linear time kernelization for feedback vertex set, 2016. URL: `https://arxiv.org/abs/1608.01463`.

**51** Yoichi Iwata and Keigo Oka. Fast dynamic graph algorithms for parameterized problems. In *Proc. SWAT 2014*, volume 8503 of *Lecture Notes Comput. Sci.*, pages 241–252, 2014.

**52** Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. Linear-time FPT algorithms via network flow. In *Proc. SODA 2014*, pages 1749–1761, 2014.

**53** Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proc. SODA 2013*, pages 1131–1142, 2013.

**54** Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3SUM conjecture. In *Proc. SODA 2016*, pages 1272–1287, 2016.

**55** 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.

**56** Stefan Kratsch and Magnus Wahlström. Representative sets and irrelevant vertices: New tools for kernelization. In *Proc. FOCS 2012*, pages 450–459, 2012.

**57** Daniel Lokshtanov, N. S. Narayanaswamy, Venkatesh Raman, M. S. Ramanujan, and Saket Saurabh. Faster parameterized algorithms using linear programming. *ACM Trans. Algorithms*, 11(2):15:1–15:31, 2014.

**58** Daniel Lokshtanov, M. S. Ramanujan, and Saket Saurabh. Linear time parameterized algorithms for subset feedback vertex set. In *Proc. ICALP 2015*, volume 9134 of *Lecture Notes Comput. Sci.*, pages 935–946, 2015.

**59** Daniel Lokshtanov, M. S. Ramanujan, and Saket Saurabh. A linear time parameterized algorithm for directed feedback vertex set, 2016. URL: `https://arxiv.org/abs/1609.04347`.

**60** Aleksander Madry. Navigating central path with electrical flows: From flows to matchings, and back. In *Proc.FOCS 2013*, pages 253–262, 2013.

**61** Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *Proc. STOC 2010*, pages 457–464, 2010.

**62** Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *Proc. STOC 2010*, pages 603–610, 2010.

**63** Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011.

**64** Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006.

**65** Mihai Pătraşcu and Mikkel Thorup. Don't rush into a union: take time to find your roots. In *Proc. STOC 2011*, pages 559–568, 2011.

**66** Shay Solomon. Fully dynamic maximal matching in constant update time. In *Proc. FOCS 2016*, pages 325–334, 2016.

**67** Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proc. STOC 2000*, pages 343–350, 2000.

**68** René van Bevern. Towards optimal and expressive kernelization for $d$-hitting set. *Algorithmica*, 70(1):129–147, 2014.

**69** Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proc. STOC 2012*, pages 887–898, 2012.

**70** Magnus Wahlström. Half-integrality, LP-branching and FPT algorithms. In *Proc. SODA 2014*, pages 1762–1781, 2014.

**71** Christian Wulff-Nilsen. Faster deterministic fully-dynamic graph connectivity. In *Proc. SODA 2013*, pages 1757–1769, 2013.