

Dynamic Kernels for Hitting Sets and Set Packing

Max Bannach  

Universität zu Lübeck, Germany

Zacharias Heinrich  

Universität zu Lübeck, Germany

Rüdiger Reischuk 

Universität zu Lübeck, Germany

Till Tantau 

Universität zu Lübeck, Germany

Abstract

Computing small kernels for the hitting set problem is a well-studied computational problem where we are given a hypergraph with n vertices and m hyperedges, each of size d for some small constant d , and a parameter k . The task is to compute a new hypergraph, called a *kernel*, whose size is polynomial with respect to the parameter k and which has a size- k hitting set if, and only if, the original hypergraph has one. State-of-the-art algorithms compute kernels of size k^d (which is a polynomial kernel size as d is a constant), and they do so in time $m \cdot 2^d \text{poly}(d)$ for a small polynomial $\text{poly}(d)$ (which is a linear runtime as d is again a constant).

We generalize this task to the *dynamic* setting where hyperedges may continuously be added or deleted and one constantly has to keep track of a size- k^d hitting set kernel in memory (including moments when no size- k hitting set exists). This paper presents a *deterministic* solution with *worst-case* time $3^d \text{poly}(d)$ for updating the kernel upon hyperedge inserts and time $5^d \text{poly}(d)$ for updates upon deletions. These bounds nearly match the time $2^d \text{poly}(d)$ needed by the best static algorithm per hyperedge. Let us stress that for constant d our algorithm maintains a dynamic hitting set kernel with *constant, deterministic, worst-case* update time that is independent of n , m , and the parameter k . As a consequence, we also get a deterministic dynamic algorithm for keeping track of size- k hitting sets in d -hypergraphs with update times $O(1)$ and query times $O(c^k)$ where $c = d - 1 + O(1/d)$ equals the best base known for the static setting.

2012 ACM Subject Classification Theory of computation \rightarrow Design and analysis of algorithms; Theory of computation \rightarrow Fixed parameter tractability

Keywords and phrases Kernelization, Dynamic Algorithms, Hitting Set, Set Packings

Digital Object Identifier 10.4230/LIPIcs.IPEC.2021.7

Related Version *Technical Report*: <https://eccc.weizmann.ac.il/report/2019/146/> [3]

1 Introduction

The hitting set problem is a fundamental combinatorial problem that asks, given a hypergraph, whether there is a small vertex subset that intersects (“hits”) each hyperedge. Many interesting problems reduce to it: a dominating set of a graph is just a hitting set in the hypergraph that contains for every vertex a hyperedge consisting of the vertex’s closed neighborhood; for any fixed graph H , the question of whether we can delete k vertices from a graph G in order to make G an H -free graph can be reduced to the hitting set problem for the hypergraph to which each occurrence of H in G contributes one hyperedge – and this problem in turn generalizes problems such as TRIANGLE-DELETION and CLUSTER-VERTEX-DELETION [1]. The hitting set problem also finds applications in the area of descriptive complexity, as a fragment of first-order logic can be reduced to it [9].



© Max Bannach, Zacharias Heinrich, Rüdiger Reischuk, and Till Tantau;
licensed under Creative Commons License CC-BY 4.0

16th International Symposium on Parameterized and Exact Computation (IPEC 2021).

Editors: Petr A. Golovach and Meirav Zehavi; Article No. 7; pp. 7:1–7:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The hitting set problem is NP-complete [25] and its parameterized version p_k -HITTING-SET is $W[2]$ -complete [14]. However, if we restrict the size of hyperedges to at most some constant d , the resulting problem p_k - d -HITTING-SET lies in FPT [20] and even has polynomial kernels. In particular, $d = 2$ is the vertex cover problem, which is still NP-complete, but one of the best-investigated parameterized problems. Already the jump from $d = 2$ to $d = 3$ turns out to be nontrivial in this setting. In detail, the inputs for our algorithms are a hypergraph $H = (V, E)$ and an upper bound k for the size of a hitting set X wanted (a set for which $e \cap X \neq \emptyset$ holds for all $e \in E$). We think of the numbers $n = |V|$ and $m = |E|$ as large numbers, of k as a (relatively small) parameter, and of $d = \max_{e \in E} |e|$ as a small constant (already the cases $d = 3$ and $d = 4$ are of high interest).

Parameterized algorithms for the hitting set problem proceed in two steps: First, the input (H, k) is *kernelized*, which means that we quickly compute a (small) new hypergraph K such that H has a size- k hitting set iff K has one. Afterwards the problem is solved on K using an expensive algorithm based on search trees or iterative compression. The currently best algorithm for computing a kernel is due to Fafianie and Kratsch [17], see also [4, 32] for some recent developments. The cited algorithm outputs a kernel of size k^d (meaning that K has at most k^d hyperedges) in time $m \cdot 2^d \text{poly}(d)$ (meaning that time $2^d \text{poly}(d)$ is needed on average per hyperedge of H). The best algorithms for solving the hitting set problem on the computed kernel K run in time $O(c^k)$, where the exact value of $c = d - 1 + O(1/d)$ is a subject of ongoing research [33, Section 6], [19, 21, 28], and [10, Section 4]. In summary, on input (H, k) one can solve the hitting set problem in time $O(2^d \text{poly}(d) \cdot m + c^k)$.

Our objective in this paper is to transfer (only) the first part of solving the hitting set problem (namely the computation of the kernel K) into the dynamic setting. Instead of a single hypergraph H being given at the beginning, there is a sequence $H_0, H_1, H_2, H_3, \dots$ of hypergraphs each of which differs from the previous one by a single edge being either added or deleted. One continuously has to keep track of hitting set kernels $K_0, K_1, K_2, K_3, \dots$ for the current H_i (including moments when H_i has no size- k hitting set). Our aim is to compute the updated kernel K_{i+1} from K_i in constant time based solely on the knowledge which edge was added to or deleted from H_i in order to obtain H_{i+1} .

Doing the necessary bookkeeping to dynamically manage a hitting set kernel is not easy. As an example, consider two hypergraphs H and \tilde{H} with disjoint vertex sets, where H is a clear no-instance (like a matching of size $k + 1$) while \tilde{H} is a hard, borderline case that can only be reduced to a relatively large kernel \tilde{K} . A dynamic kernel algorithm that works on $H \cup \tilde{H}$ must be able to cope with the situation that we first add all the edges of H (at which point a natural kernel would be a trivial size-1 no-instance K), followed by all the edges of \tilde{H} (which even reinforces that the trivial no-instance K is a correct kernel for the ever-larger hypergraph), followed by a deletion of the edges from H . At some point during these deletions, a dynamic kernel algorithm must switch from the constant-size K to the large kernel \tilde{K} . Previous work from the literature [2] shows that it is already tricky to achieve this switch in time polynomial in the size of kernels K and \tilde{K} . The challenge we address is to do the updates in *constant worst-case* time, which forces our dynamic algorithm to spread the necessary changes over time while neither resorting to amortization nor to randomization.

Note that we only give a dynamic algorithm for keeping the *kernel* up-to-date with constant update times – we make no claims concerning the time needed to actually compute a hitting set for the current kernel K_i (and, thus, for the current H_i). Phrased in terms of dynamic complexity theory, there are two different problems for which we present algorithms with differing *update times* (the time needed for updating internal data structures) and *query times* (the time needed to construct an output upon request): For the first problem of (just)

computing hitting set kernels K for inputs H , we present a dynamic algorithm with *constant* update time and *zero* query time (since the current kernel K_i is explicitly stored in memory as an adjacency matrix at all times). For the second problem of computing size- k hitting sets X for inputs H , our dynamic algorithm also has constant update time (to keep track of kernels K_i), but has a query time of c^k (to compute X_i from K_i). Since in both cases our update times are constant and since it is not hard to see that one cannot improve the query times beyond the time needed by the fastest static algorithm, these bounds are optimal.

Main Result: A Fully Dynamic Hitting Set Kernel. In the fully dynamic case where edges may be inserted and deleted over time, the hypergraph may repeatedly switch between having and not having a size- k hitting set. This turns out to be a big obstacle for updating a kernel in just a few steps. Dynamic kernels have already been constructed by Alman, Mnich, and Williams [2]. They present a p_k -VERTEX-COVER kernel with $O(k)$ worst-case update time and $O(1)$ amortized update time. For the p_k - d -HITTING-SET they achieve a kernel of size $(d-1)!k(k+1)^{d-1}$ with update time $(d!)^d \cdot k^{O(d^2)}$.

In this paper, for each fixed number d we present a fully dynamic algorithm that maintains a p_k - d -HITTING-SET kernel of size $O(k^d)$ with constant update times.

► **Theorem 1.** *For every $d \geq 2$ there is a deterministic, fully dynamic kernel algorithm for the problem p_k - d -HITTING-SET that maintains at most $\sum_{i=0}^d k^i \leq (k+1)^d$ hyperedges in the kernel, has worst-case insertion time $3^d \text{poly}(d)$, and worst-case deletion time $5^d \text{poly}(d)$. In particular, as d is a constant, the dynamic kernel algorithm performs both insertions and deletions in time that is constant and independent of the input and parameter k .*

► **Corollary 2.** *There is a fully dynamic algorithm for p_k - d -HITTING-SET with update time $O(1)$ and query time $O(c^k)$, where $c = d - 1 + O(1/d)$.*

In order to achieve update times independent of k , this paper makes three major improvements on the general sunflower approach [1]. First, relevant objects are handled hierarchically. This allows an inductive construction and an analysis that improves the bounds on the kernel size as well as the update time. Second, we replace the notion of *strong edges* (see [2]) by *needed edges* to be defined later. Whenever a flower is formed, the replacement of its petals can be handled much more easily this way. Finally, the use of b -flowers (see also [17]) instead of generalized sunflowers [2] decreases the size of the kernel even further.

Our kernel is a *full kernel* in the sense of [11]: It does not just preserve a single size- k solution, but all of them. Therefore, we can use the kernel for counting and enumeration problems; and we can even use the whole kernel as approximate solution. The kernel size is optimal insofar as p_k - d -HITTING-SET has no kernel of size $O(k^{d-\epsilon})$ unless $\text{coNP} \subseteq \text{NP/poly}$ [13]. Note that if we feed the hyperedges of a *static* hypergraph to our algorithm one-at-a-time, we compute a *static* hitting set kernel in time $3^d \text{poly}(d) \cdot m$. Since the currently best algorithms for that tasks, see [4, 17, 30], run in time $2^d \text{poly}(d) \cdot m$, our algorithm is not far from the best static runtime: the difference just lies in the constant factor 3^d versus 2^d .

Extension to Set Packing. Our kernelization can be adapted for p_k -MATCHING and the more general p_k - d -SET-PACKING: The input (H, k) is as before, but the question is whether there is a *packing* $P \subseteq E$ with $|P| \geq k$ (that is, $e \cap f = \emptyset$ for any two different $e, f \in P$).

► **Theorem 3.** *For every $d \geq 2$ there is a deterministic, fully dynamic kernel algorithm for the problem p_k - d -SET-PACKING that maintains at most $\sum_{i=0}^d (d(k-1))^i \leq d^d k^d$ hyperedges in the kernel, has worst-case insertion time $3^d \text{poly}(d)$, and worst-case deletion time $5^d \text{poly}(d)$.*

Related Work. Ever-better kernels for p_k - d -HITTING-SET are due to Flum and Grohe [20], van Bevern [30], and Fafianie and Kratsch [17]. Damaschke studied *full* kernels for the problem, which are kernels that contain all small solutions [11]. There are also optimized algorithms for specific values of d : for instance the algorithm by Buss and Goldsmith [7] for $d = 2$, or by Niedermeier and Rossmanith [28] and Abu-Khzam [1] for $d = 3$.

Dynamic algorithms can be used in a variety of monitoring applications such as maintaining a minimum spanning tree [22] or connected components [23]. There is also a recent trend in studying dynamic approximation algorithms, for instance for VERTEX-COVER [6]. Algorithms that maintain a solution for a dynamically changing input can also be studied using descriptive complexity, as suggested by Patnaik and Immerman [29]. A recent break-through result in this area is that reachability is contained in DynFO [12].

Iwata and Oka [24] were the first to combine kernelization and dynamic algorithms by studying a dynamic quadratic kernel for p_k -VERTEX-COVER. Their dynamic kernel algorithm requires $O(k^2)$ update time and works in a promise model where at all times it is guaranteed that there actually *is* a size- k vertex cover in the input graph. Alman, Mnich, and Williams extended this line of research by studying dynamic parameterized algorithms for a broad range of problems [2]. Among others, they provided a p_k -VERTEX-COVER kernel with $O(k)$ worst-case update time and $O(1)$ amortized update time that works in the fully dynamic model. Their generalization to a fully dynamic algorithm for p_k - d -HITTING-SET with a slightly larger kernel size and nonconstant update time has already been mentioned above. Recent advances in dynamic FPT-algorithms were achieved by a dynamic data structure that maintains a optimum-height elimination forest for a graph of bounded treedepth [8].

Organization of This Paper. After a short introduction to dynamic algorithms, data structures, and parameterized complexity in Section 2, we first illustrate the algorithm for the special case of p_k -VERTEX-COVER in Section 3. Then, in Section 4, we generalize the algorithm to p_k - d -HITTING-SET. In Section 5 we argue that with slight modifications, the same algorithm can be used to maintain a polynomial kernel for p_k - d -SET-PACKING. In this publication we focus on explaining the core concepts – detailed proofs and technical details can be found in the technical report [3].

2 A Framework for Parametrized Dynamic Algorithms

Our aim is to *dynamically* maintain *kernels* with *minimal update time*. To formalize this, let us begin with the definition of *kernels* and then explain properties of *dynamic* kernels.

Parameterized Hypergraph Problems and Kernels. A d -hypergraph is a pair $H = (V, E)$ consisting of a set V of *vertices* and a set E of *hyperedges* with $e \subseteq V$ and $|e| \leq d$ for all $e \in E$. Let $n = |V|$ and $m = |E|$. The degree of v is $\deg_H(v) = |\{e \in E \mid v \in e\}|$. A *uniform d -hypergraph* has $|e| = d$ for all $e \in E$, e. g., a *graph* is a uniform 2-hypergraph. We use $\binom{V}{d}$ to denote the set $\{e \subseteq V \mid |e| = d\}$ of all size- d hyperedges and let $\binom{V}{\leq d} = \{e \subseteq V \mid |e| \leq d\}$. *Parameterized hypergraph problems* are sets $Q \subseteq \Sigma^* \times \mathbb{N}$, where *instances* $(H, k) \in \Sigma^* \times \mathbb{N}$ consist of a hypergraph H and a *parameter* k . A parameterized problem is in FPT if the question $(H, k) \in Q$ can be decided in time $f(k) \cdot (|V| + |E|)^c$ for some computable function f and constant c . It is known that $Q \in \text{FPT}$ holds iff *kernels* can be computed for Q in polynomial time [15]. Kernels of *polynomial size* are of special interest: For a polynomial σ , a σ -kernel for an instance $(H, k) \in \Sigma^* \times \mathbb{N}$ of a problem Q is another instance $(H', k') \in \Sigma^* \times \mathbb{N}$ with $|H'| \leq \sigma(k)$, $k' \leq \sigma(k)$, and $(H, k) \in Q \iff (H', k') \in Q$.

Dynamic Hypergraphs and Dynamic Kernels. One might consider several properties of a hypergraph that could change in a dynamic way. In this paper we consider as fixed and immutable the bound d on the hyperedge sizes, the vertex set V , and also the parameter k . That means only the most specific one, the hyperedge set E , will change dynamically. We assume that initially it is the empty set:

► **Definition 4** (Dynamic Hypergraphs). *A dynamic hypergraph consists of a fixed vertex set $V = \{v_1, \dots, v_n\}$ and a sequence o_1, o_2, o_3, \dots of update operations, where each o_j is either $\text{insert}(e_j)$ or $\text{delete}(e_j)$ for a hyperedge $e_j \subseteq V$.*

A dynamic hypergraph defines a sequence of hypergraphs H_0, H_1, \dots with $H_0 = (V, \emptyset)$, $H_j = (V, E(H_{j-1}) \cup \{e_j\})$ for $o_j = \text{insert}(e_j)$, and with $H_j = (V, E(H_{j-1}) \setminus \{e_j\})$ for $o_j = \text{delete}(e_j)$. For convenience (and without loss of generality) we assume only missing hyperedges are inserted and only existing ones deleted. A *dynamic* hypergraph algorithm gets the update sequence of a dynamic hypergraph as input and has to output a sequence of solutions, one for each H_i . Crucially, the solution for H_i must be generated before the next operation o_{i+1} is read. While after each update we could solve the problem from scratch for H_i , we may do better by taking into account that the difference between H_{i-1} and H_i is small. With the help of an internal *auxiliary data structure* A_i that the algorithm updates alongside the graphs, one might be able to solve the original problem faster after each update. *The problem we wish to solve dynamically* is to compute for each H_i a kernel K_i (as opposed to the problem of solving the parameterized problem Q itself).

► **Definition 5** (Dynamic Kernel Algorithm). *Let Q be a parameterized problem and σ a polynomial. A dynamic kernel algorithm ALGO for Q with kernel size $\sigma(k)$ has three methods:*

1. $\text{ALGO.init}(n, k)$ gets the size n of V and the parameter k as inputs, neither of which will change during a run of the algorithm, and must initialize an auxiliary data structure A_0 and a kernel K_0 for (H_0, k) and Q and σ (observe that $H_0 = (V, \emptyset)$ holds).
2. $\text{ALGO.insert}(e)$ gets a hyperedge e to be added to H_{i-1} and must update A_{i-1} and K_{i-1} to A_i and K_i with, again, K_i being a kernel for (H_i, k) and Q and σ .
3. $\text{ALGO.delete}(e)$ removes an edge instead of adding it.

One could also require that only the data structure A_i is updated in each step, while a kernel K_i would only be needed to be computed upon a query request. This would allow to differentiate between update times and query times for computing kernels. By requiring that the kernel K_i is explicitly computed at each step alongside A_i , our definition implies a query time of zero for computing K_i . However, solving the query $(H_i, k) \in Q$ using K_i may take exponential time in k . Concerning the update times, an efficient dynamic kernel algorithm should of course compute A_i and K_i faster than a static kernelization that processes H_i completely. The best one could hope for is constant time for the initialization and per update, even independent of the parameter k – and this is exactly what we achieve in this paper.

Data Structures for Dynamic Algorithms. The A_i rely on data structures such as objects and arrays. We additionally use a novel data structure called *relevance list*, which are ordinary lists equipped with a *relevance bound* $\rho \in \mathbb{N}$: the first ρ elements are said to be *relevant*, while the others are *irrelevant*. This data structure supports insertion and deletion, querying the relevance status of an element, and querying the last relevant element – each in $O(1)$ time. For concrete implementations and an analysis, please see the technical report [3].

3 Dynamic Vertex Cover with Constant Update Time

In order to better explain the ideas behind our dynamic kernel algorithm, we first tackle the case $d = 2$ in this section and show how we can maintain kernels of size $O(k^2)$ for the vertex cover problem with update time $O(1)$. The idea is based on a well-known *static* kernel: Buss [7] noticed that in order to cover all edges of a graph $G = (V, E)$ with k vertices, we *must* pick any vertex with more than k neighbors (let us call such vertices *heavy*). If there are more than k^2 edges after all heavy vertices have been picked and removed, no vertex cover of size k is possible (since each light vertex can cover at most k edges).

To turn this idea into a dynamic kernel, let us first consider only insertions. Initially, new edges can simply be added to the kernel; but at some point a vertex v “becomes heavy.” In the static setting one would remove v from the graph and decrease the parameter by 1. In the dynamic setting, however, removing v with its adjacent edges would take time $O(k)$ rather than $O(1)$. Instead, we leave v in the graph, but do *not* add further edges containing v to the kernel once v becomes heavy. We call the first $k + 1$ edges *relevant for the vertex* and the rest *irrelevant*. By putting the relevant edges of a heavy vertex in the kernel, we ensure that this vertex still must be chosen for any vertex cover. By leaving out the irrelevant edges, we ensure a kernel size of at most $O(k^2)$. More precisely, if the kernel size now threatens to exceed $k^2 + k + 1$, then any additional edges will be *irrelevant for the kernel* since the already inserted edges already form a proof that no size- k vertex cover exists.

Being relevant for a vertex is a “local” property: For an edge $e = \{u, v\}$, the vertex u may consider e to be relevant, while v may consider it to be irrelevant. An edge only “makes it to the kernel” when it is relevant for both endpoints – then it will be called *needed*. It is not obvious that this is how the case of a “disagreement” should be resolved and that this is the right notion of “needed edges” – but Lemma 8 shows that it leads to a correct kernel.

A Dynamic Vertex Cover Kernel Algorithm. We now turn the sketched ideas into a formal algorithm in the sense of Definition 5. The initialization sets up the auxiliary data structures: One relevance list L_v per vertex v to keep track of the edges that are relevant for v and one relevance list L to keep track of the edges that are relevant for the kernel. The code violates the requirement that the *initialization procedures* should run in constant time, but a known code transformation [27] for ensuring this will be discussed in the general hitting set case.

```

1  method DYNKERNELVC.init( $n, k$ ) //  $V = \{v_1, \dots, v_n\}$  holds by definition
2  for  $v \in V$  do
3     $L_v \leftarrow$  new RELEVANCE LIST( $k + 1$ ) // Keep track of relevant edges for a vertex
4   $L \leftarrow$  new RELEVANCE LIST( $k^2 + k + 1$ ) // Keep track of relevant edges for the kernel

```

The insert operation adds an edge e to the relevance lists of both endpoints of e . Furthermore, it also adds e to L if it is *needed*, which meant “relevant for both sides”.

```

5  method DYNKERNELVC.insert( $e$ )
6     $L_u.append(e)$ ;  $L_v.append(e)$ 
7    check if needed( $e$ )
8
9  function check if needed( $e$ ) // assume  $e = \{u, v\}$ 
10   if  $L_u.is\ relevant(e) \wedge L_v.is\ relevant(e)$  then
11      $L.append(e)$ 

```

The delete operation for an edge e is more complex: When $e = \{u, v\}$ is removed from the lists L_u , L_v , and L , formerly irrelevant edges may suddenly become relevant from the point of view of these three lists and, thus, possibly also needed. Fortunately, we know which

edge e' may suddenly have become relevant for a list: After the removal of e , the edge e' that is now the last relevant edge stored in the list is the (only) one that may have become relevant – and relevance lists keep track of the last relevant element.

```

12 method DYNKERNELVC.delete( $e$ ) // assume  $e = \{u, v\}$ 
13    $L.delete(e)$ 
14    $L_u.delete(e); L_v.delete(e)$ 
15   check if needed( $L_u.last\ relevant$ ); check if needed( $L_v.last\ relevant$ )

```

Correctness and Kernel Size. The relevant edges in L clearly have some properties that we would expect of a kernel: First, there are at most $k^2 + k + 1$ of them (for the simple reason that L caps the number of relevant edges in line 4) – which is exactly the size that a kernel should have. Second, it is also easy to see from the code of the algorithm that all operations run in time $O(1)$. Two lemmas make these observations precise, where $R(L)$ denotes the set of relevant edges in a list L and $E(L)$ denotes all edges in L ; and where we say that a dynamic algorithm *maintains an invariant* if that invariant holds for its auxiliary data structure right after the *init* method has been called and after every call to *insert* and *delete*.

► **Lemma 6.** DYNKERNELVC *maintains the invariant* $|R(L)| \leq k^2 + k + 1$.

► **Lemma 7.** DYNKERNELVC.*insert* and DYNKERNELVC.*delete* run in time $O(1)$.

The crucial, much less obvious property of the algorithm is stated in the next lemma, whose proof contains a non-trivial recursive analysis showing that irrelevant edges must already be covered by relevant edges inserted earlier.

► **Lemma 8.** DYNKERNELVC *maintains the invariant that* $(V, R(L))$ *and the current graph* (V, E) *have the same size- k vertex covers.*

Put together, we get the following special case of Theorem 1:

► **Theorem 9.** DynKernelVC *is a dynamic kernel algorithm for* p_k -VERTEX-COVER *with update time* $O(1)$ *and kernel size* $k^2 + k + 1$.

Proof. Lemmas 6, 7, and 8 together state that at all times during a run of the algorithm DYNKERNELVC the graph $(V, R(L))$ has at most $k^2 + k + 1$ edges and has the same size- k vertex covers as the current graph. Thus, $(V, R(L))$ is *almost* a kernel *except* that $R(L)$ is actually a linked list of edges (with potentially large vertex identifiers).

However, we can simultaneously keep track of an adjacency matrix of a graph K with the vertex set $V_K = \{1, \dots, 2(k^2 + k + 1)\}$ and with an edge set E_K that is always isomorphic to $R(L)$, that is, $E_K \sim R(L)$. In particular, K has a size- k hitting set if, and only if, G has one. See [3] for technical details.

The update times are constant. The time needed for DynKernelVC.*init*(n, k) can be made constant with a special new-initialized-with construct based on a technique in [27] as already mentioned and discussed in more detail below. ◀

4 Dynamic Hitting Set Kernels

The hitting set problem is a generalization of VERTEX-COVER to hypergraphs. However, allowing larger hyperedges introduces considerable complications into the algorithmic machinery. Nevertheless, we still seek and prove an update time that is constant. More precisely, it is independent of $n = |V|$, $m = |E|$, and the parameter k , while it does depend on d (in

fact even exponentially). Such an exponential dependency on d seems currently unavoidable, since a direct consequence of our dynamic algorithm is a static algorithm with running time $3^d \text{poly}(d) \cdot m$, and the currently best static algorithm runs in time $2^d \text{poly}(d) \cdot m$.

The first core idea of our algorithm concerns a replacement notion for the “heavy vertices” from the previous section. *Sunflowers* [16] are usually a stand-in (see [20, Section 9.1] and [32, 5]), but they are hard to find and especially hard to manage dynamically. Instead, we use an idea first proposed by Fafianie and Kratsch [17], but adapted to our dynamic setting: a generalizations of sunflowers, which we call *b-flowers* for different parameters $b \in \mathbb{N}$ that will be easier to keep track of dynamically.

The second core idea is to recursively reduce each case d to the case $d-1$: For a fixed $d > 2$, we compute a set of hyperedges relevant for the kernel (the set $R(L)$, but now called $R(L^d[\emptyset])$ in the more general case), but *additionally* we dynamically keep track of an instance for $p_k-(d-1)$ -HITTING-SET and merge the dynamic kernel for this instance (which we get from the recursion) with the list of hyperedges relevant for the kernel.

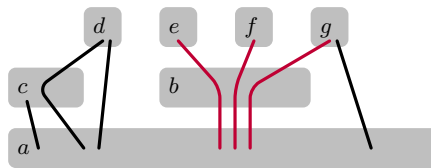
4.1 From High-Degree Vertices in Graphs to Flowers in Hypergraphs

A *sunflower* in a d -hypergraph $H = (V, E)$ is a collection of hyperedges $S \subseteq E$ such that there is a set $c \subseteq V$, called the *core*, with $x \cap y = c$ for all distinct pairs $x, y \in S$. For example, the edges adjacent to a heavy vertex v form a (large) sunflower with core $\{v\}$. In general, any size- k hitting set has to intersect with the core of a sunflower if it has more than k edges – which means that replacing large sunflowers by their cores is a reduction rule for p_k-d -HITTING-SET. This rule yields a kernel since the *Sunflower Lemma* [16] states that every d -hypergraph with more than $k^d \cdot d!$ hyperedges contains a sunflower of size $k + 1$.

Unfortunately, it is not easy to find sunflowers for larger d in the first place, let alone to keep track of them in a dynamic setting with constant update times. Rather than trying to find all sunflowers, we use a more general concept called *b-flowers*.

► **Definition 10.** For a hypergraph $H = (V, E)$ and $b \in \mathbb{N}$, a *b-flower* with core c is a set $F \subseteq E$ such that $c \subseteq e$ for all $e \in F$ and $\deg_{(V,F)}(v) \leq b$ for all $v \in V - c$.

Note that a 1-flower is exactly a sunflower and, thus, *b-flowers* are in fact a generalization of sunflowers, see Figure 1 for an example.



■ **Figure 1** A hypergraph $H = (\{a, b, c, d, e, f, g\}, E)$ in which each hyperedge $e \in E$ is drawn as a line and contains all vertices it “touches”. The three red edges form a 1-flower (a sunflower) with core $\{a, b\}$. The hyperedges $\{a, c\}, \{a, d\}, \{a, g\}, \{a, b, e\}$ also form a 1-flower, now with with core $\{a\}$, but if we add the hyperedges $\{a, b, f\}$ and $\{a, c, d\}$, we no longer have a 1-flower – but still a 2-flower with core $\{a\}$. All edges together form a 3-flower with core $\{a\}$.

► **Lemma 11.** Let F be a *b-flower* with core c in H and X a size- k hitting set of H . If $|F| > b \cdot k$, then $X \cap c \neq \emptyset$ (“ X must hit c ”).

Proof. If we had $X \cap c = \emptyset$, then each $v \in X$ could hit at most b hyperedges in F since $\deg_{(V,F)}(v) \leq b$. Then F can contain at most $b \cdot |X|$ hyperedges, contradicting $|F| > b \cdot k$. ◀

4.2 Dynamic Hitting Set Kernels: A Recursive Approach

As previously mentioned, the core idea behind our main algorithm is to recursively reduce the case d to $d - 1$. To better explain this idea, we illustrate how the (already covered) case $d = 2$ can be reduced to $d = 1$ and how this in turn can be reduced to $d = 0$. Following this, we present the complete recursive algorithm, prove its correctness, and analyze its runtime.

Recall that DYNKERNELVC adds up to $k + 1$ edges per vertex v into the kernel $R(L)$ to ensure that v “gets hit.” In the recursive hitting set scenario we ensure this differently: When we notice that v is “forced” into all hitting sets, we add a new hyperedge $\{v\}$ to an internal 1-hypergraph used exclusively to keep track of the forced vertices (clearly the only way to hit $\{v\}$ is to include v in the hitting set). When, later on after a deletion, we notice that a singleton hyperedge is no longer forced, we remove it from the internal 1-hypergraph once more. Since we have to ensure that not too many new hyperedges make it into the final kernel, we keep track of a *dynamic kernel of the internal 1-hypergraph* (using a dynamic hitting set algorithm for $d = 1$) and then *join* this kernel with $R(L)$.

Using a hypergraph to track the forced vertices allows us to change the relevance bounds of the algorithm: For the lists L_v these were $k + 1$, but since we explicitly “force” $\{v\}$ into the solution by generating a new hyperedge, it is enough to set the bound to k . Similarly, the bound for the original list L was set to $k^2 + k + 1$ since this constitutes a proof that no size- k vertex cover exists. In the new setting with the relevance bound for L_v lowered to k , we can also lower the relevance bound for L to k^2 : All vertices $v \in V$ have a degree of at most k in $R(L)$ and, thus, k vertices can hit at most k^2 hyperedges. If L contains more elements, we consider the (unhittable) empty hyperedge as *forced* and add it to the 1-hypergraph.

In order to dynamically keep track of a kernel for the internal 1-hypergraph, we proceed similarly: We simply put all its hyperedges (which have size 1 or 0) in a list (called $L^1[\emptyset]$ in the algorithm). If the number of hyperedges in this list exceeds k , we immediately know that no hitting set of size k exists; and we “recursively remember this” by inserting the empty set into yet another internal 0-hypergraph – this is the recursive call to $d = 0$.

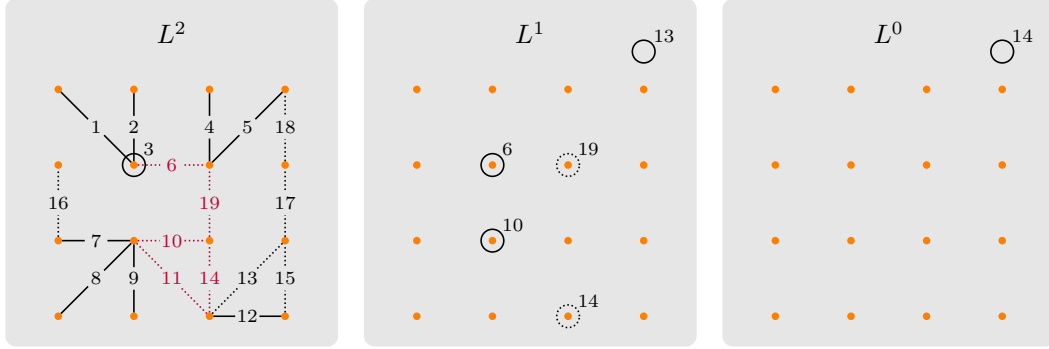
Managing Needed and Forced Hyperedges. In the general setting (now for arbitrary d), we need a uniform way to keep track of lists like the L_v and L for the many different internal hypergraphs. We do this using arrays L^i for $i \in \{0, \dots, d\}$ with domains $\binom{V}{\leq i}$, one for each i -hypergraph, where each $L^i[s]$ stores a relevance list. The list $L^i[s]$ has relevance bound $k^{i-|s|}$ and we only stores edges $e \in \binom{V}{\leq i}$ with $e \supseteq s$ in it.

The idea behind this construction is as follows. For $d = 2$ the list $L^2[\{v\}]$ represents the list L_v of DYNKERNELVC and $L^2[\emptyset]$ represents the list L . The lists $L^2[\{u, v\}]$ are new and will only store a single element and are only added to simplify the code: When an edge $e = \{u, v\}$ is inserted into the 2-hypergraph, we add it to $L^2[e]$, but more importantly also to $L^2[\{u\}]$ and $L^2[\{v\}]$. If it is *relevant* for both lists, we call it *needed* and add it also to $L^2[\emptyset]$. If $L^2[s]$ contains an irrelevant edge, then s is *forced*, and we insert it into $L^1[s]$. For L^1 , the array that manages the internal 1-hypergraph, we have similar rules for being needed and forced. An example of how this works is shown in Figure 2. The next two definitions generalize the idea of *needed* and *forced* hyperedges to arbitrary d and lie at the heart of our algorithm. The earlier rules for $d = 2$ are easily seen to be special cases:

► **Definition 12 (Needed Hyperedges and the Need Invariant).** *A hyperedge e is needed in a list $L^i[s]$ with $s \subsetneq e$ if $e \in R(L^i[t])$ holds for all $t \subseteq e$ with $s \subsetneq t$. A dynamic algorithm maintains the Need Invariant if for all $e \in \binom{V}{\leq d}$, all $s \subsetneq e$, and all $i \in \{0, \dots, d\}$, the list $L^i[s]$ contains e iff e is needed in it.*

7:10 Dynamic Kernels for Hitting Sets and Set Packing

► **Definition 13** (Forced Hyperedges and the Force Invariant). *A set of vertices s is forced by $L^i[s]$ into $L^{i-1}[s]$ or just forced by $L^i[s]$ if $L^i[s]$ has an irrelevant hyperedge. A dynamic algorithm maintains the Force Invariant if for all $i \in \{1, \dots, d\}$ and all $s \in \binom{V}{\leq i}$, the list $L^{i-1}[s]$ contains s iff s is forced by $L^i[s]$.*



■ **Figure 2** The data stored in the lists L^2 , L^1 , and L^0 for $k = 3$ and a dynamic 2-hypergraph with 16 (orange) vertices created with 19 edge insertions (numbers indicate insertion times; there are no deletions in this example). Normal edges are shown as straight lines, singleton edges $\{v\}$ as circles around v , and the empty set as an empty circle. In L^2 , the members of $L^2[\emptyset]$ are drawn in black. They are all relevant for both endpoints and thus needed in $L^2[\emptyset]$. The red edges are not relevant for one of the endpoints and thus neither needed in nor added to $L^2[\emptyset]$. Among the black edges, only the first $k^2 = 9$ are relevant, the rest (dotted) are irrelevant. In L^1 , we store the “forced s ” that L^2 forces into L^1 at the indicated timestamps: each time, it is the first time an irrelevant edge e is inserted into $L^2[s]$. After the first three s (two singletons at timestamps 6 and 10 and then the empty set at timestamp 13) got inserted into $L^1[\emptyset]$, further edges are irrelevant and trigger the insertion of the empty set into $L^0[\emptyset]$.

We will show in Lemmas 18 and 21 that the union $K = \bigcup_{i=0}^d R(L^i[\emptyset])$ is the sought kernel: Each $R(L^i[\emptyset])$ contains (only) those hyperedges e that have not already been taken care of by having forced a subset s of e into the internal $(i - 1)$ -hypergraph.

In the following, we develop code that ensures that the Need Invariant and the Force Invariant hold at all times. We will show that this is the case both for an insert operation and also for delete operations. Then we show that the invariants imply that $K = \bigcup_{i=0}^d R(L^i[\emptyset])$ is a kernel for the hitting set problem. Finally, we analyze the runtimes.

Initialization. The initialization creates the arrays L^i and the relevance lists.

```

1 method DYNKERNELHS.init( $n, k, d$ )
2 // Keep track of relevant edges per vertex ( $V = \{v_1, \dots, v_n\}$  holds by definition):
3 for  $i \in \{0, \dots, d\}$  do
4    $L^i \leftarrow$  new ARRAY( $\binom{V}{\leq i}$ ) initialized with
5     (new RELEVANCE LIST( $k^{i-|s|}$ )) for  $s \in \binom{V}{\leq i}$ 

```

The construct `new ARRAY(D) initialized with $f(s)$ for $s \in D$` allocates a new array with domain D and then immediately set the value of each entry $s \in D$ to $f(s)$. (So, in our case, each $L^i[s]$ will be a new, empty relevance list with relevance bound $k^{i-|s|}$.) The important point is that both allocation and pre-filling can be done in *constant* time using the standard trick to work with uninitialized memory [27].

Independently of the *time* needed for the allocation, observe that the amount of memory we allocate is about $O(n^d)$ – which is already too much in almost any practical setting for $d = 3$, see [31, Chapter 5] for a discussion of experimental findings. However, we will only

use a very small fraction of the allocated memory: The only lists $L^i[s]$ that are non-empty at any point during a run of the algorithms are those where $s \subseteq e \in E$ holds. This means that we actually only need space $O(2^d|E|)$ to manage the non-empty lists if we use hash tables. Of course, this entails a typically non-constant overhead per access for managing the hash tables, which is why our analysis is only for the wasteful implementation above. For a clever way around this problem in the *static* setting, see [32].

► **Lemma 14.** *The Need and Force Invariant hold after the `init` method has been called.*

Proof. All lists are empty after the initialization. ◀

Insertions. We view insertions as a special case of “forcing an edge,” namely as forcing it into the lists of L^d . Adding an edge e to a list $L^i[e]$ can, of course, change the set of relevant edges in $L^i[e]$, which means that e may also be needed in lists $L^i[s]$ for $s \subsetneq e$. It is the job of the method `fix needs downward` to add e to the necessary lists.

```

6  method DYNKERNELHS.insert(e)
7    call insert(e, d) // The hyperedges of  $H$  always get inserted into  $L^d$ 
8
9  function insert(s, i)
10   if  $L^i[s]$  does not already contain  $s$  then // Sanity check
11      $L^i[s]$ .append(s) //  $s$  is always needed in  $L^i[s]$ 
12     call fix force(s, i)
13     call fix needs downward(s, s, i)
14
15   function fix needs downward(s, p, i)
16     // Ensure that the Need Invariant holds for  $s$  with respect to all  $L^i[s']$  with  $s' \subseteq p$ ,
17     // assuming that the Need Invariant holds for  $s$  with respect to all  $L^i[s^*]$  with  $s^* \supseteq p$ :
18     for  $s' \subsetneq p$  in decreasing order of size do // Add  $s$  to all  $L^i[s']$  where  $s$  is needed
19       if  $L^i[s']$  does not contain  $s$  then // Sanity check
20         if  $\forall v \in p - s' : s \in R(L^i[s' \cup \{v\}])$  then // Is  $s$  needed for  $L^i[s']$ ?
21            $L^i[s']$ .append(s) // Yes: it is relevant for all its direct and hence all its supersets
22           call fix force( $s'$ , i)
23
24   function fix force(s, i)
25     if  $L^i[s]$ .has irrelevant elements then // Is  $s$  forced?
26     call insert(s, i - 1)

```

The method `fix needs downward` is more complex than necessary here, but we will need the extra flexibility for the delete method later on: For two sets of vertices s and p with $s \supseteq p$ and a fixed number i , let us say that *the Need Invariant holds for s above p* if for all $s' \supseteq p$ we have $s \in E(L^i[s'])$ iff s is needed for $L^i[s']$. Let us say that *the Need Invariant holds for s below s'* if for all $s' \subseteq p$ we have $s \in E(L^i[s'])$ iff s is needed for $L^i[s']$. In the context of the insert operation, `fix needs downward` always gets called with $s = p$, meaning that in the following lemma the premise (“the Need Invariant holds for s above p ”) is trivially true.

► **Lemma 15.** *Let s and p with $s \supseteq p$ be sets of vertices and let i be fixed. Suppose the Need Invariant holds for s above p . Then after the call `fix needs downward(s, p, i)` the Need Invariant will also hold for s' below p .*

Proof. We need to show that the code ensures for all $s' \subseteq p$ that if s is needed in $L^i[s']$, it gets inserted. It is the job of line 20 to test whether such an insertion is necessary. The line tests whether $\forall v \in p - s' : s \in R(L^i[s' \cup \{v\}])$ holds. By Definition 12 of needed hyperedges,

■ **Table 1** Handling of an insertion for $d = 3$ and $k = 2$. The upper part shows for selected relevance lists a snapshot of their relevant elements (left), their irrelevant elements (right), the list lengths, and the relevance bounds. In lines $L^i[s]$ where the length exceeds the bound (in red), s is forced into $L^{i-1}[s]$. The insertion of $e = \{u, v, w\}$ triggers: (i) e is added to the list $L^3[e]$; (ii) since e is relevant in $L^3[e]$, it is added to the lists for $\{u, v\}$, $\{u, w\}$, and $\{v, w\}$ as well; (iii) e becomes needed in $L^3[\{u\}]$ and gets inserted; (iv) since $L^3[\{u\}]$ was already at maximum capacity ($k^2 = 4$), e becomes the first irrelevant element in this list; (v) this forces $\{u\}$ into $L^2[\{u\}]$; (vi) there $\{u\}$ is the first element and hence relevant and also needed in $L^2[\emptyset]$, where it gets inserted.

$E(L^i[s]) =$	$R(L^i[s])$	\cup	$E(L^i[s]) \setminus R(L^i[s])$	size \leq bound?
$E(L^3[\{u, v, w\}]) =$	\emptyset	\cup	\emptyset	$0 \leq k^0 = 1$
$E(L^3[\{u, v\}]) =$	$\{\{u, v, x\}\}$	\cup	\emptyset	$1 \leq k^1 = 2$
$E(L^3[\{v\}]) =$	$\{\{u, v, x\}\}$	\cup	\emptyset	$1 \leq k^2 = 4$
$E(L^3[\{u\}]) =$	$\{\{u, v, x\}, \{u\}, \{u, x, y\}, \{u, z\}\}$	\cup	\emptyset	$4 \leq k^2 = 4$
$E(L^3[\emptyset]) =$	$\{\{u, v, x\}, \{u\}, \{u, x, y\}, \{u, z\}\}$	\cup	\emptyset	$4 \leq k^3 = 8$
$E(L^2[\{u\}]) =$	\emptyset	\cup	\emptyset	$0 \leq k^1 = 2$
$E(L^2[\emptyset]) =$	\emptyset	\cup	\emptyset	$0 \leq k^2 = 4$
Insertion of $e = \{u, v, w\}$ now yields:				
$E(L^3[\{u, v, w\}]) =$	$\{\{u, v, w\}\}$	\cup	\emptyset	$1 \leq k^0 = 1$
$E(L^3[\{u, v\}]) =$	$\{\{u, v, x\}\{u, v, w\}\}$	\cup	\emptyset	$2 \leq k^1 = 2$
$E(L^3[\{v\}]) =$	$\{\{u, v, x\}\{u, v, w\}\}$	\cup	\emptyset	$2 \leq k^2 = 4$
$E(L^3[\{u\}]) =$	$\{\{u, v, x\}, \{u\}, \{u, x, y\}, \{u, z\}\}$	\cup	$\{\{u, v, w\}\}$	$5 > k^2 = 4$
$E(L^3[\emptyset]) =$	$\{\{u, v, x\}, \{u\}, \{u, x, y\}, \{u, z\}\}$	\cup	\emptyset	$4 \leq k^3 = 8$
$E(L^2[\{u\}]) =$	$\{\{u\}\}$	\cup	\emptyset	$1 \leq k^1 = 2$
$E(L^2[\emptyset]) =$	$\{\{u\}\}$	\cup	\emptyset	$1 \leq k^2 = 4$

what we are *supposed* to test is whether for all $t \subseteq s$ with $s' \subsetneq t$ we have $s \in R(L^i[t])$. Observe that the property of being needed is “upward closed”: if s is needed in $L^i[p]$, it is also needed in all $L^i[s^*]$ with $p \subseteq s^* \subseteq s$. This implies that by processing the hyperedges s' in descending order of size (line 18), s will be needed for $L^i[s']$ iff s is needed for all the hyperedges $t = s' \cup \{v\}$ that are one element larger than s . This is exactly what we test. ◀

► **Lemma 16.** *The Need and Force Invariant are maintained by the insert method.*

Proof. For the Need Invariant, observe that whenever the *fix force* method adds an edge s to $L^i[s]$ in line 11, it also calls *fix needs*(s, s, i) right away. By Lemma 15, this ensures that s is inserted exactly into those $L^i[s']$ for $s' \subseteq s$ where it is needed. For the Force Invariant, observe that we only *add* elements to lists of L^i , which means that they can only *become* forced – they cannot lose this status through an addition of an edge. However, after any insertion of s into any list of L^i (namely, in lines 11 and 21) we immediately call *fix forced*, which inserts s into $L^{i-1}[s]$ if s is forced. ◀

Deletions. The delete operation has to delete an edge e from all places where it might have been inserted to, which is just from all lists $L^d[s]$ for $s \subseteq e$. However, removing e from such a list can have two side-effects: First, it can cause $L^d[s]$ to lose its last irrelevant element, changing the status of s from “forced” to “not forced” and we need to “unforce” it (remove it from $L^{d-1}[s]$), which may recursively entail new deletions. Furthermore, removing e

from $L^d[s]$ may make a previous irrelevant hyperedge (the first irrelevant hyperedge of $L^d[s]$) relevant. Then one has to fix the needs for this hyperedge once more, which may entail new inserts and forcings, but no new deletions (see Table 2 for an example).

```

27 method DYNKERNELHS.delete(e)
28   call delete(e, d)
29
30 function delete(s, i)
31   if  $L^i[s]$  contains s then // Sanity check
32     // Delete s and subsets of s if no longer forced
33   for  $s' \subseteq s$  do
34      $L^i[s']$ .delete(s) // Delete e from all lists that could contain it
35     if not  $L^i[s']$ .has irrelevant elements then // Has  $s'$  now lost its forced status?
36       call delete( $s'$ ,  $i - 1$ )
37
38   // Restore Need Invariant for hyperedges that have suddenly become relevant
39   for  $s' \subseteq s$  do
40      $f \leftarrow L^i[s']$ .last relevant
41     call fix needs downward( $f$ ,  $s'$ ,  $i$ ) // (Only) the last relevant may have changed

```

■ **Table 2** For the situation illustrated in the upper part, we delete the edge $e = \{u, v, w\}$. This triggers: (i) e gets deleted from all $L^3[s]$ with $s \subseteq e$; (ii) $\{u, v, z\}$ becomes relevant for $\{u, v\}$ in L^3 ; (iii) since that was the last irrelevant edge for the set $\{u, v\}$, the edge $\{u, v\}$ gets deleted from the graph represented by L^2 ; (iv) $\{u, z\}$ becomes relevant for $\{u\}$ in L^2 ; (v) as this was the last irrelevant edge, $\{u\}$ gets deleted from L^1 ; (vi) $\{u, z\}$ becomes relevant for $\{u\}$ and needed for $L^2[\emptyset]$; (vii) $\{u, v, z\}$ is now also needed in $L^3[\{u\}]$ and, thus, in $L^3[\emptyset]$ as well.

$E(L^i[s]) =$	$R(L^i[s])$	\cup	$E(L^i[s]) \setminus R(L^i[s])$	size \leq bound?
$E(L^3[\{u, v\}]) =$	$\{\{u, v, y\}, \{u, v, w\}\}$	\cup	$\{\{u, v, z\}\}$	$3 > k^1 = 2$
$E(L^3[\{u, y\}]) =$	$\{\{u, y, v\}, \{u, y, z\}\}$	\cup	$\{\{u, y, x\}\}$	$3 > k^1 = 2$
$E(L^3[\{u, z\}]) =$	$\{\{u, z, v\}, \{u, z, r\}\}$	\cup	$\{\{u, z, y\}\}$	$3 > k^1 = 2$
$E(L^3[\{u\}]) =$	$\{\{u, y, v\}, \{u, v, w\}, \{u, z, r\}\}$	\cup	\emptyset	$3 \leq k^2 = 4$
$E(L^3[\emptyset]) =$	$\{\{u, y, v\}, \{u, v, w\}, \{u, z, r\}\}$	\cup	\emptyset	$3 \leq k^3 = 8$
$E(L^2[\{u\}]) =$	$\{\{u, v\}, \{u, y\}\}$	\cup	$\{\{u, z\}\}$	$3 > k^1 = 2$
$E(L^2[\emptyset]) =$	$\{\{u, v\}, \{u, y\}\}$	\cup	\emptyset	$2 \leq k^2 = 4$
$E(L^1[\{u\}]) =$	$\{\{u\}\}$	\cup	\emptyset	$1 \leq k^0 = 1$
$E(L^1[\emptyset]) =$	$\{\{u\}\}$	\cup	\emptyset	$1 \leq k^1 = 2$
Deletion of $e = \{u, v, w\}$ now yields:				
$E(L^3[\{u, v\}]) =$	$\{\{u, v, y\}, \{u, v, z\}\}$	\cup	\emptyset	$2 \leq k^1 = 2$
$E(L^3[\{u, y\}]) =$	$\{\{u, y, v\}, \{u, y, z\}\}$	\cup	$\{\{u, y, x\}\}$	$3 > k^1 = 2$
$E(L^3[\{u, z\}]) =$	$\{\{u, z, v\}, \{u, z, r\}\}$	\cup	$\{\{u, z, y\}\}$	$3 > k^1 = 2$
$E(L^3[\{u\}]) =$	$\{\{u, y, v\}, \{u, z, r\}, \{u, v, z\}\}$	\cup	\emptyset	$3 \leq k^2 = 4$
$E(L^3[\emptyset]) =$	$\{\{u, y, v\}, \{u, z, r\}, \{u, v, z\}\}$	\cup	\emptyset	$3 \leq k^3 = 8$
$E(L^2[\{u\}]) =$	$\{\{u, y\}, \{u, z\}\}$	\cup	\emptyset	$2 \leq k^1 = 2$
$E(L^2[\emptyset]) =$	$\{\{u, y\}, \{u, z\}\}$	\cup	\emptyset	$2 \leq k^2 = 4$
$E(L^1[\{u\}]) =$	\emptyset	\cup	\emptyset	$0 \leq k^0 = 1$
$E(L^1[\emptyset]) =$	\emptyset	\cup	\emptyset	$0 \leq k^1 = 2$

► **Lemma 17.** *The Need and Force Invariant are maintained by the delete method.*

Kernel. As stated earlier, the dynamic kernel maintained by DYNKERNELHS is the set $K = \bigcup_{i=0}^d R(L^i[\emptyset])$. (K is given only indirectly via d linked lists, but one can do the same transformations as in the proof of Theorem 9 to obtain a compact matrix representation.)

Correctness. We have already established that the algorithm maintains the Need Invariant and the Force Invariant. Our objective is now to show that DYNKERNELHS does, indeed, maintain a kernel at all times. We start with the size:

► **Lemma 18.** *DYNKERNELHS maintains the invariant $|K| \leq k^d + k^{d-1} + \dots + k + 1$.*

Proof. The *init*-method installs a relevance bound of k^i for $L^i[\emptyset]$ for all $i \in \{0, \dots, d\}$. ◀

Lemma 21 shows the crucial property that the current K has a hitting set of size k iff the current hypergraph does. The proof hinges on the following two lemmas on “flower properties”:

► **Lemma 19.** *DYNKERNELHS maintains the invariant that for all $i \in \{0, \dots, d\}$ and all $s \in \binom{V}{\leq i-1}$, the set $E(L^i[s])$ is a $k^{i-|s|-1}$ -flower with core s .*

Proof. First, for all $e \in E(L^i[s])$ we have $s \subseteq e$ since in all places in the *insert*-method where we append an edge e to a list $L^i[s]$, we have $s \subseteq e$ (in line 11 we have $e = s$ and in line 21 we have $s \subsetneq e$ by line 18). Second, consider a vertex $v \in V - s$. We have to show that $\deg_{(V, E(L^i[s]))}(v) \leq k^{i-|s|-1}$ (recall Definition 10) or, spelled out, that v lies in at most $k^{i-|s|-1}$ hyperedges $e \in E(L^i[s])$. By the Need Invariant, all $e \in E(L^i[s])$ are needed. In particular, for $t = s \cup \{v\}$ Definition 12 tells us $e \in R(L^i[t])$. Therefore, we have $\{e \in E(L^i[s]) \mid v \in e\} \subseteq R(L^i[s \cup \{v\}])$ and the latter set has a maximum size of $k^{i-|s \cup \{v\}|} = k^{i-|s|-1}$ due to the relevance bound installed in line 5. ◀

► **Lemma 20.** *DYNKERNELHS maintains the invariant that for all $X \in \binom{V}{\leq k}$ and for all $i \in \{1, \dots, d\}$ and all $s \in \binom{V}{\leq i}$, if s is forced into L^{i-1} and if X hits all elements of $E(L^i[s])$, then X hits s .*

Proof. By Definition 13, “being forced into L^{i-1} ” means that $L^i[s]$ has an irrelevant edge. In particular, $|E(L^i[s])| > k^{i-|s|}$. By Lemma 19, $E(L^i[s])$ is a $k^{i-|s|-1}$ -flower with core s . By Lemma 11, since $|E(L^i[s])| > k^{i-|s|} = k \cdot k^{i-|s|-1}$, we know that X hits s , as claimed. ◀

► **Lemma 21.** *DYNKERNELHS maintains the invariant that H and K have the same size- k hitting sets.*

Run-Time Analysis. It remains to bound the run-times of the insert and delete operations.

► **Lemma 22.** *DYNKERNELHS.insert(e) runs in time $3^d \text{poly}(d)$.*

Proof. The call `DYNKERNELHS.insert(e)` will result in at least one call of `insert(s, i)`: The initial call is for $s = e$ and $i = d$, but the method *fix force* may cause further calls for different values. However, observe that *all* subsequently triggered calls have the property $s \subsetneq e$ and $i < d$. Furthermore, observe that `insert(s, i)` returns immediately if s has already been inserted. We will establish a time bound $t_{\text{insert}}(|s|, i)$ on the total time needed by a call of `insert(s, i)` and a time bound $t_{\text{insert}}^*(|s|, i)$ where we *do not count the time needed by the recursive calls* (made to `insert` in line 26), that is, for a “stripped” version of the

method where no recursive calls are made. We can later account for the missing calls by summing up over all calls that could possibly be made (but we count each only once, as we just observed that subsequent calls for the same parameters return immediately). In a similar fashion, let us try to establish time bounds $t_{\text{fix}}(|s'|, i)$ and $t_{\text{fix}}^*(|s'|, i)$ on the time needed (including or excluding the time needed by calls to *insert*) by a call to the method *fix needs downward*(s, s', i) (note that, indeed, these times are largely independent of s and its size – it is the size of s' that matters).

The starred versions are easy to bound: We have $t_{\text{insert}}^*(|s|, i) = O(1) + t_{\text{fix}}^*(|s|, i)$ as we call *fix needs downward* for $s' = s$. We have $t_{\text{fix}}^*(|s'|, i) = 2^{|s'|} \text{poly } |s'|$ since the run-time is clearly dominated by the loop in line 18, which iterates over all subsets s'' of s' . For each of these $2^{|s'|}$ many sets, we run a test in line 20 that needs time $O(|s'|)$, yielding a total run-time of $t_{\text{fix}}^*(|s'|, i) = O(|s'|2^{|s'|})$. For the unstarred version we get:

$$\begin{aligned} t_{\text{insert}}(|s|, i) &= t_{\text{insert}}^*(|s|, i) + \sum_{s' \subsetneq s, j \in \{|s'|, \dots, i-1\}} t_{\text{insert}}^*(|s'|, j) \\ &= t_{\text{insert}}^*(|s|, i) + \sum_{c=0}^{|s|-1} \underbrace{\binom{|s|}{c}}_{\text{number of } s' \subsetneq s \text{ with } |s'|=c} \sum_{j=c}^{i-1} t_{\text{insert}}^*(c, j) \end{aligned}$$

Plugging in the bound $2^c \text{poly}(c)$ for $t_{\text{insert}}^*(c, j)$, we get that everything following the binomial can be bounded by $(d-c)2^c \text{poly}(c) = 2^c \text{poly}'(c)$. This means that the main sum we need to bound is $\sum_{c=0}^{|s|-1} \binom{|s|}{c} 2^c \leq \sum_{c=0}^{|s|} \binom{|s|}{c} 2^c$. The latter is equal to $3^{|s|}$, which yields the claim. ◀

► **Lemma 23.** *DYNKERNELHS.delete(e) runs in time $5^d \text{poly}(d)$.*

The proof, to be found in the technical report version, is similar to the insertion case, but with some complications resulting from the fact that deletions may trigger insertions.

Proof of Theorem 1. The claim follows from Lemmas 18, 21, 22, and 23. ◀

5 Dynamic Set Packing Kernels

Like the static kernel [1], the dynamic kernel algorithm we have developed in the previous section also works, after a slight modification, for the set packing problem, which is the “dual” of the hitting set problem: Instead of trying to “cover” *all* hyperedges using as few vertices as possible, we must now “pack” *as many* hyperedges as possible. These superficially quite different problems allow similar kernel algorithms because correctness of the dynamic hitting set kernel algorithm hinges on Lemma 11, which states that every size- k hitting set X must hit the core of any b -flower F with $|F| > b \cdot k$. It leads to the central idea behind the complex management of the lists $L^i[s]$: The lists $L^i[s]$ were all b -flowers for different values of b by construction and the moment one of them gets larger than $b \cdot k$, we stop adding hyperedges to its relevant part and instead “switch over to the core s ” by adding s to $L^{i-1}[s]$. It turns out that a similar lemma also holds for set packings:

► **Lemma 24.** *Let F be a b -flower with core c in a d -hypergraph $H = (V, E)$ and let $|F| > b \cdot d \cdot (k-1)$. If $E \cup \{c\}$ has a packing of size k , so does E .*

Proof. Let P be the size- k packing of $E \cup \{c\}$. If $c \notin P$, we are done, so assume $c \in P$. For each $p \in P - \{c\}$, consider the hyperedges in $e \in F$ with $p \cap e \neq \emptyset$. Since p has at most d elements v and since each v lies in at most b different hyperedges of the b -flower F , we conclude that p intersects with at most $d \cdot b$ hyperedges in F . However, this means that the $(k-1)$ different $p \notin P - \{c\}$ can intersect with at most $(k-1) \cdot b \cdot d$ hyperedges in F . In particular, there is a hyperedge $f \in F$ with $f \cap p = \emptyset$ for all $p \in P - \{c\}$. Since $F \subseteq E$, we get that $P - \{c\} \cup \{f\}$ is a packing of E of size k . ◀

Keeping this lemma in mind, suppose we modify the relevance bounds of the lists $L^i[s]$ as follows: Instead of setting them to $k^{i-|s|}$, we set them to $(d(k-1))^{i-|s|}$. Then all lists are b -flowers for a value of b such that whenever more than $b \cdot d(k-1)$ hyperedges are in $L^i[s]$, the set s gets forced into $L^{i-1}[s]$. Lemma 24 now essentially tells us that instead of considering the flower $E(L^i[s])$, it suffices to consider the core s (see also [3, Theorem 3] for more details). Thus, simply by replacing line 5 inside the *init* method as follows, we get a dynamic kernel algorithm for p_k - d -SET-PACKING:

```
5      (new RELEVANCE LIST( $(d(k-1))^{i-|s|}$ )) for  $s \in \binom{V}{\leq i}$  // Modified relevance bounds
```

6 Conclusion

We have introduced a fully dynamic algorithm that maintains a p_k - d -HITTING-SET kernel of size $\sum_{i=0}^d k^i \leq (k+1)^d$ with update time $5^d \text{poly}(d)$ – which is a *constant, deterministic, worst-case* bound – and zero query time. Since p_k - d -HITTING-SET has no kernel of size $O(k^{d-\epsilon})$ unless $\text{coNP} \subseteq \text{NP}/\text{poly}$ [13], and since the currently best static algorithm requires time $|E| \cdot 2^d \text{poly}(d)$ [30], this paper essentially settles the dynamic complexity of computing hitting set kernels. While it seems possible that the update time can be bounded even tighter with an amortized analysis, we remark that this could, at best, yield an improvement from the *already constant* worst-case time $5^d \text{poly}(d)$ to an amortized time of $2^d \text{poly}(d)$.

Our algorithm has the useful property that any size- k hitting set of a kernel is a size- k hitting set of the input graph. Therefore, we can also dynamically provide the following “gap” approximation with constant query time: Given a dynamic hypergraph H and a number k , at any time the algorithm either correctly concludes that there is no size- k hitting set, or provides a hitting set of size at most $\sum_{i=0}^d k^i$. With a query time that is linear with respect to the kernel size, we can also greedily obtain a solution of size dk , which gives a simple d -approximation. A “real” dynamic approximation algorithm, however, should combine the concept of α -approximate pre-processing algorithms [18, 26] with dynamic updates of the hypergraph. This seems manageable if we allow only edge insertions, but a solution for the general case is not obvious to us.

References

- 1 F. N. Abu-Khazam. A Kernelization Algorithm for d -Hitting Set. *Journal of Computer and System Sciences*, 76(7):524–531, 2010. doi:10.1016/j.jcss.2009.09.002.
- 2 Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. *ACM Trans. Algorithms*, 16(4):1–46, July 2020. doi:10.1145/3395037.
- 3 Max Bannach, Zacharias Heinrich, Rüdiger Reischuk, and Till Tantau. Dynamic kernels for hitting sets and set packing. Technical Report TR19-146, Computational Complexity Foundation, 2019. URL: <https://eccc.weizmann.ac.il/report/2019/146>.
- 4 Max Bannach, Malte Skambath, and Till Tantau. Kernelizing the hitting set problem in linear sequential and constant parallel time. In *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22–24, 2020, Tórshavn, Faroe Islands*, pages 9:1–9:16, 2020. doi:10.4230/LIPIcs.SWAT.2020.9.
- 5 Max Bannach and Till Tantau. Computing Hitting Set Kernels By AC^0 -Circuits. *Theory Comput. Syst.*, 64(3):374–399, 2020. doi:10.1007/s00224-019-09941-z.
- 6 S. Bhattacharya, M. Henzinger, and G. F. Italiano. Deterministic Fully Dynamic Data Structures for Vertex Cover and Matching. In *Proceedings of the 26th ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4–6, 2015*, pages 785–804, 2015. doi:10.1137/1.9781611973730.54.

- 7 J. F. Buss and J. Goldsmith. Nondeterminism Within P. *SIAM Journal on Computing*, 22(3):560–572, 1993. doi:10.1137/0222038.
- 8 Jiehua Chen, Wojciech Czerwinski, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michal Pilipczuk, Manuel Sorge, Bartłomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10–13, 2021*, pages 796–809. SIAM, 2021. doi:10.1137/1.9781611976465.50.
- 9 Y. Chen, J. Flum, and X. Huang. Slice-wise Definability in First-Order Logic with Bounded Quantifier Rank. In *Proceedings of the 26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20–24, 2017, Stockholm, Sweden*, pages 19:1–19:16, 2017. doi:10.4230/LIPIcs.CSL.2017.19.
- 10 M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer Berlin Heidelberg, 2015.
- 11 P. Damaschke. Parameterized Enumeration, Transversals, and Imperfect Phylogeny Reconstruction. *Theoretical Computer Science*, 351(3):337–350, 2006. doi:10.1016/j.tcs.2005.10.004.
- 12 S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. Reachability Is in DynFO. *Journal of the ACM*, 65(5):33:1–33:24, 2018. doi:10.1145/3212685.
- 13 H. Dell and D. van Melkebeek. Satisfiability Allows No Nontrivial Sparsification Unless the Polynomial-Time Hierarchy Collapses. *Journal of the ACM*, 61(4):23:1–23:27, 2014. doi:10.1145/2629620.
- 14 R. G. Downey and M. R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. doi:10.1007/978-1-4471-5559-1.
- 15 Rodney G. Downey, Michael R. Fellows, and Ulrike Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In Ronald L. Graham, Jan Kratochvíl, Jaroslav Nešetřil, and Fred S. Roberts, editors, *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future, Proceedings of a DIMACS Workshop, Střirín Castle, Czech Republic, May 19–25, 1997*, volume 49 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 49–99. DIMACS/AMS, 1997. doi:10.1090/dimacs/049/04.
- 16 P. Erdős and R. Rado. Intersection Theorems for Systems of Sets. *Journal of the London Mathematical Society*, 1(1):85–90, 1960.
- 17 Stefan Fafianie and Stefan Kratsch. A shortcut to (sun)flowers: Kernels in logarithmic space or linear time. In *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science, MFCS 2015, Milan, Italy, August 24–28, 2015*, volume 9235 of *Lecture Notes in Computer Science*, pages 299–310. Springer, 2015. doi:10.1007/978-3-662-48054-0_25.
- 18 Michael R. Fellows, Ariel Kulik, Frances A. Rosamond, and Hadas Shachnai. Parameterized approximation via fidelity preserving transformations. *J. Comput. Syst. Sci.*, 93:30–40, 2018. doi:10.1016/j.jcss.2017.11.001.
- 19 Henning Fernau. A top-down approach to search-trees: Improved algorithmics for 3-hitting set. *Algorithmica*, 57(1):97–118, 2010. doi:10.1007/s00453-008-9199-6.
- 20 J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006. doi:10.1007/3-540-29953-X.
- 21 Fedor V. Fomin, Serge Gaspers, Dieter Kratsch, Mathieu Liedloff, and Saket Saurabh. Iterative compression and exact algorithms. *Theor. Comput. Sci.*, 411(7–9):1045–1053, 2010. doi:10.1016/j.tcs.2009.11.012.
- 22 M. Henzinger and V. King. Maintaining Minimum Spanning Forests in Dynamic Graphs. *SIAM Journal on Computing*, 31(2):364–374, 2001. doi:10.1137/S0097539797327209.
- 23 J. Holm, K. de Lichtenberg, and M. Thorup. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

- 24 Y. Iwata and K. Oka. Fast Dynamic Graph Algorithms for Parameterized Problems. In *Proceedings of the 14th Scandinavian Symposium and Workshop on Algorithm Theory, SWAT 2014, Copenhagen, Denmark, July 2–4, 2014*, pages 241–252, 2014. doi:10.1007/978-3-319-08404-6_21.
- 25 R. M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, pages 85–103, 1972. doi:10.1007/978-1-4684-2001-2_9.
- 26 Daniel Lokshtanov, Fahad Panolan, M. S. Ramanujan, and Saket Saurabh. Lossy kernelization. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19–23, 2017*, pages 224–237. ACM, 2017. doi:10.1145/3055399.3055456.
- 27 Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.
- 28 R. Niedermeier and P. Rossmanith. An Efficient Fixed-Parameter Algorithm for 3-Hitting Set. *Journal of Discrete Algorithms*, 1(1):89–102, 2003. doi:10.1016/S1570-8667(03)00009-1.
- 29 S. Patnaik and N. Immerman. DynFO: A Parallel, Dynamic Complexity Class. *Journal of Computer and System Sciences*, 55(2):199–209, 1997. doi:10.1006/jcss.1997.1520.
- 30 R. van Bevern. Towards Optimal and Expressive Kernelization for d -Hitting Set. *Algorithmica*, 70(1):129–147, September 2014. doi:10.1007/s00453-013-9774-3.
- 31 René van Bevern. *Fixed-Parameter Linear-Time Algorithms for NP-hard Graph and Hypergraph Problems Arising in Industrial Applications*, volume 1 of *Foundations of Computing*. Universitätsverlag der TU Berlin, 2014. doi:10.14279/depositonce-4131.
- 32 René van Bevern and Pavel V. Smirnov. Optimal-size problem kernels for d -hitting set in linear time and space. *Information Processing Letters*, 163(105998), 2020. doi:10.1016/j.ipl.2020.105998.
- 33 Magnus Wahlström. *Algorithms, measures and upper bounds for satisfiability and related problems*. PhD thesis, Linköping University, Sweden, 2007.