Electronic Colloquium on Computational Complexity, Report No. 146 (2019)

Dynamic Kernels for Hitting Sets and Set Packing

Max Bannach Zacharias Heinrich Rüdiger Reischuk Till Tantau

Institute for Theoretical Computer Science,
Universität zu Lübeck
Lübeck, Germany

{bannach,zacharias.heinrich,reischuk,tantau}@tcs.uni-luebeck.de

October 31, 2019

Abstract

Computing kernels for the hitting set problem (the problem of finding a size-k set that intersects each hyperedge of a hypergraph) is a well-studied computational problem. For hypergraphs with m hyperedges, each of size at most d, the best algorithms can compute kernels of size $O(k^d)$ in time $O(2^d m)$. In this paper we generalize the task to the *dynamic* setting where hyperedges may be continuously added and deleted and we always have to keep track of a hitting set kernel (including moments when no size-k hitting set exists). We present a deterministic solution, based on a novel data structure, that needs worst-case time $O^*(3^d)$ for updating the kernel upon hyperedge inserts and time $O^*(5^d)$ for updates upon deletions – thus nearly matching the time $O^*(2^d)$ needed by the best static algorithm per hyperedge. As a novel technical feature, our approach does not use the standard replace-sunflowers-by-their-cores methodology, but introduces a generalized concept that is actually easier to compute and that allows us to achieve a kernel size of $\sum_{i=0}^{d} k^i$ rather than the typical size $d! \cdot k^d$ resulting from the Sunflower Lemma. We also show that our approach extends to the dual problem of finding packings in hypergraphs (the problem of finding k pairwise disjoint hyperedges), albeit with a slightly larger kernel size of $\sum_{i=0}^{d} d^i(k-1)^i$.

ISSN 1433-8092

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. Even special cases of the hitting set problem are of high interest: the vertex cover problem is exactly the hitting set problem in which all hyperedges have cardinality two. Furthermore, many interesting problems reduce to the hitting set problem: a dominating set of a graph is just a hitting set in the hypergraph that contains for every vertex a hyperedge that consists of the closed neighborhood of that vertex; 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 in which each occurrence of H contributes a hyperedge – this problem in return 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 [5].

Being a powerful problem, HITTING-SET is unsurprisingly NP-complete [17] and its parameterized version p_k-HITTING-SET is W[2]-complete [10]. 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 [12] and has polynomial kernels. In detail, there is a polynomial-time algorithm that reduces any instance of p_k-d-HITTING-SET to a membership-equivalent instance of size at most $O(k^d \cdot d! \cdot d^2)$ [12]. In this paper we generalize the task to the *dynamic* setting where hyperedges may be continuously added and deleted and we always have to keep track of a hitting set kernel (including moments when no size-k hitting set exists).

Our Results. For each fixed number d, we present a dynamic algorithm that maintains a kernel of size $O(k^d)$ for the hitting set problem with size-d hyperedges. Formally: For fixed $d \in \mathbb{N}$ let p_k-d-HITTING-SET be the parametrized problem whose *instances* are pairs (H, k) where $H = (V, E)$ is a *d-hypergraph* (that is, $|e| \leq d$ holds for all $e \in E$) and $k \in \mathbb{N}$ is the *parameter,* and the *question* is whether there is a *hitting set* $X \subseteq V$ for H with $|X| \leq k$ (that is, $X \cap e \neq \emptyset$ holds for all $e \in E$). We prove:

Theorem 1.1. *There is a fully dynamic kernel algorithm for* p_k-d-HITTING-SET *with at most* $k^d + k^{d-1} + \cdots + k + 1$ *hyperedges in the kernel, insertion time* $O^*(3^d)$*, and deletion time* $O^*(5^d)$*.*

Some remarks are in order: First, we stress that the maximum update time $O^*(5^d)$, which means $O(5^d \cdot d^\ell)$ for some fixed number ℓ, is *independent* of both k and $|E|$. Second, the algorithm works in the fully dynamic case where the hypergraph may switch repeatedly between having and not having a size-k hitting set. Third, our kernel is a *full kernel* in the sense of [7]: 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 as well; and we can even use the whole kernel as approximate solution. Fourth, the kernel size is optimal insofar as p_k-d-HITTING-SET has no kernel of size $O(k^{d-\epsilon})$ unless coNP $\subseteq$ NP/poly [9]. Lastly, 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 $O^*(3^d \cdot |E|)$. Since the currently best algorithm for that task runs in time $O^*(2^d \cdot |E|)$ [20], the run time of our dynamic algorithm is not very far from the best static run time.

Our main technical tool to prove Theorem 1.1 are combinatorial objects that we call *flowers.* They are a generalization of sunflowers [11] with two useful properties: First, they are easy to find and to maintain in a dynamic setting. Second, similar to the Sunflower Lemma, a hypergraph without a flower cannot be too large. Our dynamic algorithm maintains an involved data structure to track all flowers that are present in the given hypergraph, as well as all flowers that are created by the cores of other flowers. In this way, the algorithm is able to dynamically remove existing flowers (and sunflowers, too) from a given hypergraph.

This provides the kernel claimed above, but can also be adapted to other problems with a sunflower methodology. In particular, we can (in a surprisingly simple way) adapt our algorithm

to produce kernels for p_k-MATCHING and the more general p_k-d-SET-PACKING problem: The *instances* for this problem are also d-hypergraphs and a *parameter* k, but the *question* is whether there is a *packing* $P \subseteq E$ with $|P| \geq k$ (that is, $e \cap f = \emptyset$ must hold for any two different $e, f \in P$).

Theorem 1.2. *There is a fully dynamic kernel algorithm for* p_k-d-SET-PACKING *with at most* $\sum_{i=0}^{d}(d(k-1))^i$ *hyperedges in the kernel, insertion time* $O^*(3^d)$*, and deletion time* $O^*(5^d)$*.*

Related Work. Parameterized complexity is a fast-growing subfield of both, complexity theory and modern algorithm design. There are many textbooks that present an overview over the many facets of this field [6, 10, 12]. A useful tool of this field is kernelization: algorithms that reduce a given instance with a guarantee (see for instance the textbook by Fomin et al. for an recent overview [13]). The first kernel for p_k-d-HITTING-SET is due to Flum and Grohe [12], and a refined version was presented by van Bevern [20]. Damaschke studied *full* kernels for the problem, which are kernels that contain all small solutions [7]. There are also optimized algorithms for specific values of d: for instance the algorithm by Buss and Goldsmith [4] for $d = 2$, or by Niedermeier and Rossmanith [18] and Abu-Khzam [1] for $d = 3$.

Dynamic algorithms can be used in a variety of monitoring applications, for instance for maintaining a minimum spanning tree [14] or connected components [15]. There is also a recent trend in obtaining dynamic approximation algorithms, for instance for MATCHING and VERTEX-COVER [3]. Algorithms that maintain a solution for a dynamically changing input can also be studied with descriptive complexity, as suggested by Patnaik and Immerman [19]. A recent break-through result in this area is that reachability is contained in DynFO [8].

Iwata and Oka where the first to combine both fields by studying a dynamic quadratic kernel for p_k-VERTEX-COVER – the restriction of the hitting set problem to ordinary graphs [16]. However, their algorithm requires $O(k^2)$ update time and works only in the dynamic promise model: the algorithm assumes that, at any time, there actually *is* a size-k vertex cover in the input graph. The algorithm was improved by Alman, Mnich, and Williams [2] to $O(k)$ worst case update time and $O(1)$ amortized update time and it works in the full dynamic model. That paper provides a fully dynamic algorithm for p_k-d-HITTING-SET that produces a kernel of size $d! \cdot d^{d+1} \cdot k^{O(d^2)}$ with an update time of $(d!)^d \cdot k^{O(d^2)}$ [2].

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. Appendix A contains technical details on the implementation of basic dynamic data structures.

2 A Framework for Parametrized Dynamic Algorithms

Our aim is to *dynamically* maintain *small kernels* for graph problems with *minimal update time.* To better formalize this, we begin with the standard definition of *kernels* and then explain which properties a *dynamic* (kernel) algorithm should have. Since we are interested in constant update times, some remarks on standard data structures will also be of interest.

Parameterized Hypergraph Problems and Kernels. The inputs for parameterized hypergraph problems are, of course, *hypergraphs,* which are pairs $H = (V, E)$ consisting of a set V of *vertices* of some size $n = |V|$ and a set E of *hyperedges* with $e \subseteq V$ for all $e \in E$. Let $\deg_H(v) = \big|\{e \in E \mid v \in e\}\big|$ denote the degree of v in H. For a number $d \in \mathbb{N}$, a *d-hypergraph* H has $|e| \leq d$ for all $e \in E$. A *uniform d-hypergraph* has $|e| = d$ for all $e \in E$. In particular, a *graph*

is just a uniform 2-hypergraph. We use the notation $\binom{V}{d}$ to denote the set $\{ e \subseteq V \mid |e| = d \}$ of all size-d hyperedges in V and similarly let $\binom{V}{\leq d} = \{ e \subseteq V \mid |e| \leq d \}$.

Parameterized hypegraphs 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. The p$_k$-d-HITTING-SET and p$_k$-d-SET-PACKING problems from the introduction are examples. Note that in both cases k is the parameter while d is fixed; the special cases for $d = 2$ are exactly p$_k$-VERTEX-COVER and p$_k$-MATCHING.

A core question of parameterized complexity theory is, which parameterized problems Q are in the class FPT of *fixed-parameter tractable* problems. This means that $(H, k) \in^? Q$ can be decided in time $f(k) \cdot (|V||E|)^{O(1)}$ for some computable function f. It is well-known that $Q \in$ FPT holds if, and only if, *kernels* can be computed for Q in polynomial time. Kernels of polynomial size are of special interest:

Definition 2.1 (Polynomial Kernel)**.** Let $Q \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem and let p be a polynomial. A *kernel for an instance* $(w, k) \in \Sigma^* \times \mathbb{N}$ *and the problem* Q *(and* p*)* is another instance $(w', k') \in \Sigma^* \times \mathbb{N}$ with $|w'| \leq p(k)$, $k' \leq p(k)$, and $(w, k) \in Q \iff (w', k') \in Q$.

Kernel algorithms normally ensure $k' \leq k$ and in our paper we always have $k' = k$. It is well-known that for p$_k$-d-HITTING-SET and p$_k$-d-SET-PACKING polynomial kernels can be computed in polynomial time. The objective of this paper is to maintain such kernels in a *dynamic* setting.

Dynamic Hypergraphs and Dynamic Kernels. Of the many different aspects of a hypergraph that could possibly 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. Only the hyperedge set E will change dynamically and it will start out as the empty set:

Definition 2.2 (Dynamic Hypergraphs)**.** A *dynamic hypergraph* consists of a fixed vertex set $V = \{v_1, \ldots, v_n\}$ and a sequence $o_1, o_2, o_3, \ldots$ of *update operations*, where each o_i is either *insert*(e_j) or *delete*(e_j) for a hyperedge $e_j \subseteq V$.

The dynamic hypergraph defines a sequence of hypergraphs $H_0, H_1, \ldots$ in the obvious way: Starting with the empty hypergraph $H_0 = (V, \emptyset)$, let $H_i = (V, E(H_{i-1}) \cup \{e_j\})$ for $o_i = \mathit{insert}(e_j)$, and $H_i = (V, E(H_{i-1}) \setminus \{e_j\})$ for $o_i = \mathit{delete}(e_j)$. Note that we may restrict the sequences to consist only of nonredundat operations: $e_j \notin E(H_{i-1})$ as o_i is an insert, and $e_j \in E(H_{i-1})$ in case of a delete; these requirements can easily be checked before executing o_i.

A *dynamic* hypergraph algorithm for a hypergraph problem 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 just solve the problem from scratch for the updated H_i, we may be able to do better by taking into account that the difference between successive graphs H_{i-1} and H_i are very small: By keeping track of an internal *auxiliary data structure* A_i that the algorithm updates alongside the actual input graph, we may be able to solve the original problem very quickly after each update. In our setting, what we wish to compute for each H_i is a kernel:

Definition 2.3 (Dynamic Kernel Algorithm)**.** Let Q be a parameterized problem and σ a polynomial. A *dynamic kernel algorithm* ALGO *for* Q *with kernel size* $\sigma(k)$ consists of three methods:

1. 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, k') for (H_0, k) and Q and σ (observe that $H_0 = (V, \emptyset)$ holds).
2. 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, k') being a kernel for (H_i, k) and Q and σ.
3. ALGO.*delete*(e) removes an edge instead of adding it.

An efficient dynamic kernel algorithm should compute K_i (and also A_i) faster than a standard static kernel algorithm that processes a whole H_i in polynomial time with respect to the graph size n. The best one could hope for would be the other extrem: constant time per update, even independent of the parameter k – and this is exactly what we achieve in this paper (though the update time *does* depend exponentially on d). Note that we do *not* count the time to create the initial empty data structure A_0, which will typically have polynomial size, but which is essentially empty. It only depends on the size n of the vertex set V and possibly k and d.

Data Structures for Dynamic Algorithms. The sequence of auxiliary data A_i relies on standard data structures such as objects, arrays, and maps. In addition, a novel data structure used are *relevance lists,* which are lists equipped with a *relevance bound* ρ*:* the first ρ elements in the list are said to be *relevant,* while the others are *irrelevant.* This data structure supports insertion and deletion of elements, querying the relevance status of a given element, and querying the last relevant element – each in $O(1)$ time. Implementation details and a formal analysis of these lists, and for all other data structures used, are provided in Appendix A.

3 Dynamic Vertex Cover with Constant Update Time

Our first dynamic algorithm maintains kernels of size $O(k^2)$ for the vertex cover problem with update time $O(1)$. It is based on a well-known *static* kernel for the vertex cover problem: Buss [4] 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*). Once all heavy vertices have been picked and removed, if there are still more than k^2 edges, then it is easy to see that no vertex cover of size k is possible (since light vertices can cover at most k edges).

To turn this idea into a dynamic kernel, let us first consider only insertions. Initially, new graph edges can just be added to the kernel; but at some point some vertex v "becomes heavy." In the static setting one would now 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 and its edges in the graph, but do *not* add further v-edges 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 ensures 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 3.3 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 2.3. 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:

```
1 method DynKernelVC.init(n, k) // V = {v_1, ..., v_n} holds by definition
2   for v ∈ V do
3     L_v ← new relevance list(k + 1) // Keep track of edges relevant for a vertex
4   L ← new relevance list(k^2 + k + 1) // Keep track of edges relevant for the kernel
```

The insert operation adds an edge to the relevance lists of both endpoints of the edge. Furthermore, it also adds the edge 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  function check if needed(e) // assume e = {u, v}
9    if L_u.is relevant(e) ∧ L_v.is relevant(e) then
10     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 e' 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.

```
11 method DynKernelVC.delete(e) // assume e = {u, v}
12   L.delete(e)
13   L_u.delete(e); L_v.delete(e)
14   check if needed(L_u.last relevant); check if needed(L_v.last relevant)
```

In the code, we tacitly assume that borderline cases like a delete on a non-existing edge or inserts on already-existing edges are handled sensibly.

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. 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 3.1. DynKernelVC *maintains the invariant* $|R(L)| \leq k^2 + k + 1$.

Proof. The relevance list L is setup in the *init* method to have at most the claimed number of relevant elements. □

Lemma 3.2. DynKernelVC.*insert and* DynKernelVC.*delete run in time* $O(1)$.

Proof. The codes itself clearly only needs time $O(1)$ and it calls only operations on relevance lists, all of which also only take time $O(1)$ by Lemma A.1, which is presented in the appendix. □

Third and much less obvious, $R(L)$ and E always have the same size-k vertex covers:

Lemma 3.3. DynKernelVC *maintains the invariant that* $(V, R(L))$ *and the current graph* (V, E) *have the same size-*k *vertex covers.*

Proof. One direction is trivial since $R(L) \subseteq E$. For the other direction, consider a size-k vertex cover X of $R(L)$, that is, a set X with $|X| = k$ and $X \cap e \neq \emptyset$ for all $e \in R(L)$. We need to show that $X \cap e \neq \emptyset$ holds for all $e \in E$. We distinguish three cases: $e \in R(L)$, $e \in E(L) - R(L)$, and $e \in E - E(L)$.

Case 1: The edge is in L and is relevant. The first case is trivial: If $e \in R(L)$, then by assumption we have $X \cap e \neq \emptyset$ as claimed.

Case 2: The edge is in L, but is irrelevant. For the second case, we need an observation:

Claim. *The degree of vertices in $(V, R(L))$ is at most $k+1$.*

Proof. Consider any $v \in V$. All edges in $R(L)$ that contain v must be *relevant edges with respect to L_v* since the function *L.append if needed* only allows such edges to enter L. However, the *init* method setup L_v to contain at most $k+1$ relevant edges. □

Using this observation, we see that the second case ($e \in E(L) - R(L)$) cannot happen: L can only have an irrelevant edge if there are already k^2+k+1 relevant edges in $R(L)$. However, by the claim, each of the k many $x \in X$ covers at most $k+1$ edges in $R(L)$, implying that X covers at most $k(k+1) = k^2+k$ edges of $R(L)$. In particular, contrary to the assumption, one edge of $R(L)$ is not covered by X.

Case 3: The edge is not even in L. For the third case, let $e \in E - E(L)$, that is, let $e = \{u, v\}$ be an edge that "did not make it into the L list." This can only happen because it was irrelevant for L_u or L_v (or both).

Recall that when e is irrelevant for a list L_u, this means that u has more than $k+1$ adjacent edges in E and, hence, u *must* be present in any vertex cover of the graph $G = (V, E)$. If all the relevant edges of u are also present in $R(L)$, then u has exactly $k+1$ neighbors in the graph $(V, R(L))$ and, in particular, its vertex cover X must include u. Unfortunately, it may happen that even though a vertex u has some irrelevant adjacent edges in E, not all relevant edges of L_u make it into L: After all, the other endpoint v of an edge $e = \{u, v\}$ may *also* have irrelevant adjacent edges and e may happen to be one of them. We can now try to apply the same argument to v; but may again find yet another edge e' and another vertex w that causes v to have a degree less than $k+1$ in $R(L)$. Fortunately, it turns out that after a finite number of steps, we arrive at a vertex that *must* be present in X. Furthermore, starting from this vertex, we can track back to show that eventually we must have $u \in X$. The details are as follows.

Claim. *There is an ordering $u_1, \ldots, u_q$ of the vertices of degree at least $k+1$ in E such that for each $i \in \{1, \ldots, q\}$ there are at least $k+1-(i-1)$ edges in $R(L)$ of the form $\{u_i, v\}$ with $v \notin \{u_1, \ldots, u_{i-1}\}$.*

Proof. In the current graph G, each edge e has a time t_e when it entered the graph and these times define a total order on the edges in E. For each vertex v, let $l(v)$ be the *last relevant edge of L_v*, that is, the edge returned by L_v*.last relevant*. Order the vertices of degree at least $k+1$ in E according to the following rule: For $i < j$ we must have that $t_{l(u_i)} \leq t_{l(u_j)}$ (if two vertices u_i and u_j happen to have the same last relevant edge, they can be ordered arbitrarily).

Consider any u_i. Then all edges from u_i to any vertex $v \notin \{u_1, \ldots, u_{i-1}\}$ are *relevant for L_v* since the last relevant edge of L_v is an edge that came *later* than the edge $\{u_i, v\}$ and, hence, $\{u_i, v\}$ is relevant for L_v. However, this means that the only edges of $R(L_{u_i})$ that do not get passed to L can be those of the form $\{u_i, u_j\}$ for some $j \in \{1, \ldots, i-1\}$. Clearly, since L_{u_i} has $k+1$ relevant edges and only $i-1$ do not get passed, we get the claim. □

This claim has the implication that $\{u_1, \ldots, u_q\} \subseteq X$. We show by induction on i that $\{u_1, \ldots, u_i\} \subseteq X$ holds. The case $i = 0$ is trivial. For the inductive step from $i-1$ to i, consider u_i. By the claim, there are $k+1-(i-1)$ edges in $R(L)$ from u_i to vertices $v \notin \{u_1, \ldots, u_{i-1}\}$. Since by the induction hypothesis we have $\{u_1, \ldots, u_{i-1}\} \subseteq X$, if we do not have $u_i \in X$, then the $X - \{u_1, \ldots, u_{i-1}\}$ must contain enough vertices to cover the $(k+1)-(i-1)$ edges between u_i and vertices not in $\{u_1, \ldots, u_{i-1}\}$. However, $|X - \{u_1, \ldots, u_{i-1}\}| \leq k-(i-1)$ and, thus, this is impossible.

This concludes the third case: If $e \in E - E(L)$, then one or both elements of e must be one of the u_i – and we just saw that all of them are in X. Hence, as claimed, $X \cap e \neq \emptyset$. □

Put together, we get the following special case of Theorem 1.1 (the proof contains some further details on how $R(L)$ is kept in memory):

Theorem 3.4. 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 3.1, 3.2, and 3.3 together state that at all times during a run of the algorithm DynKernelVC the hypergraph $(V, R(L))$ has at most $k^2 + k + 1$ hyperedges 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, while a kernel should be a mathematical object whose encoding *only* depends on k (encoding the lists takes something like $O(k^2 \log n)$ since we need $O(\log n)$ bits to encode a vertex number). Furthermore, the whole idea behind kernelizations is, of course, that we should be able to perform further computations on the kernel once it has been determined. Thus, we do not wish to spend a non-constant time like $O(k^2)$ to transform the lists into something "usable" when we actually use the kernel to find a solution.

Fortunately, it turns out that we can keep track of a "real" kernel in the form of an adjacency matrix still with update times $O(1)$. We need some terminology: For a set E of edges, let $\bigcup E = \{\, x \mid \exists e \in E \colon x \in e \,\}$ denote the set of all vertices mentioned in any edge of E. For two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ let us write $G_1 \sim G_2$ if G_1 and G_2 are isomorphic. For two edge sets E_1 and E_2, let us write $E_1 \sim E_2$ if $(\bigcup E_1, E_1) \sim (\bigcup E_2, E_2)$ (so vertices that are not involved in any edges are ignored).

Our objective is the following: We wish to dynamically keep track of a graph $K = (V_K, E_K)$ with the fixed vertex set $V_K = \{1, \ldots, 2(k^2 + k + 1)\}$ such that we always have $E_K \sim R(L)$. (Note that vertices not covered by E_K or not by $R(L)$ are not relevant for the isomorphism; but they are also not relevant for the vertex cover problem.) If we can maintain such an isomorphic graph, then K is continuously a vertex cover kernel for the current graph G in the sense of Definition 2.3: First, trivially, K has a vertex set size of at most $O(k^2)$. Second, from Lemma 3.3 we know that $(V, R(L))$ and G have the same size-k vertex covers; and in particular $(V, R(L))$ *has* a vertex cover of size at most k if, and only if, K, with $E_K \sim R(L)$, has one.

We now show how to modify DynKernelVC so that it keeps track of a "real" kernel K with update times $O(1)$. The change is that whenever an edge e enters or leaves $R(L)$, we update K in time $O(1)$ such that $R(L) \sim E_K$ still holds. To achieve this, we use the following auxiliary data structures, all of which are initialized in the *init* method (in addition to the lists L_v and L):

1. An adjacency matrix of Boolean entries storing $E_K \subseteq \binom{V_K}{2}$, indicating which edges are present in K,
2. a mapping ι that stores for each vertex of $\bigcup R(L)$ to which vertex in V_K it corresponds (and $\iota(x) = \perp$ for $x \notin \bigcup R(L)$),
3. an array D that stores for each $v \in V_K$ the degree of v in K, and
4. a list Z of *zero degree vertex intervals in* K. Each element of the list is a pair (a, b) of numbers from V_K that stands for the interval $[a, b]$. The semantics is that the union of the intervals should be exactly the set of vertices in V_K that have degree 0 in K. Clearly, we can initialize the Z with the single interval $[1, 2(k^2 + k + 1)]$ to ensure that this holds at the beginning.

Translated to code, we get:

```
1  method DynKernelVC.init(n, k)
2    ... // As before in the original code of the init method
3
4    // New structures for keeping track of K:
5    E_K ← new ARRAY(binom(V_K, 2))
6    ι ← new ARRAY({1, ..., n})
7    D ← new ARRAY(V_K)
```

```
8    Z ← new List
9    Z.append([1, 2(k² + k + 1)])
```

Let us now see how these auxiliary data structures allow us to keep track of E_K such that $E_K \sim R(L)$ holds when edges enter or leave $R(L)$:

Inserting Edges. Suppose $e = \{u, v\}$ is about to be added to $R(L)$. First, we test whether $u \notin \bigcup R(L)$ holds (by testing whether $\iota[u] = \perp$ holds). In this case, consider the first interval $[a, b]$ in Z (such an interval must exist since there will never be more than $2(k^2 + k + 1)$ vertices in $R(L)$ and, hence, there is always a vertex of degree 0 in K when a new vertex is about to enter $\bigcup R(L)$). If $a = b$, remove this interval from Z, otherwise replace it by $[a+1, b]$. We think of this as "allocating" a and will store in ι that u gets mapped to a. Next, if $v \notin \bigcup R(L)$ holds, we allocate a vertex from V_K for it. Then both u and v have corresponding vertices in V_K and we store an edge between them in E_K and adjust the values in D accordingly: The function *check if needed* inside the whole algorithm gets replaced as follows:

```
10   function check if needed(e) // assume e = {u, v}
11     if L_u.is relevant(e) and L_v.is relevant(e) then
12       L.append(e)
13       // New part:
14       if L.is relevant(e) then
15         allocate(u)
16         allocate(v)
17         if E_K[ι[u], ι[v]] = false then
18           E_K[ι[u], ι[v]] ← true
19           D[ι[u]] ← D[ι[u]] + 1
20           D[ι[v]] ← D[ι[v]] + 1
```

where the function *allocate* works as follows:

```
21   function allocate(u)
22     if ι[u] = ⊥ then
23       [a, b] ← first element of L
24       ι[u] ← a
25       if a = b then
26         remove first element of L
27       else
28         replace first element of L by [a + 1, b]
```

Observe that after the above steps, $E_K \sim R(L)$ holds and all auxiliary data structures hold the proper values.

Deleting Edges. Suppose $e = \{u, v\}$ is about to be deleted from $R(L)$. The code for this case is straightforward:

```
29   method DynKernelVC.delete(e)
30     if L.is relevant(e) then
31       // When e ∈ R(L), remove the corresponding edge from E_K:
32       E_K[ι[u], ι[v]] ← false
33       D[ι[u]] ← D[ι[u]] − 1 // Adjust the degrees
34       D[ι[v]] ← D[ι[v]] − 1
35       if D[ι[u]] = 0 then Z.append([ι[u], ι[u]]) // "Free" the vertex ι[u]
36       if D[ι[v]] = 0 then Z.append([ι[v], ι[v]]) // "Free" the vertex ι[v]
37
38     ... // Now the original code of the delete method
```

Once more, $E_K \sim R(L)$ holds after the updates and all auxiliary data structure have also been updated correctly. □

4 Dynamic Hitting Set Kernels

The hitting set problem is the natural generalization of the vertex cover problem 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 $|V|$, $|E|$, and k, while it does depend on d (in fact even exponentially). Such an exponential dependency on d seems currently unavoidable, as a direct consequence of our dynamic algorithm is a static algorithm with running time $O^*(3^d \cdot |E|)$, and the currently best static algorithm with a linear dependency on $|E|$ runs in time $O^*(2^d \cdot |E|)$.

The first core idea of our algorithm concerns a replacement notion for the "heavy vertices" from the previous section. For this purpose the notion of *sunflowers* [11] is usually used – but, they are hard to find and manage, especially dynamically. To overcome this difficulty, we introduce a generalization of sunflowers called *b-flowers* for different parameters $b \in \mathbb{N}$ that will be easier to keep track of.

The second core idea is to recursively reduce each case d to the case $d-1$: For a fixed $d > 2$, as in the dynamic algorithm for the vertex cover problem, 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

Kernelization algorithms for p_k-d-HITTING-SET typically rely on so-called *sunflowers,* as suggested by Flum and Grohe [12, Section 9.1]. 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* of the sunflower, 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 with more than k edges – which means that replacing large sunflowers in hypergraphs by their cores is a reduction rule for p_k-d-HITTING-SET. Even better, this simple rule is guaranteed to yield a kernel since the Erdős–Rado *Sunflower Lemma* [11] 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 introduce a more general concept called *b-flowers.* These structures are simpler than sunflowers and, especially, easier to find.

Definition 4.1. 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. A comprehensive example can be found in Figure 1.

The following property of b-flowers will be essential for our dynamic kernelization strategy (it implies that we can replace sufficiently large flowers by their cores):

Lemma 4.2. *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 explain how the (already covered) case $d = 2$ can

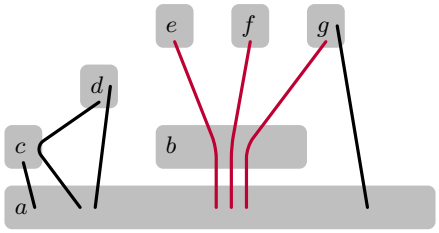

Figure 1: A hypergraph $H = (\{a, b, c, d, e, f, g\}, E)$ in which each hyperedge $e \in E$ is drawn as a line and contains all vertices it "touches". There is a 1-flower (a sunflower) with core $\{a\}$ and the four edges $\{a, c\}$, $\{a, d\}$, $\{a, g\}$, and either $\{a, b, e\}$ or $\{a, b, f\}$; just not both as b would have degree 2. There is a 2-flower with core $\{a\}$ and six edges (all except for one that includes b). There also is a 1-flower with core $\{a, b\}$, consisting of the three red edges. Finally there is a 3-flower with core $\{a\}$ containing all edges.

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 exclusively managed 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 an internal 1-hypergraph to keep track of 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 internal 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 (it will be 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 (internal, except for $i = d$) 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 store edges $e \in \binom{V}{\leq i}$ with $e \supseteq s$ in it.

The idea is that, 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 then also add it 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 4.3 (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 $i \in \{0, \ldots, d\}$, the list $L^i[s]$ contains e if, and only if, e is needed in it.

Definition 4.4 (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 $i \in \{1, \ldots, d\}$ and all $s \in \binom{V}{<i}$, the list $L^{i-1}[s]$ contains s if, and only if, s is forced by $L^i[s]$.

We will show in Lemmas 4.10 and 4.13 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 been already been taken care of by having forced a subset s of e into the internal $(i-1)$-hypergraph.

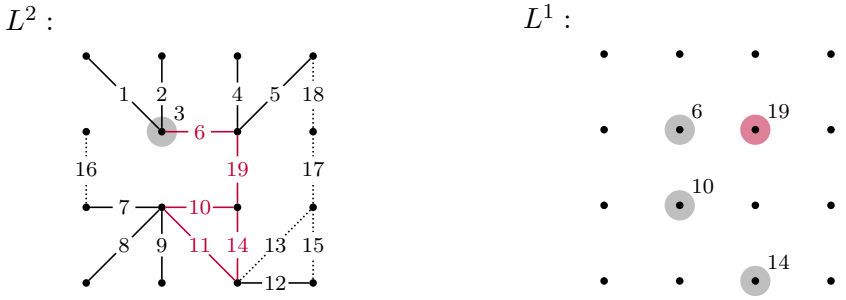

Figure 2: A hitting set instance with $k = 3$ and a dynamic 2-hypergraph with 16 vertices, where the sequence of operations consists of 19 edge insertions. Edges of cardinality 2 are illustrated by a dash, while singelton edges are illustrated by a gray dot that surrounds the corresponding vertex. In both cases, the small number indicates the timestamp at which the edge has been inserted. The left figure illustrates the data structure L^2, which (besides other information) stores all inserted edges. Edges $\{u, v\}$ that are black in the picture are needed by both of their endpoints and, thus, relevant in $L^2[\{u\}]$ and $L^2[\{v\}]$. In contrast, red edges are not. The only singelton edge (the gray dot) is needed by its vertex. Black edges that are solid are needed by the L^2-kernel and, thus, are relevant in $L^2[\emptyset]$. The black dotted edges are not relevant in $L^2[\emptyset]$. The right figure shows the data structure L^1, which encodes the 1-hypergraph of "forced subedges." The gray singelton edges correspond to the "forced subedges" that we move down from L^2 to L^1 at the corresponding timestamp. All three gray singelton edges are relevant in $L^1[\emptyset]$. The red singelton edge that is inserted at timestamp 19 is not relevant in $L^1[\emptyset]$ and, thus, at this timestamp an empty edge is created in $L^0[\emptyset]$.

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.

Initializiation. 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, ..., v_n} holds by definition):
3    for i ∈ {0, ..., d} do
4      L^i ← new array(binom(V, ≤i))
5      for s ∈ binom(V, ≤i) do
6        L^i[s] ← new relevance list(k^{i−|s|})
```

A remark on the run-time of this method may be in order: Recall that we opted not to count the time needed for the initialization since the initial data structure "typically [...] is essentially empty" as we wrote. However, while creating the initially empty arrays in line 4 can, indeed, typically be done very quickly, allocating the roughly n^d (empty) lists $L^i[s]$ in the loop in the next lines will take quite some time. In an actual implementation, we would *not* create any of these lists inside the *init* method: Rather, only when the dynamic algorithm actually tries to access any particular $L^i[s]$ and finds $L^i[s] = \bot$, a list would be created.

Lemma 4.5. *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.

```
1  method DynKernelHS.insert(e)
2    call insert(e, d) // The hyperedges of H always get inserted into L^d
3
4  function insert(s, i)
5    if L^i[s] does not already contain s then // Sanity check
6      L^i[s].append(s) // Add s to the trivial list ...
7      call fix force(s, i)
8      call fix needs downward(s, s, i)
9
10 function fix needs downward(s, p, i)
11   // Ensure that the Need Invariant holds for s with respect to all L^i[s'] with s' ⊆ p,
12   // assuming that the Need Invariant holds for s with respect to all L^i[s*] with s* ⊇ p:
13   for s' ⊊ p in decreasing order of size do // Add s to all L^i[s'] where s is needed
14     if not L^i[s'] contains s then // Sanity check
15       if ∀v ∈ p − s' : s ∈ R(L^i[s' ∪ {v}]) then // Is s needed for L^i[s']?
16         L^i[s'].append(s) // Yes: it is relevant for all its direct and hence all its supersets
17         call fix force(s', i)
18
19 function fix force(s, i)
20   if L^i[s].has irrelevant elements then // Is s forced?
21     call insert(s, i − 1)
```

Example 4.6. Let us illustrate how the dynamic algorithm handles the insertion and deletion of hyperedges. We consider a dynamic 3-hypergraph with vertex set $V = \{u, v, w, x, y, z\}$ and we assume we wish to compute a hitting set kernel for $k = 2$. For both operations, we present one example in the form of a table that shows the hypergraph *before* the operation takes place (marked with a $\star$) and the data structures *after* the operation was applied (marked with a †). In order to keep things clear, the tables only contain a few selected relevance lists (left), their relevant contents (left of the line in the middle), their irrelevant content (to the right of the line), and finally the size of the list and its bound. We highlight the size in red if it exceeds the bound of the list, that is, if the list contains irrelevant elements (and is, thus, forced).

We start with the insertion of the hyperedge $e = \{u, v, w\}$. Before this operation takes place, there are already some hyperedges in the graph (as shown in the $\star$-part of the following table), but no hyperedge is forced yet. Observe that e is not part of the graph and that the subedge $s = \{u\}$ is already at its bound. Inserting e has several effects (as shown the †-part of the following table): (i) e is added to the list $L^3[e]$; (ii) since e is relevant in $L^3[e]$, e is added to the lists of $\{u, v\}$, $\{u, w\}$, and $\{v, w\}$ as well; (iii) e becomes needed in $L^3[\{u\}]$; (iv) since it was already at its bound, e becomes the first irrelevant element in this list; (v) this forces $\{u\}$ into $L^2[\{u\}]$; (vi) since there are no other edge in L^2, the edge is obviously needed in $L^2[\emptyset]$.

	$L^i[s] =$	$\{R(L^i[s])$	$E(L^i[s]) \setminus R(L^i[s])\}$	size/bound
$\star$	$L^3[\{u,v,w\}]$	$\{$	$\}$	0/1
	$L^3[\{u,v\}]$	$\{\{u,v,x\}$	$\}$	1/2
	$L^3[\{v\}]$	$\{\{u,v,x\}$	$\}$	1/4
	$L^3[\{u\}]$	$\{\{u,v,x\},\{u\},\{u,x,y\},\{u,z\}$	$\}$	4/4
	$L^3[\emptyset]$	$\{\{u,v,x\},\{u\},\{u,x,y\},\{u,z\}$	$\}$	4/8
	$L^2[\{u\}]$	$\{$	$\}$	0/2
	$L^2[\emptyset]$	$\{$	$\}$	0/4
†	$L^3[\{u,v,w\}]$	$\{\{u,v,w\}$	$\}$	1/1
	$L^3[\{u,v\}]$	$\{\{u,v,x\}\{u,v,w\}$	$\}$	2/2
	$L^3[\{v\}]$	$\{\{u,v,x\}\{u,v,w\}$	$\}$	2/4
	$L^3[\{u\}]$	$\{\{u,v,x\},\{u\},\{u,x,y\},\{u,z\}$	$\{u,v,w\}\}$	5/4
	$L^3[\emptyset]$	$\{\{u,v,x\},\{u\},\{u,x,y\},\{u,z\}$	$\}$	4/8
	$L^2[\{u\}]$	$\{\{u\}$	$\}$	1/2
	$L^2[\emptyset]$	$\{\{u\}$	$\}$	1/4

We apply some further (not specified) insertions and deletions on the resulting graph, leading to a situation as illustrated in the $\star$-part of the following table. Observe that we now have a set of edges that force (among others) the set $\{u, v\}$ in L^3 and the set $\{u\}$ in L^2. From this graph we delete the edge $e = \{u, v, w\}$, which triggers the following sequence of events: (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.

	$L^i[s] =$	$\{R(L^i[s])$	$E(L^i[s]) \setminus R(L^i[s])\}$	size/bound
$\star$	$L^3[\{u,v\}]$	$\{\{u,v,y\},\{u,v,w\}$	$\{u,v,z\}\}$	3/2
	$L^3[\{u,y\}]$	$\{\{u,y,v\},\{u,y,z\}$	$\{u,y,x\}\}$	3/2
	$L^3[\{u,z\}]$	$\{\{u,z,v\},\{u,z,r\}$	$\{u,z,y\}\}$	3/2
	$L^3[\{u\}]$	$\{\{u,y,v\},\{u,v,w\},\{u,z,r\}$	$\}$	3/4
	$L^3[\emptyset]$	$\{\{u,y,v\},\{u,v,w\},\{u,z,r\}$	$\}$	3/8
	$L^2[\{u\}]$	$\{\{u,v\},\{u,y\}$	$\{u,z\}\}$	3/2
	$L^2[\emptyset]$	$\{\{u,v\},\{u,y\}$	$\}$	2/4
	$L^1[\{u\}]$	$\{\{u\}$	$\}$	1/1
	$L^1[\emptyset]$	$\{\{u\}$	$\}$	1/2
$\dagger$	$L^3[\{u,v\}]$	$\{\{u,v,y\},\{u,v,z\}$	$\}$	2/2
	$L^3[\{u,y\}]$	$\{\{u,y,v\},\{u,y,z\}$	$\{u,y,x\}\}$	3/2
	$L^3[\{u,z\}]$	$\{\{u,z,v\},\{u,z,r\}$	$\{u,z,y\}\}$	3/2
	$L^3[\{u\}]$	$\{\{u,y,v\},\{u,z,r\},\{u,v,z\}$	$\}$	3/4
	$L^3[\emptyset]$	$\{\{u,y,v\},\{u,z,r\},\{u,v,z\}$	$\}$	3/8
	$L^2[\{u\}]$	$\{\{u,y\},\{u,z\}$	$\}$	2/2
	$L^2[\emptyset]$	$\{\{u,y\},\{u,z\}$	$\}$	2/4
	$L^1[\{u\}]$	$\{$	$\}$	0/1
	$L^1[\emptyset]$	$\{$	$\}$	0/2

This concludes the example. ⌟

The method *fix needs downward* is more complex than strictly necessary, 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 $s \in E(L^i[s'])$ if, and only if, 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 $s \in E(L^i[s'])$ if, and only if, 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 4.7. *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. Clearly, to prove the lemma, we need to show that the code of the method ensures for all $s' \subseteq p$ that if s is needed in $L^i[s']$, it gets inserted. It is the job of line 15 to test whether such an insertion is necessary. The line tests whether $\forall v \in p - s' \colon s \in R(L^i[s' \cup \{v\}])$ holds. By Definition 4.3 of needed hyperedges, what we are *supposed* to test is whether for all $t \subseteq s$ with $s' \subsetneq t$ we have $s \in R(L^i[t])$. However, just 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 assuming we process the hyperedges s' in descending order of size (which we do, see line 13), s will be needed for $L^i[s']$ if, and only if, 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 4.8. *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 6, it also calls *fix needs*(s,s,i) right away. By Lemma 4.7, 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 6 and 16) we immediately call *fix forced*, which inserts s into $L^{i-1}[s]$ if s is forced. □

Deletions. The delete operation has to check and 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 e 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, making it the last relevant hyperedge. Then one has to fix the needs for this hyperedge once more, which may entail new inserts and forcings, but no new deletions. In detail:

```
1  method DynKernelHS.delete(e)
2    call delete(e, d)
3
4  function delete(s, i)
5    if L^i[s] contains s then // Sanity check
6      // Delete s and subsets of s if no longer forced
7      for s' ⊆ s do
8        L^i[s'].delete(s) // Delete e from all lists that could contain it
9        if not L^i[s'].has irrelevant elements then // Has s' now lost its forced status?
10         if |s'| < i then // Can it even be in L^{i-1}
11           call delete(s', i − 1)
12
13     // Restore Need Invariant for hyperedges that have suddenly become relevant
14     for s' ⊆ s do
15       f ← L^i[s'].last relevant
16       call fix needs downward(f, s', i) // (Only) the last relevant may have changed
```

Lemma 4.9. *The Need and Force Invariant are maintained by the delete method.*

Proof. Proving the Need and Force Invariants for the delete operation is trickier than for the insert operation since a delete can, internally, trigger insert operations – namely in line 16. For this reason, we prove by induction on i that the Need and Force Invariants still hold for all elements in all $L^j[s']$ for $j \leq i$, $s' \subset s$ after a call to $delete(s, i)$. For $i = 0$ this is trivial since the only possible s is $\emptyset$ and the loop only considers $s' = \emptyset$, deletes it from $L^0[\emptyset]$, and does nothing else.

For the inductive step, first consider the Need Invariant on s. The loop removes s from $L^i[s]$ and also from all $L^i[s']$ (the loop from line 7 executes a remove operation for each $s' \subseteq s$ in the next line). This ensures the Need Invariant on s. Next, observe that removing a hyperedge from a list $L^i[s']$ can only reduce the number of irrelevant hyperedge, meaning that s' can only change its status from forced to unforced. If this happens, as tested in line 9, we recursively remove s' from $L^{i-1}[s']$. By the induction hypothesis, this will maintain the Need and Force Invariants on all the L^j for $j < i$.

While we have now correctly accounted for the needed and the forced status of s and its subsets $s' \subseteq s$, the removal of an edge s from a list $L^i[s']$ can have a second side-effect, besides (possibly) unforcing s': It can also make a previously irrelevant hyperedge relevant. This happens when, firstly, s used to be a relevant hyperedge in $L^i[s']$ and, secondly, there was a (first) irrelevant hyperedge f in $L^i[s]$. In this case, the mechanics of relevance lists automatically change the relevance status of f from irrelevant to relevant. Note that at most one edge is deleted from $L^i[s']$ during a call of $delete(s, i)$, namely s, and hence at most one hyperedge f can become relevant per list $L^i[s']$. Note that more than one hyperedge can be deleted from the same list $L^{i-1}[s']$ by recursive calls during a single call of $delete(s, i)$ – but by the induction hypothesis the Need and Force Invariants are maintained by the calls $delete(s', i - 1)$.

When a hyperedge f becomes relevant in a list $L^i[s']$, this *may* change the need status of f in sets $s'' \subsetneq s$: Previously, we had $f \notin R(L^i[s'])$ and, hence, also $f \notin R(L^i[s''])$ for all $s'' \subsetneq s'$. Now, however, f might be needed in some of the lists $L^i[s'']$ "further down." To address this, we call *fix needs*(f, s', i) in line 16, which will ensure that the Need Invariant of f is fixed below s' (see Lemma 4.7) *provided* the Need Invariant did hold for f *above* s'. However this was the case: the very fact that $f \in E(L^i[s'])$ used to hold shows that f was already relevant and present everywhere above s' (otherwise, f would not have made it into $L^i[s']$). Since we do not know whether f was needed before, a sanity check is in order to prevent edges from being inserted multiple times.

Crucially, observe that both the Need Invariant and the Force Invariant now hold for *all* hyperedges whose relevance status may have changed, namely s (as shown earlier) and all f in line 15. No other hyperedges in L^i change their relevance status (and for L^j with $j < i$ the invariants hold by the inductive assumption). □

Kernel. As stated earlier, the dynamic kernel maintained by DynKernelHS is the set $K = \bigcup_{i=0}^{d} R(L^i[\emptyset])$. (As stated, K is given only indirectly via d linked lists, but one can do the same transformations as in the proof of Theorem 3.4 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 4.10. DynKernelHS *maintains the invariant* $|K| \leq k^d + k^{d-1} + \cdots + k + 1$.

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

Lemma 4.13 below shows the crucial property that the current K has a hitting set of size k if, and only if, the current hypergraph does. The proof hinges on two lemmas, which we prove first and which show that the lists we manage are flowers:

Lemma 4.11. DynKernelHS *maintains the invariant that for all* $i \in \{0, \ldots, d\}$ *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 6 we have $e = s$ and in line 16 we have $s \subsetneq e$ by line 13). 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 4.1) 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 4.3 tells us $e \in R(L^i[t])$. Thus, $\{\, e \in E(L^i[s]) \mid v \in e \,\} \subseteq R(L^i[s \cup \{v\}])$ 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 6. □

Lemma 4.12. DynKernelHS *maintains the invariant that for all* $X \in \binom{V}{\leq k}$ *and for all* $i \in \{1, \ldots, d\}$ *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 4.4, "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 4.11, $E(L^i[s])$ is a $k^{i-|s|-1}$-flower with core s. By Lemma 4.2, since $|E(L^i[s])| > k^{i-|s|} = k \cdot k^{i-|s|-1}$, we know that X hits s, as claimed. □

Lemma 4.13. DynKernelHS *maintains the invariant that* H *and* K *have the same size-*k *hitting sets.*

Proof. For the first direction, let X be a size-k hitting set of $H = (V, E)$. For $i = d$, $i = d-1$, $\ldots$, $i = 1$, and $i = 0$ we show inductively that all lists $L^i[s]$ for all $s \in \binom{V}{\leq i}$ only contain hyperedges that are hit by X. For $i = d$ the claim is trivial since the lists $L^d[s]$ contain only edges from E, all of which are hit by X by assumption. Now assume that the claim holds for i and consider any $s \in L^{i-1}[s']$ for some $s' \in \binom{V}{\leq i-1}$. By the Need Invariant, this can only happen if $s \in L^{i-1}[s]$ holds. By the Force Invariant, this means that s is forced by $L^i[s]$. By Lemma 4.12, this means that X hits s.

Since X hits all hyperedges in all lists, it also hits all hyperedges in the kernel, which is just a union of such lists.

For the second direction, let X be a size-k hitting set of K. Let $e \in \binom{V}{\leq d}$ be an arbitrary hyperedge (not necessarily in E). We show by induction on i that if $e \in E(L^i[e])$, then e gets hit by X. This will show that X hits all of H: The *insert*-method ensures that for all $e \in E$ we have $e \in E(L^d[e])$ and, hence, they all get hit by X.

The case $i = 0$ is trivial since we can only have $e \in L^0[e]$ for $e = \emptyset$ and $L^0[\emptyset]$ is part of the kernel K and all its elements get hit by assumption (actually, $\emptyset \in K$ means that the assumption that X hits the kernel is never satisfied; the implication is true anyway). Next, consider a larger i and a hyperedge $e \in E(L^i[e])$.

First assume that $e \in E(L^i[s]) - R(L^i[s])$ holds for some $s \subseteq e$. Then s is forced by $L^i[s]$ since it contains an irrelevant edge (namely e). By the Force Invariant, we know that $s \in L^{i-1}[s]$ holds and, by the inductive assumption, that X hits s. Since $s \subseteq e$, X hits e as claimed.

Second assume that $e \notin E(L^i[s]) - R(L^i[s])$ holds for all $s \subseteq e$. Suppose there is an $s \subseteq e$ with $e \notin E(L^i[s])$. Then there is also an s that is inclusion-maximal, meaning that for all $t \subseteq e$ with $s \subsetneq t$ we have $e \in E(L^i[t])$ and hence also $e \in R(L^i[t])$ since $e \notin E(L^i[t]) - R(L^i[t])$. However, by definition, this means that e is needed in $L^i[s]$ and, hence, $e \in E(L^i[s])$ contrary to the assumption. In particular, we now know that for $s = \emptyset$ we have $e \in E(L^i[s])$ and, thus, also $e \in R(L^i[s]) = R(L^i[\emptyset]) \subseteq K$. Since X hits all of K, it also hits e, as claimed. □

Run-Time Analysis. It remains to bound the run-times of the insert and delete operations. We show that they depend only on d, albeit exponentially, and do *not* depend on k nor on $|V|$.

Lemma 4.14. DynKernelHS.*insert*(e) *runs in time* $O^*(3^d)$.

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. This allows us to assume that for each $s \subsetneq e$ and $i < d$ at most one call of the form *insert*(s, i) is made (further calls are immediately suppressed).

The following notation will be convenient: 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 21), that is, the starred time bound 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, due to the above argument that subsequent calls for the same parameters can be suppressed).

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 $t^*_{\text{fix}}(|s'|, i) = O^*(2^{|s'|})$ since the run-time is clearly dominated by the loop in line 13, which iterates over all subsets s'' of s'. For each of these

$2^{|s'|}$ many sets, we run a test in line 15 that needs time $O(|s'|)$, yielding a total run-time of $t^*_{\text{fix}}(|s'|, i) = O(|s'|2^{|s'|}) = O^*(2^{|s'|})$.

For the unstarred version we account for the recursive calls by summing up over all possible such calls:

$$\begin{aligned} t_{\text{insert}}(|s|, i) &= t^*_{\text{insert}}(|s|, i) + \textstyle\sum_{s' \subsetneq s, j \in \{|s'|, \ldots, i-1\}} t^*_{\text{insert}}(|s'|, j) \\ &= t^*_{\text{insert}}(|s|, i) + \textstyle\sum_{c=0}^{|s|-1} \underbrace{\binom{|s|}{c}}_{\text{number of } s' \subseteq s \text{ with } |s'|=c} \sum_{j=c}^{i-1} t^*_{\text{insert}}(c, j) \end{aligned}$$

Pluggin in the bound $O^*(2^c)$ for $t^*_{\text{insert}}(c, j)$, we get that everything following the binomial can be bounded by $O^*((d-c)2^c) = O^*(2^c)$. 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 of the theorem.

For later reference, we also establish a bound on $t_{\text{fix}}(|s'|, i)$. The crucial observation is that *all recursive calls made inside fix needs downward*(s, s', i) *are insert*(s'', j) *with* $s'' \subseteq s'$ *and* $j < i$. In particular, the size of s is not relevant for the number of recursive calls, but the size of s' is. We get:

$$t_{\text{fix}}(|s'|, i) = t^*_{\text{fix}}(|s'|, i) + \textstyle\sum_{s'' \subsetneq s', j \in \{|s'|, \ldots, i-1\}} t^*_{\text{insert}}(|s''|, j).$$

This has the same solution as we had earlier (only now depending on s' rather than s), namely $t_{\text{fix}}(|s'|, i) = O^*(3^{|s'|})$. □

Lemma 4.15. DynKernelHS.*delete*(e) *runs in time* $O^*(5^d)$.

Proof. Similar to the analysis of the *insert* method, let $t_{\text{delete}}(|s|, i)$ denote the run-time needed by *delete*(s, i) and let $t^*_{\text{delete}}(|s|, i)$ the time to delete *excluding* the time needed by the recursive calls made to *delete*$(s', i-1)$ inside this method. In other words, we do not count the (huge) time actually needed in line 11, where a recursive call is made, and will once more later on account for this time by summing over all $t_{\text{delete}}(|s|, i)$; but $t^*_{\text{delete}}(|s|, i)$ *will* include the run-time needed for the second loop, starting at line 14, where we (possibly) fix the Need Invariant for many f (this loop does not involve any recursive calls to the *delete* method). Note that – as in the insertion case – if there are multiple calls of *delete*(s, i) for the same s and i, we only need to count one of them since all subsequent ones return immediately (and could be suppressed).

A call to *delete*(s, i) clearly spends at most time $O^*(2^{|s|})$ in the first loop (starting on line 7) if we ignore the recursive calls. For the second loop, we iterate over all $s' \subseteq s$ and for each of them we call *fix needs downwards*(f, s', i):

$$t^*_{\text{delete}}(|s|, i) = O^*(2^{|s|}) + \textstyle\sum_{s' \subseteq s} t_{\text{fix}}(|s'|, i).$$

With the bound of $O^*(3^{|s'|})$ established in the proof of Lemma 4.14 for $t_{\text{fix}}(|s'|, i)$, we can focus on bounding $\sum_{s' \subseteq s} 3^{|s'|} = \sum_{c=0}^{|s|} \binom{|s|}{c} 3^c$ and this is equal to $4^{|s|}$. Thus, $t^*_{\text{delete}}(|s|, i) = O^*(4^{|s|})$.

Finally, we can now bound the total run-time of the delete method by summing over all recursive calls:

$$t_{\text{delete}}(|s|, i) = t^*_{\text{delete}}(|s|, i) + \textstyle\sum_{s' \subseteq s, j \in \{|s'|, \ldots, i-1\}} t^*_{\text{delete}}(|s'|, j).$$

Plugging in $O^*(4^{|s'|})$ for $t^*_{\text{delete}}(|s'|, j)$ we get that the crucial sum is

$$\textstyle\sum_{s' \subseteq s} 4^{|s'|} = \sum_{c=0}^{|s|} \binom{|s|}{c} 4^c = 5^{|s|},$$

yielding $t_{\text{delete}}(|s|, i) = O^*(5^{|s|})$ as claimed. □

Putting it all together, we get:

Proof of Theorem 1.1. The claim follows from Lemmas 4.10, 4.13, 4.14, and 4.15. □

5 Dynamic Set Packing Kernels

A bit surprisingly, the dynamic kernel algorithm we developed in the previous section works, after a slight modification, also 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 4.2, 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 hyperedge 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 5.1. *Let F be a b-flower with core c in a d-hypergraph $H = (V, E)$ and $|F| > b \cdot d \cdot (k-1)$. 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 5.1 now essentially tells us that instead of considering the flower $E(L^i[s])$, it suffices to consider the core s. Thus, simply by replacing line 6 inside the *init* method as follows, we get a dynamic kernel algorithm for p$_k$-d-SET-PACKING:

6 $L^i[s] \leftarrow$ ***new*** RELEVANCE LIST$\big((d(k-1))^{i-|s|}\big)$ // Modified relevance bounds

Proof of Theorem 1.2. We have to show that there is an algorithm DYNAMICSPKERNEL that is a dynamic kernel algorithm for p$_k$-d-SET-PACKING with at most $\sum_{i=0}^{d}(d(k-1))^i$ hyperedges in the kernel, insertion time $O^*(3^d)$, and deletion time $O^*(5^d)$.

Clearly, the analysis of the kernel size and of the runtimes is identical to the hitting set case. We only need to show that an analogue of Lemma 4.13 holds, which stated that DYNAMICHSKERNEL maintains the invariant that H and K have the same size-k hitting sets. We now have to show for $H = (V, E)$:

Claim. DYNAMICHSKERNEL *maintains the invariant that E has a size-k packing if, and only if, K does.*

Proof. We start with an observation: For every hyperedge $e \in E$ there is a subset $s \subseteq e$ with $s \in K$. To see this, for a given e consider the smallest i such that there is an $s \subseteq e$ with $s \in E(L^i[s'])$ for some $s' \subseteq s$ (such an i, s, and s' must exist, since at least for $s = s' = e$ and $i = d$ we have the property $s \in E(L^i[s'])$). If we have $s \in R(L^i[\emptyset])$, we have $s \in K$ as claimed. Otherwise, there must be an inclusion-maximal $t \subseteq s'$ such that $s \in E(L^i[t]) - R(L^i[t])$ (as we have $s \notin R(L^i[\emptyset])$, but $s \in E(L^i[s'])$). But, then, t would be forced into $L^{i-1}[t]$ and hence $t \in E(L^{i-1}[t])$ would hold, violating the minimality of i.

We now prove the claim by proving two directions. The first direction is easy: Consider a packing P of E. By the above observations, for every $p \in P$ there is a set $s_p \in K$ with $s_p \subseteq p$. Then $\{s_p \mid p \in P\}$ is a packing of size k in K.

For the second direction, let P be a packing of K of size k. For a number $i \in \{0, \dots, d\}$ let $A_i = \bigcup_{s \in \binom{V}{\leq i}} E(L^i[s])$ be the set of all hyperedges "mentioned in $L^i[s]$ for some s" and let $B_i = \bigcup_{j=i}^{d} A_j$ be the "hyperedges mentioned in some $L^d[s]$, $L^{d-1}[s]$, …, $L^i[s]$ for some s." Observe that $K \subseteq B_0$ and that $E = A_d = B_d$ (since $e \in L^d[e]$ holds for all $e \in E$ and no edges $e \notin E$ make it into any $L^d[s]$). Since P is a packing of $K \subseteq B_0$, we know that B_0 has a size-k packing. We show by induction on i that all B_i have a size-k packing and, hence, in particular $B_d = E$ as claimed.

For the inductive step, let B_{i-1} have a size-k packing and let P be one of these with the minimum number of elements that do not already lie in B_i (that is, which lie only in A_{i-1}). If the number is zero, P is already a size-k packing of B_i; so let $p \in P - B_i$. By the force property, a hyperedge p can lie in A_{i-1} only because it was forced, that is, because $L^i[s]$ has an irrelevant hyperedge. This means that $E(L^i[s]) \subseteq B_i$ is a $(d(k-1))^{i-|s|-1}$-flower (by Lemma 4.11 where we clearly just have to replace k by $d(k-1)$). Since $|E(L^i[s])|$ is larger than the relevance bound of $(d(k-1))^{i-|s|}$, Lemma 5.1 tells us that there is a set $f \in E(L^i[s])$ such that $P - \{p\} \cup \{f\}$ is also a packing of B_i. Since $f \in B_i$, this violates the assumed minimality of P. Thus, P must already have been a size-k packing of B_i. □

□

6 Conclusion

We have introduced a fully dynamic algorithm that maintains a p_k-d-HITTING-SET kernel of size $O(k^d)$ in update time $O^*(5^d)$. Since p_k-d-HITTING-SET has no kernel of size $O(k^{d-\epsilon})$ unless coNP $\subseteq$ NP/poly [9], and since the currently best static algorithm requires time $O^*(2^d \cdot |E|)$ [20], we have essentially settled the dynamic complexity of computing hitting set kernels. It seems possible that the update time can be bounded even tighter with an amortized analysis.

The 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: 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 $\sum_{i=1}^{d} k^i$. A natural next step would be to turn this idea into a "real" dynamic approximation algorithm for HITTING-SET.

A Implementation Details of Data Structures

The dynamic kernel algorithms that we present in this paper internally employ different standard dynamic data structures like linked lists or small arrays that allow update operations in time $O(1)$. In the following, for completeness, we sketch how these basic data structures can be implemented so that all basic operations work in constant time.

We will often store and treat mathematical entities like edges or sequences as *objects* in the sense of object-oriented programming. As is customary, they are just blocks of memory storing the object's current *attributes* and the object can be referenced with a pointer to the start of the memory block. For an object X we write $X.attribute$ for the current value of an attribute.

By *arrays* we refer to the usual notion of arrays that store a value for each index number from an immutable domain $D = \{1, \dots, r\}$. We write $A[i]$ for the value stored at position $i \in D$, write $A[i] \leftarrow v$ to indicate that we store the value v (typically an object or a number) at the ith position in A, write $A[i] = \perp$ to indicate that nothing is stored at an address i, and write $new\textsc{array}(D)$ for the operation that allocates a new, empty array with domain D. For convenience, we also allow domains D that are not sets of numbers, but whose elements can easily be mapped to numbers. For instance, we would also allow the domain $D = \binom{V}{2}$ of undirected edges since we can easily map D to $\left\{1, \dots, \binom{n}{2}\right\}$. We can then write $G[e] \leftarrow v$ to store a value v for an edge $e = \{u, v\}$. Clearly, for hypergraphs this can be generalized to $D = \binom{V}{\leq d}$ for fixed constants d since this D, too, can easily be mapped to elements of $\left\{1, \dots, \sum_{i=0}^{d} \binom{n}{i}\right\}$. By storing arrays as tables of size $O(r)$, reading from and writing to an array can be done in time $O(1)$ for any reasonable machine model. Unfortunately, this model of storing values is not very memory-efficient when $A[i] = \perp$ holds for most i. In this case it is better to store A as a hash table. In practice, hash tables also allow us to read and write in time $O(1)$. For our purposes, we just assume in the following that in whatever way arrays are really implemented, reading and writing from arrays can be done in time $O(1)$.

Maps (also known as *associative arrays*) are similar to arrays, but may be indexed by *keys* k, which can be arbitrary objects, and not just by numbers from a small domain. We still write $M[k]$ for the value v stored at the key k (and $M[k] = \perp$ if nothing is stored) and write $M[k] \leftarrow v$ to indicate that we store the value v for the key k, possibly replacing any previous value stored for k. Implementing maps is normally much trickier than implementing arrays, but we will only need and use maps that store values for *a constant number of keys.* In this case, even if we implement accesses using just a linear search in a normal array, all reading and writing can be done in time $O(1)$.

We will use the standard data structure of *doubly-linked lists* a lot, which we will just refer to as *lists.* We consider lists L to be objects that store pointers to the first and last *cell* of the list. Each cell stores pointers to the next and the previous cell in the list plus a pointer to an object, called the *payload* of the cell. For a list L, we write $L.append(x)$ to indicate that a new cell c gets created with x as its payload and then c is added to the list at the end (and the last and, possibly, first cells stored in L are updated appropriately).

Quite less standard, when creating a cell c for a list L, we also store the cell c in x: We assume that x has an attribute *lists* that is a map and we execute $x.lists[L] \leftarrow c$. In other words, inside x, we store a back-pointer to the cell c. This allows us to perform the operation $L.delete(x)$ without being given the cell c: We first lookup the cell c that has x as its payload in the map $x.lists$ and can then easily remove the cell from the doubly-linked list in constant time. Storing back-pointers in objects allows us to remove elements in time $O(1)$ provided that (i) no element is added more than once to a list (this will always be the case) and (ii) each element is added only to a constant number of lists (this will also always be the case).

The final dynamic data structure that we will need in our algorithms is more specific to the needs of the present paper: *relevance lists.* These are normal lists with a parameter $\rho \in \mathbb{N}$ in which the first ρ elements are "more relevant" than later elements (with respect to the order of

the elements inside the list). Once a relevance list L for a bound ρ has been allocated by the call *new* RELEVANCE LIST(ρ), we wish the following to hold for its elements: If there are only ρ or less elements in L, all of them are relevant; but if there are more, all elements after the ρth element are *irrelevant.* The operations we wish to support (in time $O(1)$) in addition to the normal list operations *append* and *delete* are $L.is\ relevant(x)$, which should return whether x is one of the first ρ elements in L, and $L.last\ relevant$, which should return the last relevant element of the list, respectively. For convenience, we will also use $L.first\ irrelevant$, which is the successor of $L.last\ relevant$ (and thus $\perp$ if there are no irrelevant elements), and $L.has\ irrelevant\ elements$, which just checks whether $L.last\ relevant$ has a successor.

Note that it is not immediately clear how the two additional operations of relevance list can be implemented in time $O(1)$: The relevance status of an element can change when far-away elements get added or deleted. The following lemma shows how this can be achieved:

Lemma A.1. *A relevance list can be implemented such that the methods $L.is\ relevant(x)$ and $L.last\ relevant$ run in time $O(1)$.*

Proof. A relevance list object L stores the immutable bound ρ as an attribute. It also stores the length of the list using an additional attribute (just increment or decrement it as needed). To keep track of which elements x are relevant with respect to L and which one is the last of them, we use two kinds of "trackers": First, we store one bit of information in each element x as follows. In x we have, in addition to the map attribute *lists* mentioned earlier, another attribute *relevances*. It is also a map and we set $x.relevances[L] \leftarrow true$ for relevant x and set $x.relevances[L] \leftarrow false$ otherwise. Clearly, if we can keep these values up-to-date, we can implement $L.is\ relevant(x)$ simply as returning $x.relevances[L]$. Second, in L we store a pointer to the last relevant element in an attribute *last relevant*. Once more, if we can keep this pointer up-to-date, we can trivially access it in time $O(1)$.

To keep the introduced trackers up to date, first consider the operation $L.append(x)$: Before we insert x, we check whether the length of L is at most $\rho - 1$. If so, after x has been appended, it is flagged as relevant ($x.relevances[L] \leftarrow true$) and $L.last\ relevant \leftarrow x$; and otherwise it is flagged as irrelevant and $L.last\ relevant$ is not changed. Next, consider the operation $L.delete(x)$. If x is not relevant ($x.relevances[L] = false$), we can simply delete it. However, if x is relevant, deleting x will make the first irrelevant element (if it exists) relevant: before deleting x, if $L.last\ relevant$ has a successor s, we set $L.last\ relevant \leftarrow s$ and $s.relevances[L] \leftarrow true$. As a special case, if x happens to be the last relevant element and has no successor, set $L.last\ relevant$ to the predecessor of x.

Note that all operations needed to keep the trackers up-to-date can be implemented to run in time $O(1)$, yielding the claim. □

References

[1] F. N. Abu-Khzam. 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] J. Alman, M. Mnich, and V. Vassilevska Williams. Dynamic Parameterized Problems and Algorithms. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 41:1–41:16, 2017. doi:10.4230/LIPIcs.ICALP.2017.41.

[3] 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.

[4] J. F. Buss and J. Goldsmith. Nondeterminism Within P. *SIAM Journal on Computing*, 22(3):560–572, 1993. doi:10.1137/0222038.

[5] Y. Chen, J. Flum, and X. Huang. Slicewise 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.

[6] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer Berlin Heidelberg, 2015.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] P. Erdős and R.Rado. Intersection Theorems for Systems of Sets. *Journal of the London Mathematical Society*, 1(1):85–90, 1960.

[12] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006. doi:10.1007/3-540-29953-X.

[13] F. V. Fomin, D. Lokshtanov, S. Saurabh, and M. Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, 2018.

[14] 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.

[15] 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.

[16] 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, Copenhagen, Denmark, July 2-4, 2014*, pages 241–252, 2014. doi:10.1007/978-3-319-08404-6_21.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il