The Dynamic Complexity of Acyclic Hypergraph Homomorphisms

Nils Vortmeier\textsuperscript{1} and Ioannis Kokkinis\textsuperscript{2}

\textsuperscript{1} University of Zurich, Switzerland  
nils.vortmeier@uzh.ch
\textsuperscript{2} National Technical University of Athens, Greece, and  
University of the Aegean, Greece  
ikiokkinis@aegean.gr

Abstract. Finding a homomorphism from some hypergraph $Q$ (or some relational structure) to another hypergraph $D$ is a fundamental problem in computer science. We show that an answer to this problem can be maintained under single-edge changes of $Q$, as long as it stays acyclic, in the DynFO framework of Patnaik and Immerman that uses updates expressed in first-order logic. If additionally also changes of $D$ are allowed, we show that it is unlikely that existence of homomorphisms can be maintained in DynFO.

Keywords: Dynamic Complexity · Conjunctive Queries · Hypergraph Homomorphisms.

1 Introduction

Many important computational problems can be phrased as the question “is there a homomorphism from $Q$ to $D$?”, where $Q$ and $D$ are hypergraphs, or more generally, relational structures. Examples include evaluation and minimisation of conjunctive queries \cite{4} and solving constraint satisfaction problems, see \cite{10}.

The problem HOM – is there a homomorphism from $Q$ to $D$? – is \textsc{np}-complete in its general form. In the static setting it is well understood which restrictions on $Q$ or $D$ render the problem tractable \cite{1,4,16}. A particular restriction of great importance in databases is to demand that $Q$ is acyclic \cite{1}. This restriction of HOM, we call it the Acyclic Hypergraph Homomorphism problem AHH, can be solved in polynomial time by Yannakakis’ algorithm \cite{25} and is complete for the complexity class \textsc{logcfl} \cite{12}, the class of problems that can be reduced in logarithmic space to a context-free language.

We are interested in a dynamic setting where the input of a problem is subject to changes. The complexity-theoretic framework DynFO for such a dynamic setting was introduced by Patnaik and Immerman \cite{20} and it is closely related to a setting of Dong, Su and Topor \cite{8}. In this setting, a relational input structure is subject to a sequence of changes, which are usually insertions of single tuples into a relation, or deletions of single tuples from a relation. After each change, additionally stored auxiliary relations are updated as specified by first-order update
formulas. The class DynFO contains all problems for which the update formulas can maintain the answer for the changing input.

With few exceptions, for example in parts of [19], research in the DynFO framework takes a data complexity viewpoint: all context-free languages [11] and all problems definable in monadic second-order logic MSO [7] are in DynFO if the context-free language or the MSO-definable problem is fixed and not part of the input. Every fixed conjunctive query is trivially in DynFO, as such a query can be expressed in first-order logic and updates defined by first-order formulas can just compute the result from scratch after every change; however, there are also non-trivial maintenance results for fixed conjunctive queries for subclasses of DynFO [11,26]. The complexity results for Hom and AHH of [12,25] are however from a combined complexity perspective: both \( Q \) and \( D \) are part of the input.

**Contributions.** In this paper we study the combined complexity of AHH in the dynamic setting. As inputs we allow hypergraphs and general relational structures over some fixed schema \( \tau \).

As our main positive result, we show that AHH(\( \tau \)) is in DynFO for every schema \( \tau \), if \( Q \) is subject to insertions and deletions of hyperedges but stays acyclic, and \( D \) may initially be arbitrary but is not changed afterwards. A main building block for this result is a proof that a join tree for \( Q \) can be maintained in DynFO in such a way that after a single change to \( Q \) the maintained join tree only changes by a constant number of edges. We show that given a join tree for \( Q \) we can maintain the answer to AHH(\( \tau \)) under changes of single edges of the join tree. The main result follows by compositionality properties of DynFO.

We also give a hardness result for the case that also \( D \) is subject to changes. If AHH(\( \tau \)) is in DynFO for every schema \( \tau \) under changes of \( Q \) and \( D \), then all LOGCFL-problems are in (a variant of) DynFO, which we believe not to be the case. So, this result is a strong indicator that maintenance under changes of \( D \) is not possible in DynFO. Note that this result does not follow immediately from the fact that AHH is LOGCFL-complete: the NL-complete problem of reachability in directed graphs is in DynFO [6] as well as a PTIME-complete problem [20], and these results do not imply that all NL-problems and even all PTIME-problems are in DynFO, as this class is not known to be closed under the usual classes of reductions.

**Further related work.** In databases, Incremental View Maintenance is concerned with updating the result of a database query after a change of the input, see [15] for an overview. Koch [17] shows that a set of queries that include conjunctive queries can be maintained incrementally by low-complexity updates. A system for maintaining the result of Datalog-like queries under changes of the data and the queries is described in [13].

**Organisation.** We introduce preliminaries and the DynFO framework in Section 2. Section 3 contains the maintenance result for AHH under changes of \( Q \), the hardness result for changes of \( D \) is presented in Section 4. We conclude in Section 5. This paper accompanies [23] and contains more proof details.
2 Preliminaries and Setting

We introduce some concepts and notation that we need throughout the paper. See also [21] for an overview of Dynamic Complexity. We assume familiarity with first-order logic $\text{FO}$, and refer to [18] for basics of Finite Model Theory.

A (purely relational) schema $\tau$ consists of a finite set of relation symbols with a corresponding arity. A structure $D$ over schema $\tau$ with finite domain $D$ has, for every $k$-ary relation symbol $R \in \tau$, a relation $R^D \subseteq D^k$. We assume that all structures come with a linear order $\leq$ on their domain $D$, which allows us to identify $D$ with $\{1, \ldots, n\}$, for $n = |D|$. We also assume that first-order formulas have access to this linear order and to compatible relations $+$ and $\times$ encoding addition and multiplication on $\{1, \ldots, n\}$.

The dynamic complexity framework. In the dynamic complexity framework as introduced by Patnaik and Immerman [20], the goal of a dynamic program is to answer a standing query to an input structure $\mathcal{I}$ under changes. To do so, the program stores and updates an auxiliary structure $A$, which is over the same domain as $\mathcal{I}$. This structure consists of a set of auxiliary relations.

The set of admissible changes to the input structure is specified by a set $\Delta$ of change operations. We mostly consider the change operations $\text{INS}_R$ and $\text{DEL}_R$ for a relation $R$ of the input structure. A change $\delta(\bar{a})$ consists of a change operation $\delta \in \Delta$ and a tuple $\bar{a}$ over the domain of $\mathcal{I}$. The change $\text{INS}_R(\bar{a})$ inserts the tuple $\bar{a}$ into the relation $R$ and the change $\text{DEL}_R(\bar{a})$ deletes $\bar{a}$ from $R$.

For every change operation $\delta \in \Delta$ and every auxiliary relation $S$, a dynamic program has a first-order update rule that specifies how $S$ is updated after a change over $\delta$. Such a rule is of the form on change $\delta(\bar{p})$ update $S(\bar{x})$ as $\varphi^S_\delta(\bar{p}; \bar{x})$, where the update formula $\varphi^S_\delta$ is a first-order formula over the combined schema of $\mathcal{I}$ and $A$. After a change $\delta(\bar{a})$ is applied, the relation $S$ is updated to $\{\bar{b} \mid (\mathcal{I}, A) \models \varphi^S_\delta(\bar{a}; \bar{b})\}$.

We say that a dynamic program $P$ maintains a query $Q$ under changes from $\Delta$ if a dedicated auxiliary relation $\text{ANS}$ contains the answer to $Q$ for the current input structure after each sequence of changes over $\Delta$. The class $\text{DynFO}$ contains all queries that can be maintained by dynamic programs with first-order update rules, starting from initially empty input and auxiliary relations. We also say that $Q$ can be maintained in $\text{DynFO}$ under $\Delta$ changes.

In this paper we are interested in scenarios where only parts of the input are subject to changes. To have a meaningful setting we then have to allow non-empty initial input relations. We then say that a query can be maintained in $\text{DynFO}$ starting from non-empty inputs. Sometimes we then also allow the auxiliary relations to be initialised within some complexity bound. We say that a query $Q$ is in $\text{DynFO}$ with $\mathcal{C}$ initialisation, for a complexity class $\mathcal{C}$, if there is a $\mathcal{C}$-algorithm $A$ such that $Q$ can be maintained in $\text{DynFO}$ if for an initial input $\mathcal{I}_0$ the initial auxiliary relations are set to the result of $A$ applied to $\mathcal{I}_0$.

The reductions usually used in dynamic complexity are bounded first-order reductions [20]. A reduction $f$ is bounded if there is a global constant $c$ such that if the structure $\mathcal{D}'$ is obtained from the structure $\mathcal{D}$ by inserting or deleting
one tuple, then \( f(D') \) can be obtained from \( f(D) \) by inserting and/or deleting at most \( c \) tuples. We will not directly employ these reductions here, but we will use the simple proof idea to show that \( \text{DynFO} \) is closed under these reductions (see [20]): if a query \( Q \) can be maintained by a dynamic program \( P \) under insertions and deletions of single tuples, then there is also a dynamic program that can maintain \( Q \) under insertions and deletions of up to \( c \) tuples, for any constant \( c \). That dynamic program can be obtained by nesting \( c \) copies of the update formulas of \( P \).

**Hypergraphs and Homomorphisms.** We use the term **hypergraph** in a very broad sense. For this paper, a hypergraph \( H \) is just a relational structure over a purely relational schema \( \tau = \{ E_1, \ldots, E_m \} \), that is, a structure \( H = (V, E_1, \ldots, E_m) \), where the domain \( V \) is a set of nodes and the relations \( E_1, \ldots, E_m \) are sets of (labelled) hyperedges. This definition implies that the maximal size of any hyperedge, that is, the maximal arity of a relation \( E_i \), is a constant that only depends on \( \tau \). Sometimes we ignore the labels and denote \( H \) as a tuple \( (V, E) \), where \( E = E_1 \cup \cdots \cup E_m \) is the set of all hyperedges.

A **spanning forest** of an undirected graph \( G = (V, E) \) is defined in the usual way. We encode a spanning forest as a structure \( (V, F, P) \) where \( F \) is the set of spanning edges and \( P \) is a ternary relation that describes paths in the spanning forest. A tuple \((s, t, u) \in P\) indicates that \( s \) and \( t \) are in the same connected component of the spanning forest and \( (2) \) the unique path from \( s \) to \( t \) in the spanning forest is via the node \( u \). Patnaik and Immerman [20] have shown that spanning forests with this encoding can be maintained in \( \text{DynFO} \) under insertions and deletions of single edges [20, Theorem 4.1].

A **join forest** \( J(H) \) of a hypergraph \( H = (V, E_1, \ldots, E_m) \) is a forest whose nodes are the hyperedges of \( H \), such that if two hyperedges \( e, e' \) have a node \( v \in V \) in common, then they are in the same connected component of \( J(H) \) and all nodes on the unique path from \( e \) to \( e' \) in \( J(H) \) are hyperedges of \( H \) that also include \( v \). We encode a join forest using relations \( F_{ij} \) and \( P_{ijk} \) with the same intended meaning as for spanning forests, where \( i, j, k \in \{1, \ldots, m\} \). The arity of \( F_{ij} \) is the sum of the arities of \( E_i \) and \( E_j \), a tuple \((e, e') \in F_{ij}\) indicates that \( J(H) \) has an edge between the hyperedges \( e \in E_i \) and \( e' \in E_j \). The use of \( P_{ijk} \) is analogous.

We define that a hypergraph is **acyclic** if it has a join forest. This definition coincides with the notion of \( \alpha \)-acyclicity introduced by Fagin [9]. See also [12, Section 2.2] for a detailed discussion of this notion.

A **homomorphism** from a hypergraph \( H = (V, E_1^H, \ldots, E_m^H) \) to a hypergraph \( H' = (V', E_1^{H'}, \ldots, E_m^{H'}) \) is a map \( h : V \rightarrow V' \) that preserves the hyperedge relations. So, for all relations \( E_i \) and all tuples \((v_1, \ldots, v_\ell) \) over \( V \), where \( \ell \) is the arity of \( E_i \), if \((v_1, \ldots, v_\ell) \in E_i^H \) is a hyperedge of \( H \), then \((h(v_1), \ldots, h(v_\ell)) \in E_i^{H'} \) is a hyperedge of \( H' \).

The main problem we study is the Acyclic Hypergraph Homomorphism problem \( \text{AHH}(\tau) \), where \( \tau \) is a fixed schema. It asks, for two given hypergraphs \( Q \) and \( D \) over schema \( \tau \) (where \( Q \) is acyclic), also called **query hypergraph** and **data hypergraph** respectively, whether there is a homomorphism from \( Q \) to \( D \).
3 Maintenance under Changes of the Query Hypergraph

The goal of this section is to show that AHH can be maintained under changes of the query hypergraph $Q$, as long as it stays acyclic. We also show that a DynFO program can recognise that a change would make $Q$ cyclic. So, we do not need to assume that only changes occur that preserve acyclicity, if we allow the program to “deny” all other changes.

We introduce some notation of [12]. The weighted hyperedge graph $wg(H)$ of a hypergraph $H$ is the undirected weighted graph $wg(H) = (V, E, w)$ whose nodes $V$ are the hyperedges of $H$ and the set $E$ contains an undirected edge $(e, e')$ if $e, e'$ are different hyperedges of $H$ that have at least one node in common. The weight $w((e, e'))$ of such an edge is the number of nodes that $e$ and $e'$ have in common.

The weight $w(H)$ of a hypergraph $H$ is the sum over the degrees of the non-isolated nodes of $H$, where each degree is decremented by one. So, if for $H = (V, E_1, \ldots, E_m)$ the set $V_{ni} \subseteq V$ contains all nodes of $H$ that appear in at least one hyperedge, then $w(H) = \sum_{v \in V_{ni}}(\deg(v) - 1)$.

The following lemma provides the basis for our approach. It was originally proven in [2], we follow the presentation of [12, Proposition 3.5].

**Lemma 1** ([2], see also [12]). Let $H$ be a hypergraph.

(a) The hypergraph $H$ is acyclic if and only if the weight $w(H)$ of $H$ is equal to the weight $w(msf(wg(H)))$ of a maximal-weight spanning forest of $wg(H)$.

(b) If $H$ is acyclic, then $msf(wg(H))$ is a join forest of $H$.

Using this lemma, we prove that a dynamic program can maintain acyclicity of hypergraphs, as well as a join forest that only changes moderately when the input hypergraph is changed.

**Theorem 2.** Let $\tau = \{E_1, \ldots, E_m\}$ be a fixed schema. The following can be maintained in DynFO under insertions and deletions of single hyperedges:

(a) whether a hypergraph over $\tau$ is acyclic, and

(b) a join forest for an acyclic hypergraph $H$ over $\tau$, as long as $H$ stays acyclic. Moreover, there is a global constant $c_\tau$ such that if $J(H)$ is the maintained join forest for $H$ and $J(H)'$ is the maintained join forest after a single hyperedge is inserted or deleted, then $J(H)$ and $J(H)'$ differ by at most $c_\tau$ edges.

The proof follows the idea that is brought forth by Lemma 1. We show that a maximal-weight spanning forest of $wg(H)$ and its weight can be maintained. This weight is compared with the weight of $H$, which is easy to maintain. If the weights are equal, then $H$ is acyclic and the spanning forest is a join forest.

Already Patnaik and Immerman [20] describe how a spanning forest of an undirected graph can be maintained under changes of single edges, and their procedure [20, Theorem 4.1] can easily be extended towards maximal-weight
spanning forests. However, we face the problem that inserting and deleting hyperedges of $\mathcal{H}$ implies insertions and deletions of nodes of $\text{WG}(\mathcal{H})$. While a spanning forest can easily be maintained in $\text{DynFO}$ under node insertions, it is an open problem to maintain a spanning forest under node deletions: if the spanning forest is a star and its center node is deleted, then it seems that a spanning forest of the remaining graph needs to be defined from scratch, which is not possible using $\text{FO}$ formulas. We circumvent this problem and show that we can maintain a spanning forest where the degree of every node is bounded by a constant.

**Proof.** We show how a maximal-weight spanning forest of $\text{WG}(\mathcal{H})$ and the weight $\text{w}(\mathcal{H})$ can be maintained; the result then follows using Lemma 1.

We start with the weight $\text{w}(\mathcal{H})$. If a hyperedge $e$ is inserted, then the weight of the hypergraph increases by the number of nodes it contains that were not isolated before the insertion. Similarly, if $e$ is deleted, then the weight decreases by the number of nodes it contains that do not become isolated. This update can easily be expressed by first-order formulas.

Now we consider maintaining a maximal-weight spanning forest of $\text{WG}(\mathcal{H})$.

Let $a_{\text{max}}$ be the maximal arity of a relation in $\tau$. Any hyperedge of $\mathcal{H} = (V, E_1, \ldots, E_m)$ can only include at most $a_{\text{max}}$ many nodes and there are at most $r \equiv 2^{a_{\text{max}}} - 1$ many different non-empty sets of nodes that a fixed hyperedge can have in common with any other hyperedge. We show how to maintain a maximal-weight spanning forest of $\text{WG}(\mathcal{H})$ where each node has degree at most $2r$. More specifically, for any node $e$ of $\text{WG}(\mathcal{H})$ (which is a hyperedge of $\mathcal{H}$) and each non-empty set $A$ of nodes appearing in $e$, the maintained spanning forest contains at most two edges $(e, e_1)$, $(e, e_2)$ such that the set of nodes that $e$ has in common with $e_1$ and $e_2$, respectively, is exactly $A$. We call this property Invariant ($\star$).

We assume that our auxiliary relations contain a maximal-weight spanning forest $\text{S}(\mathcal{H})$ and its weight, and that $\text{S}(\mathcal{H})$ satisfies Invariant ($\star$). This is trivially satisfied by an empty spanning forest for an initially empty hypergraph. We show how the invariant can be satisfied again after a change.

In the following, we say that $e'$ is an $A$-neighbour of $e$ if these hyperedges have exactly the nodes $A$ in common. The number of $A$-neighbours of $e$ (that $e$ has an edge to in $\text{S}(\mathcal{H})$) is called its $A$-degree (with respect to $\text{S}(\mathcal{H})$).

**Insertion of a hyperedge $e$.** Suppose a hyperedge $e$ is inserted into the hypergraph $\mathcal{H}$, resulting in the hypergraph $\mathcal{H}'$. Let $B$ be the set of nodes that occur in $e$. For each non-empty $A \subseteq B$, let $E_A$ be the edges of $e$ in $\text{WG}(\mathcal{H}')$ to its $A$-neighbours. We adapt $\text{S}(\mathcal{H})$ in stages, one stage per subset $A$, and in each stage the spanning forest is changed by at most two edges. As the number of stages is bounded by the constant $r$, the maintained spanning forests before and after the update only differ by a constant number of edges.

Let $A_1, \ldots, A_\ell$ be a sequence of all non-empty subsets of $B$, partially ordered by their size, starting with the largest. So, $A_1 = B$. Stage $i$ for an arbitrary $1 \leq i \leq \ell$ works as follows. Suppose that $S_{i-1}$ is a maximal spanning forest of the graph that results from $\text{WG}(\mathcal{H})$ by adding the node $e$ and the edge set
\[ \bigcup_{j \leq i-1} E_{A_j} \] and that satisfies Invariant (⋆) (so \( S_0 \) is a maximum spanning forest for \( \text{WG}(\mathcal{H}) \cup \{e\} \emptyset \) and therefore also for \( \text{WG}(\mathcal{H}) \)). Let \( N \) be the \( A_i \)-neighbours of \( e \). The hyperedges in \( N \) form a clique in \( \text{WG}(\mathcal{H}) \), as they all have at least the nodes \( A_i \) in common, so they are in the same connected component \( C \) of \( S_{i-1} \).

We consider two cases.

First, if \( e \) is not in \( C \), then let \( e' \in N \) be some hyperedge that has \( A_i \)-degree at most 1 with respect to \( S_{i-1} \). Such an \( e' \) needs to exist as \( S_{i-1} \) is a forest. Then \( S_i \equiv S_{i-1} \cup \{(e,e')\} \) clearly is a spanning forest of \( \text{WG}(\mathcal{H}) \cup \bigcup_{j \leq i} E_{A_j} \). It is maximal, as replacing some spanning edge \((e_1,e_2)\) by another edge from \( e \) to an \( A_i \)-neighbour cannot increase the weight: if this would be the case for some edge \((e,e'')\), then \( S_{i-1} \) cannot be maximal, because the edge \((e',e'')\) has at least the same weight as the edge \((e,e'')\) and replacing \((e_1,e_2)\) by \((e',e'')\) would therefore create a spanning forest with larger weight than \( S_{i-1} \). Invariant (⋆) is also satisfied by \( S_i \).

Second, if \( e \) is in \( C \), then let \((e_1,e_2)\) be the minimal-weight edge in \( S_{i-1} \) on the path from \( e \) to any hyperedge in \( N \). If the weight of this edge is at least \(|A_i|\), then \( S_i \equiv S_{i-1} \), as the weight of the spanning forest cannot be increased by incorporating an edge from \( E_{A_i} \). Otherwise, \( S_i \) results from \( S_{i-1} \) by removing the edge \((e_1,e_2)\) and adding an edge from \( e \) to one of its \( A_i \)-neighbours with \( A_i \)-degree at most 1 with respect to \( S_{i-1} \). With the same arguments as in the other case, \( S_i \) is a maximal-weight spanning forest of \( \text{WG}(\mathcal{H}) \cup \bigcup_{j \leq i} E_{A_j} \), it also satisfies Invariant (⋆).

The updates of the spanning forest relation \( F_{ij} \) can be expressed by first-order formulas using the relations \( P_{ijk} \). These relations can be updated as in the proof of \cite{20} Theorem 4.1, as the relations \( F_{ij} \) change only by a constant number of tuples. The weight of the spanning forest can also be updated easily.

**Deletion of a hyperedge** \( e \). Suppose \( e \) is deleted from \( \mathcal{H} \), resulting in the hypergraph \( \mathcal{H'} \). As the maintained spanning forest \( S(\mathcal{H}) \) satisfies Invariant (⋆), the degree of \( e \) in \( S(\mathcal{H}) \) is bounded by the constant \( r \). Therefore, the procedure of \cite{20} Theorem 4.1] only needs to be applied for a constant number of edge deletions. If by a deletion of a spanning tree edge a connected component of \( S(\mathcal{H}) \) decomposes into two components \( C_1 \) and \( C_2 \), then we need to ensure that a potentially selected replacement edge results in a spanning forest that satisfies Invariant (⋆) again. So, suppose that the components \( C_1 \) and \( C_2 \) are connected in \( \text{WG}(\mathcal{H}') \), and let \((e_1,e_2)\) be a maximal-weight edge that connects them. Let \( A \) be the set of nodes that \( e_1 \) and \( e_2 \) have in common. Without loss of generality, we suppose that \( e_1 \) and \( e_2 \) have \( A \)-degree at most 1 with respect to \( S(\mathcal{H}) \). If this is not the case for example for \( e_1 \), then there needs to be a \( A \)-neighbour \( e'_1 \) in \( C_1 \) that has \( A \)-degree at most 1 with respect to \( S(\mathcal{H}) \), which then can be used instead of \( e_1 \). Adding \((e_1,e_2)\) to the remaining spanning forest will therefore result in a maximal-weight spanning forest for the changed hypergraph which satisfies Invariant (⋆). Also, the maintained spanning forest differs only by at most \( 2r \) edges from its previous version. The weight of the spanning forest can easily be updated as well. \( \square \)
We now present the main maintenance result of this paper.

**Theorem 3.** Let \( \tau = \{ E_1, \ldots, E_m \} \) be a fixed schema. The problem \( \text{AHH}(\tau) \) can be maintained in \( \text{DynFO} \), starting from an arbitrary initial hypergraph \( \mathcal{D} \) and an initially empty hypergraph \( \mathcal{Q} \), under insertions and deletions of single hyperedges of \( \mathcal{Q} \), as long as this hypergraph stays acyclic.

The proof uses the idea of Yannakakis' algorithm \( [20] \) for evaluating a conjunctive query. This algorithm processes a join tree for a query \( \mathcal{Q} \) in a bottom-up fashion. In a first step, for each node \( \mathcal{E}_1(\bar{x}) \) of the join tree (which is a hyperedge of \( \mathcal{Q} \)) all assignments \( \bar{y} \) for its variables are stored such that \( \mathcal{E}_i(\bar{y}) \) exists in the data hypergraph \( \mathcal{D} \). Then, bottom-up, each inner node removes all of its variable assignments that are not consistent with the assignments of its children. So, an assignment \( \bar{y} \) for a node \( \mathcal{E}_1(\bar{x}) \) is removed if there is a child \( \mathcal{E}_j(\bar{x}') \) of \( \mathcal{E}_1(\bar{x}) \) such that no stored assignment \( \bar{y}' \) of that child agrees with \( \bar{y} \) on the common variables \( \bar{x} \) and \( \bar{x}' \).

All remaining stored assignments for \( \mathcal{E}_i(\bar{x}) \) can be extended to a homomorphism for the subhypergraph of \( \mathcal{Q} \) that consists of the hyperedges that are in the subtree of the join tree rooted at \( \mathcal{E}_i(\bar{x}) \). A homomorphism from \( \mathcal{Q} \) to \( \mathcal{D} \) exists if after the join tree is processed the root has remaining assignments.

**Proof.** Let \( \mathcal{Q} \) be an acyclic hypergraph over some schema \( \tau \) and let \( \mathcal{D} \) be a hypergraph over the same schema. Also, let \( J(\mathcal{Q}) \) be a join forest of \( \mathcal{Q} \).

We adapt a technique that was used by Gelade, Marquardt and Schwentick \( [11] \) to show that regular tree languages can be maintained in a subclass of \( \text{DynFO} \). For each triple \( \mathcal{E}_i, \mathcal{E}_j, \mathcal{E}_k \) of symbols from \( \tau \) we maintain an auxiliary relation \( H_{ijk}(\bar{r}, \bar{x}_1, \bar{x}_2, \bar{y}_1, \bar{y}_2) \) with the following intended meaning. A tuple \( (\bar{r}, \bar{x}_1, \bar{x}_2, \bar{y}_1, \bar{y}_2) \) is in \( H_{ijk} \) if

1. the hyperedges \( \mathcal{E}_i(\bar{r}), \mathcal{E}_j(\bar{x}_1) \) and \( \mathcal{E}_k(\bar{x}_2) \) are present in \( \mathcal{Q} \) and in the same connected component \( C \) of \( J(\mathcal{Q}) \),
2. when we consider \( \mathcal{E}_i(\bar{r}) \) to be the root of \( C \) then \( \mathcal{E}_j(\bar{x}_1) \) is a descendant of \( \mathcal{E}_i(\bar{r}) \) and \( \mathcal{E}_k(\bar{x}_2) \) is a descendant of \( \mathcal{E}_j(\bar{x}_1) \), and
3. if we assume that there is a homomorphism \( h_2 \) of the subtree of \( C \) rooted at \( \mathcal{E}_k(\bar{x}_2) \) into \( \mathcal{D} \) such that \( h_2(\bar{x}_2) = \bar{y}_2 \), then it follows that there also is a homomorphism \( h_1 \) of the subtree of \( C \) rooted at \( \mathcal{E}_j(\bar{x}_1) \) into \( \mathcal{D} \) such that \( h_1(\bar{x}_1) = \bar{y}_1 \).

Phrased differently, \( (\bar{r}, \bar{x}_1, \bar{x}_2, \bar{y}_1, \bar{y}_2) \in H_{ijk} \) means that the hyperedges in \( J(\mathcal{Q}) \) which, considering \( \mathcal{E}_i(\bar{r}) \) to be the root, are in the subtree of \( \mathcal{E}_j(\bar{x}_1) \) but not in the subtree of \( \mathcal{E}_k(\bar{x}_2) \), can be mapped into \( \mathcal{D} \) by a homomorphism that maps the elements \( \bar{x}_1 \) to \( \bar{y}_1 \) and the elements \( \bar{x}_2 \) to \( \bar{y}_2 \).

If \( (\bar{r}, \bar{x}_1, \bar{x}_2, \bar{y}_1, \bar{y}_2) \in H_{ijk} \) holds we say that \( \bar{y}_1 \) is a valid partial assignment for \( \mathcal{E}_j(\bar{x}_1) \) down to \( \mathcal{E}_k(\bar{x}_2) \).

Notice that from these relations one can first-order define relations \( H'_{ij}(\bar{r}, \bar{x}, \bar{y}) \) with the intended meaning that \( (\bar{r}, \bar{x}, \bar{y}) \in H'_{ij} \) if

1. the hyperedges \( \mathcal{E}_i(\bar{r}) \) and \( \mathcal{E}_j(\bar{x}) \) are in the same connected component \( C \) of \( J(\mathcal{Q}) \), and
(2) when we consider $E_i(\bar t)$ to be the root of $C$ then there is a homomorphism $h$ of the subtree of $C$ rooted at $E_j(\bar x)$ into $D$ such that $h(\bar x) = \bar y$.

For this, a first-order formula existentially quantifies a hyperedge $E_k(\bar x_2)$ and a tuple $\bar y_2$ of elements, and checks that $E_k(\bar x_2)$ is a leaf of the component $C$ with root $E_i(\bar t)$, that the hyperedge $E_k(\bar y_2)$ exists in $D$ and that $(\bar t, \bar x, \bar x_2, \bar y, \bar y_2) \in H_{ijk}$ holds. Whether a node is a leaf in a join tree can be expressed using the join tree’s paths relations $P_{ijk}$, all other conditions are clearly first-order expressible. We assume in the following that these relations are available. If $(\bar t, \bar x, \bar y) \in H'_{ij}$ holds we say that $\bar y$ is a valid partial assignment for $E_j(\bar x)$.

We argue next that if we can maintain these auxiliary relations under insertions and deletions of single edges of the join forest, then the statement of the theorem follows.

Notice that from the auxiliary relations a first-order formula can express whether a homomorphism from $Q$ to $D$ exists. To check this, a formula needs to express that for every connected component of $J(Q)$ there is a homomorphism from this component to $D$. This is the case if for each hyperedge $E_i(\bar t)$ of $Q$ there is a tuple $\bar y$ such that $(\bar t, \bar t, \bar y)$ is in $H'_{ii}$.

It remains to argue that it is sufficient to maintain the auxiliary relations under changes of single edges of the join forest. From Theorem 2 we know that a join forest $J(Q)$ for $Q$ can be maintained in DynFO under insertions and deletions of single hyperedges, as long as it stays acyclic. Moreover, after each edge change, the maintained join forest only differs in a constant number of edges from its previous version. If we have a dynamic program that is able to process single edge changes of the join forest, then by nesting its update formulas $c$ times we can obtain a dynamic program $P'$ that is able to process $c$ edge changes at once.

In summary, a dynamic program $P$ for AHH maintains a join forest as described by Theorem 2 and after every change of a hyperedge it uses $P'$ to update the auxiliary relations and to decide whether a homomorphism exists.

Now we explain how the relations $H_{ijk}$ can be maintained by first-order formulas under insertions and deletions of single edges of the join forest. For notational simplicity we assume that the schema $\tau$ of $Q$ consists of a single relation $E$. It follows that we only have one auxiliary relation $H$ that needs to be maintained.

**Edge insertions.** When an edge $(e_1, e_2)$ is inserted into the join forest, the two connected components $C_1$ of $E(e_1)$ and $C_2$ of $E(e_2)$ get connected. The auxiliary relations for all other connected components remain unchanged. We explain under which conditions a tuple $\bar t = (\bar t, \bar x_1, \bar x_2, \bar y_1, \bar y_2)$ is contained in the updated version of $H$, where we assume that $E(\bar t)$ is from $C_1$. For hyperedges from $C_2$ the reasoning is symmetric. We assume that $E(\bar x_1)$ is a descendant of $E(\bar t)$ and $E(\bar x_2)$ is a descendant of $E(\bar x_1)$ in the combined connected component rooted at $E(\bar t)$; if this is not the case, $\bar t$ is not in the updated version of $H$.

We distinguish three cases. First, assume that $E(\bar x_1)$ and $E(\bar x_2)$ are in $C_1$. If $E(e_1)$ is not in the subtree of $E(\bar x_1)$ or is in the subtree of $E(\bar x_2)$, then no change regarding $\bar t \in H$ is necessary.
Otherwise, let $E(\bar{x}_{\text{LCA}})$ be the lowest common ancestor of $E(\bar{x}_2)$ and $E(e_1)$ in the join tree with root $E(\bar{r})$ and let $E(\bar{x}_1), \ldots, E(\bar{x}_m)$ be the children of $E(\bar{x}_{\text{LCA}})$, where $E(\bar{x}_1)$ is the ancestor of $E(\bar{x}_2)$ and $E(\bar{x}_m)$ is the ancestor of $E(e_1)$. With the help of the old version of $H$ first-order formulas can determine the valid partial assignments for all children $E(\bar{x}_i)$ for $i \geq 2$ and the valid partial assignments for $E(\bar{x}_1)$ down to $(E(\bar{x}_2), \bar{y}_2)$. This is immediate for all $E(\bar{x}_i)$ with $i < m$, we now explain it for $E(\bar{x}_m)$.

To check whether a tuple $\bar{y}_m$ is a valid partial assignment for $E(\bar{x}_m)$, a first-order formula can first determine the valid partial assignments for $E(e_2)$ for the component $C_2$ with root $E(e_2)$, which are given by $H'$. With this information it can check which valid partial assignments for $E(e_1)$ for the component $C_1$ with root $E(\bar{r})$ are still valid for the union of $C_1$ and $C_2$ with root $E(\bar{r})$. To do so, it checks for a (formerly) valid assignment $E(e_1)$ whether there is a valid assignment for $E(e_2)$ such that they agree on the shared elements. The tuple $\bar{y}_m$ is a valid partial assignment for $E(\bar{x}_m)$ if it is a valid partial assignment for $E(\bar{x}_m)$ down to $(E(e_1), \bar{y}_e)$, for some valid partial assignment $\bar{y}_e$ for $E(e_1)$.

With the information on the children, a first-order formula can determine the valid partial assignments for $E(\bar{x}_{\text{LCA}})$ down to $(E(\bar{x}_2), \bar{y}_2)$.

The tuple $\bar{t}$ is in the updated version of $H$ if and only if $(\bar{r}, \bar{x}_1, \bar{x}_{\text{LCA}}, \bar{y}_1, \bar{y}_{\text{LCA}})$ is in the old version of $H$, for some valid partial assignment $\bar{y}_{\text{LCA}}$ of $E(\bar{x}_{\text{LCA}})$ down to $(E(\bar{x}_2), \bar{y}_2)$.

As a second case, assume that $E(\bar{x}_1)$ is in $C_1$ and $E(\bar{x}_2)$ is in $C_2$. This case is very similar to the case we just discussed and we do not spell out the details.

As a last case, assume that $E(\bar{x}_1)$ and $E(\bar{x}_2)$ are both in $C_2$. This case is very simple, as $t \in H$ holds after the update precisely if $(\bar{e}_2, \bar{x}_1, \bar{x}_2, \bar{y}_1, \bar{y}_2) \in H$ holds before the update.

In all cases, the stated conditions can be expressed by first-order formulas. This is because the schema $\tau$ is fixed and therefore the arity of $E$ is constant, it follows that formulas can quantify over hyperedges and assignments. Also, formulas can determine whether a node is in a subtree of another node and the lowest common ancestor of two nodes using the paths relation $P_{ijk}$ of the join forest.

Edge deletions. When an edge $(e_1, e_2)$ is deleted from the join forest, one connected component is split into the two connected components $C_1$ of $E(e_1)$ and $C_2$ of $E(e_2)$. Again, the auxiliary relations for all other connected components remain unchanged. As for the insertion case, we explain under which conditions a tuple $\bar{t} = (\bar{r}, \bar{x}_1, \bar{x}_2, \bar{y}_1, \bar{y}_2)$ is contained in the updated version of $H$, for a root $E(\bar{r})$ from $C_1$. We assume that $E(\bar{x}_1)$ is a descendant of $E(\bar{r})$ and $E(\bar{x}_2)$ is a descendant of $E(\bar{x}_1)$ in the component $C_1$ rooted at $E(\bar{r})$; otherwise, $\bar{t}$ is not in the updated version of $H$.

If $E(e_1)$ is not in the subtree of $E(\bar{x}_1)$ or is in the subtree of $E(\bar{x}_2)$, then no change regarding $\bar{t} \in H$ is necessary. Otherwise the update is performed very
similarly to the corresponding insertion case detailed above. The only difference is the way the valid partial assignments for $E(e_1)$ are determined. Notice that a first-order formula can determine the valid partial assignments for all (remaining) children of $E(e_1)$, as they are given by the relation $H'$. A tuple $\bar{y}_{e_1}$ is a valid partial assignment for $E(e_1)$ if the hyperedge $E(\bar{y}_{e_1})$ exists in $D$ and if all children have a valid partial assignment that agrees with $\bar{y}_{e_1}$ on the shared elements.

4 Hardness under Changes of the Data Hypergraph

We have seen in the previous section that one can maintain the existence of homomorphisms in $\text{DynFO}$ if only the query hypergraph $Q$ may change and the data hypergraph $D$ remains the same. The dynamic program we constructed for the proof of Theorem 3 can not directly cope with changes of $D$. This is because $Q$ might contain several hyperedges $E_i(\bar{x}_1), \ldots, E_i(\bar{x}_m)$ over a single relation $E_i$: if a change of $D$ occurs, then the number of nodes in the join tree for which we have to take this change into account when updating partial valid assignments is a priori unbounded. If we disallow multiple hyperedges over the same relation in $Q$, then we can actually allow a change to replace an arbitrary number of $D$-hyperedges, as long as each change only affects a single relation of $D$. Such a restriction of $Q$ translates to self-join free acyclic conjunctive queries.

Corollary 4. Let $\tau = \{E_1, \ldots, E_m\}$ be a fixed schema. As long as $Q$ remains acyclic and contains at most one hyperedge $E_i(\bar{x})$ for each relation $E_i \in \tau$, the problem $\text{AHH}(\tau)$ can be maintained in $\text{DynFO}$ under insertions and deletions of single hyperedges of $Q$ and under arbitrary changes of a single relation of $D$.

Proof sketch. To adapt the proof of Theorem 3 it suffices to show how after changing some relation $E_i$ of $D$ one can determine the valid partial assignments for the single node $E_i(\bar{x})$ in the join forest $J(Q)$. As the valid partial assignments for its children did not change, this only involves to check for each tuple $\bar{y}$ such that the hyperedge $E_i(\bar{y})$ exists in $D$ whether each child in $J(Q)$ has a valid partial assignment that agrees with $\bar{y}$ on all elements it has in common with $\bar{x}$.

In the remainder of this section, we will see that if $Q$ might be an arbitrary acyclic hypergraph, then a maintenance result for $\text{AHH}(\tau)$ under changes of $D$ is unlikely, even if in turn $Q$ is not allowed to change.

Gottlob et al. [12] show that it is $\text{LOGCFL}$-complete to decide whether from a given acyclic hypergraph $Q$ there is a homomorphism into a hypergraph $D$. The complexity class $\text{LOGCFL}$ contains all problems that can be reduced in logarithmic space to a context-free language. This class is contained in $\text{AC}^1$, contains $\text{NL}$ and is equivalent to logspace-uniform $\text{SAC}^1$, the class of problems decidable by logspace-uniform families of semi-unbounded Boolean circuits of polynomial size and logarithmic depth. A semi-unbounded Boolean circuit consists of or-gates with unbounded fan-in and and-gates with fan-in 2. There are no negation
gates, but for each input gate $x_i$ there is an additional input gate $\neg x_i$ that carries the negated value of $x_i$.

In their article, Gottlob et al. [12] show that there is a schema $\tau$ such that every $\text{SAC}^1$ problem can be reduced in logarithmic space to $\text{AHH}(\tau)$. We slightly adapt their construction and show that the hardness result also holds for bounded logspace reductions. Furthermore, if a reduction $f$ maps an instance $x$ to an instance $f(x)$, then the change to $f(x)$ induced by a change to $x$ is first-order definable.

**Theorem 5** (adapted from [12, Theorem 4.8]).

(a) There is a schema $\tau$ that contains at most binary relations such that $\text{AHH}(\tau)$ is hard for $\text{LOGCFL}$ under logspace reductions.

(b) Let $L \in \text{LOGCFL}$. There is a logspace reduction $f_L$ from $L$ to $\text{AHH}(\tau)$ that satisfies the following properties. Assume that $x, x'$ are instances of $L$ with $|x| = |x'|$ and let $(Q, D) = f_L(x)$ and $(Q', D') = f_L(x')$. Then:

(i) $Q = Q'$,

(ii) if $x$ and $x'$ differ only in one bit, then $D'$ differs from $D$ by at most $c$ hyperedges, for a global constant $c$, and

(iii) $D'$ is first-order definable from $D, x$ and $x'$.

**Proof.** Let $L$ be a problem from $\text{LOGCFL}$. As $\text{LOGCFL} = \text{logspace-uniform } \text{SAC}^1$, there is a logspace-uniform family $(C_n)_{n \in \mathbb{N}}$ of circuits that decides $L$, where a circuit $C_n$ has size at most $n^k$ for some $k \in \mathbb{N}$, logarithmic depth in $n$, and the fan-in of every and-gate is bounded by 2. Without loss of generality, see [12, Lemma 4.6], we can assume that $C_n$ also has the following normal form:

1. The circuit consists of layers of gates, and the gates of layer $i$ receive all their inputs from gates at layer $i - 1$,

2. All layers either only contain or-gates or only contain and-gates,

3. The first layer after the inputs consists of or-gates,

4. If layer $i$ is a layer of or-gates, then layer $i + 1$ only consists of and-gates, and vice versa,

5. The output gate is an and-gate.

A circuit of this form accepts its input if and only if a proof tree can be homomorphically mapped into it. A proof tree $T_n$ for a circuit $C_n$ in normal form has the same depth as $C_n$ and an and-gate as its root. Each and-gate of the proof tree has two or-gates as its children, and each or-gate has one child, which is a gate labelled with the constant 1 for an or-gate at the lowest layer, and an and-gate for all other or-gates. Note that a proof tree is acyclic.

If there is a homomorphism that maps each constant 1 of the proof tree to an input gate of the circuit that is set to 1, each and-gate of the proof tree to an and-gate of the circuit, and for each and-gate of the proof tree its two children to different or-gates in the circuit, then all gates of the circuit that are in the image of the homomorphism evaluate to 1 for the current input. Therefore, the output gate also evaluates to 1, and the circuit accepts its input. It is also clear that if
the circuit accepts its input, then there is a homomorphism from the proof tree into the circuit.

We encode circuits and proof trees over the schema \( \tau = \{0, 1, \text{OR, AND-LEFT, AND-RIGHT}\} \). Each gate \( g \) is encoded by a tuple \( \text{ENC}(g) \) of \( k \) elements. If \( g \) is an and-gate with children \( g_1, g_2 \), then this is encoded by tuples \( (\text{ENC}(g), \text{ENC}(g_1)) \in \text{AND-LEFT} \) and \( (\text{ENC}(g), \text{ENC}(g_2)) \in \text{AND-RIGHT} \). If \( g \) is an or-gate and \( g' \) is one of its children, then this is encoded by the tuple \( (\text{ENC}(g), \text{ENC}(g')) \in \text{OR} \). The relations \( 0 \) and \( 1 \) are used to encode constants and assignments of input gates in the obvious way.

We use the two relations \( \text{AND-LEFT}, \text{AND-RIGHT} \) to ensure that a homomorphism from a proof tree to a circuit maps the two children of an and-gate to two different or-gates.

From the proof of [12, Theorem 4.8] it follows that from an input \( x \) of \( L \) with \(|x| = n\) the corresponding circuit \( C_n(x) \), which results from \( C_n \) by assigning constants to its inputs gates as specified by \( x \), and the corresponding proof tree \( T_n \) can be computed in logarithmic space. In conclusion, this proves that the function \( f_L \) that maps \( x \) to \((T_n, C_n(x))\) is a logspace reduction from \( L \) to AHH(\( \tau \)), and therefore that AHH(\( \tau \)) is hard for \( \text{LOGCFL} \) under logspace reductions.

We now proceed to prove part \( \text{(b)} \) of the theorem statement. Consider two input instances \( x, x' \) for \( L \) with \(|x| = |x'| = n \). Both \( x \) and \( x' \) are inputs of the circuit \( C_n \), so the same proof tree is constructed for them by \( f_L \), yielding part \( \text{(b) i} \). The only differences in the images of \( f_L \) are the assignments of constants to the input gates of \( C_n \). If \( x \) and \( x' \) only differ in one bit, say, the first bit that is represented by the input gate \( g_1 \), then we have \( \text{ENC}(g_1) \in 0 \) and \( \text{ENC}(\neg g_1) \in 1 \) for one input, and \( \text{ENC}(g_1) \in 1 \) and \( \text{ENC}(\neg g_1) \in 0 \) for the other input. So, the encodings of the circuit only differ by 4 tuples, which implies part \( \text{(b) ii} \). Towards part \( \text{(b) iii} \) we can ensure that these tuples are first-order definable by using an appropriate encoding \( \text{ENC} \) of the gates, for example by encoding the \( i \)-th input gate by the \( i \)-th tuple in the lexicographic ordering of \( k \)-tuples over the domain. \( \square \)

Building on the hardness result of Theorem [5] we can show that if AHH(\( \tau \)) can be maintained in DynFO under changes of \( D \), then all \( \text{LOGCFL} \)-problems are in DynFO if we allow a \( \text{PTIME} \) initialisation. This would be a breakthrough result, as there are already problems in uniform \( \text{AC}^0[2] \) (problems decidable by uniform circuits with polynomial size, constant depth and not-, and-, or- and modulo 2-gates with arbitrary fan-in), a much smaller complexity class, that we do not know how to maintain in DynFO [23].

**Theorem 6.** If for arbitrary schema \( \tau \) the problem AHH(\( \tau \)) can be maintained in DynFO under insertions and deletions of single hyperedges from \( Q \) and \( D \), as long as \( Q \) stays acyclic, then every problem \( L \in \text{LOGCFL} \) can be maintained in DynFO with \( \text{PTIME} \) initialisation under insertions and deletions of single tuples.

The same even holds under the condition that AHH(\( \tau \)) can only be maintained under changes of single hyperedges of \( D \), but starting from an arbitrary
initial acyclic hypergraph $Q$, even if a \textbf{PTIME} initialisation of the auxiliary relations is allowed. So, we can take this theorem as a strong indicator that AHH might not be in DynFO under changes of $D$.

\textbf{Proof.} Let $L \in \text{LOGCFL}$ be arbitrary. Let $\tau$ be the schema and $f_L$ the reduction guaranteed to exist by Theorem 5 such that $f_L$ is a reduction from $L$ to AHH$(\tau)$. Let $\mathcal{P}$ be a dynamic program that maintains AHH$(\tau)$ under insertions and deletions of single hyperedges. We construct a dynamic program $\mathcal{P}'$ with \textbf{PTIME} initialisation that maintains $L$.

For an initially empty input structure $\mathcal{I}$ over a domain of size $n$, the initialisation first constructs the corresponding SAC$^1$-circuit $C_n(\mathcal{I})$, with the input bits set as given by $\mathcal{I}$, and the proof tree $T_n$ and stores them in auxiliary relations. This is possible in LOGSPACE $\subseteq$ \textbf{PTIME}. Then, using polynomial time, it simulates $\mathcal{P}$ for a sequence of insertions that lead to $C_n(\mathcal{I})$ and $T_n$ from initially empty hypergraphs and stores the produced auxiliary relations.

When a change of $\mathcal{I}$ occurs, $\mathcal{P}'$ identifies the constantly many changes of $C_n(\mathcal{I})$ that are induced by the change, which is possible in first-order logic thanks to Theorem 5 and simulates $\mathcal{P}$ for these changes. \hfill $\square$

5 Conclusion and Further Work

In this paper we studied under which conditions the problem AHH can be maintained in DynFO. Our main result is that this problem is in DynFO under changes of single hyperedges of the query hypergraph $Q$, on the condition that it remains acyclic. This result directly implies that the result of acyclic Boolean conjunction queries can be maintained in DynFO. As the corresponding dynamic program, see proof of Theorem 3, also maintains partial assignments of existing homomorphisms, this can straightforwardly be extended also to non-Boolean acyclic conjunctive queries.

We have also seen that it is unlikely that AHH is in DynFO under changes of the data hypergraph $D$.

In the static setting, the homomorphism problem is not only tractable for acyclic hypergraphs $Q$, but for a larger class of graphs [6] which includes the class of graphs with \textit{bounded treewidth}, see [12]. It is therefore interesting whether our DynFO maintenance result can also be extended to allow for cyclic hypergraphs $Q$, in particular to allow hypergraphs of treewidth at most $k$, for some $k$. Results of this form would probably require an analogous result to Theorem 2, so, that a tree decomposition of some width $f(k)$ can be maintained for every hypergraph with treewidth at most $k$, and that any change of the hypergraph leads to a maintained tree decomposition that can be obtained from its previous version by a constant number of changes.

Outside the DynFO framework, maintenance of tree decompositions for graphs with treewidth $k = 2$, that is, series-parallel graphs, is considered in [3], but a change of the graph may affect a logarithmic number of nodes of the tree decomposition. Preliminary unpublished results show (using different techniques
than \cite{3} that for graphs with treewidth 2 a tree decomposition can indeed be maintained in DynFO. It is so far unclear whether tree decompositions can also be maintained in a way that only a constant-size part changes after a change of the graph.

Acknowledgements.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 682588.

References

1. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. Journal of the ACM (JACM) 30(3), 479–513 (1983). https://doi.org/10.1145/2402.322389
2. Bernstein, P.A., Goodman, N.: Power of natural semijoins. SIAM Journal on Computing 10(4), 751–771 (1981). https://doi.org/10.1137/0210059
3. Bodlaender, H.L.: Dynamic algorithms for graphs with treewidth 2. In: van Leeuwen, J. (ed.) Graph-Theoretic Concepts in Computer Science, 19th International Workshop, WG ’93, Utrecht, The Netherlands, June 16-18, 1993, Proceedings. Lecture Notes in Computer Science, vol. 790, pp. 112–124. Springer (1993). https://doi.org/10.1007/3-540-57899-4_45
4. Chandra, A.K., Merlin, P.M.: Optimal implementation of conjunctive queries in relational data bases. In: Hopcroft, J.E., Friedman, E.P., Harrison, M.A. (eds.) Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA. pp. 77–90. ACM (1977). https://doi.org/10.1145/800105.803397
5. Dalmau, V., Kolaitis, P.G., Vardi, M.Y.: Constraint satisfaction, bounded treewidth, and finite-variable logics. In: Hentenryck, P.V. (ed.) Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2470, pp. 310–326. Springer (2002). https://doi.org/10.1007/3-540-46135-3_21
6. Datta, S., Kulkarni, R., Mukherjee, A., Schwentick, T., Zeume, T.: Reachability is in DynFO. J. ACM 65(5), 33:1–33:24 (2018). https://doi.org/10.1145/3212685
7. Datta, S., Mukherjee, A., Schwentick, T., Vortmeier, N., Zeume, T.: A strategy for dynamic programs: Start over and muddle through. Log. Methods Comput. Sci. 15(2) (2019). https://doi.org/10.23638/LMCS-15(2:12)2019
8. Dong, G., Su, J., Topor, R.W.: Nonrecursive incremental evaluation of datalog queries. Ann. Math. Artif. Intell. 14(2-4), 187–223 (1995). https://doi.org/10.1007/BF01530820
9. Fagin, R.: Degrees of acyclicity for hypergraphs and relational database schemes. Journal of the ACM (JACM) 30(3), 514–550 (1983). https://doi.org/10.1145/2402.322390
10. Feder, T., Vardi, M.Y.: The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. SIAM J. Comput. 28(1), 57–104 (1998). https://doi.org/10.1137/S0097539794266766
11. Gelade, W., Marquardt, M., Schwentick, T.: The dynamic complexity of formal languages. ACM Trans. Comput. Log. 13(3), 19:1–19:36 (2012). https://doi.org/10.1145/2287718.2287719

12. Gottlob, G., Leone, N., Scarcello, F.: The complexity of acyclic conjunctive queries. J. ACM 48(3), 431–498 (2001). https://doi.org/10.1145/382780.382783

13. Green, T.J., Olteanu, D., Washburn, G.: Live programming in the LogicBlox system: A MetaLogiQL approach. Proc. VLDB Endow. 8(12), 1782–1791 (2015). https://doi.org/10.14778/2824032.2824075

14. Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. J. ACM 54(1), 1:1–1:24 (2007). https://doi.org/10.1145/1206035.1206036

15. Gupta, A., Mumick, I.S.: Maintenance of Materialized Views: Problems, Techniques, and Applications, p. 145–157. MIT Press, Cambridge, MA, USA (1999)

16. Hell, P., Nesetril, J.: On the complexity of H-coloring. J. Comb. Theory, Ser. B 48(1), 92–110 (1990). https://doi.org/10.1016/S0095-8956(90)90132-1

17. Koch, C.: Incremental query evaluation in a ring of databases. In: Paredaens, J., Gucht, D.V. (eds.) Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA. pp. 87–98. ACM (2010). https://doi.org/10.1145/1807085.1807100

18. Libkin, L.: Elements of Finite Model Theory. Springer (2004). https://doi.org/10.1007/978-3-662-07003-1

19. Muńoz, P., Vortmeier, N., Zeume, T.: Dynamic Graph Queries. In: Martens, W., Zeume, T. (eds.) 19th International Conference on Database Theory (ICDT 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 48, pp. 14:1–14:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.ICDT.2016.14

20. Patnaik, S., Immerman, N.: Dyn-FO: A parallel, dynamic complexity class. J. Comput. Syst. Sci. 55(2), 199–209 (1997). https://doi.org/10.1006/jcss.1997.1520

21. Schwentick, T., Vortmeier, N., Zeume, T.: Sketches of dynamic complexity. SIGMOD Rec. 49(2), 18–29 (2020). https://doi.org/10.1145/3442322.3442325

22. Venkateswaran, H.: Properties that characterize LOGCFL. J. Comput. Syst. Sci. 43(2), 380–404 (1991). https://doi.org/10.1016/0022-0000(91)90020-6

23. Vortmeier, N., Kokkinis, I.: The dynamic complexity of acyclic hypergraph homomorphisms, accepted for publication at the International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2021.

24. Vortmeier, N., Zeume, T.: Dynamic complexity of parity exists queries. In: Fernández, M., Muscholl, A. (eds.) 28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain. LIPIcs, vol. 152, pp. 37:1–37:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.CSL.2020.37

25. Yannakakis, M.: Algorithms for acyclic database schemes. In: Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings. pp. 82–94. IEEE Computer Society (1981), https://dl.acm.org/doi/10.5555/1286831.1286840

26. Zeume, T.: The dynamic descriptive complexity of k-clique. Inf. Comput. 256, 9–22 (2017). https://doi.org/10.1016/j.ic.2017.04.005