Abstract

Hypertree width is a measure of the degree of cyclicality of hypergraphs. A number of relevant problems from different areas, e.g., the evaluation of conjunctive queries in database theory or the constraint satisfaction in AI, are tractable when their underlying hypergraphs have bounded hypertree width. However, in practical contexts like the evaluation of database queries, we have more information besides the structure of queries. For instance, we know the number of tuples in relations, the selectivity of attributes and so on. In fact, all commercial query-optimizers are based on quantitative methods and do not care on structural properties.

In this paper, in order to combine structural decomposition methods with quantitative approaches, the notion of weighted hypertree decomposition is defined. Weighted hypertree decompositions are equipped with cost functions, that can be used for modeling many situations where there is further information on the given problem, besides its hypergraph representation. The complexity of computing hypertree decompositions having the smallest weights, called minimal hypertree decompositions, is analyzed. It is shown that in many cases tractability is lost if weights are added. However, it is proven that, under some—not very severe—restrictions on the allowed cost functions and on the target hypertrees, optimal weighted hypertree decompositions can be computed in polynomial time. For some easier hypertree weighting functions, this problem is also highly parallelizable. Then, a cost function modeling query evaluation costs is provided, and it is shown how to exploit weighted hypertree decompositions for determining (logical) query plans for answering conjunctive queries. Finally, some preliminary results of an experimental comparison of this query optimization technique with the query optimizer of a commercial DBMS are presented.

© 2006 Elsevier Inc. All rights reserved.

Keywords: Query processing; Computational complexity; Decomposition methods; Hypergraphs; Relational databases
1. Introduction

1.1. Conjunctive queries and structural decomposition methods

Conjunctive queries (CQs) have been studied for a long time in database theory. This class of queries, equivalent in expressive power to the class of Select-Project-Join queries, is probably the most fundamental and most thoroughly analyzed class of database queries. While the evaluation of conjunctive queries is known to be an NP-complete problem in general [6], and in PTIME for the restricted class of acyclic queries [16,40], several recent papers [7,10,15,21–23,27], exploit structural query properties to identify and analyze very large parameterized classes of conjunctive queries whose evaluation is tractable. Note that all these results refer to the combined complexity of database queries, where both the database and the query are given in input [38]. In the restricted cases where either the query or the database is fixed the problem may be easier [30,38].

The recent renewed interest in tractable classes of conjunctive queries has two main motivations. First, it is well known that the problem of conjunctive query containment is essentially the same as the problem of CQ evaluation [6]. Conjunctive query containment is of central importance in view-based query processing [1] that arises, e.g., in the context of data warehousing. Second, conjunctive query evaluation is essentially the same problem as constraint satisfaction, one of the major problems studied in the field of AI (see, e.g., [8,31]), and there has been a lot of recent interaction between the areas of query optimization and constraint satisfaction (see Vardi’s survey paper [39]).

In this paper, we adopt the logical representation of a relational database [2,37], where data tuples are identified with logical ground atoms, and conjunctive queries are represented as datalog rules. In particular, a Boolean conjunctive query (BCQ) is represented by a rule whose head is variable-free, i.e., propositional. Some relevant polynomially solvable classes of conjunctive queries are determined by structural properties of the query hypergraph (or of query graph—the primal graph of the query hypergraph, also called Gaifman graph). The hypergraph $\mathcal{H}(Q)$ associated with a conjunctive query $Q$ is defined as $\mathcal{H}(Q) = (V, E)$, where the set $V$ of vertices consists of all variables occurring in the body of $Q$, while the set $E$ of hyperedges contains the set $\text{var}(A)$ of all variables occurring in $A$, for each atom $A$ in the rule body. As an example, consider the following query

$$Q_0: \text{ans} \leftarrow s_1(A, B, D) \land s_2(B, C, D) \land s_3(B, E) \land s_4(D, G) \land s_5(E, F, G) \land s_6(E, H) \land s_7(F, I) \land s_8(G, J).$$

Figure 1 shows its associated hypergraph $\mathcal{H}(Q_0)$.

A structural query decomposition method\(^1\) is a method of appropriately transforming a conjunctive query into an equivalent tree query (i.e., acyclic query given in form of a join tree) by organizing its atoms into a polynomial number of clusters, and suitably arranging the clusters as a tree (see Fig. 1). Each cluster contains a number of atoms. After performing the join of the relations corresponding to the atoms jointly contained in each cluster, we obtain a join tree of an acyclic query which is equivalent to the original query. The resulting query can be answered in polynomial time by Yannakakis’s well-known algorithm [40]. In the case of a Boolean query, it can be answered in polynomial time. The tree of atom-clusters produced by a structural query decomposition method on a given query $Q$ is referred to as the decomposition of $Q$. Figure 1 also shows two possible decompositions of our example query $Q_0$. A decomposition of $Q$ can be seen as a query plan for $Q$, requiring to first evaluate the join of each cluster, and then to process the resulting join tree bottom-up (following Yannakakis’s algorithm).

The efficiency of a structural decomposition method essentially depends on the maximum size of the produced clusters, measured (according to the chosen decomposition method) either in terms of the number of variables or in terms of the number of atoms. For a given decomposition, this size is referred-to as the width of the decomposition. For example, if we adopt the number of atoms as the efficiency measure, then the width of both the decompositions shown in Fig. 1 is 2. Intuitively, the complexity of transforming a given decomposition into an equivalent tree query is exponential in its width. In fact, the evaluation cost of each of the (polynomially many) clusters is bounded by the cost of performing the (at most) $w$ joins of its relations. The overall cost (transformation + evaluation of the resulting acyclic query) is $O(n^{w+1} \log n)$, where $n$ is the size of the input problem, that is, the size of the query and the database encoding [15].

\(^1\) In the field of constraint satisfaction, the same notion is known as structural CSP decomposition method, cf. [15].
Once we fix a bound $k$ for such a width, any structural method $D$ identifies a class of queries that can be answered in polynomial time, namely, all those queries having $k$-bounded $D$-width.\footnote{Intuitively, the $D$-width of a query $Q$ is the minimum width of the decompositions of $Q$ obtainable by method $D$.}

The main structural decomposition methods are based on the notions of Biconnected Components \cite{AD90}, Tree Decompositions \cite{CRS84,CRS85,DRS92,RS85}, Hinge Decompositions \cite{NPS01}, and Hypertree Decompositions \cite{GP98,KP04,KE03}. Among them, the Hypertree Decomposition Method (HYPERTREE) seems to be the most powerful method, as a large class of cyclic queries has a low hypertree-width, and in fact, it strongly generalizes all other structural methods \cite{GP96}. More precisely, this means that every class of queries recognized as tractable according to any structural method $D$ (has $k$-bounded $D$-width), is also tractable according to HYPERTREE (has $k$-bounded HYPERTREE-width), and that there are classes of queries that are tractable according to HYPERTREE, but not tractable w.r.t. $D$ (have unbounded $D$-width). Importantly, for any fixed $k > 0$, deciding whether a hypergraph has hypertree width at most $k$ is feasible in polynomial time, and is actually highly parallelizable, as this problem belongs to LOGCFL \cite{GP98} (see Appendix A, for properties and characterizations of this complexity class).

1.2. Limits to the applicability of structural decomposition methods

Despite their very positive computational properties, all the above structural decomposition methods, including Hypertree Decomposition, are often unsuited for some real-world applications; two decompositions having the same width might be very different from the viewpoint of the efficient evaluation in practice. For instance, in a practical database context, one may prefer query plans (i.e., minimum-width decompositions) that minimize the number of clusters having the largest cardinality.

Even more importantly, structural decompositions methods focus “only” on structural query features, while they completely disregard “quantitative” aspects of the database instance, which may dramatically affect the query-evaluation time. For instance, in query answering, the computation of an arbitrary hypertree decomposition (having minimum width) might not be satisfactory, since it does not take into account important quantitative factors, such as relation sizes, attribute selectivity, and so on. These factors are flattened in the query hypergraph (which considers only the query structure), while their suitable exploitation can significantly reduce the actual cost of query evaluation. On the other hand, query optimizers of commercial DBMSs are based solely on quantitative methods and do not care on structural properties at all. Indeed, since computing an optimal plan is an NP-hard problem, all the commercial DBMSs compute approximations of optimal query plans. They restricting the search space to query plans having a very simple structure (e.g., left-deep trees), and then try to find the best plan among them, by estimating their evalua-
tion costs according to some cost-models exploiting the quantitative information on the input database. It follows that, even on some low-width queries with a guaranteed polynomial-time evaluation upper-bound, commercial DBMSs may also take time \( O(n^\ell) \), which is exponential in the length \( \ell \) of the query, rather than on its width. On some relevant applications with many atoms involved, as for the queries used for populating datawarehouses, this may lead to unacceptable costs. In fact, very often such queries are not very intricate and have low hypertree width, though not necessarily acyclic.

1.3. Contribution of the paper

To overcome the above-mentioned drawbacks, we aim at combining the structural decomposition methods with quantitative approaches.

To this end, we generalize the notion of HYPERTREE, by equipping hypertree decompositions with polynomial-time weight functions that may encode quantitative aspects of the query database. It is worthwhile noting that such weight functions may well encode further (possibly alternative) requirements that may be useful in different applications of structural decomposition methods [9,14]. For instance, one may want to minimize the size of hypergraph separators, or the number of vertices with the largest width, and so on.

Computing a minimal weighted-decomposition is, in general, harder than computing a standard decomposition and tractability may be lost. We extensively study the computational complexity of this problem, we prove hardness results and we identify useful tractable cases. In summary, the main contributions of this paper are:

- We define the notion of hypertree weighting function (short: HWF) and of minimal hypertree decompositions of a hypergraph \( H \) w.r.t. a given HWF \( \omega_H \) and a class of decompositions \( \mathcal{C}_H \).
- We show that computing such a minimal decomposition is NP-hard for general HWFs, even for acyclic hypergraphs.
- We show that this problem remains NP-hard even if we consider very simple weighting functions, called vertex aggregation functions, and restrict the search space to the class of \( k \)-bounded hypertree decompositions, for any fixed \( k \geq 4 \).
- We prove that, surprisingly, computing minimal hypertree decompositions is tractable if we consider normal-form (NF) hypertree decompositions, which are equivalent to the general ones in the unweighted framework. Moreover, this tractability result holds for a class of functions, called tree aggregation functions (TAFs) and defined over semirings, that is much larger than the class of vertex aggregation functions. We design an algorithm for the efficient computation of such minimal decompositions.
- We investigate what happens in the (frequent) case where the given TAF is logspace computable, and hence not inherently sequential. We show that deciding whether the minimum weight of the hypertree decompositions in normal-form is below some given threshold is in LOGCFL and hence is highly parallelizable. Moreover, we show that this problem is hard also for this class and we thus get another natural complete problem for this interesting complexity class. Note that this result is not obvious, as the LOGCFL-hardness of recognizing (unweighted) \( k \)-bounded hypertree decompositions is still unknown. We then prove that computing such minimal decompositions is in the functional version of LOGCFL, that is, it is in \( L^{\text{LOGCFL}} \).
- We show how the notion of weighted hypertree decomposition can be used for generating effective query plans for the evaluation of conjunctive queries, by combining structural and quantitative information. To this end, we describe a suitable TAF cost \( H(Q) \), which encodes traditional query-plan cost estimations, based on the size of database relations and attribute selectivity.
- We implement the algorithm for the computation of the minimal decompositions with respect to cost \( H(Q) \), that correspond to query plans that are optimal w.r.t. our cost-model, and we report the result of some preliminary experimental comparisons of the query plans computed by our algorithm with those generated by the internal optimization module of a leader DBMS system (which will be referred to by the fantasy name CommDB, for licence restrictions). This is not intended to be a thorough comparison with such a system, as we only tried a small set of queries. Our aim here is just to show that exploiting the structure of the query may lead to significant computational savings, and in fact, the preliminary results confirm this intuition. We refer the interested reader to [13] for further experiments both with CommDB and with the DBMS PostgreSQL. In particular, since the latter DBMS is open-source, a query optimizer based on weighted hypertree decompositions has been implemented directly inside the system.
1.4. Organization of the paper

In Section 2, we recall the definitions of hypertree decomposition and hypertree width, and we present a new normal form for hypertree decompositions. In Section 3, we present the framework of weighted hypertree decompositions, and investigate the complexity of the main problems related to the computation of hypertree decompositions having the minimal weights. In Section 4, we define the tree aggregation functions (TAFs), provide some examples, and describe a polynomial time algorithm (called \(\text{minimal-}\)k-decomp) for computing \(k\)-bounded minimal NF hypertree decompositions w.r.t. a TAF. For the sake of presentation, we defer the (quite involved) formal proof of correctness of algorithm \(\text{minimal-}\)k-decomp to Section 7. The tractability result for decompositions in normal forms is then sharpened in Section 5, where we show that, for some parallelizable TAFs, computing minimal NF hypertree decomposition is in \(L^{\mathrm{LOGCFL}}\). In Section 6, we describe a specialization of \(\text{minimal-}\)k-decomp for conjunctive query planning. In Section 8 we draw our conclusions. Finally, in Appendix A, we provide some basic notions of alternating Turing machines and of the complexity class LOGCFL, which are useful in some proofs of our results.

2. Hypertree decompositions

In this section, we recall some basic definitions of hypergraphs and hypertree decompositions. For detailed descriptions of the latter notion, see [14,17]. Then, we describe a new normal form for hypertree decompositions, which leads to more efficient algorithms.

2.1. Basic notions

A hypergraph \(\mathcal{H}\) is a pair \((V, H)\), where \(V\) is a set of vertices and \(H\) is a set of hyper-edges such that for each \(h \in H\), \(h \subseteq V\). For the sake of simplicity, we always denote \(V\) and \(H\) by \(\text{var}(\mathcal{H})\) and \(\text{edges}(\mathcal{H})\), respectively. We use the term \(\text{var}\) because, in our context, hypergraph vertices correspond to query variables. Moreover, for a set of hyperedges \(S\), \(\text{var}(S)\) denotes the set of variables occurring in \(S\), that is \(\bigcup_{h \in S} h\). For the sake of presentation, without loss of generality, we assume connected hypergraphs.

An important structural property for hypergraphs is acyclicity. Actually, there are different notions of hypergraph acyclicity and different equivalent characterizations. We consider here the more liberal notion known as \(\alpha\)-acyclicity, together with its characterization in terms of join trees [3,4,28]: A hypergraph \(\mathcal{H}\) is \(\alpha\)-acyclic if and only if it has a join tree. A join tree \(JT(\mathcal{H})\) for a hypergraph \(\mathcal{H}\) is a tree whose vertices are the edges of \(\mathcal{H}\) such that, whenever the same variable \(X \in V\) occurs in two edges \(A_1\) and \(A_2\) of \(\mathcal{H}\), then \(X\) occurs in each vertex on the unique path linking \(A_1\) and \(A_2\) in \(JT(\mathcal{H})\). In other words, the set of vertices in which \(X\) occurs induces a (connected) subtree of \(JT(\mathcal{H})\). We will refer to this condition as the Connectedness Condition of join trees.

The hypertree-width is a measure of the degree of cyclicity of hypergraphs. A hypertree for a hypergraph \(\mathcal{H}\) is a triple \((T, \chi, \lambda)\), where \(T = (N, E)\) is a rooted tree, and \(\chi\) and \(\lambda\) are labeling functions which associate each vertex \(p \in N\) with two sets \(\chi(p) \subseteq \text{var}(\mathcal{H})\) and \(\lambda(p) \subseteq \text{edges}(\mathcal{H})\). If \(T' = (N', E')\) is a subtree of \(T\), we define \(\chi(T') = \bigcup_{v \in N'} \chi(v)\). We denote the set of vertices \(N\) of \(T\) by \(\text{vertices}(T)\), and the root of \(T\) by \(\text{root}(T)\). Moreover, for any \(p \in N\), \(T_p\) denotes the subtree of \(T\) rooted at \(p\).

**Definition 2.1.** [17] A hypertree decomposition of a hypergraph \(\mathcal{H}\) is a hypertree \(HD = (T, \chi, \lambda)\) for \(\mathcal{H}\) which satisfies all the following conditions:

1. For each edge \(h \in \text{edges}(\mathcal{H})\), there exists \(p \in \text{vertices}(T)\) such that \(h \subseteq \chi(p)\) (we say that \(p\) covers \(h\));
2. For each variable \(Y \in \text{var}(\mathcal{H})\), the set \(\{p \in \text{vertices}(T) \mid Y \in \chi(p)\}\) induces a (connected) subtree of \(T\);
3. For each \(p \in \text{vertices}(T)\), \(\chi(p) \subseteq \text{var}(\lambda(p))\);
4. For each \(p \in \text{vertices}(T)\), \(\text{var}(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)\).

Note that the inclusion in condition (4) is actually an equality, because condition (3) implies the reverse inclusion.

An edge \(h \in \text{edges}(\mathcal{H})\) is strongly covered in \(HD\) if there exists \(p \in \text{vertices}(T)\) such that \(\text{var}(h) \subseteq \chi(p)\) and \(h \in \lambda(p)\). In this case, we say that \(p\) strongly covers \(h\). A hypertree decomposition \(HD\) of hypergraph \(\mathcal{H}\) is a complete decomposition of \(\mathcal{H}\) if every edge of \(\mathcal{H}\) is strongly covered in \(HD\).
The width of a hypertree decomposition \( \langle T, \chi, \lambda \rangle \) is \( \max_{p \in \text{vertices}(T)} |\lambda(p)| \). The \textsc{hypertree width} \( hw(H) \) of \( H \) is the minimum width over all its hypertree decompositions. A \( c \)-width hypertree decomposition of \( H \) is \textit{optimal} if \( c = hw(H) \).

In analogy to join trees of acyclic hypergraphs, we will refer to condition (2) above as the \textit{Connectedness Condition}.

For instance, Fig. 1 shows two hypertree decompositions of the query \( Q_0 \) in Section 1. Both decompositions have width two and are complete decompositions of \( Q_0 \).

Given any hypertree decomposition \( HD \) of \( H \), we can easily compute a complete hypertree decomposition of \( H \) having the same width. Note that the acyclic hypergraphs are precisely those hypergraphs having hypertree width one. Indeed, any join tree of an acyclic hypergraph \( H \) trivially corresponds to a hypertree decomposition of \( H \) of width one. Furthermore, if a hypergraph \( H' \) has a hypertree decomposition of width one, then, from this decomposition, we can easily compute a join tree of \( H' \) [17].

Intuitively, if \( H \) is a cyclic hypergraph, the \( \chi \) labeling selects the set of variables to be fixed in order to split the cycles and achieve acyclicity; \( \lambda(p) \) “covers” the variables of \( \chi(p) \) by a set of edges.

Let \( k \) be a fixed positive integer. From the results in [17], it follows that deciding whether a hypergraph has hypertree width at most \( k \) (also, \( k \)-bounded hypertree width) is in \textsc{LOGCFL}, and hence in polynomial time. Moreover, computing a hypertree decomposition of width at most \( k \) (if any) can be done efficiently.

### 2.2. A normal form for hypertree decompositions

Observe that, according to Definition 2.1, a hypergraph may have some (usually) undesirable hypertree decompositions. For instance, a licit decomposition may contain a vertex \( p \) with some useless hyperedge \( h \in \lambda(p) \) such that \( h \cap \chi(p) = \emptyset \), i.e., whose variables do not actually contribute to the decomposition.

Thus, we next define a new normal form for hypertree decompositions, which is stronger than the previous similar notion in [17], and avoids these kind of redundancies. An important feature of this new normal form, exploited in the algorithms presented in this paper, is that it allows us to define easily the \( \chi \) labeling of a vertex, just on the basis of its \( \lambda \) labeling and of the hypergraph component (to be decomposed) associated with it.

Let \( H \) be a hypergraph, and let \( V \subseteq \text{var}(H) \) be a set of variables and \( X, Y \in \text{var}(H) \). \( X \) is \([V]\)-adjacent to \( Y \) if there exists an edge \( h \in \text{edges}(H) \) such that \( \{X, Y\} \subseteq (h - V) \). A \([V]\)-path \( \pi \) from \( X \) to \( Y \) is a sequence \( X = X_0, \ldots, X_\ell = Y \) of variables such that: \( X_i \) is \([V]\)-adjacent to \( X_{i+1} \), for each \( i \in \{0, \ldots, \ell - 1\} \). A set \( W \subseteq \text{var}(H) \) of variables is \([V]\)-connected if \( \forall X, Y \in W \) there is a \([V]\)-path from \( X \) to \( Y \). A \([V]\)-component is a maximal \([V]\)-connected non-empty set of variables \( W \subseteq (\text{var}(H) - V) \). For any \([V]\)-component \( C \), let \( \text{edges}(C) = \{h \in \text{edges}(H) \mid h \cap C \neq \emptyset\} \).

Let \( HD = \langle T, \chi, \lambda \rangle \) be a hypertree decomposition of \( H \). For any vertex \( v \) of \( T \), we will often use \( v \) as a synonym of \( \chi(v) \). In particular, \([v]\)-component \( [\chi(v)] \)-component; the term \([v]\)-path is a synonym of \( [\chi(v)] \)-path; and so on.

**Definition 2.2.** A hypertree decomposition \( HD = \langle T, \chi, \lambda \rangle \) of a hypergraph \( H \) is in \textit{normal form (NF)} if for each vertex \( r \in \text{vertices}(T) \), and for each child \( s \) of \( r \), all the following conditions hold:

1. there is (exactly) one \([r]\)-component \( C_r \) such that \( \chi(T_s) = C_r \cup (\chi(s) \cap \chi(r)) \);
2. \( \chi(s) \cap C_r \neq \emptyset \), where \( C_r \) is the \([r]\)-component satisfying condition (1);
3. for each \( h \in \lambda(s), h \cap \text{var(\text{edges}(C_r))} \neq \emptyset \), where \( C_r \) is the \([r]\)-component satisfying condition (1);
4. \( \chi(s) = \text{var(\text{edges}(C_r))} \cap \lambda(s) \), where \( C_r \) is the \([r]\)-component satisfying condition (1).

Note that, from condition (4) above, the label \( \chi(s) \) of any vertex \( s \) of a hypertree decomposition in normal form can be computed just from its label \( \lambda(s) \) and from the \([r]\)-component satisfying condition (1) associated with its parent \( r \).

As for the weaker normal form defined in [17], NF hypertree decompositions have the following fundamental property.

**Theorem 2.3.** For each \( k \)-width hypertree decomposition of a hypergraph \( H \) there is a \( k \)-width hypertree decomposition of \( H \) in normal form.
Proof. Recall that the original definition of normal form in [17], denoted hereafter by NF, has the same conditions (1) and (2) as the new NF in Definition 2.2, and a condition (3) \( \var(s) \cap \chi(r) \subseteq \chi(s) \), which is a weaker form of condition (4), while the above condition (3) is missing.

From [17] we know that, if there is a \( k \)-width hypertree decomposition of a hypergraph \( H \), then there is a \( k \)-width hypertree decomposition \( HD = (T, \chi, \lambda) \) of \( H \) in NF. It thus suffices to prove that, from \( HD \), we may build a \( k \)-width hypertree decomposition \( HD' = (T', \chi', \lambda') \) of \( H \) in NF.

We first observe that, from conditions (1) and (2) of hypertree decompositions and condition (1) of NF,

(i) \( \var(edges(C_r)) \cap \chi(r) \subseteq \chi(s) \),

because all variables in \( \var(edges(C_r)) \) occur in the subtree \( T_s \) rooted at the child \( s \) of \( r \).

Moreover, from conditions (3) and (4) of hypertree decompositions, we know that \( \chi(s) = \var(\lambda(s)) \cap \chi(T_s) \), which after condition (1) of NF yields: \( \chi(s) = \var(\lambda(s)) \cap (C_r \cup \chi(s) \cap \chi(r)) = (\var(\lambda(s)) \cap C_r) \cup (\var(\lambda(s)) \cap (\chi(s) \cap \chi(r))) \). Finally, since \( \chi(s) \subseteq \var(\lambda(s)) \), we have that, for any (non-root) vertex \( s \) of a decomposition in NF,

(ii) \( \chi(s) = (C_r \cap \var(\lambda(s))) \cup (\chi(s) \cap \chi(r)) \).

Then, the hypertree \( HD' \) is defined as follows. We set \( T' = T \), we label the roots of \( HD \) and \( HD' \) in the same way, and by a breadth-first visit of \( T' \) we set, for every \( s \) having a vertex \( r \) as its parent,

\[
\lambda'(s) = \{ h \in \lambda(s) \mid h \cap \var(edges(C_r)) \neq \emptyset \} \quad \text{and} \quad \chi'(s) = (C_r \cap \var(\lambda(s))) \cup (\var(edges(C_r)) \cap \chi'(r)),
\]

where \( C_r \) is the \( [r] \)-component in \( HD \) satisfying condition (1) of NF.

We next show that \( HD' \) is a \( k \)-width hypertree decomposition of \( H \) in normal form.

First, observe that, at any step, the above transformation does not affect the \( C_r \) components, and hence the same sets of variables play this role in both \( HD \) and \( HD' \). Indeed, comparing the expression for \( \chi'(s) \) with the expression (ii) for \( \chi(s) \), we see that we are just discarding those variables from \( \chi(r) \) that do not belong to \( \var(edges(C_r)) \), and are thus useless for covering hyperedges and breaking connected components. Similarly, for \( \lambda'(s) \), we are discarding those edges that are not covering any variable in \( \var(edges(C_r)) \).

Note that labeling \( \lambda' \) and \( \chi' \) satisfy conditions (3) and (4) of Definition 2.2, respectively. Indeed, for the latter, note that \( \var(edges(C_r)) \) is given by the disjoint union of \( C_r \) and \( \var(edges(C_r)) \cap \chi(r) \), which is a subset of \( \chi(s) \) (from (i)) and thus a subset of \( \var(\lambda(s)) \). Furthermore, since \( HD \) and \( HD' \) have the same \( C_r \) components, the remaining conditions (1) and (2) of Definition 2.2 are fulfilled, and therefore \( HD' \) is in normal form.

Moreover, we just delete edges from \( \lambda \), and thus the width of \( HD' \) is at most equal to the width of \( HD \). We next show that \( HD' \) is, in fact, a hypertree decomposition of \( H \). Conditions (1) and (3) of Definition 2.1 clearly hold.

For the remaining conditions, recall that visiting the decomposition tree top-down, the components of the form \( C_r \) decrease monotonically (with respect to set inclusion). This property is also related to the correspondence of these components to the escape space of a robber, which is monotonically shrunk at each step by the marshals who try to catch her, in the game characterization of hypertree decompositions [19]. Thus, a variable discarded from \( \chi(s) \) because it does not occur in \( \var(edges(C_r)) \), will not occur in any component in the subtree \( T'_s \) rooted at \( s \), and hence in \( \chi'(T'_s) \), by construction.

It follows that condition (4) of Definition 2.1 is also fulfilled by \( HD' \) and, from the same line of reasoning, also condition (2). □

3. Weighted hypertree decompositions

In this section, we consider hypertree decompositions with an associated weight, and we analyze the complexity of the main problems related to the computation of the best decompositions.

Formally, given a hypergraph \( H \), a hypertree weighting function (short: HWF) \( \omega_H \) is any polynomial-time function that maps each hypertree decomposition \( HD = (T, \chi, \lambda) \) of \( H \) to a real number, called the weight of \( HD \). We assume in proofs and algorithms that HWFs are given intentionally, as suitable encodings of deterministic Turing transducers.

For instance, a very simple HWF is the function \( \omega^{\max}_H(HD) = \max_{p \in \text{vertices}(T)} |\lambda(p)| \), which weights a hypertree decomposition \( HD \) just on the basis of its worst vertex, that is the vertex with the largest \( \lambda \) label, which also determines the width of the decomposition.
Example 3.1. In many applications, finding a decomposition having the minimum width is not the best that can be done. Minimizing the number of vertices having the largest width \( w \) can be considered and, for decompositions having the same numbers of such vertices, minimizing the number of vertices having width \( w - 1 \), and continuing so on, in a lexicographical way. To this end, we can define the \( \text{HWF} \) decompositions in Definition 2.1, while the \( \text{HDF} \) and the class of hypertree decompositions in normal form; for instance, the decomposition \( \text{HD} \) with the same numbers of such vertices, minimizing the number of vertices having width.

Consider again the query \( Q_0 \) of the Introduction, and the hypertree decomposition \( \text{HD}' \) of \( \mathcal{H}(Q_0) \) shown on the right in Fig. 1. Then, \( \omega_{\text{HDF}}^\text{lex}(\mathcal{H}(Q_0)) = 4 \times 9^0 + 3 \times 9^1 \). Observe that \( \text{HD}' \) is not the best decomposition w.r.t. \( \omega_{\text{HDF}}^\text{lex} \) and the class of hypertree decompositions in normal form; for instance, the decomposition \( \text{HD}'' \) shown on the bottom of Fig. 1 is better than \( \text{HD}' \), as \( \omega_{\text{HDF}}^\text{lex}(\text{HD}'') = 6 \times 9^0 + 1 \times 9^1 \).

Note that these examples are still based on structural criteria only, as no information on the input database is exploited. In fact, in the next section, we will define a more sophisticated hypertree weighting function, specifically designed for query evaluation.

3.1. Minimal decompositions

Since computing a minimum-width hypertree decomposition is NP-hard, it is easy to see that, in the general case, finding the hypertree decompositions having the minimum weight is NP-hard, too. However, we are usually interested only in decompositions belonging to some tractable class, for instance the hypertree decompositions having small width.

Let \( k > 0 \) be a fixed integer and \( \mathcal{H} \) a hypergraph. We define \( k\text{HD}_{\mathcal{H}} \) (respectively, \( k\text{NFD}_{\mathcal{H}} \)) as the class of all hypertree decompositions (respectively, normal-form hypertree decompositions) of \( \mathcal{H} \) having width at most \( k \).

We recall that, given a hypergraph \( \mathcal{H} \), deciding whether \( k\text{NFD}_{\mathcal{H}} \neq \emptyset \) (and hence \( k\text{HD}_{\mathcal{H}} \neq \emptyset \), after Theorem 2.3) is in LOGCFL [17].

Definition 3.2. Let \( \mathcal{H} \) be a hypergraph, \( \omega_{\mathcal{H}} \) a weighting function, and \( \mathcal{C}_{\mathcal{H}} \) a class of hypertree decompositions of \( \mathcal{H} \). Then, a hypertree decomposition \( \text{HD} \in \mathcal{C}_{\mathcal{H}} \) is minimal w.r.t. \( \omega_{\mathcal{H}} \) and \( \mathcal{C}_{\mathcal{H}} \), denoted by \([\omega_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}]\)-minimal, if there is no \( \text{HD}' \in \mathcal{C}_{\mathcal{H}} \) such that \( \omega_{\mathcal{H}}(\text{HD}') < \omega_{\mathcal{H}}(\text{HD}) \).

For instance, the \([\omega_{\text{HDF}}^\text{lex}, \text{kHD}_{\mathcal{H}}]\)-minimal hypertree decompositions are exactly the \( k \)-bounded optimal hypertree decompositions in Definition 2.1, while the \([\omega_{\text{HDF}}^\text{lex}, \text{kHD}_{\mathcal{H}}]\)-minimal hypertree decompositions correspond to the lexicographically minimal decompositions described above. As an example, it is easy to see that the hypertree decomposition \( \text{HD}'' \) of \( \mathcal{H}(Q_0) \), shown in Fig. 1, is a \([\omega_{\text{HDF}}^\text{lex}, \text{kHD}_{\mathcal{H}(Q_0)}]\)-minimal hypertree decomposition, while \( \text{HD}' \) is not.

It is not difficult to show that, for general weighting functions, the computation of minimal hypertree decompositions is a difficult problem even if we consider very simple classes of bounded hypertree decompositions. In fact, we next consider, for any hypergraph \( \mathcal{H} = (V, H) \), the class \( \mathcal{J}\mathcal{T}_{\mathcal{H}} \) of all its join trees. More precisely, \( \mathcal{J}\mathcal{T}_{\mathcal{H}} \) will be the class of all width 1 complete hypertree decompositions \( \text{HD} = (T, \chi, \lambda) \) corresponding to join trees of \( \mathcal{H} \), i.e., such that \( |T| = |H| \) and, for any hyperedge \( h \) of \( \mathcal{H} \), there is a vertex \( p \) in \( T \) with \( \lambda(p) = \{h\} \) and \( \chi(p) = h \). Note that \( \mathcal{J}\mathcal{T}_{\mathcal{H}} \neq \emptyset \) if and only if \( \mathcal{H} \) is acyclic.

Theorem 3.3. Given a hypergraph \( \mathcal{H} \) and an \( \text{HWF} \) \( \omega_{\mathcal{H}} \), computing a \([\omega_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}]\)-minimal hypertree decomposition (if any) is NP-hard. Hardness holds for any \( k > 0 \), and even if \( \mathcal{C}_{\mathcal{H}} \) is the class of join trees \( \mathcal{J}\mathcal{T}_{\mathcal{H}} \).

Proof. The reduction is from the NP-hard problem of coloring a graph by 3 colors. Given a graph \( G \), we can build in polynomial time an acyclic hypergraph \( \mathcal{H}(G) \) and an \( \text{HWF} \) \( \omega_{\mathcal{H}(G)} \) such that the \([\omega_{\mathcal{H}(G)}, \mathcal{J}\mathcal{T}_{\mathcal{H}(G)}]\)-minimal hypertree decompositions correspond to legal 3-colorings of \( G \). Intuitively, the acyclic hypergraph has (exponentially) many possible join trees, and the idea is to exploit their shapes in such a way that join trees corresponding to legal colorings have weight 0, while all other join trees have weight 1.

Then, \( \mathcal{H}(G) = (\bar{V} \cup \bar{V}' \cup \{C\}, H) \), where \( \bar{V} = \{V_1, \ldots, V_n\} \) is the set of vertices of \( G \) and \( \bar{V}' = \{V'_1, \ldots, V'_n\} \). For each vertex \( V_i \in \bar{V}' \), we say that \( V_i \in \bar{V} \) is the vertex associated with \( V'_i \). The set \( H \) contains a big hyperedge \( g = \bar{V} \cup \{C\} \), a hyperedge \( \{V_i, C\} \), for each \( 1 \leq i \leq n \), and a binary hyperedge \( \{V_j, V_i\} \), for each edge \( \{V_j, V_i\} \) of the graph \( G \). Observe that \( \mathcal{H}(G) \) is acyclic, because the big hyperedge \( g \) “absorbs” possible cycles in \( G \).
The weighting function \( \omega_H(G) \) is defined as follows. For any decomposition \( HD = (T, \chi, \lambda) \in \mathcal{JT}_H(G) \), \( \omega_H(G)(HD) = 0 \) if the following conditions hold:

1. all the hyperedges of the form \( \{V', C\} \) are covered in at most 3 subtrees \( T_{c_1}, T_{c_2}, \) and \( T_{c_3} \) rooted at three children \( c_1, c_2, \) and \( c_3 \) of a vertex \( r \), with \( \chi(r) = V \cup \{C\} \), which covers the big hyperedge \( g \);
2. let \( T_c \) be any subtree as in point (1). Then, for any pair of vertices \( n_j, n_t \) in \( T_c \) such that \( \{V'_j, C\} \subseteq \chi(n_j) \) and \( \{V'_t, C\} \subseteq \chi(n_t) \), hypergraph \( H(G) \) has no hyperedge \( \{V_j, V_t\} \).

Otherwise, \( \omega_H(G)(HD) = 1 \). Note that \( \omega_H(G)(HD) \) evaluates any given hypertree \( HD \) in polynomial time. Moreover, the above reduction is feasible in polynomial time (w.r.t. the input size \( |G| \)).

Assume that the graph has a 3-coloring \( col \). Then, there is a width 1 hypertree decomposition for \( H(G) \) (i.e., a join tree) defined as follows. The root covers the hyperedge \( g \) and there are at most three children of \( r \) such that the subtrees rooted at these vertices cover all hyperedges of the form \( \{V, C\} \), and if \( \{V'_j, C\}, \{V'_t, C\} \) belong to the same subtree \( T_r \), \( col \) assigns the same color to both variables \( V_j \) and \( V_t \) of \( G \). Since \( col \) is a legal coloring for \( G \), also the condition in the above point (2) is satisfied, and \( \omega_H(G) \) assigns weight 0 to this decomposition.

Conversely, assume there is a hypertree decomposition \( HD \) such that \( \omega_H(G)(HD) = 0 \). Recall that all hyperedges must be covered, by condition (1) of Definition 2.1. Moreover, no hyperedge of the form \( \{V', C\} \) may occur in two different subtrees of the vertex \( r \) covering \( g \), which satisfies the condition in point (1). Indeed, \( \chi(r) \) does not contain any primed variable \( V' \) and \( HD \) fulfills the connectedness condition. Moreover, after the condition in point (2), a map that assigns the same color \( c \) to all, and only variables occurring in a subtree \( T_r \) rooted at a child of \( r \) clearly encodes a 3-coloring of \( G \).

Then, the weight of any \([\omega_H(G), \mathcal{JT}_H(G)]\)-minimal decomposition of \( H(G) \) is 0 if and only if \( G \) is 3-colorable, and thus computing any such a decomposition amounts to solving the 3-colorability problem.

For the sake of completeness, observe that, for the more general class of all \( k \) bounded hypertree decompositions \( kHD_H(G) \) (where \( k \) is any fixed positive constant), the above hardness proof may significantly be simplified. Given a graph \( G \), consider the acyclic hypergraph \( H(G) = (V, \{V\}) \), where \( V \) is the set of vertices of \( G \) and the hypergraph has only one big hyperedge containing all those vertices. The weighting function \( \omega_{H_G}(G) \) is defined as follows. For any decomposition \( HD = (T, \chi, \lambda) \in kHD_H(G) \), \( \omega_{H_G}(G)(HD) = 0 \) if the \( \lambda \) label of all vertices of \( T \) is the one big hyperedge, the \( \chi \) label of the root \( r \) contains all vertices, and \( r \) has three children whose \( \chi \) labels form a partition of the vertices of \( G \) encoding a 3-coloring of this graph. All other decompositions of \( H(G) \) get weight 1. Then, it is easy to check that \([\omega_{H_G}(G), kHD_H(G)]\)-minimal decompositions of \( H(G) \) are in a one-to-one correspondence with the 3-coloring of \( G \).

In this case, the main source of complexity is the \( \mathbb{HNF} \), which can evaluate hypertree decompositions looking at the whole tree, and weighting for instance the shape of the tree or other arbitrary relationships among vertices.

One may thus wonder whether, restricting attention to simpler \( \mathbb{HNF} \)s the problem becomes any easier. Let \( \mathcal{H} \) be a hypergraph. A vertex aggregation function is a weighting function of the form \( \Lambda^v_H(HD) = \sum_{p \in \text{vertices}(T)} v_H(p) \), where \( v_H \) is any polynomial time function that associates a non-negative real number to any vertex of the hypertree decomposition \( HD \).

Therefore, in vertex aggregation functions, all the power is in the local (restricted to single vertices) function \( v_H \), and the weighting function just returns the sum of such values.

For instance, if we let \( v^\text{lex}_H(p) = B^{(\text{edges}(T))}-1 \), where \( B = |\text{edges}(T)| + 1 \), then the vertex aggregation function \( \Lambda^v_H \) is exactly the \( \mathbb{HNF} \omega_H^\text{lex} \) described in Example 3.1, that allows us to single out the lexicographically minimal hypertree decompositions.

Unfortunately, the next result shows that even in this restricted setting computing minimal decompositions is NP-hard.

**Theorem 3.4.** Given a hypergraph \( \mathcal{H} \) and a vertex aggregation function \( \Lambda^v_H \), computing a \([\Lambda^v_H, kHD_{\mathcal{H}}]\\)-minimal hypertree decomposition of \( \mathcal{H} \) (if any) is NP-hard, for any fixed \( k \geq 4 \).
Both the following conditions:

(1) for each edge $h$ of $\mathcal{H}$, there exists $p \in N$ such that $h \in \lambda(p)$;
(2) for each edge $h$ of $\mathcal{H}$, the set $\{p \in N \mid h \in \lambda(p)\}$ induces a (connected) subtree of $T$;
(3) for each variable $Y \in \text{var}(\mathcal{H})$, the set $\{p \in N \mid Y \in \lambda(p)\} \cup \{p \in N \mid Y \text{ occurs in some edge } h \in \lambda(p)\}$ induces a (connected) subtree of $T$.

The width of the query decomposition $\langle T, \lambda \rangle$ is $\max_{p \in N} |\lambda(p)|$. The query-width $qw(\mathcal{H})$ of $\mathcal{H}$ is the minimum width over all its query decompositions. A query decomposition for $\mathcal{H}$ is pure if, for each vertex $p \in N$, $\lambda(p) \subseteq \text{edges}(\mathcal{H})$.

Recall that deciding the existence of a query decomposition of width at most 4 is $\text{NP}$-hard [17], and hardness holds even for pure query decompositions, where the labels of the decomposition tree contains only hyperedges. By inspecting that hardness proof, it can be observed that the problem is $\text{NP}$-hard even for the slightly liberal notion obtained by deleting condition (2) above. Let us denote by $\text{H-QUERY}$ this generalization of query decomposition. More precisely, we say that an H-QUERY decomposition of a hypergraph $\mathcal{H}$ is a pair $\text{HD} = \langle T, \lambda \rangle$ which satisfies both the following conditions:

(1) for each edge $h$ of $\mathcal{H}$, there exists $p \in N$ such that $h \in \lambda(p)$;
(2) for each variable $Y \in \text{var}(\mathcal{H})$, the set $\{p \in N \mid Y \in \lambda(p)\}$ induces a (connected) subtree of $T$.

The width of the H-QUERY decomposition $\langle T, \lambda \rangle$ is $\max_{p \in N} |\lambda(p)|$. The H-QUERY width $\text{hqw}(\mathcal{H})$ of $\mathcal{H}$ is the minimum width over all its H-QUERY decompositions.

Comparing this definition to Definition 2.1 of hypertree decompositions, observe that in H-QUERY decompositions all variables occurring in the labels obey the (connectedness) condition (3), while in hypertree decompositions only variables in the $\chi$ labeling have to satisfy the connectedness condition, while hyperedges in the $\lambda$ labeling should only cover these variables. Also, for both kinds of decompositions, the width is measured in terms of cardinality of the hyperedge ($\lambda$) labeling.

Therefore, there is a one-to-one correspondence between H-QUERY decompositions for $\mathcal{H}$ of width $w$ and hypertree decompositions $\text{HD} = \langle T, \chi, \lambda \rangle$ of $\mathcal{H}$ having width $w$ and such that, for each $p \in \text{vertices}(T)$, we require that all variables matter for the connectedness property, i.e., that $\chi(p) = \text{var}(\lambda(p))$. To see the correspondence, just notice that this additional requirement entails that conditions (3) and (4) of Definition 2.1 are always satisfied.

We reduce the problem of deciding whether there is an H-QUERY decomposition of width at most 4 to the problem of computing a $[A^\text{L}_k, k\text{HD}_\mathcal{H}]$-minimal hypertree decomposition of $\mathcal{H}$, for any fixed $k \geq 4$.

In [17] it is shown that, for every query $Q$, $\text{hw}(\mathcal{H}(Q)) \leq qw(Q)$, and by the same proof we get that, for every hypergraph $\mathcal{H}$, the more liberal notion of H-QUERY is also less powerful than hypertree decompositions, i.e., $\text{hw}(\mathcal{H}) \leq \text{hqw}(\mathcal{H})$. Therefore, all hypertree decompositions corresponding to H-QUERY decompositions of width at most 4 belong to the class $k\text{HD}_\mathcal{H}$.

For any hypertree decomposition $\text{HD}' = \langle T', \chi', \lambda' \rangle$ of $\mathcal{H}$, we define the vertex evaluation function $v_{\mathcal{H}}$ as follows: for each $p \in \text{vertices}(T')$, $v_{\mathcal{H}}(p) = \max\{|\text{var}(\lambda'(p)) - \chi'(p)|, |\lambda'(p)| - 4\}$.

Since $A^\text{L}_k$ takes the sum of all these values, and $|\text{var}(\lambda'(p)) - \chi'(p)| \geq 0$ (for any $p$), it follows that the weight of the $[A^\text{L}_k, k\text{HD}_\mathcal{H}]$-minimal hypertree decompositions of $\mathcal{H}$ is 0 if and only if there is a hypertree decomposition in $k\text{HD}_\mathcal{H}$ whose width is at most 4 and such that $|\text{var}(\lambda'(p)) - \chi'(p)| = 0$ for any $p$ in the decomposition tree, that is, if and only if there is an H-QUERY decomposition for $\mathcal{H}$ of width at most 4.

Therefore computing any such a minimal hypertree decomposition amounts to solving the NP-hard query decomposition problem. \qed
Thus, restricting the kind of hypertree weighting functions is not sufficient to ensure tractability. Indeed, the above result shows that even the search space of all \( k \)-bounded hypertree decompositions is too wide to be efficiently explored. We need a further restriction of this class of decompositions.

4. Efficiently-computable minimal hypertree decompositions

In this section, we show that, by slightly restricting the class of allowed decompositions, it is possible to compute in polynomial time any minimal weighted hypertree decomposition, even with respect to hypertree weighting functions more general than the vertex aggregation functions described above.

In fact, we would like to use \( \mathbb{HWF} \)'s whose evaluation of a hypertree decomposition does not depend only on the vertices as isolated entities, but also on the relationships between any vertex and its children in the tree. Moreover, we can think of other operators besides the simple summation.

4.1. Tree aggregation functions

Let \( \langle \mathbb{R}^+, \oplus, \text{min}, \bot, \infty \rangle \) be a semiring, that is, \( \oplus \) is a commutative, associative, and closed binary operator, \( \bot \) is the neuter element for \( \oplus \) (e.g., \( 0 \) for +, \( 1 \) for \( \times \), etc.) and the absorbing element for \( \text{min} \), and \( \text{min} \) distributes over \( \oplus \).\(^3\)

Given a function \( g \) and a set of elements \( S = \{p_1, \ldots, p_n\} \), we denote by \( \bigoplus_{i \in S} g(p_i) \) the value \( g(p_1) \oplus \cdots \oplus g(p_n) \).

**Definition 4.1.** Let \( \mathcal{H} \) be a hypergraph. Then, a tree aggregation function (short: TAF) is any hypertree weighting function of the form

\[
\mathcal{F}^{\oplus, v, e}_{\mathcal{H}}(HD) = \bigoplus_{p \in N} \left( v_{\mathcal{H}}(p) \oplus \bigoplus_{(p, p') \in E} e_{\mathcal{H}}(p, p') \right),
\]

associating an \( \mathbb{R}^+ \) value with the hypertree \( HD = (N, E, \chi, \lambda) \), where \( v_{\mathcal{H}} : N \mapsto \mathbb{R}^+ \) and \( e_{\mathcal{H}} : N \times N \mapsto \mathbb{R}^+ \) are two polynomial functions evaluating vertices and edges of hypertrees, respectively.

Note that every vertex aggregation function corresponds to a tree aggregation function with the same \( v_{\mathcal{H}} \), and with \( \oplus = + \) and the constant function \( \bot \) as the edge evaluation function \( e_{\mathcal{H}} \).

**Example 4.2.** Another simple example of tree aggregation function is \( \mathcal{F}^{\max, v^{w}, \bot}_{\mathcal{H}}(HD) \), where \( v^{w}_{\mathcal{H}}(p) = |\lambda(p)| \). Observe that this TAF is equal to \( \omega^{w} \), and thus its minimal decompositions are those having the minimum possible width.

In some applications it could also be useful to minimize the size of the largest vertex separator in \( HD \), where a separator \( \text{sep}(p, q) \) is defined as \( \chi(p) \cap \chi(q) \) [9]. This can be easily obtained by using the tree aggregation function \( \mathcal{F}^{\max, \bot, \text{sep}}_{\mathcal{H}}(HD) \), where \( e^{\text{sep}}_{\mathcal{H}}(p, q) = |\text{sep}(p, q)| \). Of course, a more sophisticated minimization of the size of separators may be obtained using a lexicographical criterion as for the width, through the TAF \( \mathcal{F}^{\bot, \text{sep}}_{\mathcal{H}}(HD) \), where \( e^{\text{sep}}_{\mathcal{H}}(p, q) = (|N| + 1)^{|\text{sep}(p, q)| - 1} \), and \( N \) is the set of vertices of the decomposition tree of \( HD \).

Observe that in the above examples we used either the vertex evaluation function or the edge evaluation function. By exploiting both functions \( v_{\mathcal{H}} \) and \( e_{\mathcal{H}} \), we can obtain more sophisticated and powerful tree aggregation functions.

**Example 4.3.** Given a query \( Q \), let \( HD = (T, \chi, \lambda) \) be a hypertree decomposition in normal form for \( \mathcal{H}(Q) \). For any vertex \( p \) of \( T \), let \( E(p) \) denote the relational expression \( E(p) = \forall_{h \in \lambda(p)} \prod_{\chi(p)} \text{rel}(h) \), i.e., the join of all relations corresponding to hyperedges in \( \lambda(p) \), suitably projected onto the variables in \( \chi(p) \). Given also an incoming node \( p' \) of \( p \) in the decomposition \( HD \), we define \( v^{*}_{\mathcal{H}(Q)}(p) \) and \( e^{*}_{\mathcal{H}(Q)}(p, p') \) as follows:

- \( v^{*}_{\mathcal{H}(Q)}(p) \) is the estimate of the cost of evaluating the expression \( E(p) \), and
- \( e^{*}_{\mathcal{H}(Q)}(p, p') \) is the estimate of the cost of evaluating the semi-join \( E(p) \bowtie E(p') \).

\(^3\) For the sake of presentation, we refer to \text{min} and hence to minimal hypertree decompositions. However, it is easy to see that all the results presented in this paper can be generalized easily to any semiring, possibly changing \( \text{min} \), \( \mathbb{R}^+ \), and \( \infty \).
Let \( \text{cost}_{\cal H}(Q) \) be the TAF \( \mathbb{F}^{v^*,e^*}_{\cal H}(Q) \) (HD), determined by the above functions. Intuitively, \( \text{cost}_{\cal H}(Q) \) weights the hypertree decompositions of the query hypergraph \( \cal H(Q) \) in such a way that minimal hypertree decompositions correspond to “optimal” query evaluation plans. We will come back to this TAF in Section 6, which is devoted to the relationship between query optimization and minimal hypertree decompositions.

Note that any method for computing the estimates for the evaluation of relational algebra operations may be employed for \( v^* \) and \( e^* \). In particular, in our experiments with such minimal hypertree decompositions reported in Section 6, we adopt the standard techniques described in [12,25].

Clearly, all these powerful weighting functions would be practically useless without a polynomial time algorithm for the computation of minimal hypertree decompositions. We show that, unlike the traditional (non-weighted) framework, working with normal-form hypertree decompositions, rather than with any kind of bounded-width hypertree decomposition, does matter. Indeed, it turns out that computing such minimal hypertree decompositions with respect to any tree aggregation function is a tractable problem. And, in fact, in the following section we exhibit a polynomial time method for computing minimal hypertree decompositions w.r.t. any tree aggregation function, over all \( k \)-bounded normal-form hypertree decompositions of \( \cal H \).

4.2. Algorithm minimal-\( k \)-decomp

The computation of minimal hypertree decompositions w.r.t. TAFs can be carried out in polynomial time by exploiting the algorithm minimal-\( k \)-decomp, shown in Fig. 2. The algorithm works as follows. It maintains a weighted directed bipartite graph \( CG \), called the Candidates Graph, that collects all information we need for computing the desired decompositions. Its nodes are partitioned in two sets \( N_{\text{sub}} \) and \( N_{\text{sol}} \), representing the subproblems that we have to solve and the candidates to their solutions, respectively. Nodes in \( N_{\text{sub}} \) have the form \((R,C)\), where \( R \) is a set of at most \( k \) edges of \( \cal H \), called a \( k \)-vertex, and \( C \) is an \([\text{var}(R)]\)-component. Moreover, there is a special node \((\emptyset,\text{var}(\cal H))\) that represents the whole problem. Nodes in \( N_{\text{sol}} \) have the form \((S,C')\), where \( S \) is a \( k \)-vertex, \( C' \) is a component to be decomposed, \( \text{var}(S) \cap C' \neq \emptyset \) and, \( \forall h \in S, h \cap \text{var}(\text{edges}(C')) \neq \emptyset \).

Intuitively, this node could be the root of a hypertree decomposition for the sub-hypergraph induced by \( \text{var}(\text{edges}(C')) \). The node \((S,C')\) has an arc pointing to all nodes of the form \((R',C') \in N_{\text{sub}}\) for which it is a candidate solution, that is, for which \( \text{var}(\text{edges}(C')) \cap \text{var}(R') \subseteq \text{var}(S) \) holds. Moreover, it has a number of incoming nodes of the form \((S,C'') \in N_{\text{sol}}\), for each \([\text{var}(S)]\)-component \( C'' \) that is included in \( C' \), as each of these nodes represents a subproblem of \((S,C')\) (or, more precisely, of any \((R',C') \in N_{\text{sub}}\), the node \((S,C')\) is connected to).

For every node \( p' \in N_{\text{sub}} \), we initially set \( \text{weight}(p') := v_{\cal H}(p') \). Then, since \( \oplus \) is associative, commutative, and closed, we can update this weight by setting \( \text{weight}(p') := \text{weight}(p') \oplus (\text{weight}(p) \oplus e_{\cal H}(p', p)) \), as soon as we know any descendant \( p \) of \( p' \) in the decomposition tree. These descendants of \( p' \) are obtained by a suitable filtering of the nodes connected to its incoming nodes in \( N_{\text{sub}} \), corresponding to its subproblems. Specifically, for each node \( q \in N_{\text{sub}} \) such that \( q \in \text{incoming}(p') \), \( p \) is any node in \( N_{\text{sol}} \) such that \( \text{weight}(p) \oplus e_{\cal H}(p', p) = \min_{p'' \in \text{incoming}(q)} (\text{weight}(p'') \oplus e_{\cal H}(p', p'')) \).

If a node \( q \in N_{\text{sub}} \) has no candidate solutions, i.e., if \( \text{incoming}(q) = \emptyset \), then it is not solvable. We immediately exploit this information by removing all of its outgoing nodes, for which it was a subproblem. On the other hand, if it has some candidates, whenever all of them have been completely evaluated, it can propagate this information to its outgoing nodes. Then, since \( \min \) distributes over \( \oplus \), we can safely select as its solution its minimum-weighted incoming node in \( N_{\text{sol}} \).

Once all the nodes have been processed, the information encoded in the weighted graph \( CG \) is enough to compute every minimal hypertree decomposition of \( \cal H \) in normal form having width at most \( k \), if any. One of these hypertrees is eventually selected through the simple recursive procedure \( \text{Select-hypertree} \).

**Theorem 4.4 (Correctness of minimal-\( k \)-decomp).** Let \( \cal H \) be a hypergraph and \( \mathbb{F}^{v^*,e^*}_{\cal H} \) be a TAF.

- If \( k \text{NFD}_{\cal H} = \emptyset \), minimal-\( k \)-decomp outputs failure.
- If \( k \text{NFD}_{\cal H} \neq \emptyset \),
Input: A hypergraph \( \mathcal{H} \), a tree aggregation function \( \mathcal{F} \).
Output: An \( [F^{H} \circ \mathcal{F}] - \text{minimal hypertree decomposition of } \mathcal{H} \), if any; otherwise, failure.

\[ \begin{align*}
&\text{Var } CG = (N_{\text{sol}} \cup N_{\text{sub}}, A, \text{weight}); \text{weighted directed bipartite graph; } \\
&\quad HD = ((N_{\text{sol}}, E), \chi, \lambda); \text{hypertree of } \mathcal{H}; \\

&\text{Begin} \\
&\quad \text{(* Build the Candidates Graph *)} \\
&\quad N_{\text{sub}} := \{(\emptyset, \text{var}(\mathcal{H}))\} \cup \{(R, C) \mid R \text{ is a } k\text{-vertex and } C \text{ is an } \text{var}(R)\text{-component} \}; \\
&\quad N_{\text{sol}} := \{(S, C) \mid S \text{ is a } k\text{-vertex, } C \text{ is any } \text{var}(R)\text{-component (for some } R), \text{var}(S) \cap C \neq \emptyset \text{ and,} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ Quad
Theorem 4.5. Given a hypergraph $\mathcal{H}$ and a $TAF F_{\mathcal{H}}^{\oplus,v,e}$, computing an $[F_{\mathcal{H}}^{\oplus,v,e}, kNFD_{\mathcal{H}}]$-minimal hypertree decomposition of $\mathcal{H}$ (if any) is feasible in polynomial time.

Proof. Let $\mathcal{H}$ be a hypergraph and $F_{\mathcal{H}}^{\oplus,v,e}$ a $TAF$, and let $n$ and $m$ be the number of edges and the number of vertices of $\mathcal{H}$, respectively. Moreover, let $c_\oplus, c_{\min}, c_v$, and $c_e$ be the maximum costs of evaluating the operators $\oplus$ and $\min$, and the functions $v_{\mathcal{H}}$ and $e_{\mathcal{H}}$, respectively. Recall that, by definition of $TAF$s, these costs are bounded by a polynomial of their input sizes, and hence by a polynomial of the size of the given problem instance. We denote by $\Psi$ the number of $k$-vertices of $\mathcal{H}$, that is,

$$\Psi = \sum_{i=1}^{k} \binom{n}{i} = \sum_{i=1}^{k} \frac{n!}{i!(n-i)!}.$$  

Note that, for each $k$-vertex $R$, there are at most $O(m) \lfloor R \rfloor$-components. Therefore, the graph $CG$ has $O(\Psi m)$ nodes in $N_{sub}$ and $O(\Psi^2 m)$ nodes in $N_{sol}$. Moreover, each node in $N_{sub}$ has $O(\Psi)$ incoming arcs at most, and each node in $N_{sol}$ has $O(m)$ incoming arcs at most. Then, it can be checked that building $CG$ costs $O(\Psi^2 m^2)$, and computing the weights according to minimal-$k$-decomp costs $O(\Psi^2 m^2 c_{\oplus} + \Psi^2 m c_{\min} + \Psi^2 m c_{v} + \Psi^2 m c_{e}).$ Thus, an upper bound of the overall complexity is given by the latter expression, which is clearly polynomial since $\Psi$ is $O(n^k)$.

Note that the last observation in the above proof provides a readable but very inaccurate upper bound of the cost of computing an $[F_{\mathcal{H}}^{\oplus,v,e}, kNFD_{\mathcal{H}}]$-minimal hypertree decomposition of $\mathcal{H}$. Indeed, for practical purposes, it is worthwhile noting that $\Psi$ and $O(n^k)$ differ significantly. For instance, for $k = 3$ and $n = 5$, $n^k = 125$, while $\Psi = 25$; for $k = 4$ and $n = 10$, $n^k = 10000$, while $\Psi = 385$.

5. Highly parallelizable cases

In this section, we show that by restricting the power of tree aggregation functions a little, we can prove that computing minimal NF hypertree decompositions is a highly parallelizable task.

Of course, this is not possible without any restriction, as the weighting function may be a P-complete function, in general. We say that a $TAF F_{\mathcal{H}}^{\oplus,v,e}$ is smooth if it evaluates any given hypertree decomposition $HD$ of $\mathcal{H}$ in logspace. More precisely, if $\oplus, v_{\mathcal{H}}$ and $e_{\mathcal{H}}$ are Turing transducers whose space employed on both the output tape and the work tape is $O(\log(|\mathcal{H}|))$.

Note that this is a wide class of functions, comprising many interesting $TAF$s. In fact, with the exception of $cost_{\mathcal{H}(Q)}$, all hypertree weighting functions described so far in this paper are smooth. For instance, counting the size of a separator is feasible in logspace, and encoding such a number requires logspace in the size of the given hypertree decomposition. The lexicographical weighting function requires more attention, as the space required for computing $\omega^{lex}_{\mathcal{H}}(HD)$ is $O(w \log(|HD|))$, where $w$ is the width of the given hypertree decomposition $HD$ of $\mathcal{H}$. Therefore, $\omega^{lex}_{\mathcal{H}}$ is smooth if we focus (as is usually the case) on classes of bounded-width hypertree decompositions where $w \leq k$, for some fixed constant $k$.

We first consider the problem of deciding whether there is a normal-form hypertree decomposition $HD$ of $\mathcal{H}$ such that $F_{\mathcal{H}}^{\oplus,v,e}(HD) \leq t$, for some given threshold $t \geq 0$. Interestingly, we show that this problem is LOGCFL-complete and hence in $NC^2$ (see Appendix A, for details on this complexity class, and for its characterization in terms of alternating Turing machines).

It is worth noting that, by proving also the hardness for LOGCFL, we get a new nice natural complete problem for this class, after the recent results in [15]. We remark that we miss such a result for traditional (unweighted) hypertree decompositions (and, similarly, for the notion of bounded treewidth). Indeed, we know that deciding whether $\mathcal{H} \in kHD_{\mathcal{H}}$ is in LOGCFL [17], but the hardness for this class has not been proven, and is still an open problem.

Theorem 5.1. Given a hypergraph $\mathcal{H}$, a smooth $TAF F_{\mathcal{H}}^{\oplus,v,e}$, and a number $t \geq 0$, deciding whether there is a hypertree decomposition $HD \in kNFD_{\mathcal{H}}$ such that $F_{\mathcal{H}}^{\oplus,v,e}(HD) \leq t$ is LOGCFL-complete. Hardness holds for every fixed $k > 0$, and even if we consider an acyclic hypergraph $\mathcal{H}$ and the class $\mathcal{J}\mathcal{T}_{\mathcal{H}}$ of its join trees (instead of $kNFD_{\mathcal{H}}$).
Proof. **Hardness.** We describe a logspace reduction from the LOGCFL-complete problem of answering an acyclic Boolean conjunctive query \( Q \): \( \text{ans} \leftarrow s_1(\bar{X}_1) \land \cdots \land s_m(\bar{X}_m) \) over a database \( \text{DB} \) [16]. Assume without loss of generality that \( Q \) is connected, and that there is no pair of identical tuples in \( \text{DB} \), and no pair of atoms in \( Q \) with the same set of variables. Therefore, we often write simply \( s_i \) for \( s_i(\bar{X}_i) \). Moreover, \( R_i \) denotes the relation in \( \text{DB} \) associated with predicate \( s_i \), with \( 1 \leq i \leq m \). The answer to this query is \( \text{true} \) if and only if there is an assignment \( \rho \), called satisfying assignment, which assigns to each query atom \( s_i \) a tuple \( T_i \in R_i \) such that, for each pair of atoms \( s_i, s_j \), tuple \( T_i = \rho(s_i) \) matches tuple \( T_j = \rho(s_j) \), that is, for the set of variables \( \bar{Y} \) these atoms have in common, the corresponding values in the two tuples are equal, denoted by \( T_i[\bar{Y}] = T_j[\bar{Y}] \).

From \((Q, \text{DB})\), we build the following hypergraph \( \mathcal{H} = (V, H) \). The set of variables \( V \) of \( \mathcal{H} \) is given by \( \bar{X} \cup \bar{T} \), where \( \bar{X} \) is the set of variables occurring in \( Q \), and \( \bar{T} \) is a set of variables corresponding to the tuples occurring in database \( \text{DB} \). Moreover, for each atom \( s_i(\bar{X}_i) \), the hyperedge \( h_i = \bar{X}_i \cup R_i \) and the hyperedges \( h_{ij} = \bar{X}_i \cup \{T_j\} \), for every \( T_j \in R_i \), belong to \( H \). These are the only hyperedges in \( H \). Therefore, since each tuple occurs in (exactly) one relation, \(|H| = m + |\text{DB}|\), where the latter term denotes the number of tuples in the database.

Notice that \( \mathcal{H} \) is an acyclic hypergraph, because \( Q \) is an acyclic query and tuple variables introduce no cycles.

As an example, consider the acyclic query \( Q : \text{ans} \leftarrow s_1(A, B) \land s_2(A, C) \land s_3(B, D) \land s_4(B, E) \), over a database \( \text{DB} = \{R_1, R_2, R_3, R_4\} \), where \( R_1 = \{T_1, T_2, T_3\} \), \( R_2 = \{T_4, T_5\} \), \( R_3 = \{T_6, T_7\} \), and \( R_4 = \{T_8, T_9\} \). For instance, tuple \( T_1 \) may be a pair \( \langle a, b \rangle \), where \( a \) and \( b \) belong to the domains of \( A \) and \( B \), respectively. Figure 3 shows: (A) a join tree for \( Q \), and (B) the hyperedges of hypergraph \( \mathcal{H}_{Q_4} \), obtained by means of the above construction.

We define a \( \text{TAF} \) as follows. Given a hypertree decomposition \( \mathcal{H} \),

\[ v(p) = \max(|\lambda(p)| - 1, |\text{var}(\lambda(p)) - \chi(p)|); \]

- for any pair of vertices \((r, s)\) in the decomposition tree, \( e(r, s) = 0 \) if \( \chi(r) = \chi(s) = h_{ij} \) as in the point above, and \( \chi(s) = h_i \), that is, \( s \) corresponds to the hyperedge \( \bar{X}_i \cup R_i \) containing all variables occurring in query atom \( s_i \) and all tuples in \( R_i \);
- for all pairs \((r, s)\) not satisfying any condition above, \( e(r, s) = 1 \).

![Diagram](image)

Fig. 3. Example of reduction in Theorem 5.1: (A) a join tree for \( Q_4 \); (B) the hyperedges of \( \mathcal{H}_{Q_4} \); and (C) a width 1 hypertree decomposition for \( \mathcal{H}_{Q_4} \) encoding a satisfying assignment for \( Q_4 \).
Notice that both (a suitable encoding of) $F^{+,v,e}$ and $H$ may be computed by a logspace Turing transducer, and that $F^{+,v,e}$ is a smooth TAF.

We claim that the answer to $Q$ on DB is true if and only if there is a minimal hypertree decomposition $HD \in kNFD_H$ such that $F^{+,v,e}(HD) = 0$.

(Only if part.) Recall that $Q$ is acyclic and let $JT$ be a join tree of $Q$, or equivalently of the hypergraph $\mathcal{H}(Q)$ associated with $Q$ (the usual one, not the hypergraph $\mathcal{H}$ in the construction of this proof).

Assume the answer to $Q$ on DB is true and let $\rho$ be a satisfying assignment for $Q$ on DB. Then, consider the following hypertree $HD_\rho = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$. For each vertex $s_i$ of the join tree $JT$, $T$ contains a vertex $p_{ij}$ such that $\lambda(p_{ij}) = \{h_{ij}\}$ and $\chi(p_{ij}) = \text{var}(\lambda(p_{ij})) \subseteq T_j$, where $T_j = \rho(s_i)$. Moreover, all these vertices are connected in $T$ in the same way as their corresponding vertices in $JT$. Furthermore, for any such a $p_{ij}, T$ contains the edge $(q_i, p_{ij})$, where $\lambda(q_i) = \{h_i\}$ and $\chi(q_i) = \tilde{X}_i \cup R_i$. For instance, Fig. 3 shows hypertree $HD_{\rho_t}$ for the assignment $\rho_t$ such that: $\rho_t(s_1) = T_2, \rho_t(s_2) = T_4, \rho_t(s_3) = T_6,$ and $\rho_t(s_4) = T_9.$

Note that this hypertree is in fact a hypertree decomposition of $\mathcal{H}$ in normal form. As far as condition (1) is concerned, observe that every vertex $q_i$ with $\chi(q_i) = \tilde{X}_i \cup R_i$ covers all hyperedges of the form $\tilde{X}_i \cup \{T_j\}$. Also, the connectedness condition is guaranteed because the construction is guided by $JT$, and conditions (3) and (4) are satisfied because, for every $p \in \text{vertices}(HD_\rho)$, $\text{var}(\lambda(p)) = \chi(p)$. Finally, let us compute $F^{+,v,e}(HD_\rho)$. For every $p$, the latter observation and the fact that $|\lambda(p)| = 1$ entail $\nu(p) = 0$. Moreover, every vertex of the form $p_{ij}$, which encodes the choice of tuple $T_j$ for $s_i$, is connected with vertices of the form $p_{ab}$ where $T_j$ matches $T_b$, and with vertex $q_i$ covering all (remaining) tuples for atom $s_i$. Therefore, function $e$ assigns 0 to all edges in the decomposition tree, whence $F^{+,v,e}(HD_\rho) = 0$.

(If part.) Assume there is a width $k$ NF hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$ such that $F^{+,v,e}(HD) = 0$. Observe that, to get weight 0, all edges and all vertices of $HD$ must be weighted 0 by $e$ and $v$, respectively, as $\oplus = +$. Then, consider the following properties of $HD$:

1. Every vertex of $HD$ has width 1, and it is either of the form $p_i$ or of the form $p_{ij}$, where we adopt the same notation as in the other side of the proof. That is, we have $\lambda(p_i) = \{h_i\}$ or $\lambda(p_{ij}) = \{h_{ij}\}$ (where $i$ and $j$ identify an atom $s_i$ and a tuple $T_j \in R_i$, respectively), and $\text{var}(\lambda(p)) = \chi(p)$, for every vertex $p$. Indeed, these are the only cases where $\nu(p) = 0$.

2. Decomposition tree $T$ contains at least one vertex of the form $p_{ij}$ for each atom $s_i$. By contradiction, if $T$ misses any vertex of the form $p_{ab}$ for some atom $s_a(\tilde{X}_a)$, then variables $\tilde{X}_a$ should necessarily be covered by vertex $p_a$, with $\chi(p_a) = \tilde{X}_a \cup R_a$. Since we assumed there is no vertex of the form $p_{ab}$ and $\mathcal{H}$ is a connected hypergraph, $p_a$ must be adjacent to some vertex $p'$ associated with another atom in $Q$. However, in this case, $e(p_a, p') = 1$ (as well as $e(p', p_a) = 1$). Contradiction.

3. There is a set $S$ of vertices of $T$ that contains exactly one vertex of the form $p_{ij}$ for each atom $s_i$, and that induces a (connected) subtree of $T$. To see that this property holds, let $S$ be any maximal subset of $T$ containing only this kind of vertices and inducing a subtree of $T$. If $S$ misses all the vertices for some atom $s_a$, since $S$ is maximal and such a vertex for $s_a$, say $p_{ab}$, belongs to $T$ (from the property in point (2)), it follows that there is a vertex of the form $p_z$ in the path of $T$ from $p_{ab}$ to a vertex $p_{ij} \in S$. This means that $p_z$ is not a leaf (has degree at least 2) and, by definition of weighting function $e$, the only way to get 0 is that $p_z$ is only connected to vertices of the form $p_{\Sigma}$. However, this is impossible, because $HD$ is in normal form and such a vertex below $p_z$ would violate condition (2) of Definition 2.2, as it is completely covered by $p_z$. Moreover, assume by contradiction that $S$ contains a pair of vertices $p_{ij}$ and $p_{ij'}$ for the same atom $s_i$. Then, from point (1), $\chi(p_{ij}) = \tilde{X}_i \cup \{T_j\}$ and $\chi(p_{ij'}) = \tilde{X}_i \cup \{T_j\}$. From the connectedness condition, for each $p$ in the path of the subtree induced by $S$ that connect them, $\tilde{X}_i \subseteq \chi(p)$. Thus, $p_{ij}$ is directly connected to some $p_{ij''}$ with $\chi(p_{ij''}) = \tilde{X}_i \cup \{T_{j''}\}$, with $T_j$ not matching $T_{j''}$ (two different tuples for the same set of variables). It follows that $e$ assigns weight 1 to the edge connecting them, which contradicts $F^{+,v,e}(HD) = 0$.

Observe that the subtree of $T$ induced by set $S$ in point (3) corresponds to a join tree of the acyclic query $Q$. Then, define $\rho_S$ as follows: for each $s_i$ in $Q$, $\rho_S(s_i) = T_j$, where $T_j \in \chi(p_{ij})$ is the tuple variable occurring in the vertex $p_{ij} \in S$ associated with $s_i$. From the connectedness condition of join trees, the definition of $v$ and $e$, and the fact that $F^{+,v,e}(HD) = 0$, it immediately follows that $\rho_S$ is a satisfying assignment for $Q$ on DB, whence the answer of $Q$ is true.
Input: A hypergraph $\mathcal{H}$, a TAF $F^{\oplus,v,e}_{\mathcal{H}}$, and a number $t \geq 0$.
Output: Yes, if $\exists \mathcal{H} \in kNFD_{\mathcal{H}}$ such that $F^{\oplus,v,e}_{\mathcal{H}}(HD) \leq t$; No, otherwise.
Begin (* MAIN *)
Output decomposable$_k(\emptyset, \text{var}(\mathcal{H}), t)$;
End.

Boolean Function decomposable$_k(R: k$-vertex; $C: [\text{var}(R)]$-component; $t'$: number);
Begin
Guess $S \subseteq \text{edges}(\mathcal{H})$, with $|S| \leq k$;
Check that all the following conditions hold:
C1: $\text{var}(S) \cap C \neq \emptyset$ and, $\forall h \in S$, $h \cap \text{var}(\text{edges}(C)) \neq \emptyset$;
C2: $\text{var}(\text{edges}(C)) \cap \text{var}(R) \subseteq S$;
If this check fails Then
Output false;
Else
let $C := \{C' \subseteq \text{var}(\mathcal{H}) \mid C'$ is a $[\text{var}(S)]$-component and $C' \subseteq C\}$;
let $i := v_{\mathcal{H}}((R, C))$;
For each $C' \in C$ Do
Guess a number $t_{C'} \geq 0$, and let $i := i \oplus t_{C'} \oplus e_{\mathcal{H}}((R, C), (S, C'))$;
If not decomposable$_k(S, C', t_{C'})$ Then Output false;
If $i > t'$ Then
Output false;
Else
Output true;
End Function;

Fig. 4. Algorithm threshold-$k$-decomp in Theorem 5.1;

For completeness, observe that, from the above reasoning, hardness clearly holds even if we consider the class of the join trees $\mathcal{J}_{\mathcal{H}}$ of $\mathcal{H}$, instead of the bounded NF hypertree decompositions $kNFD_{\mathcal{H}}$.

Membership. Figure 4 shows Algorithm threshold-$k$-decomp that, given a hypergraph $\mathcal{H}$, a TAF $F^{\oplus,v,e}_{\mathcal{H}}$, and a number $t \geq 0$, outputs Yes, if $\exists \mathcal{H} \in kNFD_{\mathcal{H}}$ such that $F^{\oplus,v,e}_{\mathcal{H}}(HD) \leq t$, and No, otherwise.

Algorithm threshold-$k$-decomp is based on a recursive procedure decomposable$_k$ that gets as its parameters a $k$-vertex $R$ (i.e., a set of at most $k$ hyperedges), a $[\text{var}(R)]$-component $C$ to be “decomposed,” and a number $t' \leq 0$. The latter number may be viewed as a budget for the procedure. It means that it should be able to find an NF hypertree decomposition $HD_C$ of the sub-hypergraph induced by $\text{var}(\text{edges}(C))$ having width at most $k$, and such that $F^{\oplus,v,e}_{\mathcal{H}}(HD_C)$ is at most $t'$. To this end, decomposable$_k$ non-deterministically guesses a $k$-vertex $S$ and checks whether

C1: it may encode a child of $R$ in a possible NF decomposition (satisfies conditions (2) and (3) of Definition 2.2), and
C2: all variables belonging to both $\text{var}(\text{edges}(C))$ and $\text{var}(R)$ should occur in $S$, too (otherwise, the connectedness condition would be violated).

Then, the procedure computes the set of $[\text{var}(S)]$-components $C$ included in $C$, and guesses a budget $t_{C'}$ for each component $C' \in C$. In the subsequent steps, decomposable$_k$ checks, by recursive calls to itself, that each $C' \in C$ may be further decomposed by a suitable choice of a $k$-vertex (to become a child of $S$), and that the weight of this decomposition will be $t_{C'}$ at most. Furthermore, if all the recursive calls succeed, by looking at variable $i$, it checks that

$$v_{\mathcal{H}}((R, C)) \oplus \bigoplus_{C' \in C} (t_{C'} \oplus e_{\mathcal{H}}((R, C), (S, C'))) \leq t'.$$

If $F^{\oplus,v,e}_{\mathcal{H}}$ is smooth, then Algorithm threshold-$k$-decomp can be implemented on a logspace alternating Turing machine $M$ with a polynomially-bounded proof-tree, and thus the problem is in LOGCFL [33] (see Appendix A for formal definitions of these notions). Existential configurations of $M$ are used for guessing both the set $S$ of $k$-hyperedges that is candidate to belong to some decomposition, and the maximum weight $t_{C'}$ associated with any component of the form $C'$, as described above. The only exception is the starting configuration, where the weight is not guessed, but just initialized with the given threshold $t$ (suitably encoded through a logspace pointer to the input
Note that any certificate for the computation of $M_k$ may form having width at most $2$ transducers [18].

Let $w$ be a logspace Turing transducer that encodes a hypertree in $\mathcal{H}$ whose width is at most $k$ and $F_{\mathcal{H}}^{\oplus,v,e}(HD) \leq t$, if and only if there is a sequence of $k$-vertex and weight guesses that leads to a proof tree of $M$ corresponding to $HD$. □

Moreover, we show that such a minimal decomposition is a parallelizable task, as it belongs to the functional version of LOGCFL.

**Theorem 5.2.** Given a hypergraph $\mathcal{H}$ and a smooth TAF $F_{\mathcal{H}}^{\oplus,v,e}$, computing an $[F_{\mathcal{H}}^{\oplus,v,e}, kNFD_{\mathcal{H}}]$-minimal hypertree decomposition of $\mathcal{H}$ (if any) is feasible in $L^{LOGCFL}$.

**Proof.** Let $M'$ be a logspace Turing transducer with an oracle in LOGCFL that acts as follows. First, $M'$ computes the minimal weight $w$ (w.r.t. $F_{\mathcal{H}}^{\oplus,v,e}$) over all hypertree decompositions in $kNFD_{\mathcal{H}}$. This is feasible via a binary search over all possible output values for $F_{\mathcal{H}}^{\oplus,v,e}$. Note that, since the TAF is smooth, the maximum possible weight is determined by the size of the largest bounded-width hypertree decomposition of $\mathcal{H}$ in normal form. As observed in the proof above, we know from [17] that this size is bounded by a polynomial of the hypergraph size, and thus $\|t_{\text{max}}\|$ is $O(\|\mathcal{H}\|)$. At each step, $M'$ asks its oracle whether there is some $HD \in kNFD_{\mathcal{H}}$ such that $F_{\mathcal{H}}^{\oplus,v,e}(HD) \leq t$, for some $0 < t \leq t_{\text{max}}$. This is a feasible query because it is in LOGCFL, from Theorem 5.1.

From Theorem A.3, there is an $L^{LOGCFL}$ transducer $M''$ that computes a LOGCFL certificate of acceptance for the logspace alternating Turing machine $M$ that implements Algorithm threshold-$k$-decomp (see Appendix A). Note that any certificate for the computation of $M$ on inputs $\mathcal{H}$ and $w$ encodes a hypertree in $kNFD_{\mathcal{H}}$ having the (minimum) weight $w$.

Finally, the composition of $M'$ and $M''$ belongs to $L^{LOGCFL}$, from the closure under composition of $L^{LOGCFL}$ transducers [18]. □

From the closure properties of LOGCFL and $L^{LOGCFL}$, it is easy to see that computing a minimal NF hypertree decomposition is in $L^{LOGCFL}$ even if we consider the extension of smooth TAFs where the logspace Turing transducers for $\oplus, v,$ and $e$ may also query a LOGCFL oracle, that is, even if these functions belong to $L^{LOGCFL}$.

Moreover, these tractability results also hold for some further restrictions of the class of NF bounded-width hypertree decompositions, such as the bounded-width decompositions in reduced normal form, recently defined in [24].

6. Minimal decompositions and optimal query plans

In this section, we show an application of minimal weighted hypertree decompositions to query optimization. In this context, quantitative methods and structural decomposition methods have been completely separated worlds so far, since the structural properties of queries have been deeply investigated in the literature, but rarely used in practice.
However, many meaningful queries are quite long, but often acyclic or close to acyclic queries. Because of their sizes, it may happen that quantitative methods are unable to find optimal query plans for such queries and then take a large amount of time for answering them, though they are structurally simple and easily solvable. See, e.g., [29], for some recent interesting experiments with bounded treewidth queries.

As an example scenario, consider the data warehouse (DW) framework, in which a number of batch queries are executed on the reconciled operational database for populating or refreshing the DW. These queries acquire data of interest, rearrange them to comply with the multidimensional model, and possibly group them to the required granularity. Note that these queries are typically very different from OLAP queries. Indeed, while the latter queries are executed on star schemes (or similar simple schemes), these populating queries usually span several tables in the reconciled scheme in order to update both dimension and fact tables. Thus, they are often long queries involving many join operations. In this context, the choice of a good query-execution strategy is therefore particularly relevant, because the differences among the execution times of different query plans can be several orders of magnitude.

Recall that $\text{cost}_H(Q)$ is the TAP $F^+, \psi^*, e^*$ described in Example 4.3, and let $\text{cost}-k$-decomp the specialization of minimal-$k$-decomp implementing $\text{cost}_H(Q)$. Then, any \([\text{cost}_H(Q), kNFD_H(Q)]\)-minimal weighted hypertree decomposition, which can be computed by algorithm $\text{cost}-k$-decomp, represents an effective plan for evaluating $Q$ and is, in fact, an optimal plan according to the given cost model (and to the class of $k$-bounded NF hypertree decompositions). Algorithm $\text{cost}-k$-decomp has been implemented in C++, and is currently available on the web [34]. Note that the current version only works for conjunctive queries, which are equivalent to Select-Project-Join queries. Ongoing works focus on possible applications of hypertree decompositions to more general queries, and thus future implementations of $\text{cost}-k$-decomp will incorporate these extensions.

In order to better understand the algorithm $\text{cost}-k$-decomp, we next propose an explicative example, for the computation of an optimal query plan for the conjunctive query $Q_1$, defined as follows:

$$\text{ans} \leftarrow a(S, X, X', C, F) \land b(S, Y, Y', C', F') \land c(C, C', Z) \land d(X, Z) \land e(Y, Z) \land f(F, F', Z') \land g(X', Z') \land h(Y', Z') \land j(J, X, Y, X', Y').$$

Assume we have to evaluate this query on a database $\text{DB}$, whose quantitative statistics information are reported in Fig. 5, which shows for each atom $p$ occurring in $Q_1$, the number of tuples in the relation $\text{rel}(p)$ associated with $p$, denoted by $|p|$, and for each variable $X$ occurring in $p$, the selectivity of the attribute corresponding to $X$ in $\text{rel}(p)$, i.e., the number of distinct values $X$ takes on $\text{rel}(p)$. These data are obtained by means of the command ANALYZE TABLE in CommDB.

| atom | $|a|$ | $|b|$ | $|c|$ | $|d|$ | $|e|$ | $|f|$ | $|g|$ | $|h|$ | $|j|$ |
|------|------|------|------|------|------|------|------|------|------|
| atom a. | 4606 | S | X | X' | C | F |
| SELECTIVITY | 14 | 24 | 16 | 21 | 15 |
| atom b. | 2808 | S | Y | Y' | C' | F' |
| SELECTIVITY | 17 | 5 | 12 | 20 | 7 |
| atom c. | 1748 | C | C' | Z' |
| SELECTIVITY | 18 | 7 | 19 |
| atom d. | 3756 | X | Z |
| SELECTIVITY | 18 | 7 |
| atom e. | 3554 | Y | Z |
| SELECTIVITY | 21 | 13 |
| atom f. | 2892 | F | F' | Z' |
| SELECTIVITY | 20 | 7 | 6 |
| atom g. | 4573 | X' | Z' |
| SELECTIVITY | 22 | 16 |
| atom h. | 3390 | Y' | Z' |
| SELECTIVITY | 15 | 12 |
| atom j. | 4234 | J | X | Y | X' | Y' |
| SELECTIVITY | 18 | 8 | 18 | 22 | 10 |

Fig. 5. Cardinality and selectivity for relations in query $Q_1$. 
Figure 6 shows a minimal hypertree decomposition of width 2 for $Q_1$ computed by $\text{cost-}k\text{-decomp}$, if we fix the bound $k$ to 2. In the figure, each vertex $v$ has a new label marked by the symbol $\$\$ that represents the estimated cost for evaluating the subtree rooted at $v$. In particular, for each leaf $\ell$, this number will be equal to the estimate for the cost of the relational expression $E(\ell)$ (see Example 4.3); for the root $r$ of the hypertree, this number gives the estimated cost of the whole evaluation of $Q_1(\text{DB})$. In our example, this cost is 3,521,741.

Observe that $Q_1$ is a cyclic query, having hypertree width 2, thus there is no hypertree decomposition of width 1 for $Q_1$, and $k = 2$ is the lowest bound such that $\text{cost-}k\text{-decomp}$ is able to compute decomposition for $Q_1$. However, any value larger than 2 is feasible, and so we have run $\text{cost-}k\text{-decomp}$ with $k$ ranging from 2 to 5. For $k = 3$, we obtain the estimated cost 1,373,879, while for both $k = 4$ and $k = 5$ we obtain 854,867. Figure 7 shows a minimal hypertree decomposition computed by $\text{cost-}k\text{-decomp}$, with $k = 4$.

It turns out that, even if the hypertree width of $Q_1$ is 2, for the given quantitative information on $\text{DB}$ and the cost model we have chosen, the bound 4 leads to the best query plans. This is not surprising, as a larger bound $k$ allows us to explore a larger set of decompositions.

In order to assess the efficacy of the query plans generated by $\text{cost-}k\text{-decomp}$, we have replaced the query planner of CommDB by $\text{cost-}k\text{-decomp}$, and compared the execution time obtained by the new method vs CommDB execution time, on a number of benchmark queries. We remark that these experiments are far from being a full comparison between our prototype and CommDB. Rather, they aim at checking whether Algorithm $\text{cost-}k\text{-decomp}$ might have also a practical impact and may speed-up the evaluation of database queries having structural properties to be exploited. For a systematic analysis of the performance of $\text{cost-}k\text{-decomp}$, we refer the interested reader to [13], where new results on experiments carried out with both CommDB and PostgreSQL are reported. In particular, since the latter DBMS is open-source, a query optimizer based on weighted hypertree decompositions has been implemented directly inside the system.

We next present some experiments carried out by using CommDB as evaluation engine. Each query used for the comparison is evaluated over CommDB twice:

(i) by using its internal optimization module, allowing the exploitation of all the quantitative information available for the database (by means of the command $\text{ANALYZE TABLE}$, invoked for each table);
(ii) with the query plan generated by $\text{cost-}k\text{-decomp}$, whose execution is enforced by supplying a suitable translation in terms of views and hints ($\text{NO\_MERGE, ORDERED}$) to CommDB. Experiments with $\text{cost-}k\text{-decomp}$ have been conducted for $k$ ranging over $(2..5)$.

In particular, we used the algorithm $\text{cost-}k\text{-decomp}$ for computing a cost-based hypertree decomposition $HD$ whose associated query plan for answering $Q_1$ on $DB$ guarantees the minimum estimated cost over all (normal form) $k$-bounded hypertree decomposition of $Q_1$. We also report experiments for queries $Q_2$ and $Q_3$. Query $Q_2$ consists of 8 atoms and 9 distinct variables, and query $Q_3$ is made of 9 atoms, 12 distinct variables, and 4 output variables. All of these queries have hypertree width 2. In these experiments, we used randomly generated synthetic data. Moreover, we did not allow indices on the relations, in order to focus just on the less-physical aspects of the optimization task. The experiments have been performed on a 1600MHz/256MB Pentium IV machine running Windows XP Professional.

The results of the experiments are displayed in Fig. 8. On all considered queries, the evaluation of the query plans generated by our approach is significantly faster than the evaluation which exploits the internal query optimization module of CommDB. For query $Q_1$, we report the costs of constructing the plan and the evaluation time (expressed in seconds), for different values of $k$ ($2..5$). Figure 8(A) displays the ratio between the query evaluation times (CommDB vs $\text{cost-}k\text{-decomp}$) over a database of 1500 data tuples. It is worth noting that a higher value of $k$ permits a larger number of hypertree decompositions to be considered, and can therefore allow a better plan to be generated; but it obviously causes a computational overhead due to a larger search space to be explored by $\text{cost-}k\text{-decomp}$. In Fig. 8(A) we observe that the ratio between the query evaluation times (CommDB vs $\text{cost-}k\text{-decomp}$) increases for higher values of $k$ from $k = 2$ to $k = 4$, where the overhead for the plan computation is less than the benefit obtained thanks to the improved plan quality. For $k = 5$, instead, the overhead for plan computation overcomes the query evaluation benefit (as in fact there is no benefit at all, because 4 already leads to the minimum possible weight for the hypertree decompositions of this query). Thus, the best overall performance is obtained by $\text{cost-}k\text{-decomp}$ for $k = 4$, which seems empirically a good bound to be used in practice for queries with less than 10 atoms. Queries $Q_2$ and $Q_3$ show a similar behavior of this ratio, and we report in Fig. 8(B) only the (absolute) execution times for CommDB and $\text{cost-}k\text{-decomp}$ for $k = 3$, over a database of 1500 tuples.

We conclude this section with a small remark on a technical issue. As explained in [17], for answering queries we need complete decompositions, where each hyperedge is strongly covered (see Definition 2.1), i.e., each query atom occurs entirely at least once. However, NF decompositions are not necessarily complete, and there are hypergraphs having no complete NF hypertree decompositions at all. Nevertheless, from any incomplete (NF) hypertree decomposition $HD$ of a hypergraph $H$ we can compute easily a complete hypertree decomposition [17]: For any hyperedge $h$ that is covered by a vertex $r$ in $HD$, but not strongly covered by any vertex in $HD$, we add a child $s$ to $r$ with $\lambda(s) = \{h\}$.
and $\chi(s) = h$. Note that the resulting decomposition, say $HD'$, is not in normal form because for such a pair of vertices $r$ and $s$, $\chi(s) \subseteq \chi(r)$ holds. Indeed, in this case, $HD'$ in normal form would entail $\chi(T_s) = C_r \cup \chi(s)$, where $C_r$ is the $[r]$-component associated with $s$, as required in Definition 2.2. Note that, by definition of $[r]$-component, $C_r \cap \chi(r) = \emptyset$ and hence $C_r \cap \chi(s) = \emptyset$, too. It follows that $C_r = \emptyset$, because $s$ is a leaf and thus $\chi(T_s) = \chi(s)$. However, this is impossible, as the components cannot be empty.

Of course, we can just look for a minimal NF decomposition $HD$ and then use the transformed complete decomposition $HD'$ for query answering. However, in this way we are not sure that the final decomposition $HD'$ is still minimal. Indeed, in principle, the above transformation can lead to decompositions with a bad weight, thus losing minimality. To overcome this problem, we have to force the algorithm to compute directly complete minimal decompositions. This may be guaranteed by adding a fresh variable to each query atom. Algorithm $\text{cost-}k\text{-decomp}$ also deals with these issues, by suitably modifying the query (and consequently the related data information), and then filtering such fresh variables, in its output phase.

### 7. Proof of correctness of algorithm $\text{minimal-}k\text{-decomp}$

In this section, we show that algorithm $\text{minimal-}k\text{-decomp}$ described in Section 4 is sound and complete. We need some preliminary definitions and results.

Given a hypertree decomposition in normal form $HD = (T, \chi, \lambda)$ of a hypergraph $\mathcal{H}$, for each vertex $s$ of $T$, we denote by $\text{treecomp}(s)$ the set

- $\text{treecomp}(s) = \text{var}(\mathcal{H})$, if $s$ is the root of $T$, otherwise
- $\text{treecomp}(s) = C_r$ where $C_r$ is the unique $[r]$-component such that $\chi(T_s) = C_r \cup (\chi(r) \cap \chi(s))$ and $s$ is a child of $r$ in $T$.

Recall that the term $[r]$-component is used as a synonymous of $[\chi(r)]$-component as long as no confusion arises. The following lemma points out some properties of hypertree decompositions in normal form that will be useful in the following.

**Lemma 7.1.** Let $\mathcal{H}$ be a hypergraph such that $\text{hw}(\mathcal{H}) \leq k$, and let $HD = (T, \chi, \lambda)$ be a hypertree decomposition in $k\text{NFD}_\mathcal{H}$. Then, for any vertex $s \in \text{vertices}(T)$, the following properties hold:

1. $\chi(T_s) = \text{treecomp}(s) \cup \chi(s)$;
2. if $s$ is a child of $r$ in $T$, then $\text{treecomp}(s) \subseteq \text{treecomp}(r)$;
3. each $[\chi(s)]$-component $C$ such that $C \subseteq \text{treecomp}(s)$ is also a $[\text{var}(\lambda(s))]$-component;
4. each $[\text{var}(\lambda(s))]$-component $C$ such that $C \subseteq \text{treecomp}(s)$ is also a $[\chi(s)]$-component;
5. if $s$ is a leaf of $T$, then $\text{treecomp}(s) \subseteq \chi(s)$;
6. if $s$ is a non-leaf node in $T$, then the set $C = \{C \subseteq \text{treecomp}(s) \mid C$ is an $[s]$-component$\}$ is a partition of $\text{treecomp}(s) - \chi(s)$;
7. for any $[s]$-component $C$ such that $C \subseteq \text{treecomp}(s)$ there is a child, say $s_C$, of $s$ in $T$ such that $C = \text{treecomp}(s_C)$.

**Proof.**

1. If $s$ is the root of $T$, the property follows by observing that $\text{treecomp}(s) = \text{var}(\mathcal{H})$. Assume that $s$ is the child of a node $r$ of $T$. Clearly, $\text{treecomp}(s) \cup (\chi(s) \cap \chi(r))$ is contained $\text{treecomp}(s) \cup \chi(s)$; hence, it remains to show that the converse holds as well, i.e., $\text{treecomp}(s) \cup \chi(s) \subseteq \text{treecomp}(s) \cup \chi(r)$. To this aim, observe that (i) $\text{treecomp}(s) \cup \chi(s) \subseteq \text{treecomp}(s) \cup \chi(T_s)$ since $\chi(s) \subseteq \chi(T_s)$, and (ii) $\text{treecomp}(s) \cup \chi(T_s) \subseteq \text{treecomp}(s) \cup \chi(s)$ since $\chi(T_s) = \text{treecomp}(s) \cup \chi(s) \cap \chi(r)$ by condition (1) of decompositions in normal form.

2. Recall, by definition, that $\text{treecomp}(s)$ is an $[r]$-component such that $\chi(T_s) = \text{treecomp}(s) \cup (\chi(s) \cap \chi(r))$. Since $\chi(T_s) \subseteq \chi(T_r)$, then $\text{treecomp}(s) \cup (\chi(s) \cap \chi(r)) = \chi(T_s) \subseteq \chi(T_r) = \text{treecomp}(r) \cup \chi(r)$ by property (1) above. It follows that $\text{treecomp}(s)$ is a subset of $\text{treecomp}(r) \cup \chi(r)$. The result finally follows since $\text{treecomp}(s) \cap \chi(r) = \emptyset$, because of the fact that $\text{treecomp}(s)$ is an $[r]$-component.

3. Let $C$ be a $[\chi(s)]$-component, i.e., a maximally $[\chi(s)]$-connected non-empty set of variables. By condition (4) of decompositions in normal form, we have that $\chi(s) = \text{var}(\text{edges}(\text{treecomp}(s))) \cap \text{var}(\lambda(s))$; consequently, $C \cap \text{var}(\text{edges}(\text{treecomp}(s))) \cap \text{var}(\lambda(s)) = \emptyset$. Observe now that, by hypothesis, $C$ is contained in $\text{treecomp}(s)$; hence,
Let \( \var(\lambda(s)) = \emptyset \) holds as well. This relationship, together with the fact that \( \chi(s) \subseteq \var(\lambda(s)) \), implies that any \( [\var(\lambda(s))] \)-connected set which is a maximally \( [\chi(s)] \)-connected set is also a maximally \( [\var(\lambda(s))] \)-connected set as well.

(4) Let \( C \) be a \([\var(\lambda(s))] \)-component contained in \( \text{treecomp}(s) \). Since \( \chi(s) \subseteq \var(\lambda(s)) \) by condition (3) of hypertree decompositions, any \([\var(\lambda(s))] \)-component \( C \) is also a \([\chi(s)] \)-component of variables. It follows that there is a \([\chi(s)] \)-component \( C' \) with \( C' \supseteq C \). We show that in fact \( C = C' \). Indeed, if there is some \( Y \in C' - C \), then \( Y \) should be connected to some \( X \in C \) through some \([\chi(s)] \)-path that is not a \([\var(\lambda(s))] \)-path. It follows that this path should contain a variable \( Y' \) in the boundary \( \var(edges(C)) \) such that \( Y' \in \lambda(s) \) and \( Y' \notin \chi(s) \). However, this is impossible, since by condition (4) of decompositions in normal form, \( \chi(s) = \var(edges(\text{treecomp}(s))) \cap \var(\lambda(s)) \), and thus we get \( Y' \in \chi(s) \).

(5) Since \( s \) is a leaf of \( T \), then \( \chi(T_s) = \chi(s) \). Therefore, by property (1) above, we have that \( \chi(s) = \text{treecomp}(s) \cup \chi(s) \) and the result follows.

(6) Consider an \([s] \)-component \( C \) contained in \( \text{treecomp}(s) \). We preliminary show that \( C \subseteq \text{treecomp}(s) - \chi(s) \). Assume for the sake of contradiction that there is \( Y \in (C - \text{treecomp}(s) - \chi(s)) \). Since \( C \subseteq \text{treecomp}(s) \), it follows that \( Y \) is in \( \chi(s) \), thereby contradicting \( C \cap \chi(s) = \emptyset \) (that holds because \( C \) is an \([s] \)-component). Finally, disjointness and covering of all the variables in \( \text{treecomp}(s) - \chi(s) \) follow by definition of component.

(7) By property (1) above, \( \chi(T_s) = \text{treecomp}(s) \cup \chi(s) = \text{treecomp}(s) - \chi(s) \cup \chi(s) \). Hence, by property (6), the following relation holds:

\[
\chi(T_s) = \bigcup_{\{C \mid C \text{ is an } [\chi(s)] \text{-component in } \text{treecomp}(s)\}} C \cup \chi(s).
\]

Let \( C \) be a \([\chi(s)] \)-component contained in \( \text{treecomp}(s) \); since \( \chi(T_s) = \text{treecomp}(s) \cup \chi(s) \) by property (1) above, then \( C \) is contained in \( \chi(T_s) \) as well. Assume now, for the sake of contradiction, that there is no child \( s_C \) of \( s \) such that \( \text{treecomp}(s_C) = C \). Then, since the union above is disjoint, it must be the case that \( \text{treecomp}(s_C) \subseteq \chi(s) \) contradicting \( \text{treecomp}(s_C) \cap \chi(s) = \emptyset \) (holding by definition of \( \text{treecomp}(s_C) \)). □

In order to prove the correctness of algorithm \( \text{minimal}-k\text{-decomp} \), we preliminarily consider a simplified version of this algorithm that computes hypertree decompositions in normal form of width at most \( k \), but not necessarily minimal.

**Definition 7.2 (Algorithm \( k\text{-decomp} \)).** Define algorithm \( k\text{-decomp} \) as the modification of algorithm \( \text{minimal}-k\text{-decomp} \) where the instructions marked by (* select *) in Fig. 2 perform a (random) selection of any incoming node \( p \in \text{incoming}(q) \) (for a given subproblem \( q \in N_{\text{sub}} \)), rather than a minimum-weighted node in \( \text{incoming}(q) \).

Clearly enough, for a given hypergraph \( \mathcal{H} \), the set of decompositions that may be computed by \( \text{minimal}-k\text{-decomp} \) (no matter of the \( \text{TAF} \) provided in input) is a subset of the set of decompositions that may be computed by \( k\text{-decomp} \). We net prove that this latter set is exactly the set of all the hypertree decompositions in normal form of \( \mathcal{H} \) having width at most \( k \).

**Theorem 7.3 (Completeness of \( k\text{-decomp} \)).** Given a hypergraph \( \mathcal{H} \) such that \( \text{hw}(\mathcal{H}) \leq k \), for any hypertree decomposition \( \text{HD} \in knFD_{\mathcal{H}} \), there is a run of \( k\text{-decomp} \) that outputs \( \text{HD} \).

**Proof.** Let \( \text{HD} = (T, \chi', \lambda') \) be a hypertree decomposition in \( knFD_{\mathcal{H}} \). We show that there is a run of \( k\text{-decomp} \) that outputs \( \text{HD} \). Let us consider the graph \( \mathcal{CG} = (N_{\text{sol}} \cup N_{\text{sub}}, A, \text{weight}) \) build by \( k\text{-decomp} \) in the first fragment (* Build the Candidates Graph *) before any edge or node is deleted in the step (* Evaluate the Candidates Graph *).

Consider the function \( f \) defined over the vertices of \( T \) such that for any vertex \( v \) of \( T \), \( f(v) = (\lambda'(v), \text{treecomp}(v)) \).

We point out the following properties of \( f \):

- \( P_1 \): Let \( v \) be a vertex in \( T \). Then, \( f(v) \) is in \( N_{\text{sol}} \), \( \lambda(f(v)) = \lambda'(v) \), and \( \chi(f(v)) = \chi'(v) \), where \( \lambda \) and \( \chi \) are the labelings assigned the nodes in \( \mathcal{CG} \) by \( \text{cost}-k\text{-decomp} \).
Proof. If $v$ is the root of $T$, then $f(v) = (\lambda'(v), \var(H))$ is clearly in $N_{sol}$, by the construction of the candidates graph $CG$. Otherwise, i.e., if $v$ is not the root of $T$ and if $r$ is the parent of $v$ in $T$, we can show that $f(v) = (\lambda'(v),\treecomp(v)) = (\lambda'(v), C_r)$ is in $N_{sol}$. Indeed, $f(v)$ is such that:

1. $C_r$ is a $[\var(\lambda'(r))]$-component. In fact, $C_r$ coincides, by definition, with $\treecomp(v)$ and is therefore a $[\chi'(r)]$-component. By property (2) in Lemma 7.1, $C_r = \treecomp(v) \subseteq \treecomp(r)$. Hence, we can apply property (3) in Lemma 7.1, and conclude that $C_r$ is a $[\var(\lambda'(r))]$-component as well.

2. $\var(\lambda'(v)) \cap C_r \neq \emptyset$. In fact, this is precisely condition (2) of decompositions in normal form.

3. For each $h \in \lambda'(v)$, $h \cap \var(edges(C_r)) \neq \emptyset$. In fact, this is precisely condition (3) of decompositions in normal form.

The above properties suffice for concluding that $f(v)$ is included in $N_{sol}$ by algorithm $k-decomp$. To conclude the proof, observe that $k-decomp$ sets $\lambda(f(v))\to\lambda'(v)$, and $\chi(f(v))\to\var(\lambda'(v)) \cap \var(edges(C_r))$ that, in fact, corresponds to $\chi'(v)$ due to condition (4) of decompositions in normal form.

- **P2:** For any pair of vertices $v$ and $r$ in $T$ such that $r$ is the parent of $v$, there is $q \in N_{sub}$ such that both $q \in \incoming(f(r))$ and $f(v) \in \incoming(q)$ in the set of arcs $A$.

Proof. Let $v$ and $r$ be vertices such that $r$ is the unique parent of $v$. From property $P_1$, the vertex $f(v) = (\lambda'(v), \treecomp(v))$ is in $N_{sol}$. Then, letting $q = (\lambda'(r), \treecomp(v))$, it is easy to see by construction of the candidates graph that $q$ is in $N_{sub}$ and that $f(v)$ is in $\incoming(q)$. Consider now the node $f(r) = (\lambda'(r), \treecomp(r))$. From property (2) in Lemma 7.1, it holds that $\treecomp(v) \subseteq \treecomp(r)$. Therefore, $q$ is included in the set $\incoming(f(r))$ in the (* Connect its subproblems *) step.

- **P3:** For any leaf $s$ of $T$, $\incoming(f(s)) = \emptyset$.

Proof. For the sake of contradiction, assume that there is a vertex $c$ in $N_{sub}$ such that $c \in \incoming(f(s))$ holds in $A$. Then, $c$ must be of the form $(\lambda'(s), C'')$, where $C'' \subseteq \treecomp(s)$ is a $[\var(\lambda'(s))]$-component; moreover, from property (4) in Lemma 7.1, $C''$ is a $[\chi'(s)]$-component as well. Consider now property (5) in Lemma 7.1 to conclude that $\treecomp(s) \subseteq \chi'(s)$ holds, and hence that $C''$ is contained in $\chi(s)$, too. However, this contradicts the fact that $C''$ is a $[\chi'(s)]$-component.

- **P4:** Let $v$ be a vertex in $T$. For any node $p \in \incoming(f(v))$ in $A$, there exists a vertex $s_C$ child of $v$ in $T$ such that $f(s_C) = \in\incoming(p)$.

Proof. Recall that $f(v)$ has the form $(\lambda'(v), \treecomp(v))$. Any node $p \in \incoming(f(v)) \in N_{sub}$ has the form $(\lambda'(v), C')$ where $C'$ is a $[\var(\lambda'(v))]$-component contained in $\treecomp(v)$. Due to property (4) in Lemma 7.1, $C'$ is a $[\chi'(v)]$-component as well. Hence, we can apply property (7) in the same lemma for concluding that there exists a child, say $s_C$, of $v$ such that $\treecomp(s_C) = C'$. Then, the result follows by noticing that $f(s_C) = (\lambda'(s_C), C')$ and, hence, $f(s_C) \in \incoming(p)$ holds in $A$.

By combining properties $P_1$, $P_2$, $P_3$ and $P_4$, we conclude that the candidates graph $CG$ after the (* Evaluate the Candidates Graph *) step cannot be empty. Indeed, vertices corresponding to leaves in $T$ are leaves in $A$ as well, and all other internal nodes are mapped by means of a function $f$ that, in fact, preserves the childhood relationship. Moreover, the elimination of some nodes cannot affect the nodes mapped from $T$. Hence, the algorithm does not output failure, and after the (* Evaluate the Candidates Graph *) step the graph $CG$ contains $HD$.

In order to complete the proof, we describe a run of algorithm $k-decomp$ that outputs $HD$. Consider the following choices for the (* select *) steps:

- For the first (* select *), we select the node $p \in \incoming((\emptyset, \var(H)))$ such that $p = (S, \var(H))$, with $S = \lambda(root(T))$. 
• For any other (* select *) exploited for selecting an incoming node of \( q = (R, C) \), if \( R \) is the label \( \lambda'(r) \) of some vertex \( r \) in \( T \), and if \( r \) has a child \( s \) such that \( C \) is the unique \([\text{var}(R)]\)-component such that \( \chi'(T_s) = C \cup (\chi'(s) \cap \chi'(r)) \), then we choose \( p = (S, C) \) with \( S = \lambda'(s) \).

Note that, from the above properties, all these choices are feasible, and the resulting decomposition is equal to \( HD \). \( \square \)

We next prove that \( k\text{-decomp} \) is sound, i.e., that a given hypergraph \( \mathcal{H} \) with \( hw(\mathcal{H}) \leq k \), each run of \( k\text{-decomp} \) outputs a \( k \)-width hypertree decompositions in normal form. To this aim, we preliminary point out some properties of algorithm \( k\text{-decomp} \). Recall that any vertex \( s \) in the output of \( \text{cost-}k\text{-decomp} \) is a pair \( s = (S, C) \) where \( S \) is a \( k \)-vertex and \( C \) is any \([\text{var}(R)]\)-component (for some \( k \)-vertex \( R \)). Recall also that \( s \) is associated with the two sets \( \lambda(s) \) and \( \chi(s) \).

**Lemma 7.4.** Let \( \mathcal{H} \) be a hypergraph, and assume that \( k\text{-decomp} \) outputs \( HD = (T, \chi, \lambda) \). Then, for any vertex \( s = (S, C) \) in \( T \), the following conditions hold:

1. if \( s \) is a leaf of \( T \), then there is no \([\text{var}(S)]\)-component \( C' \) such that \( C' \subseteq C \);
2. if \( s \) is a leaf of \( T \), then \( \chi(s) = \text{var(edges}(C)) \);
3. for any \( h \in \text{edges}(C) \), either \( h \subseteq \chi(s) \) or there is a \([\text{var}(S)]\)-component \( C' \subseteq C \) such that \( h \in \text{edges}(C') \);
4. \( \chi(T_s) = \text{var(edges}(C)) \);
5. for any \([\text{var}(S)]\)-component \( C' \subseteq C \), there is one vertex \( s_C = (S', C') \) that is a child of \( s \) in \( T \), and all children of \( s \) in \( T \) are of this form.

**Proof.** (1) Let \( s = (S, C) \) be a leaf in \( T \), and assume, for the sake of contradiction, that there is a \([\text{var}(S)]\)-component \( C' \) such that \( C' \subseteq C \). Then, in the candidate graph \( CG = (N_{sol} \cup N_{sub}, A) \), built by \( k\text{-decomp} \) on input \( \mathcal{H} \), there exists a node \( q = (S, C') \in N_{sub} \) such that \( q \in \text{incoming}(s) \) holds in \( A \)—see the step (* Connect its subproblems *). Now, we consider two cases. In the case there is a node \( q' \in N_{sol} \) such that \( q' \in \text{incoming}(q) \) holds in \( A \), then \( s \) cannot be a leaf. Contradiction. Otherwise, i.e., in the case \( q \) is a leaf in \( A \), then both \( q \) and \( s \) are removed in the step (* Evaluate the Candidate Graph *)), contradicting \( s \) to be in the output \( T \).

(2) Let \( s = (S, C) \) be a leaf of \( T \), we show that \( \chi(s) = \text{var(edges}(C)) \). To this aim, consider an edge \( h \in \text{edges}(C) \). Since, \( \chi(s) \) is equal by construction to \( \text{var(edges}(C)) \cap \text{var}(S) \), it is sufficient to prove that for each \( X \in h \), \( X \) is in \( \text{var}(S) \) too. Let \( CG = (N_{sol} \cup N_{sub}, A) \) be the candidates graph, and let \( q = (R, C) \in N_{sub} \) be the node in \( CG \) such that \( s \in \text{incoming}(q) \) holds in \( A \) and such that, in some (* select *) step of \( k\text{-decomp} \), \( s \) is selected as a node belonging to \( \text{incoming}(q) \). In the case \( X \in \text{var}(R) \), we easily derive that \( X \) is in \( \text{var}(S) \), by exploiting the property of the construction of candidates graph guaranteeing that \( \text{var(edges}(C)) \cap \text{var}(R) \subseteq \text{var}(S) \).

Hence, it remains to consider the case in which \( X \notin \text{var}(R) \). In this case, we first show that \( X \) is in \( C \). Indeed, by contradiction that \( X \) is not in \( C \). Since \( X \) is in \( \text{var(edges}(C)) \) and \( C \) is a \([\text{var}(R)]\)-component by construction, then \( C \cup X \) is a \([\text{var}(R)]\)-component as well, contradicting the maximality of \( C \). To conclude the proof assume, for the sake of contradiction, that \( X \notin \text{var}(S) \) holds. Then, \( X \) is in some \([\text{var}(S)]\)-component, say \( C' \). Moreover, from \( \text{var(edges}(C)) \cap \text{var}(R) \subseteq \text{var}(S) \) we derive that \( C' \) is such that \( C' \subseteq C \). Indeed, for each node \( Y \in \text{var(edges}(C)) \), the existence of a \([\text{var}(S)]\)-path from \( X \) to \( Y \) implies the existence of a \([\text{var}(R)]\)-path from \( X \) to \( Y \) as well. This is a contradiction with property (1) above, stating that, for any leaf \( s = (S, C) \) in \( T \), there is no \([\text{var}(S)]\)-component \( C' \) such that \( C' \subseteq C \).

(3) Let \( s = (S, C) \) be a vertex in \( T \). We prove by structural induction over trees that, for each \( h \in \text{edges}(C) \), either \( h \subseteq \chi(s) \) or there is a \([\text{var}(S)]\)-component \( C' \subseteq C \) such that \( h \in \text{edges}(C') \).

If \( s \) is a leaf, we can exploit property (2) above and derive that \( h \subseteq \chi(s) \). Otherwise, i.e., in the case \( s \) is an internal node of \( T \), let us assume that the property holds in all subtrees rooted at the children of \( s \) in \( T \). Let \( h \) be an edge in \( \text{edges}(C) \). In the case there is a \([\text{var}(S)]\)-component \( C_j \subseteq C \) such that \( h \in \text{edges}(C_j) \) the statement trivially holds. We thus assume that \( h \cap C_j = \emptyset \), for each \([\text{var}(S)]\)-component \( C_j \subseteq C \). We next show that, for each \( X \in h \), it is the case that \( X \in \text{var}(S) \). Let \( q = (R, C) \in N_{sub} \) be the node in the candidate graph such that in the (* select *) step \( s \) is selected as a node in \( \text{incoming}(q) \). We distinguish two situations. If \( X \in \text{var}(R) \), then by construction of candidate graph we have that \( \text{var(edges}(C)) \cap \text{var}(R) \subseteq \text{var}(S) \), and the result easily follows. Otherwise, if \( X \notin \text{var}(R) \), then \( X \in C \),
as discussed in point (2) above. It follows that \( X \in \text{var}(S) \), otherwise it should belong to some \([\text{var}(S)]\)-component contained in \( C \), contradicting the assumption on \( h \).

(4) We prove by structural induction over trees that for each vertex \( s = (S, C) \) in \( T \), \( \chi(T_s) = \text{var(edges}(C)) \) holds. If \( s \) is a leaf of \( T \), the result derives from property (2) above, since \( \chi(T_s) = \chi(s) \). Hence, let \( s \) be an internal node in \( T \) and \( t_1 = (T_1, C_1), \ldots, t_n = (T_n, C_n) \) be its children. Then, \( \chi(T_s) = \chi(s) \bigcup_{i=1}^{n} \chi(T_{t_i}) \), and by the inductive hypothesis, \( \chi(T_{t_i}) = \chi(s) \bigcup_{i=1}^{n} \text{var(edges}(C_i)) \). By construction in \( k\text{-decomp} \) \( C_i \subseteq C \), for each \( C_i \), and \( \chi(s) = \text{var}(S) \cap \text{var(edges}(C)) \). Thus, we have that \( \chi(T_s) = \text{var}(S) \cap \text{var(edges}(C)) \bigcup_{i=1}^{n} \text{var(edges}(C_i)) \subseteq \text{var}(S) \cap \text{var(edges}(C)) \subseteq \text{var(edges}(C)) \). The fact that the inclusion cannot be properly easily derives from property (3) above.

(5) Let \( s = (S, C) \) be a vertex in \( T \). By construction of the candidates graph in \( k\text{-decomp} \), it is easy to see that the child of \( s \) in \( T \) are all the possible \([\text{var}(S)]\)-components contained in \( C \). The result follows by observing that, if some of these children do not have an incoming node, then it is removed in the evaluation step, and subsequently \( s \) is removed as well. \( \square \)

Given a hypertree decomposition \( HD = (T, \chi, \lambda) \) and a vertex \( v \) in \( T \), we denote by \( HD_v \) the decomposition \( (T_v, \chi_v, \lambda_v) \), where \( T_v \) is the subtree of \( T \) rooted at \( v \), and \( \chi_v \) and \( \lambda_v \) are the restrictions of \( \chi \) and \( \lambda \) to the nodes of \( T_v \). Moreover, given a hypergraph \( \mathcal{H} \) and a set of variables \( V \subseteq \text{var(\mathcal{H})} \), we denote by \( \mathcal{H}[V] \) the hypergraph containing all the edges whose variables are all in \( V \). Given a node of the candidates graph \( p = (P, C) \in N_{\text{sol}} \cup N_{\text{sub}} \), we denote by \( \mathcal{H}_p \) the hypergraph \( \mathcal{H}[\text{var(edges}(C))] \).

Finally, if \( k \geq 1 \) is a fixed constant and \( q = (R, C) \in N_{\text{sub}} \), we denote by \( k\text{NFD}_q^h \) the set of all hypertree decompositions \( HD \) in normal form of \( \mathcal{H}_q \) having width at most \( k \) and such that \( \text{var(edges}(C)) \cap \text{var(R)} \subseteq \chi(\text{root}(HD)) \). We say that these decompositions are feasible with respect to the subproblem \( q \).

Recall that, by construction, every vertex \( s = (S, C) \) selected to be part of \( T \) belongs to \( N_{\text{sol}} \) in the candidates graph \( CG \), and that all these vertices (including the root of \( T \)) have a vertex \( q = (R, C) \) in \( N_{\text{sub}} \) to which they are connected in \( CG \). Note that \( C \) is the same \([\text{var}(R)]\)-component in \( s \) and \( q \), and thus \( \mathcal{H}_s = \mathcal{H}_q \).

**Lemma 7.5.** Let \( \mathcal{H} \) be a hypergraph, and assume that \( k\text{-decomp} \) outputs \( HD = (T, \chi, \lambda) \). Let \( s \) be a vertex in \( T \) and \( q \in N_{\text{sub}} \) a node of the candidates graph \( CG \) such that \( s \in \text{incoming}(q) \). Then, \( HD_s \in k\text{NFD}_q^h \).

**Proof.** Observe that, by definition, the root of \( HD_s \) is \( s \), and thus the feasibility of this decomposition with respect to \( q \) is guaranteed by the construction of the arcs of \( CG \) in Algorithm \( k\text{-decomp} \). It remains to show that \( HD_s \) is an NF hypertree decomposition of \( \mathcal{H}_s = \mathcal{H}_q \) having width is at most \( k \).

We first prove by structural induction over trees that, for each vertex \( v = (V, C) \) in \( T \), \( HD_v = (T_v, \chi_v, \lambda_v) \) is a hypertree decomposition of the hypergraph \( \mathcal{H}[\text{var(edges}(C))] \).

**Basis:** Let \( s = (S, C) \) be a leaf of \( T_s \). We show that \( HD_s = (T_s, \chi_s, \lambda_s) \) is a decomposition of \( \mathcal{H}[\text{var(edges}(C))] \), i.e., that the following conditions (in the definition of hypertree decomposition) are satisfied:

1. For each edge \( h \in \text{edges}(\mathcal{H}[\text{var(edges}(C))]) \), there is \( p \in \text{vertices}(T_s) \) such that \( h \subseteq \chi(p) \).
2. For each \( Y \in \text{var(HD}_s) \), the set \( \{ p \in \text{vertices}(T_s) \mid Y \subseteq \chi(p) \} \) is a connected subtree of \( T_s \).
3. \( \chi(s) \subseteq \var(\lambda(s)) \).
4. \( \var(\lambda(s)) \cap \chi(T_s) \subseteq \chi(s) \).

**Induction Step:** Let \( s = (S, C) \) be a non-leaf vertex in \( T_v \) and let \( t_1 = (T_1, C_1), \ldots, t_n = (T_n, C_n) \) be its children, where each \( HD_{t_i} \) is a decomposition of the hypergraph \( \mathcal{H}[\text{var(edges}(C_i))] \), by the inductive hypothesis. We show that
$HD_r$ is a decomposition of $\mathcal{H}[\text{var}(\text{edges}(C))]$ as well. Indeed, all the properties of the hypertree decomposition are satisfied:

(1) For each edge $h \in \text{edges}(\mathcal{H}[\text{var}(\text{edges}(C))])$, there is $p \in \text{vertices}(T_s)$ such that $h \subseteq \chi(p)$.

Let $h$ be an edge in $\mathcal{H}[\text{var}(\text{edges}(C))]$. We distinguish two scenarios. In the case there exists $t_i$ such that $h \in \mathcal{H}[\text{var}(\text{edges}(C_j))]$, then the result follows by inductive hypothesis since $T_i$ is a hypertree decomposition of $\mathcal{H}[\text{var}(\text{edges}(C_i))]$. Indeed, by definition of hypertree decomposition, there is a vertex $p_i$ in $\text{vertices}(T_s)$ with $h \subseteq \chi(p_i)$.

Assume, now, that $h$ is in $\text{edges}(\mathcal{H}[\text{var}(\text{edges}(C))]) - \bigcup_{i=1}^{n} \text{edges}(\mathcal{H}[\text{var}(\text{edges}(C_i)))$. Notice that, by definition, $\text{edges}(\mathcal{H}[\text{var}(\text{edges}(C_i))]$ coincides with $\text{edges}(C_i)$, for each $C_i$. Hence, $h$ does not belong to the set $\bigcup_{i=1}^{n} \text{edges}(C_i)$. Moreover, by point (5) in Lemma 7.4, it is the case that for each $[\text{var}(S)]$-component $C' \subseteq C$, there is a child of $s$ in $T$, say $t_j$, such that $C' = C_j$. It follows that $h$ does not belong to the edges of any $[\text{var}(S)]$-component contained in $C$ as well. Thus, we can apply point (3) in Lemma 7.4 and derive that the only possibility is that $h \subseteq \chi(s)$.

(2) For each $Y \in \text{var}(HD_r)$, the set $\{p \in \text{vertices}(T_s) | Y \in \chi(p)\}$ is a connected subtree of $T_s$.

Recall that by inductive hypothesis, for each $Y \in \text{var}(HD_r)$, the subgraph of $T_s$ induced over the vertices whose labeling contains $Y$ is a connected subtree. Then, to prove the claim it suffices to show that for each variable $Y$ contained in the labeling of two subtrees, say $T_l$ and $T_j$, $Y$ is in $\chi(s)$, $\chi(t_i)$ and $\chi(t_j)$. To this aim, we preliminary notice that since $C_i$ and $C_j$ are both $[\text{var}(S)]$-components contained in $C$ (by point (5) in Lemma 7.4), then $Y$ must belong to $\text{var}(S)$. Otherwise, i.e., if $Y \notin \text{var}(S)$, then it is the case that $C_i = C_j$ by definition of components. Then, it follows that $Y$ must be contained in $\chi(s)$. Indeed, by point (4) in Lemma 7.4, $Y$ is in $\text{var}(\text{edges}(C)) = \chi(T_i)$, and $\chi(s) = \text{var}(\text{edges}(C)) \cap \text{var}(S)$ by its construction in $k-$decomp.

To conclude the proof, we show that $Y$ is also in both $\chi(t_i)$ and $\chi(t_j)$. Indeed, $k-$decomp appends $t_i$ (respectively $t_j$) as a child of $s$ only if $\text{var}(\text{edges}(C)) \cap \text{var}(S) \subseteq \text{var}(T_i)$ (respectively $\text{var}(\text{edges}(C)) \cap \text{var}(S) \subseteq \text{var}(T_j)$).

Then, since $Y$ is in $\text{var}(\text{edges}(C)) \cap \text{var}(S)$, we have that $Y$ is in both $\text{var}(T_i)$ and $\text{var}(T_j)$. Hence, to show that $Y$ is in $\chi(t_i) = \text{var}(\text{edges}(C_i)) \cap \text{var}(T_i)$ (respectively $\chi(t_j) = \text{var}(\text{edges}(C_j)) \cap \text{var}(T_j)$), it remains to see that $Y \in \text{var}(\text{edges}(C_i))$ (respectively $Y \in \text{var}(\text{edges}(C_j))$) holds. Clearly, this is the case since $Y$ is contained in the labeling of $T_l$ and $T_j$, i.e., $Y \in \chi(T_i)$ (respectively $Y \in \chi(T_j)$), and again by point (4) in Lemma 7.4, $Y \in \text{var}(\text{edges}(C_i)) = \chi(T_i)$ (respectively $Y \in \text{var}(\text{edges}(C_j)) = \chi(T_j)$).

(3) $\chi(s) \subseteq \text{var}(\chi(s))$. By construction in $k-$decomp, as for the basis case.

(4) $\text{var}(\chi(s)) \cap \chi(T_s) \subseteq \chi(s)$.

It follows immediately because, by construction in $k-$decomp, $\chi(s)$ is equal to $\text{var}(\chi(s)) \cap \text{var}(\text{edges}(C))$ and $\text{var}(\text{edges}(C)) = \chi(T_s)$, from point (4) in Lemma 7.4.

We have just proven that $HD_r$ is a hypertree decomposition. Furthermore, its width is clearly bounded by $k$. Hence, to conclude the proof we must show that $HD_r$ is in normal form. It is easy to see that conditions (2), (3) and (4) of decompositions in normal form are satisfied by the construction of the candidates graph in $k-$decomp. Thus, we next focus on condition (1).

Let $r = (R, C')$ and $s = (S, C)$ be nodes in $T_s$ such that $s$ is a child of $r$. Then, $C$ is an $[r]$-component. We show that $\chi(T_s) = C \cup (\chi(s) \cap \chi(r))$.

By the construction in $k-$decomp, the following relationships hold: $\chi(s) = \text{var}(\text{edges}(C)) \cap \text{var}(S)$, $\chi(r) = \text{var}(\text{edges}(C')) \cap \text{var}(R)$, $\text{var}(\text{edges}(C)) \cap \text{var}(R) \subseteq \text{var}(S)$ and $C \subseteq C'$. Thus, $\chi(s) \cap \chi(r) = \text{var}(\text{edges}(C)) \cap \text{var}(R)$.

Now, recall that due to point (4) in Lemma 7.4, $\chi(T_s) = \text{var}(\text{edges}(C))$. It follows that $\chi(T_s) = \text{var}(\text{edges}(C)) = C \cup \text{var}(\text{edges}(C)) \supseteq C \cup (\text{var}(\text{edges}(C)) \cap \text{var}(R)) = C \cup (\chi(s) \cap \chi(r))$. Then, it remains to show that the containment above is actually an equality: let $X$ be a variable in $\text{var}(\text{edges}(C))$, we show that $X$ is in $C \cup (\text{var}(\text{edges}(C)) \cap \text{var}(R))$ as well. We distinguish two scenarios: in the case $X$ is in $C \cup \text{var}(\text{edges}(C))$, then $X$ is also in $C \cup (\text{var}(\text{edges}(C)) \cap \text{var}(R))$. In the case $X$ is in $\text{var}(\text{edges}(C)) - C$, then $X$ must belong to $\text{var}(R)$, since $C$ is a $[\text{var}(R)]$-component by the construction in $k-$decomp. Finally, the fact that there is exactly one such a vertex $s$, follows immediately from the construction in $k-$decomp. □

**Theorem 7.6 (Soundness of $k-$decomp).** Given a hypergraph $\mathcal{H}$, $k-$decomp outputs a hypertree decomposition in $k\text{NFD}_{\mathcal{H}}$, if $\text{hw}(\mathcal{H}) \leq k$, and failure, otherwise, i.e., if $\text{hw}(\mathcal{H}) > k$. 
Proof. Let \( \mathcal{H} \) be a hypergraph such that \( \text{hw}(\mathcal{H}) \leq k \). We preliminary notice that in this case \( k\text{-decomp} \) cannot output failure. Indeed, from the same line of reasoning as in the proof of Theorem 7.3, we can observe that the candidates graph \( CG \) after the (*) Evaluate the Candidates Graph *) step cannot be empty. Let \( HD \) be the output of \( k\text{-decomp} \), and let \( v = (V,C) \) be the root of the tree produced by \( k\text{-decomp} \), where \( C \) is equal to \( \text{var}(\mathcal{H}) \) by the construction in \( k\text{-decomp} \). From Lemma 7.5, \( HD = HD_v \) is a hypertree decomposition in normal form of \( \mathcal{H}[\text{var(edges}(\text{var}(\mathcal{H})))] = \mathcal{H} \) having width at most \( k \).

To complete the proof, observe that if \( \text{hw}(\mathcal{H}) > k \) the algorithm outputs failure. Indeed, otherwise, from Lemma 7.5, \( k\text{-decomp} \) should return a decomposition \( HD \) in \( \text{kNFD}_\mathcal{H} \), which is impossible if the hypertree width of \( \mathcal{H} \) is greater than \( k \). \(\square\)

We can now focus on minimal hypertree decompositions, and on some properties of algorithm \( \text{minimal-}k\text{-decomp} \), in order to prove its correctness.

Lemma 7.7. Let \( \mathcal{H} \) be a hypergraph with \( \text{hw}(\mathcal{H}) \leq k \). Let \( CG = (N_{sol} \cup N_{sub}, A, \text{weight}) \) be the candidates graph after the code fragment marked by (*) Evaluate the Candidates Graph *) has been performed. For any vertex \( r \in N_{sol} \), consider the following expressions:

\[
\begin{align*}
(1) & \quad v_\mathcal{H}(r) \oplus \bigoplus_{q \in \text{incoming}(r)} \min_{s \in \text{incoming}(q)} \left( \text{weight}(s) \oplus e_\mathcal{H}(r,s) \right), \\
(2) & \quad v_\mathcal{H}(r) \oplus \bigoplus_{q \in \text{incoming}(r)} \min_{HD' \in \text{kNFD}_\mathcal{H}} \left( \mathbb{F}_\mathcal{H}^{\oplus,v,e}(HD') \oplus e_\mathcal{H}(r,\text{root}(HD')) \right).
\end{align*}
\]

Then, \( \text{weight}(r) = (1) = (2) \).

Proof. \( \text{weight}(r) = (1) \). We distinguish two scenarios. If \( r \) is a leaf of \( CG \), i.e., \( \text{incoming}(r) = \emptyset \), then \( \text{weight}(r) = v_\mathcal{H}(r) \) by construction, and the statement is trivially satisfied.

Assume now that \( r \) is an internal node and let \( q_1, \ldots, q_n \) be its children in \( N_{sub} \) corresponding to the subproblems to be solved for decomposing the component in \( r \). Assume by contradiction that \( \text{incoming}(q_i) = \emptyset \) for some \( i \in \{1, \ldots, n\} \). Then, by the way of processing nodes of \( \text{minimal-}k\text{-decomp} \), the node \( q_i \) as well as \( r \) are eventually removed in the step (*) Evaluate the Candidates Graph *) , thereby contradicting the fact that \( r \) is in \( CG \) after such a step.

Thus, each node of the form \( q_i \) is in \( CG \) and it is eventually processed in order to update the weight of its parent \( r \). Specifically, when \( \text{minimal-}k\text{-decomp} \) processes the node \( q_i \), it updates the weight of \( r \) by means of the expression

\[
\text{weight}(r) := \text{weight}(r) \oplus \min_{s \in \text{incoming}(q_i)} \left( \text{weight}(s) \oplus e_\mathcal{H}(r,s) \right),
\]

where \( \text{weight}(s) \) has been already computed by \( \text{minimal-}k\text{-decomp} \) and is not modified after the processing of the node \( r \) starts. Then, the result follows since (i) \( \text{weight}(r) \) is initialized to \( v_\mathcal{H}(r) \), (ii) no other update is performed by \( \text{minimal-}k\text{-decomp} \), and (iii) the ordering in processing the nodes \( q_1, \ldots, q_n \) is irrelevant, because of the commutativity of the operator \( \oplus \).

\( \text{weight}(r) = (2) \). The proof is by structural induction. The basis case where \( \text{incoming}(r) = \emptyset \) is trivial. Then, let \( r \in N_{sol} \) be an internal vertex. From the above expression (1) we have

\[
\text{weight}(r) = v_\mathcal{H}(r) \oplus \bigoplus_{q \in \text{incoming}(r)} \min_{s \in \text{incoming}(q)} \left( \text{weight}(s) \oplus e_\mathcal{H}(r,s) \right).
\]

We prove that, for each subproblem \( q \in \text{incoming}(r) \),

\[
\min_{s \in \text{incoming}(q)} \left( \text{weight}(s) \oplus e_\mathcal{H}(r,s) \right) = \min_{HD' \in \text{kNFD}_\mathcal{H}} \left( \mathbb{F}_\mathcal{H}^{\oplus,v,e}(HD') \oplus e_\mathcal{H}(r,\text{root}(HD')) \right).
\]

Consider any candidate \( s_j \in \text{incoming}(q) \). From the inductive hypothesis, we get

\[
\text{weight}(s_j) = v_\mathcal{H}(s_j) \oplus \bigoplus_{q_i \in \text{incoming}(s_j)} \min_{HD'' \in \text{kNFD}_\mathcal{H}} \left( \mathbb{F}_\mathcal{H}^{\oplus,v,e}(HD'') \oplus e_\mathcal{H}(s_j,\text{root}(HD'')) \right).
\]
Clearly enough, this is precisely the minimum cost over all hypertree decompositions in $kNFD^q_H$ having $s_j$ as their root. Indeed, the cost $F_{\mathcal{H}}^{\oplus,v,e}(HD_{s_j})$ of any hypertree decomposition $HD_{s_j} \in kNFD^q_H$ is obtained as the $\oplus$ sum of $v_{\mathcal{H}}(s_j)$ and of the weights of the decompositions rooted at $s_j$’s children, and $\min$ distributes over $\oplus$ (that is, the minimum value for the sum of such weights is the sum of the minimum possible values for these weights).

By the construction of $CG$, all vertices like $s_j$ that are candidates to be the roots of hypertree decompositions of $\mathcal{H}_q$ feasible with respect to $q$ have been connected to $q$ in $CG$. Moreover, from Lemma 7.5, after the evaluation of $CG$, all such nodes that still belong to $CG$ are roots of decompositions in $kNFD^q_H$. It follows that the vertex $s$ that minimizes $\text{weight}(s) \oplus e_{\mathcal{H}}(r,s)$ corresponds to the root of a hypertree decomposition $HD' \in kNFD^q_H$ that minimizes $F_{\mathcal{H}}^{\oplus,v,e}(HD') \oplus e_{\mathcal{H}}(r,\text{root}(HD'))$. □

We are now in the position to prove the correctness theorem.

**Theorem 4.4 (Correctness of minimal-\textit{k}-decomp).** Let $\mathcal{H}$ be a hypergraph and $F_{\mathcal{H}}^{\oplus,v,e}$ be a TAF.

- If $kNFD_{\mathcal{H}} = \emptyset$, minimal-\textit{k}-decomp outputs failure.
- If $kNFD_{\mathcal{H}} \neq \emptyset$, any run of minimal-\textit{k}-decomp outputs an $[F_{\mathcal{H}}^{\oplus,v,e},kNFD_{\mathcal{H}}]$-minimal hypertree decomposition of $\mathcal{H}$, and
  - for each $[F_{\mathcal{H}}^{\oplus,v,e},kNFD_{\mathcal{H}}]$-minimal hypertree decomposition $HD$, there exists a run of minimal-\textit{k}-decomp that outputs $HD$.

**Proof.** Recall that $k$-decomp is obtained by minimal-\textit{k}-decomp by relaxing the way in which an hypertree decomposition is selected. Hence, for a fixed hypergraph $\mathcal{H}$, the set of the possible outputs of minimal-\textit{k}-decomp running on input $\mathcal{H}$ is actually a subset of the possible outputs of $k$-decomp. Thus, by Theorem 7.6, we derive that minimal-\textit{k}-decomp on input $\mathcal{H}$ with $\text{hw}(\mathcal{H}) \leq k$ outputs a $k$-width hypertree decomposition in normal form $HD$ of $\mathcal{H}$. On the other hand, if $kNFD_{\mathcal{H}} = \emptyset$, it outputs failure. Indeed, algorithm minimal-\textit{k}-decomp just changes the strategy of selection of subproblem solutions, by taking minimal weighted nodes. The only possible failures in these steps may occur if there are no feasible choices. It follows that minimal-\textit{k}-decomp outputs failure only if $k$-decomp outputs failure, too. From Theorem 7.6, this entails $kNFD_{\mathcal{H}} = \emptyset$.

Now, let us assume $kNFD_{\mathcal{H}} \neq \emptyset$. In this case, minimal-\textit{k}-decomp outputs a hypertree decomposition $HD$ of $\mathcal{H}$ in normal form having width at most $k$. Recall that minimal-\textit{k}-decomp select the root of $HD$ as the minimum weighted node $p \in \text{incoming}(\emptyset,\text{var}(\mathcal{H}))$. From Lemma 7.7,

$$\text{weight}(p) = v_{\mathcal{H}}(p) \oplus \bigoplus_{q \in \text{incoming}(p)} \min_{HD \in kNFD^q_H} \left( F_{\mathcal{H}}^{\oplus,v,e}(HD) \oplus e_{\mathcal{H}}(p,\text{root}(HD')) \right).$$

By construction of the candidates graph $CG$, all $k$-vertices of $\mathcal{H}$ belongs to $\text{incoming}(\emptyset,\text{var}(\mathcal{H}))$, and those that still belongs to $CG$ after its evaluation are all the possible roots of any NF hypertree decomposition of $\mathcal{H}$ of width at most $k$, from Lemma 7.5. Let $r$ one of these vertices and $HD_r \in kNFD_{\mathcal{H}}$ a hypertree decomposition with $r$ as its root. Observe that the weight $F_{\mathcal{H}}^{\oplus,v,e}(HD_r)$ of this decomposition is at least $\text{weight}(r)$, since it will be the $\oplus$ sum of $v_{\mathcal{H}}(r)$ and the weight of the subtrees, corresponding to feasible hypertree decompositions of the corresponding subproblems. More precisely, as pointed out in the proof of Lemma 7.7(2), $\text{weight}(r)$ is the weight of the minimal decomposition having $r$ as its root.

It follows that $HD$ is a $[F_{\mathcal{H}}^{\oplus,v,e},kNFD_{\mathcal{H}}]$-minimal hypertree decomposition of $\mathcal{H}$, since its root $p$ has been selected as the minimum-weighted candidate in $\text{incoming}(\emptyset,\text{var}(\mathcal{H}))$.

Finally, any such a decomposition may be eventually considered, by Theorem 7.3 and the non-deterministically completeness of the selection strategy in case of ties (e.g., the random choice strategy), in the (* select *) steps of minimal-\textit{k}-decomp. □

8. Conclusion

We have presented an extension of the notion of hypertree decomposition, where hypertrees are weighted by some suitable functions, and we want to compute the hypertree decompositions having the minimum weight. This is a
natural generalization of the optimal hypertree decompositions in [17], that are the smallest width decompositions of a given query hypergraph.

The new notion has many possible applications, in all the areas where structural decomposition methods may be useful (see, e.g., [9,14]). Unfortunately, we have proved that even for very simple weighting functions, the computation of a minimal decomposition is an NP-hard task. However, if we restrict our search space to bounded width decompositions in normal form, the problem is feasible in polynomial time, for a large and interesting class of weighting functions.

In particular, we focused on a weighting function such that minimal hypertree decompositions of a query $Q$ correspond to the best plans for evaluating $Q$. Thus, we got a new hybrid technique for query planning, combining structural decomposition methods with quantitative methods which are typically adopted in commercial DBMS. The idea is to take advantage of both the information on the data and the structure of the query, in order to have a statistically good execution plan with a polynomial-time upper bound on the execution cost, guaranteed by the bounded hypertree-width of the query. We have described and implemented an algorithm for computing query plans according with the proposed technique. Also, we have carried out some preliminary and promising experiments, showing that this hybrid approach clearly outperforms the traditional quantitative-only optimizers, for many queries involving a certain number of atoms (more than four) and not very intricate (that is, having low hypertree width).

Recently, a prototype query optimizer based on the techniques described in this paper has been implemented inside the open source DBMS PostgreSQL [13]. The experiments carried out with new optimizer are very good, as it showed dramatic speed-ups on large classes of queries. Future work will concern the integration with further DBMSs (e.g., MySQL), and a thorough experimentation activity with real queries and databases.

Another interesting question to be experimentally evaluated is the true impact of restricting our attention to normal form hypertree decompositions. Indeed, note that, even if the computation of a minimal hypertree decomposition in the general case is NP-hard, the input of this problem is the query hypergraph, and not the entire database. It follows that in some applications the effort of this computation may be convenient, if it leads to very good query plans.

Acknowledgments

The authors thank the anonymous referees for their useful comments, and Alfredo Mazzitelli for his valuable work in implementing and testing the algorithm proposed in the paper.

Appendix A. LOGCFL—A class of parallelizable problems

We define and characterize the class LOGCFL which consists of all decision problems that are logspace reducible to a context-free language. An obvious example of a problem complete for LOGCFL is Greibach’s hardest context-free language [20]. There are a number of very interesting natural problems known to be LOGCFL-complete (see, e.g., [16,35,36]). The relationship between LOGCFL and other well-known complexity classes is summarized in the following chain of inclusions:

$$AC^0 \subseteq NC^1 \subseteq L \subseteq NL \subseteq LOGCFL \subseteq AC^1 \subseteq NC^2 \subseteq P.$$  

Here $L$ denotes logspace, $AC^i$ and $NC^i$ are logspace-uniform classes based on the corresponding types of Boolean circuits, $NL$ denotes nondeterministic logspace, and $P$ is polynomial time. For the definitions of all these classes, and for references concerning their mutual relationships, see [26].

Since $LOGCFL \subseteq AC^1 \subseteq NC^2$, the problems in LOGCFL are all highly parallelizable. In fact, they are solvable in logarithmic time by a concurrent-read–concurrent-write (CRCW) parallel random-access-machine (PRAM) with a polynomial number of processors, or in log$_2$-time by an exclusive-read–exclusive-write (EREW) PRAM with a polynomial number of processors.

In this paper, we use an important characterization of LOGCFL by Alternating Turing Machines. We assume that the reader is familiar with the alternating Turing machine (ATM) computational model introduced by Chandra et al. [5]. Here we assume without loss of generality that the states of an ATM are partitioned into existential and universal states.

As in [33], we define a computation tree of an ATM $M$ on an input string $w$ as a tree whose nodes are labeled with configurations of $M$ on $w$, such that the descendants of any non-leaf labeled by a universal (existential) configuration
include all (respectively one) of the successors of that configuration. A computation tree is accepting if the root is labeled with the initial configuration, and all the leaves are accepting configurations.

Thus, an accepting tree yields a certificate that the input is accepted. A complexity measure considered by Ruzzo [33] for the alternating Turing machine is the tree-size, i.e., the minimal size of an accepting computation tree.

**Definition A.1.** [33] A decision problem $P$ is solved by an alternating Turing machine $M$ within simultaneous tree-size and space bounds $Z(n)$ and $S(n)$ if, for every “yes” instance $w$ of $P$, there is at least one accepting computation tree for $M$ on $w$ of size (number of nodes) $\leq Z(n)$, each node of which represents a configuration using space $\leq S(n)$, where $n$ is the size of $w$. (Further, for any “no” instance $w$ of $P$ there is no accepting computation tree for $M$.)

Ruzzo [33] proved the following important characterization of LOGCFL:

**Proposition A.2.** [33] LOGCFL coincides with the class of all decision problems recognized by ATMs operating simultaneously in tree-size $O(n^{O(1)})$ and space $O(\log n)$.

Moreover, it is known that LOGCFL is closed under complementation and under LOGCFL reductions, and that in fact $L^{\text{LOGCFL}} = \text{LOGCFL}$, that is, logspace machines equipped with LOGCFL oracles are not more powerful than the ATMs in Proposition A.2.

Therefore, the functional version of LOGCFL has been naturally defined as the class $L^{L^{\text{LOGCFL}}}$ of functions computed by deterministic logspace Turing transducers with LOGCFL oracles. Also this class is closed under composition, i.e., the composition of two functions computable by $L^{\text{LOGCFL}}$ transducers is itself computable by a $L^{L^{\text{LOGCFL}}}$ transducer [18].

It has been observed that accepting computation trees of an alternating Turing machine $M$ for a given decision problem usually encode solutions of the corresponding search problem. In the case of LOGCFL, we know that if $M$ accepts its input, then there is an accepting computation tree having polynomial size, that we call certificate of acceptance.

For instance, given an acyclic conjunctive query $Q$ over a database $DB$, there is such an alternating Turing machine $M_b$ that decides whether $Q$ has some answer on $DB$. Roughly, $M_b$ visits top-down a join tree of $\mathcal{H}(Q)$ and, for each child $s_i$ of a vertex $r$, guesses a tuple $t_i \in R_i$ for the query atom $s_i(\bar{X}_i)$ such that $t_i$ matches the tuple chosen for its parent $r$. Therefore, any (acceptance) certificate for $M_b$ on input $(Q, DB)$ encodes an answer for this query. In particular, any certificate contains a configuration node for each query atom $s_i$, with the index of one tuple belonging to its associated relation in $DB$. These tuples, together, encode a solution of the search problem of computing an answer of $Q$ on $DB$ (if any).

It was shown that computing LOGCFL certificates is feasible in the functional version of LOGCFL and hence, by designing such alternating Turing machines for a decision problem, we immediately get also an $L^{L^{\text{LOGCFL}}}$ algorithm (hence a parallelizable algorithm) for solving the corresponding search problem.

**Theorem A.3.** [18] Let $M$ be a bounded-treesize logspace ATM recognizing a language $A$. It is possible to construct a $L^{L^{\text{LOGCFL}}}$ transducer $T$ that, for each input $w \in A$ outputs a single (polynomially-sized) accepting tree for $M$ and $w$.

**References**