Enumeration for FO Queries over Nowhere Dense Graphs

Nicole Schweikardt, Luc Segoufin, Alexandre Vigny

Enumeration for FO Queries over Nowhere Dense Graphs

Nicole Schweikardt, Humboldt-Universität zu Berlin
Luc Segoufin, INRIA and ENS Paris
Alexandre Vigny, Universität Bremen

Abstract

We consider the evaluation of first-order queries over classes of databases that are *nowhere dense*. The notion of nowhere dense classes was introduced by Nešetřil and Ossona de Mendez as a formalization of classes of "sparse" graphs and generalizes many well-known classes of graphs, such as classes of bounded degree, bounded tree-width, or bounded expansion.

It has recently been shown by Grohe, Kreutzer, and Siebertz that over nowhere dense classes of databases, first-order sentences can be evaluated in pseudo-linear time (pseudo-linear time means that for all ϵ there exists an algorithm working in time $O(n^{1+\epsilon})$, where n is the size of the database).

For first-order queries of higher arities, we show that over any nowhere dense class of databases, the set of their solutions can be enumerated with constant delay after a pseudo-linear time preprocessing. In the same context, we also show that after a pseudo-linear time preprocessing we can, on input of a tuple, test in constant time whether it is a solution to the query.

1 Introduction

Query evaluation is one of the most central tasks of a database system, and a vast amount of literature is devoted to the complexity of this problem. Given a database $\mathbf{D}$ and a query q, the goal is to compute the set $q(\mathbf{D})$ of all solutions for q over $\mathbf{D}$. Unfortunately, the set $q(\mathbf{D})$ might be much bigger than the database itself, as the number of solutions may be exponential in the arity of the query. It can therefore be insufficient to measure the complexity of answering q on $\mathbf{D}$ only in terms of the total time needed to compute the complete result set $q(\mathbf{D})$. One can imagine many scenarios to overcome this situation. We could for instance only want to compute the number of solutions or just compute the k most relevant solutions relative to some ranking function.

We consider here the complexity of the *enumeration* of the set $q(\mathbf{D})$, i.e., generating one by one all the solutions for q on $\mathbf{D}$. In this context two parameters play an important role. The first one is the *preprocessing time*, i.e. the time it takes to produce the first solution. The second one is the *delay*, i.e. the maximum time between the output of any two consecutive solutions. An enumeration algorithm is then said to be *efficient* if these two parameters are small. For the delay we aim at constant time: depending only on the query and independent from the size of the database. For the preprocessing time an ideal goal would be linear time: linear in the size of the database with a constant factor depending on the query. When both are achieved we say that the query can be enumerated with constant delay after linear preprocessing.

Constant delay enumeration after linear preprocessing cannot be achieved for all queries over all databases (this is known modulo an assumption in parameterized complexity theory, since the evaluation of boolean FO queries is AW[$*$]-complete [9]). But for restricted classes of queries and databases, several efficient enumeration algorithms have been obtained. This is the case for instance for free-connex acyclic conjunctive queries over arbitrary databases [4], first-order (FO) queries over classes of databases of bounded degree [10, 20], monadic second-order (MSO) queries over classes of databases of bounded tree-width [3, 22], and FO queries over classes of databases of bounded expansion [21].

This is the extended version of the conference contribution [29].

In some scenarios only pseudo-linear preprocessing time has been achieved. A query can be enumerated with constant delay after pseudo-linear preprocessing if for all ϵ there exists an enumeration procedure with constant delay (the constant may depend on ϵ) and preprocessing time in $O(\|\mathbf{D}\|^{1+\epsilon})$, where $\|\mathbf{D}\|$ denotes the size of the database. This has been achieved for FO queries over classes of databases of low degree [11] or of local bounded expansion [30].

A special case of enumeration is when the query is boolean. In this case the preprocessing computes the answer to the query. In order to be able to enumerate queries of a given language efficiently, it is therefore necessary to be able to solve the boolean case efficiently.

It has been shown recently that boolean FO queries can be evaluated in pseudo-linear time over nowhere dense classes of databases [17]. The notion of nowhere dense classes was introduced in [26] as a formalization of classes of "sparse" graphs and generalizes all the classes mentioned above [27] (except for the classes of low degree of [11]). Among classes of databases that are closed under subdatabases, the nowhere dense classes are the largest possible classes enjoying efficient evaluation of FO queries [24] (modulo an assumption in parameterized complexity theory). It has also been shown that over nowhere dense classes of databases, counting the number of solutions to a given FO query can be achieved in pseudo-linear time [18].

Main result In this paper we show that enumeration of FO queries on nowhere dense classes of databases can be done with constant delay after pseudo-linear preprocessing. This completes the picture of the complexity of FO query evaluation on nowhere dense classes and, due to the above mentioned result of [24], on all classes that are closed under subdatabases. We also show that for any nowhere dense class of databases, given a FO query q and a database $\mathbf{D}$ in the class, after a pseudo-linear time preprocessing we can test in constant time whether an arbitrary input tuple belongs to the result set $q(\mathbf{D})$.

Proof method Our algorithms for enumerating and testing are based on the following ingredients. Instead of Gaifman's normal form (which usually serves as a starting point for algorithmic meta-theorems) we use a normal form provided by [18]. This normal form works efficiently only in the nowhere dense case and requires using explicit distance predicates in the formulas. However, it has the advantage of controlling the quantifier-rank of the local formulas. Towards evaluating local formulas we use the result that one can compute in pseudo-linear time a representative "cover" of the database by means of neighborhoods [17]. We also make use of the game characterization of nowhere dense classes [17] showing that any neighborhood can be decomposed in finitely many steps. Then, a local formula is evaluated within a neighborhood of the cover by induction on the number of remaining steps in the game until the neighborhood is trivial. The enumeration for a combination of local formulas is then done following a scheme already present in [30]: Enumeration is supported by precomputing pointers that allow to jump from one solution to the next one. We use a pointer mechanism similar to the one used in [30] for the local bounded expansion case.

Technical challenges Constant delay enumeration after pseudo-linear preprocessing was already achieved for FO queries over classes of databases having bounded expansion [21] or local bounded expansion [30]. The nowhere dense case is significantly harder. The bounded expansion case was solved using a quantifier elimination procedure reducing all FO queries to the quantifier-free ones. It seems unlikely that such a quantifier elimination procedure exists in the nowhere dense case. Therefore, as for databases with local bounded expansion, the nowhere dense case relies on locality arguments and the neighborhood covers needed for solving the boolean case [17]. This was enough for databases with local bounded expansion, as the neighborhoods had bounded expansion and FO queries could then be evaluated on them. For the nowhere dense case we need a significantly more complicated argument using further tools: the game characterization of [17] and the Rank-Preserving Normal Form Theorem of [18].

Additional material The main results of this article have been published first in a conference paper [29]. The two biggest changes are the following:

- *An explicit storing theorem.* In the conference version of that paper and in most related work, the use of the memory is either not optimal or not explicitly mentioned. In Section 3 we provide a careful analysis of the amount of memory a RAM uses to store partial functions.
- *A stronger statement.* The proofs that FO queries can be efficiently tested and enumerated over nowhere dense classes of graphs are replaced by a proof of a more powerful result: Our main contribution, Theorem 2.3, states that after a pseudo-linear preprocessing, upon input of any tuple, we can compute in constant time the smallest next solution. This generalizes the testing problem (Corollary 2.4) and the enumeration problem (Corollary 2.5).

Organization The rest of the paper is structured as follows. Section 2 provides basic notations. Section 3 is devoted to the specific task of efficiently storing and retrieving the values of functions. Section 4 introduces some needed tools and gives an overview of our proof by proving a weaker but insightful result. Section 5 presents our main algorithm, and Section 6 concludes the paper.

2 Preliminaries and main result

By $\mathbb{N}$ we denote the non-negative integers, and we let $\mathbb{N}_{\geq 1} := \mathbb{N} \setminus \{0\}$. By $\mathbb{Q}_{>0}$ we denote the set of positive rationals. For $m, n \in \mathbb{N}$ we let $[m, n] := \{i \in \mathbb{N} \mid m \leq i \leq n\}$, and we let $[m] := [0, m-1]$.

Throughout this paper, ϵ will always be a positive real number, and ℓ, r, s, i, j, k will be elements of $\mathbb{N}$. For a tuple $\overline{x}$ of arity k, we will write x_i to denote its i-th component (for $i \in [1, k]$).

Structures and first-order queries. A relational schema is a finite set of relation symbols, each having an associated arity. A finite relational *structure* $\mathcal{A}$ over a relational schema consists of a finite set, the *domain* of $\mathcal{A}$, together with an interpretation of each relation symbol R of arity k of the schema as a k-ary relation over the domain, denoted $R(\mathcal{A})$. A *database* is a finite relational structure.

A structure $\mathcal{B}$ is a *substructure* of $\mathcal{A}$ if the domain of $\mathcal{B}$ is included in the domain of $\mathcal{A}$ and each relation of $\mathcal{B}$ is included in the corresponding relation of $\mathcal{A}$. We say that a class $\mathcal{C}$ of structures (or databases) is closed under substructures (or subdatabases) if for every structure $\mathcal{A}$ in $\mathcal{C}$ and every substructure $\mathcal{B}$ of $\mathcal{A}$ we have that $\mathcal{B}$ is in $\mathcal{C}$.

If $\mathcal{A}$ is a structure with domain A and $B \subseteq A$ is a subset of its domain, we denote by $\mathcal{A}[B]$ the substructure of $\mathcal{A}$ induced by B, i.e., $\mathcal{A}[B]$ is the structure $\mathcal{B}$ with domain B and $R(\mathcal{B}) = R(\mathcal{A}) \cap B^k$ for each relation symbol R of arity k.

Let σ and σ' be relational schemas with $\sigma \subseteq \sigma'$, and let $\mathcal{A}$ be a structure of schema σ (for short: a σ-structure). A *σ'-expansion* of $\mathcal{A}$ is a σ'-structure $\mathcal{B}$ whose domain is identical to the domain of $\mathcal{A}$ and which satisfies $R(\mathcal{B}) = R(\mathcal{A})$ for all $R \in \sigma$.

We fix a standard encoding of structures as input, see for example [1]. We denote by $\|\mathcal{A}\|$ the size of (the encoding of) $\mathcal{A}$, while $|\mathcal{A}|$ denotes the size $|A|$ of its domain. Without loss of generality we assume that the domain A comes with a linear order. If not, we arbitrarily choose one, for instance the one induced by the encoding of $\mathcal{A}$. This order induces a lexicographical order among the tuples over A.

A query is a first-order formula. We assume familiarity with first-order logic, FO, over relational structures (cf., e.g., [1, 25]). We use standard syntax and semantics for FO. In particular we write $q(\overline{x})$ to denote the fact that the free variables of the query q are exactly the variables in $\overline{x}$. The length of $\overline{x}$ is called the *arity* of the query. The *size* of a query q is the number of symbols needed to write down the formula and is denoted by $|q|$.

For a structure $\mathcal{A}$, a query $q(\overline{x})$ and a tuple $\overline{a}$ of elements of $\mathcal{A}$ of the appropriate arity, we write $\mathcal{A} \models q(\overline{a})$ to indicate that $\overline{a}$ is a solution for q over $\mathcal{A}$. We write $q(\mathcal{A})$ to denote the set of tuples $\overline{a}$ such that $\mathcal{A} \models q(\overline{a})$.

A *sentence* is a formula with no free variables, i.e., of arity 0. It is either true or false over a structure and therefore defines a property of structures, i.e., a *boolean* query. Given a relational structure $\mathcal{A}$ and a sentence q, the problem of testing whether $\mathcal{A} \models q$ is called *the model checking problem*. Often, the problem is restricted to a particular class $\mathcal{C}$ of relational structures.

Model of computation and complexity. As usual when dealing with linear time, we use Random Access Machines (RAM) with addition, multiplication, and uniform cost measure as a model of computation.

All problems encountered in this paper have two inputs: a structure $\mathcal{A}$ and a query $q(\overline{x})$. However, they play different roles as $\|\mathcal{A}\|$ is often very large while $|q|$ is generally small. We adopt the data complexity point of view [33]. When we say *linear time* we mean in time $O(\|\mathcal{A}\|)$, the constants hidden behind the "big O" depending on q, on the class $\mathcal{C}$ of structures under investigation, and possibly on further parameters that will be clear from the context. We say that a problem is solvable in *pseudo-linear time* if, for all $\epsilon > 0$, it can be solved in time $O(\|\mathcal{A}\|^{1+\epsilon})$. In this case, the constant factor also depends on ϵ. If a subroutine of a procedure depending on ϵ produces an output of size $O(\|\mathcal{A}\|^{\epsilon})$ we will say that the output is *pseudo-constant*.

Distance and neighborhoods. Fix a structure $\mathcal{A}$ of domain A. The *Gaifman graph* of $\mathcal{A}$ is the undirected graph whose set of vertices is A and whose edges are the pairs $\{a, b\}$ such that a and b occur in a tuple of some relation of $\mathcal{A}$. Given two elements a and b of A, the *distance* between a and b is the length of a shortest path between a and b in the Gaifman graph of $\mathcal{A}$. The notion of distance extends to tuples in the usual way, i.e., the distance between two tuples $\overline{a}$ and $\overline{b}$ is the minimum of the distances between a_i and b_j over all i, j.

For a positive integer r, we write $N_r^{\mathcal{A}}(a)$ for the set of all elements of A at distance at most r from a. The r-neighborhood of a in $\mathcal{A}$, denoted $\mathcal{N}_r^{\mathcal{A}}(a)$, is the substructure of $\mathcal{A}$ induced by $N_r^{\mathcal{A}}(a)$. Similarly, for a tuple $\overline{a}$ of arity k we let $N_r^{\mathcal{A}}(\overline{a}) := \bigcup_{i\in[1,k]} N_r^{\mathcal{A}}(a_i)$, and we define $\mathcal{N}_r^{\mathcal{A}}(\overline{a})$ as the substructure of $\mathcal{A}$ induced by $N_r^{\mathcal{A}}(\overline{a})$.

Nowhere dense classes of undirected graphs. For an undirected graph $G = (V, E)$ we let $|G| = |V|$ and $\|G\| = |V| + |E|$. Thus, similarly as for databases, $|G|$ denotes the size of the graph's domain, and $\|G\|$ is the size of a reasonable encoding of G.

Given two undirected graphs G and H and an integer r, the graph H is said to be a *shallow minor at depth* r of G (see [27, Section 2.1]) if H can be obtained from G by removing edges, removing vertices, and contracting into a single vertex sets of vertices of radius at most r. A class $\mathcal{C}$ of undirected graphs is *nowhere dense* if for every integer r there is an integer N such that no graph of $\mathcal{C}$ contains the clique with N elements as a shallow minor at depth r [27].

This is a robust notion, and there exist many equivalent definitions [27, 26, 17]. In particular, [17] presented a *game characterization* of nowhere dense classes that we rely on in our Sections 4 and 5 (see Definition 4.5 and Theorem 4.6). A further, equivalent characterization is based on the following notion of weak r-accessibility. Given an undirected graph G with a linear order on its vertices, and two of its vertices a, b, we say that b is *weakly r-accessible* from a if there exists a path of length at most r between a and b such that b is smaller than a and all other vertices on the path. A class $\mathcal{C}$ of undirected graphs is nowhere dense if, and only if, for all r and ϵ, there is a number N, such that for all graphs G of $\mathcal{C}$ with $|G| > N$, there is a linear order on the vertices of G, such that for all vertices a of G, the number of vertices weakly r-accessible from a is bounded by $|G|^{\epsilon}$ [27].

It is known, for example, that any class of graphs that (locally) excludes a minor or that is of (local) bounded expansion, is nowhere dense. Nowhere dense classes are "sparse" in the following sense:

Theorem 2.1 (Derived from [27, Theorem 4.1 (iv)]). *For every nowhere dense class $\mathcal{C}$ of undirected graphs there is a function $f_{\mathcal{C}}$ such that for every $\epsilon > 0$ and every G in $\mathcal{C}$, if $|G| \geq f_{\mathcal{C}}(\epsilon)$, then $\|G\| \leq |G|^{1+\epsilon}$.*

In the special case where for all integers r there is a constant c_r such that for all graphs G of $\mathcal{C}$ there is a linear order on the vertices of G, such that for all vertices a of G the number of vertices weakly r-accessible from a is bounded by c_r, the class $\mathcal{C}$ is said to have *bounded expansion*. These constants c_r were the keys to the constant delay enumeration algorithm for FO queries over classes of bounded expansion [21]. As we no longer have them in the nowhere dense case, we need a different strategy.

In the above characterization of nowhere dense classes via the notion of weak r-accessibility, we do not require the number N to be computable from the parameters r and ϵ. When N is computable from r and ϵ we say that the class is *effectively nowhere dense*. Most of the classical nowhere dense classes of

graphs, like bounded treewidth, planar graphs etc. are in fact effectively nowhere dense. Our results will show that if $\mathcal{C}$ is a nowhere dense class of graphs then for all $\epsilon > 0$ there is an algorithm depending on ϵ satisfying some properties. If $\mathcal{C}$ is furthermore effectively nowhere dense, all our algorithms will be computable from ϵ, hence providing a generic variant of our results.

From databases to colored graphs. We define *c-colored graphs* as structures over the schema $\sigma_c := \{E, C_1, \ldots, C_c\}$ where E is a binary symmetrical relation and $(C_i)_{i \leq c}$ are unary relations. A *colored graph* is a c-colored graph for some integer c.

A class $\mathcal{C}$ of colored graphs is defined to be *nowhere dense* if the class $\mathcal{C}'$ consisting of the underlying undirected graphs of all elements of $\mathcal{C}$ is nowhere dense.

Our main enumeration algorithm works for FO-queries on nowhere dense classes of colored graphs. By using standard techniques, this extends to all relational structures, as we now explain.

Given a database $\mathbf{D}$ over a schema σ, we define its *adjacency graph* $A(\mathbf{D})$ as the relational structure whose domain is $D \cup T$ where D is the domain of $\mathbf{D}$ and T is the set of tuples occurring in a relation of $\mathbf{D}$. We have one unary relation P_R per relation R of σ containing all tuples t of $R(\mathbf{D})$. We have k (symmetrical) binary relations $E_1, \ldots, E_k$ where k is the maximal arity of relations in σ. For $a \in D$ and $t \in T$, we have $A(\mathbf{D}) \models E_i(a, t)$ if and only if the element a is the i^{th} element of the tuple t. Currently, $A(\mathbf{D})$ is an undirected graph with colored vertices and colored edges. With what is following, we can build a colored graph in the sense of c-colored graph.

The *colored graph version* $A'(\mathbf{D})$ of the adjacency graph $A(\mathbf{D})$ is defined as follows.

The colors of $A'(\mathbf{D})$ are the "vertex colors" P_R of $A(\mathbf{D})$ and k further colors $(C_i)_{i \leq k}$, where k is the maximal arity of the relations in σ. The domain of $A'(\mathbf{D})$ is the domain of $A(\mathbf{D})$ plus one node per edge of $A(\mathbf{D})$: For every E_i-edge (a, t) of $A(\mathbf{D})$, we add in $A'(\mathbf{D})$ a new node v of color C_i and such that (a, v) and (v, t) are E-edges in $A'(\mathbf{D})$. In particular, $A'(\mathbf{D})$ is a colored graph of schema $\{E, C_1, \ldots, C_k, P_{R_1}, \ldots, P_{R_m}\}$ if $\mathbf{D}$ is a database of schema $\sigma = \{R_1, \ldots, R_m\}$ consisting of m relations of maximum arity k. Henceforth, we will identify the schema of $A'(\mathbf{D})$ with the schema σ_c of c-colored graphs for $c = k + m$.

The following lemma is classical and reduces relational structures to colored graphs.

Lemma 2.2. *Let $\varphi(\overline{x})$ be a* FO *query over some schema σ and let k be the maximal arity of the relations of σ. In time linear in the size of φ, we can compute a* FO *query $\psi(\overline{x})$ over σ_c, for $c = k + |\sigma|$, such that for every database $\boldsymbol{D}$ over σ, $\varphi(\boldsymbol{D}) = \psi(A'(\boldsymbol{D}))$.*

The lemma is an immediate consequence of the fact that

$$\mathbf{D} \models R(a_1, \ldots, a_j) \iff$$

$$A'(\mathbf{D}) \models \exists t \big(P_R(t) \wedge \bigwedge_{i \leq j} \exists z \big(C_i(z) \wedge E(a_i, z) \wedge E(z, t) \big) \big).$$

Let $\mathcal{C}$ be a class of relational structures. We say that $\mathcal{C}$ is *nowhere dense* if the class $\{A'(\mathbf{D}) \mid \mathbf{D} \in \mathcal{C}\}$ is nowhere dense. Note that because $A'(\mathbf{D})$ is a 1-subdivision (i.e. an edge is transformed into a path of length 2) of $A(\mathbf{D})$, the class $\{A'(\mathbf{D}) \mid \mathbf{D} \in \mathcal{C}\}$ is nowhere dense iff the class consisting of the Gaifman graphs of $A(\mathbf{D})$ for all $\mathbf{D} \in \mathcal{C}$ is nowhere dense; see [27] for further details.

We could have used another definition for nowhere dense classes of structures, using their Gaifman graphs instead of their adjacency graphs. For a fixed schema this would result in the same notion [34, Theorem 4.3.6], but when the schema is not fixed our definition is more general [19, Example 3.3.2].

A consequence of Lemma 2.2 is that the model checking, enumeration, counting, and testing problems for FO-queries reduce to the colored graph case. In the remaining part of the paper we will therefore only consider classes of colored graphs. The reader should keep in mind, though, that the results stated over colored graphs extend to relational structures.

From the definition it follows immediately that if a class of graphs is nowhere dense then the class of all its substructures is also nowhere dense. Moreover, the class of all its possible colorings is also nowhere dense. Hence without loss of generality, we can assume that all nowhere dense classes $\mathcal{C}$ of colored graphs considered from now on, are closed under substructures and all possible colorings (using arbitrarily many colors).

Enumeration. An *enumeration* algorithm for a colored graph G and a query q is divided into two consecutive phases:

- a preprocessing phase, and
- an enumeration phase, outputting one by one and without repetition the elements of the set $q(G)$.

The *preprocessing time* of the enumeration algorithm is the time taken by the preprocessing phase. Its *delay* is the maximum time between any two consecutive outputs. One can view an enumeration algorithm as a compression algorithm that computes a representation of $q(G)$, together with a streaming decompression algorithm.

We aim for enumeration algorithms with constant delay and pseudo-linear preprocessing time. By this we mean that for all $\epsilon > 0$, there is a preprocessing phase working in time $O(\|G\|^{1+\epsilon})$ and an enumeration phase with constant delay. Note that the multiplicative constants for the preprocessing phase and the delay may depend on q and ϵ.

An enumeration phase with constant delay may use a constant amount of memory while preparing the next output. Hence it may use a total amount of memory that is linear in the output size. Our enumeration algorithms will have the property that, apart from the memory used for storing the data structure (of pseudo-linear size) that is produced during the preprocessing phase, the entire enumeration phase only uses a constant amount of extra memory. In other words, our enumeration algorithm can be seen as a finite state automaton running on the index structure produced by the preprocessing phase.

All our enumeration procedures will output their tuples in lexicographical order. We will see that this is useful for queries in disjunctive normal form.

Our main result. We now state our main theorem. Recall that we assume a linear order on the domain of our structures. This order induces a lexicographical order on tuples of elements that we denote by $\leq$.

Theorem 2.3. *Let $\mathcal{C}$ be a nowhere dense class of colored graphs. There is a function f and an algorithm which, upon input of a colored graph $G \in \mathcal{C}$, an $\epsilon \in \mathbb{Q}_{>0}$, and a first-order query q, performs a preprocessing in time $f(q,\epsilon) \cdot |G|^{1+\epsilon}$ such that afterwards, upon input of a tuple $\overline{a}$, we can compute in time $f(q,\epsilon)$ the smallest (in lexicographical order) tuple $\overline{a}'$ such that $\overline{a}' \geq \overline{a}$ and $\overline{a}' \in q(G)$, or an error message in case that no such tuple $\overline{a}'$ exists.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, then f is computable.

This result both implies a constant delay enumeration and a constant time testing procedures. Testing whether a tuple is a solution is immediate from the data structure computed in Theorem 2.3, as it is enough to test on input $\overline{a}$ whether the output tuple $\overline{a}'$ is equal to $\overline{a}$. Thus, we obtain:

Corollary 2.4 (Testing solutions)**.** *Let $\mathcal{C}$ be a nowhere dense class of colored graphs. There is a function f and an algorithm which, upon input of a colored graph $G \in \mathcal{C}$, an $\epsilon \in \mathbb{Q}_{>0}$, and a first-order query q, performs a preprocessing in time $f(q,\epsilon) \cdot |G|^{1+\epsilon}$, such that afterwards, upon input of a tuple $\overline{a}$, we can test whether $G \models \varphi(\overline{a})$ in time $f(q,\epsilon)$.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, then f is computable.

The data structure of Theorem 2.3 also allows to enumerate the solutions with constant delay. Indeed, once we have outputted a solution $\overline{a}$, we derive in constant time the tuple $\overline{b}$ which immediately follows $\overline{a}$ in the lexicographical order, and then the tuple $\overline{b}'$ given by Theorem 2.3 (upon input of $\overline{b}$) is the next solution to be outputted during the enumeration phase.

Corollary 2.5 (Enumerating solutions)**.** *Let $\mathcal{C}$ be a nowhere dense class of colored graphs. There is a function f and an algorithm which, upon input of a colored graph $G \in \mathcal{C}$, an $\epsilon \in \mathbb{Q}_{>0}$, and a first-order query q, performs a preprocessing in time $f(q,\epsilon) \cdot |G|^{1+\epsilon}$, such that afterwards, the set of solutions $\varphi(G)$ can be enumerated with delay $f(q,\epsilon)$ in increasing order.*

Again, if $\mathcal{C}$ is effectively nowhere dense, then f is computable.

3 Storing functions and retrieving solutions

This section is devoted to a technical result that uses in an essential way our computational model. We will often compute partial k-ary functions associating a value to a tuple of nodes of the input graph. Such functions can be easily implemented in the RAM model using k-dimensional cubes allowing to retrieve the value of f in constant time. This requires a memory usage of $O(n^k)$. However our functions will have a domain of size pseudo-linear and can be computed in pseudo-linear time. The following theorem states that we can use the RAM model to build a data structure that stores our functions in a more efficient way.

Theorem 3.1 (Storing Theorem). *For every fixed $k \in \mathbb{N}$ and $\epsilon > 0$, there is an integer $c \in \mathbb{N}$ such that for every integer $n \in \mathbb{N}$ there is a data structure that stores the value of a k-ary function f of domain $\mathrm{Dom}(f) \subseteq [n]^k$ with:*

- *initialization time $c \cdot |\mathrm{Dom}(f)| \cdot n^\epsilon$,*
- *update time $c \cdot n^\epsilon$ whenever a pair[1] $(\overline{a}, b)$ is added to or removed from f,*
- *lookup time c,*
- *and at any point in time, the space used by the data structure is $c \cdot |\mathrm{Dom}(f)| \cdot n^\epsilon$.*

Here, lookup means that given a tuple $\overline{a} \in [n]^k$, the algorithm either answers b if $\overline{a} \in \mathrm{Dom}(f)$ and $f(\overline{a}) = b$; or $\overline{a}'$ if $\overline{a} \notin \mathrm{Dom}(f)$ and $\overline{a}' := \min\{\overline{x} \in \mathrm{Dom}(f) \ : \ \overline{x} > \overline{a}\}$; or Null if no such tuple exists.

We stress that during the lookup procedure, the data structure may be given a tuple $\overline{a}$ that is not part of the domain of the function. This will heavily be used in later sections. However, other perks of the statement will not be used. For example, we will only construct the data structure when computing f and then only use the lookup feature. In particular, we will never delete any tuple from the domain of f. Nonetheless, this result is interesting in its own. A version without updates of that theorem has been written first in [34, Section 4.1] and is inspired from [32, Figure 1].

The data structure is not that complicated. In fact its core is a trie where each pair (key,value) is a tuple $\overline{a}$ in the domain of f and its image $b = f(\overline{a})$. See [23, Section 6.3 *Digital Searching*] for more information. In order to obtain all our features, our data structure expands the trie structure in the two following ways. First, we have a backward relation from *last child* to *parent*. This relation helps us navigate the structure for the update feature. Second, we have a relation linking search paths that do not lead to a key, to the next smallest key. This relation is used for the lookup feature.

In the rest of the section, we describe the data structure together with examples. We then give an intuition of how to obtain the desired algorithmic features. As this result is not our main contribution, we wrote the actual proofs in a separated appendix.

3.1 Description of the data structure

Fix ϵ and n. Let $d := \lceil n^\epsilon \rceil$ and $h := \left\lceil \frac{1}{\epsilon} \right\rceil$. As usual, for $x \in \mathbb{Q}$, $\lceil x \rceil$ denotes the smallest integer y such that $x \leq y$.

Every $i \in [n]$ can be uniquely decomposed in base d into a string of length h whose letters are from $[0, d-1]$ since $d^h \geq n$. We arbitrarily assume that the string starts with the higher powers of d and ends with the lowest ones. Given all this, every tuple in $[n]^k$ can be decomposed into a string of length kh whose letters are from $[0, d-1]$. We then associate to the function f a partial tree $T(f)$ of maximal depth kh and degree d, where each node has 0 or d children and each leaf at depth kh represents an element of the domain of f (by looking at the sequence of child numbers in the path from the root to that leaf). The size of $T(f)$ is then $O(n^\epsilon \cdot |\mathrm{Dom}(f)|)$.

Our data structure is an encoding of $T(f)$ with extra information in order to navigate efficiently in the tree and to update it efficiently. As for leaves, to any node of $T(f)$ at depth i we can associate a string over $[0, d-1]$ of length i. Given a leaf of $T(f)$ we associate a tuple $\overline{b}$ as the smallest tuple (in lexicographical order) of the domain of f whose encoding has a prefix larger than the one of the current node.

[1] we identify f with its graph $\{(\overline{a}, b) \mid \overline{a} \in \mathrm{Dom}(f),\ f(\overline{a}) = b\}$

Each inner node of the tree associated to f is represented by $d+1$ consecutive registers in our memory each containing a pair (δ, r) where δ is either $0, 1$ or -1 and r is a value that will help us navigating in the tree.

Consider an inner node x of $T(f)$ and assume that x is the i^{th} child of y. Let R be the i^{th} register representing y and R' be the first register representing x. Then the content of R is $(1, R')$ and the content of the last register representing x contains $(-1, R)$. This encodes the parent/child relation of $T(f)$. The rest of the encoding will help updating the structure efficiently.

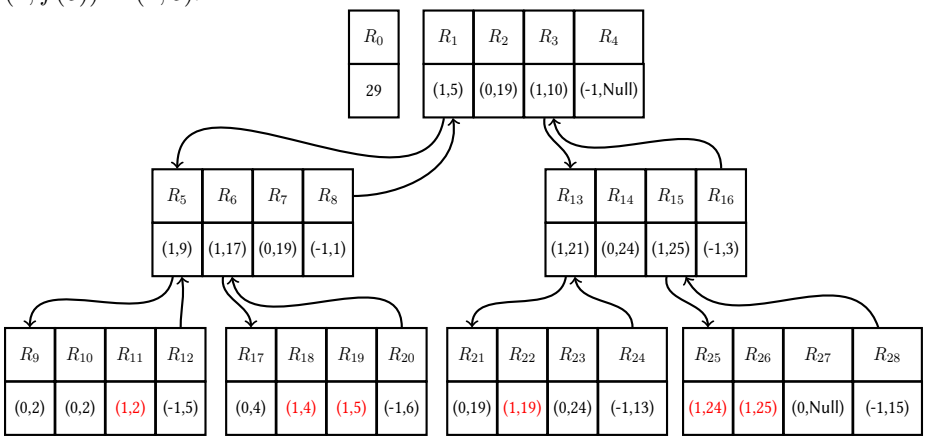
If the j^{th} child of y is a leaf, for $j \leq d$, then the content of the j^{th} register representing y is $(0, \overline{b})$ where $\overline{b}$ is the tuple associated to that leaf.

In the case when x is at depth $kh-1$ (i.e. all its children are leaves), for $i \leq d$, we set the content of the i^{th} register representing x as $(1, f(\overline{a}))$ if the i^{th} leaf of x represents a tuple $\overline{a}$ in the domain of f, and as $(0, \overline{b})$ otherwise where $\overline{b}$ is the tuple associated to that leaf.

Finally, we have a register R_0 that contains the next available (unused) register.

Our data structure is illustrated in Figure 1.

Figure 1: An example of the data structure. In this example $n = 27, \epsilon = 1/3$, therefore $d = n^{\epsilon} = 27^{1/3} = 3$ and $h = 1/\epsilon = 3$. The unary partial function f is the identity function whose domain is $\{2, 4, 5, 19, 24, 25\}$ and is not defined otherwise. Note that the decomposition of 2 in base $d = 3$ is 002, while 4 is 011, 5 is 012, 19 is 201 and so on. The leaves corresponding to the domain of f are in red. The arrows are here for readability. For instance R_1 is the first register representing the root node of $T(f)$ and its content is $(1, 5)$ because the first child of the root node of $T(f)$ is not a leaf and the first register representing it is R_5. Moreover the second register representing the root node of $T(f)$ is R_2 whose content is $(0, 19)$ because the second child of the root of $T(f)$ is a leaf and 19 is the smallest element in the domain of f whose decomposition starts with a 2. The register R_8 is the last register representing the first child of the root of $T(f)$. Its content is therefore $(-1, 1)$ because R_1 is the first register encoding the root. Finally R_{19} is the third register encoding the second child of the first child of the roof of $T(f)$. It therefore represents the number encoded by 012, i.e. 5, in the domain of f. Its content is therefore $(1, f(5)) = (1, 5)$.



3.2 Proof sketch of Theorem 3.1.

Looking up information in this data structure is pretty straightforward: Given a tuple $\overline{a}$, one can decompose this tuple into a string of length kh which precisely describes the search path for $\overline{a}$ in the data structure $T(f)$. This search can either fail by reaching a cell of the form $(0, \overline{b})$ and therefore conclude that $\overline{a}$ is not in the domain of the stored function f and that $\overline{b}$ is the smallest tuple in $\mathrm{Dom}(f)$ greater than $\overline{a}$. The only other possibility is that the search for $\overline{a}$ reaches a cell of the form $(1, b)$ implying that $\overline{a}$ is in the domain of f, and that $f(\overline{a}) = b$. All of this takes time linear in the depth kh of the tree $T(f)$, which is constant as ϵ is fixed.

Adding or removing information in the data structure is somewhat trickier. Adding a tuple may require the creation of new arrays of length d in the structure, this is where R_0 comes in handy as it points to the end of the used memory i.e. where we can start new arrays. Removing tuples may render some array useless (only composed of cells of the form $(0, \overline{b})$). While this is not an issue for the lookup procedure, such useless array must be deleted otherwise the memory usage will not remain linear. This is performed by copy/pasting the last array of the structure in the place used by the array we want to get rid of. We then have to take care of the (only) cell pointing toward the moved array, and update R_0. So far, and in both case, the procedure takes time $O(d \cdot k \cdot h)$. For the main contribution of this paper, we only use the insert and lookup procedures. The remaining procedures, together with the complete proofs can be found in the Appendix Section 7.

4 Testing distance queries

In this section we prove a weaker version of our main result where we only consider the testing problem of distance queries. This requires introducing several tools that will also be needed for proving the main result.

A distance query is a binary query testing the distance within a colored graph between two nodes. For all $r \in \mathbb{N}$, the query $\mathrm{dist}_{\leq r}(a, b)$ states that there is a path of length at most r between a and b.

Definition 4.1 (Distance queries)**.** Distance queries can formally be defined by induction as follows:

$$\begin{array}{lcl} \mathrm{dist}_{\leq 0}(x, y) & := & (x = y) \\ \mathrm{dist}_{\leq r+1}(x, y) & := & \mathrm{dist}_{\leq r}(x, y) \ \vee \ \exists z \ \big(E(x, z) \wedge \mathrm{dist}_{\leq r}(z, y)\big) \end{array}$$

A slightly different definition can make the quantifier rank of $\mathrm{dist}_{\leq r}$ equal to $\log(r)$ instead of r above, but this is not important to us. We will also make use of the following queries:

$$\begin{array}{lcl} \mathrm{dist}_{=r}(x, y) & := & \mathrm{dist}_{\leq r}(x, y) \wedge \neg\, \mathrm{dist}_{\leq r-1}(x, y) \\ \mathrm{dist}_{>r}(x, y) & := & \neg\, \mathrm{dist}_{\leq r}(x, y) \end{array}$$

Example (1-A)**.** *Consider the distance two query:*

$$q(x, y) \quad := \quad \mathrm{dist}_{\leq 2}(x, y) \quad = \quad \exists z \, \big(E(x, z) \ \wedge \ E(z, y) \big) \ \vee \ E(x, y) \ \vee \ x = y$$

In this section, we introduce notions and apply them to this example query.

The rest of the section is devoted to the proof of:

Proposition 4.2. *Let $\mathcal{C}$ be a nowhere dense class of colored graphs. There is a function f and an algorithm which, upon input of a colored graph $G \in \mathcal{C}$, an $\epsilon \in \mathbb{Q}_{>0}$, and $r \in \mathbb{N}$, performs a preprocessing in time $f(r, \epsilon) \cdot |G|^{1+\epsilon}$, such that afterwards, upon input of a tuple (a, b) of nodes of G, we can test in time $f(r, \epsilon)$ whether $(a, b) \in \mathrm{dist}_{\leq r}(G)$.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, then f is computable.

4.1 Tools for nowhere dense graphs

Since we are looking at distance queries, it is tempting to precompute the r-neighborhoods of all nodes. Unfortunately, this cannot be done in pseudo-linear time as the sum of the sizes of those neighborhoods might be too big. To overcome this situation we use a *neighborhood cover* that selects a small but sufficiently representative set of neighborhoods.

This notion was crucial already for obtaining a number of previous algorithmic meta-theorems, including e.g. [13, 12, 17].

Definition 4.3 (Neighborhood cover)**.** Given a colored graph G and a number $r \in \mathbb{N}$, an *r-neighborhood cover* of G is a collection $\mathcal{X}$ of subsets of the vertices V of G such that

$$\forall\, a \in V \;\; \exists\, X \in \mathcal{X} \;\text{ with }\; N_r^G(a) \subseteq X.$$

For $r, s \in \mathbb{N}$, an *(r,s)-neighborhood cover* of G is an r-neighborhood cover of G satisfying:

$$\forall\, X \in \mathcal{X} \;\; \exists\, a \in V \;\text{ with }\; X \subseteq N_s^G(a).$$

The number r is called the *radius* of the neighborhood cover, and the sets $X \in \mathcal{X}$ are called *bags*. The *degree* $\delta(\mathcal{X})$ of the cover is the maximal number of bags that intersect at a given node, i.e.

$$\delta(\mathcal{X}) \quad := \quad \max_{a \in V} |\{X \in \mathcal{X} \mid a \in X\}|.$$

Given a neighborhood cover of radius r of G, for every $a \in V$ we arbitrarily fix a bag containing the r-neighborhood of a and denote it by $\mathcal{X}(a)$.

Neighborhood covers with small degree can be computed efficiently on nowhere dense classes:

Theorem 4.4 ([17, Theorem 6.2])**.** *Let $\mathcal{C}$ be a nowhere dense class of colored graphs. There is a function $f_{\mathcal{C}}$ and an algorithm that, given an $\epsilon > 0$, an $r \in \mathbb{N}$, and a colored graph $G \in \mathcal{C}$ whose domain V is larger than $f_{\mathcal{C}}(r, \epsilon)$, computes in time $f_{\mathcal{C}}(r, \epsilon) \cdot |V|^{1+\epsilon}$ an $(r, 2r)$-neighborhood cover of G with degree at most $|V|^{\epsilon}$. Furthermore, if $\mathcal{C}$ is effectively nowhere dense, then $f_{\mathcal{C}}$ is computable.*

In the rest of the paper, and without loss of generality modulo taking the maximum, we will use the same function $f_{\mathcal{C}}$ in order to satisfy the requirements of Theorem 4.4 and Theorem 2.1, such that a graph with a number of vertices greater than $f_{\mathcal{C}}(0, \cdot)$ satisfies the size bound of Theorem 2.1. Notice that for a nowhere dense class $\mathcal{C}$ of colored graphs, for $G \in \mathcal{C}$ of domain V, for $\epsilon > 0$ and a neighborhood cover $\mathcal{X}$ of G with degree $|V|^{\epsilon}$, the bound on the degree implies that $\sum_{X \in \mathcal{X}} |X| \leq |V|^{1+\epsilon}$. As any edge of G can appear in at most $|V|^{\epsilon}$ distinct induced subgraphs of the form $G[X]$ we have

$$\sum_{X \in \mathcal{X}} \|G[X]\| \;\leq\; O(|V|^{\epsilon} \cdot \|G\|) \tag{1}$$

Given Theorem 4.4 and in view of our Storing Theorem 3.1, after some pseudo-linear preprocessing we are able, given a bag X and a node a, to test whether $a \in X$ in constant time and, if $a \notin X$ return the smallest b, bigger than a, such that $b \in X$. This is achieved as follows: Let $n := |V|$ and identify V with $[n]$. Let $\mathcal{X} = \{X_0, \ldots, X_{m-1}\}$ be the neighborhood cover. As it is enough to have one bag X per element of V, we can assume that $m := |\mathcal{X}| \leq n$. Now we identify $\mathcal{X}$ with the partial binary function $f_{\mathcal{X}}$ from $[n]^2$ to $\{1\}$ with $(i, a) \in \mathrm{Dom}(f_{\mathcal{X}})$ iff $i < m$ and $a \in X_i$ (for all $(i, a) \in [n]^2$). Note that $\delta(\mathcal{X}) \leq n^{\epsilon}$ implies that $|\mathrm{Dom}(f_{\mathcal{X}})| \leq n^{1+\epsilon}$. Therefore, the Storing Theorem 3.1 yields the claimed functionality.

Let's get back to our running example.

Example (1-B). *The radius of the relevant cover depends on the query. For our query q from Example (1-A), we have for all nodes a, b that*

$$G \models q(a,b) \quad \iff \quad \mathcal{N}_2^G(a) \models q(a,b).$$

Therefore if we compute a $(2,4)$-neighborhood cover $\mathcal{X}$ of G, since $N_2^G(a) \subseteq \mathcal{X}(a)$, we have that for all nodes a and b:

$$G \models q(a,b) \quad \iff \quad b \in \mathcal{X}(a) \ \wedge \ G[\mathcal{X}(a)] \models q(a,b)$$

Hence, modulo a pseudo-linear preprocessing, given two elements a, b, to test whether they are at distance ≤ 2, it is enough to restrain our attention to the bag $\mathcal{X}(a)$ of a. We will see later how this will be useful.

As illustrated by our running example, we will reduce the problem from the whole graph to a bag of the cover. The following game characterization of nowhere dense classes of colored graphs will give us an inductive parameter that will decrease when diving within a bag, ensuring termination.

Definition 4.5 (Splitter game [17, Definition 4.1])**.** Let $G = (V, E)$ be a graph and let $\lambda, r \in \mathbb{N}_{\geq 1}$. The (λ, r)-*splitter game* on G is played by two players called *Connector* and *Splitter*, as follows. We let $G_0 := G$ and $V_0 := V$. In round $i+1$ of the game, Connector chooses a vertex $c_{i+1} \in V_i$. Then Splitter picks a vertex $s_{i+1} \in N_r^{G_i}(c_{i+1})$. If $V_{i+1} := N_r^{G_i}(c_{i+1}) \setminus \{s_{i+1}\} = \emptyset$, then Splitter wins the game. Otherwise, the game continues with $G_{i+1} := G_i[V_{i+1}]$. If Splitter has not won after λ rounds, then Connector wins.

This is not exactly the same definition as the one in [17], where several nodes can be removed in one step, but it is easy to show that it is equivalent to it [31]. We say that Splitter *wins* the (λ, r)-splitter game on G if she has a winning strategy for the game. Interested readers may want to look at [34, Section 4.2.2] to find examples of Splitter's winning strategies for several classes of graphs.

Theorem 4.6 ([17, Theorem 4.2])**.** *A class $\mathcal{C}$ of graphs is nowhere dense if, and only if, for every $r \in \mathbb{N}_{\geq 1}$ there is a $\lambda(r) \in \mathbb{N}_{\geq 1}$, such that for every $G \in \mathcal{C}$, Splitter wins the $(\lambda(r), r)$-splitter game on G.*

Furthermore, if $\mathcal{C}$ is effectively nowhere dense, then λ is computable.

Remark 4.7. *In the proof of our main theorem, we will also have to compute Splitter's winning strategy efficiently: s_{i+1} should be computable from the previous moves and c_{i+1} in time $O(\|\mathcal{N}_r^{G_i}(c_{i+1})\|)$.*

This is also something that is needed in [16, 17]. However, there is a small hiccup. In Remark 4.3 of [16] and identically in Remark 4.7 from the journal version [17] it is stated that s_{i+1} is computable from the previous move and c_{i+1} in time $O(\|G_i\|)$, which is a weaker statement. A closer look at the uses of these remarks indicates that the stronger version is needed and that this is actually what is proved!

Our inductive parameter is the number of remaining rounds for Splitter before she wins the game starting with parameters $(\lambda(r'), r')$ when given an $(r'/2, r')$-neighborhood cover of G for a suitable number r'.

Example (1-C). *For our running example, we have already computed a $(2,4)$-neighborhood cover $\mathcal{X}$ of G. Let then λ be such that Splitter wins the $(\lambda, 4)$-splitter game on G. Assume that $\lambda = 1$. Then G is edgeless and any naive algorithm will work. Assume now that $\lambda > 1$. Let X be a bag of $\mathcal{X}$. By definition, there is an element c such that $X \subseteq N_4^G(c)$. Let s_X be Splitter's answer if Connector picks c in the game's first round. Then, by definition, Splitter wins the $(\lambda-1, 4)$-splitter game on $G[N_4^G(c) \setminus \{s_X\}]$. In particular, Splitter wins the $(\lambda-1, 4)$-splitter game on $G' := G[X \setminus \{s_X\}]$. Hence we can use an inductive argument within G'. For this we compute a new query $q'(x,y)$ such that for all a, b such that $\mathcal{X}(a) = X$, $G \models q(a,b)$ iff $G' \models q'(a,b)$. Recall that we already know for such pairs a, b that $G \models q(a,b)$ iff $G[X] \models q(a,b)$. It therefore remains to encode the removal of the node s_X, and this can be done by recoloring the nodes adjacent to s_X. Let R_1, R_2 be new unary predicates such that:*

$$\begin{array}{lll} w \in R_1(G') & \textit{iff} & G \models \mathrm{dist}_{\leq 1}(w, s_X) \\ w \in R_2(G') & \textit{iff} & G \models \mathrm{dist}_{\leq 2}(w, s_X). \end{array}$$

The new query $q'(x,y)$ is then defined as a disjunction of the following queries

$$\begin{array}{l} q(x,y) \;\vee\; \big(\, R_1(x) \,\wedge\, R_1(y) \,\big) \\ \vee\; \big(\, R_2(x) \,\wedge\, y = s_X \,\big) \vee\; \big(\, R_2(y) \,\wedge\, x = s_X \,\big) \\ \vee\; \big(\, x = s_X \,\wedge\, y = s_X \,\big). \end{array}$$

The first line takes care of the general case i.e. neither x nor y have been deleted. The first disjunction tests whether they are still at distance 2 in the new graph, the second one tests whether the unique node connecting x and y is s_X. The second line deals with the cases of having one of the node deleted, $y = s_X$ and $x = s_X$. As s_X is not part of G', the case $R_2(x) \wedge y = s_X$ should be understood as: If $y = s_X$, simply test whether $G' \models R_2(x)$. Similarly for the dual case. Last, if both have been deleted, the variables were talking about the same node, being at distance less than two from itself.

4.2 Testing distance queries

We are now ready to prove Proposition 4.2. Let $\mathcal{C}$ be a nowhere dense class of colored graphs and let $r \in \mathbb{N}$. We want to test efficiently whether two nodes are within distance at most r in G, for some $G \in \mathcal{C}$. Let $\epsilon > 0$.

For every $\lambda \in \mathbb{N}_{\geq 1}$ let $\mathcal{C}_\lambda$ be the subclass of $\mathcal{C}$ consisting of all $G \in \mathcal{C}$ such that Splitter wins the $(\lambda, 2r)$-splitter game on G. Clearly, $\mathcal{C}_\lambda \subseteq \mathcal{C}_{\lambda+1}$ for every λ. Since $\mathcal{C}$ is nowhere dense, by Theorem 4.6 there exists a number $\Lambda := \lambda(2r) \in \mathbb{N}$ such that $\mathcal{C} = \mathcal{C}_\Lambda$. Thus,

$$\mathcal{C}_1 \;\subseteq\; \mathcal{C}_2 \;\subseteq\; \cdots \;\subseteq\; \mathcal{C}_\Lambda \;=\; \mathcal{C}.$$

We proceed by induction on λ and prove the result for all $G \in \mathcal{C}_\lambda$. The induction base for $\lambda = 1$ follows immediately, since by definition of the splitter game every $G \in \mathcal{C}_1$ has to be edgeless, and thus a naive algorithm works. For the induction step consider a $\lambda \geq 2$ and assume that Proposition 4.2 already holds for $\lambda - 1$ (i.e., for all colored graphs in $\mathcal{C}_{\lambda-1}$).

Consider an arbitrary $G \in \mathcal{C}_\lambda$. Let $n := |V|$ be the size of the domain V of G, and let $\delta > 0$ be such that $3\delta + 2\delta^2 \leq \epsilon$.

4.2.1 The preprocessing phase

The preprocessing phase is composed of the following steps:

1. Let $f_{\mathcal{C}}(\cdot,\cdot)$ be the function provided by the neighborhood cover Theorem 4.4. If $n \leq f_{\mathcal{C}}(r,\delta)$, we use a naive algorithm to compute entirely the query result $\mathrm{dist}_{\leq r}(G)$ and trivially provide the functionality claimed by Proposition 4.2.

   From now on, consider the case where $n > f_{\mathcal{C}}(r,\delta)$. Recall from Theorem 2.1 that this implies that $\|G\| \leq O(n^{1+\delta})$.

2. Using the algorithm provided by Theorem 4.4, and since $n > f_{\mathcal{C}}(r,\delta)$, we compute a $(r,2r)$-neighborhood cover $\mathcal{X}$ of G with degree at most n^δ. Furthermore, in the same way as in [17, Lemma 6.10], we also compute for each $X \in \mathcal{X}$ a list of all $b \in V$ satisfying $\mathcal{X}(b) = X$, and we compute a node c_X such that $X \subseteq N_{2r}^G(c_X)$. All these relations can be efficiently stored and retrieved with the Storing Theorem 3.1.

3. Since $G \in \mathcal{C}_\lambda$, we know that Splitter wins the $(\lambda, 2r)$-splitter game on G. For every $X \in \mathcal{X}$ we now compute a node s_X that is Splitter's answer if Connector plays c_X in the first round of the $(\lambda, 2r)$-splitter game on G. From Remark 4.7 we know that the nodes $(s_X)_{X \in \mathcal{X}}$ can be computed within total time $O(n^{1+\delta})$.

4. For every X in $\mathcal{X}$, we define X' as $G[X \setminus \{s_X\}]$ and we compute for every $i \leq r$:

   $$R_i(X') \quad := \quad \{w \mid G[X] \models \mathrm{dist}_{\leq i}(w, s_X)\}$$

   using a simple breadth-first search.

5. By Splitter's choice of the node s_X, we know that she wins the $(\lambda-1, 2r)$-splitter game on X' for every X in $\mathcal{X}$. Therefore, for each X in $\mathcal{X}$, we spend time $O(|X'|^{1+\delta})$ for the preprocessing obtained by induction on Proposition 4.2 for the query $\mathrm{dist}_{\leq r}$. This will allows us, given (a, b) in the domain of X', to test in constant time whether $(a, b) \in \mathrm{dist}_{\leq r}(X')$.

This ends the preprocessing. We now show that it works in time $O(n^{1+\epsilon})$ as desired. Step 1 takes time $O(1)$ and Step 2 and 3 time $O(n^{1+\delta})$. Step 4 and 5 take, for each $X \in \mathcal{X}$, time $O(\|X'\|^{1+\delta})$ leading to a total time:

$$\begin{aligned} O\Big(\sum_{X\in\mathcal{X}} \|X'\|^{1+\delta}\Big) &\leq O\Big(\Big(\sum_{X\in\mathcal{X}} \|X'\|\Big)^{1+\delta}\Big) \leq O\Big(\Big(\sum_{X\in\mathcal{X}} \|G[X]\|\Big)^{1+\delta}\Big) \\ &\leq O\Big((n^{\delta}\|G\|)^{1+\delta}\Big) \qquad \text{by (1)} \\ &\leq O\left((n^{\delta}n^{1+\delta})^{1+\delta}\right) = O\left(n^{1+3\delta+2\delta^2}\right) \leq O(n^{1+\epsilon}). \end{aligned}$$

4.2.2 Test procedure

We are now given two nodes a and b. We want to answer in constant time to the question: "Is (a, b) in $\mathrm{dist}_{\leq r}(G)$?" After the preprocessing, in constant time we have access to $\mathcal{X}(a)$ and we can test whether b is in $\mathcal{X}(a)$. If it is not the case, then clearly $(a, b) \notin \mathrm{dist}_{\leq r}(G)$.

Otherwise, we have that $(a, b) \in \mathrm{dist}_{\leq r}(G)$ iff $(a, b) \in \mathrm{dist}_{\leq r}(G[\mathcal{X}(a)])$. Moreover, we have $(a, b) \in \mathrm{dist}_{\leq r}(G[\mathcal{X}(a)])$ if and only if one of the following is true (recall that $X' = G[X \setminus \{s_X\}]$):

- $a \neq s_X$ and $b \neq s_X$, and $X' \models \mathrm{dist}_{\leq r}(a, b) \vee \bigvee_{\substack{1\leq i,j\leq r-1 \\ i+j\leq r}} \big(R_i(a) \wedge R_j(b)\big)$
- $a \neq s_X$ and $b = s_X$, and $X' \models R_r(a)$
- $a = s_X$ and $b \neq s_X$, and $X' \models R_r(b)$
- $a = s_X$ and $b = s_X$.

As we can test all of this in constant time (the test that $X' \models \mathrm{dist}_{\leq r}(a, b)$ is done by using the induction hypothesis for $\lambda-1$ in Proposition 4.2), we are able to decide in constant time whether $(a, b) \in \mathrm{dist}_{\leq r}(G)$.

This ends the proof of Proposition 4.2.

5 Computing the next solution

In this section, we finally prove Theorem 2.3. Let $\mathcal{C}$ be a nowhere dense class of colored graphs. Given a colored graph G of $\mathcal{C}$ and a first-order query φ we need to construct in pseudo-linear time a data structure such that, upon input of a tuple $\overline{a}$ we can return the first tuple $\overline{b}$ such that $\overline{b}$ is greater than or equal to $\overline{a}$ in the lexicographical order and $\overline{b}$ is a solution to φ for G.

For technical reasons (explained in Section 5.1.2 below) we prove the result for FO$^+$ queries (rather than FO queries) and proceed by induction on their arity k. In FO$^+$ we are allowed to use atoms of the form $\mathrm{dist}(x, y) \leq d$ for any constant d, and when evaluated in a structure $\mathcal{A}$, $\mathrm{dist}(x, y)$ is interpreted as the distance between x and y in the Gaifman graph of $\mathcal{A}$. Allowing to use these distance-atoms does not increase the expressive power of first-order logic, as these atoms can clearly be expressed in FO, but it will lead to a different notion of quantifier-rank.

We restate Theorem 2.3 as follows.

Theorem 5.1. *For every (effectively) nowhere dense class of colored graphs $\mathcal{C}$, there is a (computable) function f_1 and an algorithm which, upon input of $k \in \mathbb{N}_{\geq 1}$, $G \in \mathcal{C}$, $\epsilon \in \mathbb{Q}_{>0}$, and an FO$^+$-query φ of arity k, performs a preprocessing in time $f_1(k, |\varphi|, \epsilon) \cdot |G|^{1+\epsilon}$, such that afterwards, upon input of a k-tuple $\overline{a}$, it computes in time $f_1(k, |\varphi|, \epsilon)$ the smallest tuple $\overline{a}' \in \varphi(G)$ such that $\overline{a}' \geq \overline{a}$ in the lexicographical order. In case that no such tuple exists, the algorithm returns Null.*

The proof of Theorem 5.1 uses the following lemma.

Lemma 5.2. *For every (effectively) nowhere dense class of colored graphs $\mathcal{C}$, there is a (computable) function f_2 and an algorithm which, upon input of $k \in \mathbb{N}_{\geq 1}$, $G \in \mathcal{C}$, $\epsilon \in \mathbb{Q}_{>0}$, and an FO$^+$-query φ of arity k, performs a preprocessing in time $f_2(k, |\varphi|, \epsilon) \cdot |G|^{1+\epsilon}$, such that afterwards, upon input of a $(k-1)$-tuple $\overline{a}$ and an element b, it computes in time $f_2(k, |\varphi|, \epsilon)$ the smallest element b' such that $b' \geq b$ and $G \models \varphi(\overline{a}, b')$. In case that no such b' exists, the algorithm returns Null.*

The proofs of Theorem 5.1 and Lemma 5.2 are nested one in the other. We actually prove the following:

- For any $k \in \mathbb{N}_{\geq 1}$, if Theorem 5.1 holds for $k-1$ and Lemma 5.2 holds for k, then Theorem 5.1 also holds for k.
- For any $k \in \mathbb{N}_{\geq 1}$, using that Theorem 5.1 and Lemma 5.2 hold for $k-1$, Lemma 5.2 also holds for k.

The first bullet is easy to prove:

Proof of the first bullet. Let $\mathcal{C}$ a nowhere dense class of colored graphs, $k \in \mathbb{N}_{\geq 1}$, $\epsilon \in \mathbb{Q}_{>0}$, $G \in \mathcal{C}$, let V be the domain of G, and let $\varphi(\overline{x}, y)$ be an FO$^+$-query of arity k. Let $\varphi'(\overline{x})$ be the query $\exists y\, \varphi(\overline{x}, y)$. Since Theorem 5.1 holds for $k-1$, we can spend time $f_1(k-1, |\varphi'|, \epsilon) \cdot |V|^{1+\epsilon}$ to compute the preprocessing for φ' on G. We also spend time $f_2(k, |\varphi|, \epsilon) \cdot |V|^{1+\epsilon}$ to compute the preprocessing of Lemma 5.2 for φ on G.

This ends the preprocessing.

We are now given a k-tuple $(\overline{a}, b)$ and we want to compute the smallest k-tuple in $\varphi(G)$ that is larger than or equal to $(\overline{a}, b)$ in the lexicographical order. Let b' be the element computed in time $f_2(k, |\varphi|, \epsilon)$ by the algorithm of Lemma 5.2 for φ and G on input of $\overline{a}$ and b. Then, the following is true:

- If b' is not Null, it is immediate to see that the desired answer is $(\overline{a}, b')$.
- If b' is Null, let $\overline{a}'$ be the answer computed in time $f_1(k-1, |\varphi|, \epsilon)$ by the algorithm given by Theorem 5.1 for φ' and G on input of the $(k-1)$-tuple $\overline{a}+1$ (where $\overline{a}+1$ is the tuple following $\overline{a}$ in the lexicographical order). Let b_0 be the smallest element of V. If $\overline{a}'$ is Null, it is immediate to see that the desired answer is Null. If $\overline{a}'$ is not Null, let b' be the element computed in time $f_2(k, |\varphi|, \epsilon)$ by Lemma 5.2 for φ and G on input of $\overline{a}'$ and b_0. As $\overline{a}'$ is not Null, b' cannot be Null, and hence the desired answer is $(\overline{a}', b')$.

Note that the algorithm works in time $f_1(k, |\varphi|, \epsilon)$ after a preprocessing in time $f_1(k, |\varphi|, \epsilon) \cdot |V|^{1+\epsilon}$, provided that $f_1(k, |\varphi|, \epsilon) > f_1(k-1, |\varphi|, \epsilon) + 2f_2(k, |\varphi|, \epsilon)$. □

The remaining part of this section is devoted to the proof of the second bullet, which is more involved. In what remains of this paper, for better readability, we replace $f(k, |\varphi|, \epsilon)$ simply by $O(\cdot)$. Recall that the "big O" hides these constants, which are computable as soon as $\mathcal{C}$ is effectively nowhere dense.

We start by introducing additional notations and tools.

5.1 Additional tools

5.1.1 Model checking result and the unary case

As we mentioned earlier, our proofs are by induction on k. The cases when k is zero, i.e. sentences, or k is one, are an immediate consequence of the following theorem.

Theorem 5.3 (Grohe, Kreutzer, Siebertz [17, Theorem 8.1])**.** *Let φ be a first-order query with at most one free variable, and let $\mathcal{C}$ be a nowhere dense class of colored graphs. There is an algorithm that, on input of $G \in \mathcal{C}$ computes $\varphi(G)$ in pseudo-linear time*[2].

We will often reference this theorem as the Model Checking Theorem or the Unary Theorem (depending on the arity of the used query being 0 or 1).

[2]Recall that this means that for all ϵ there is an algorithm. If moreover $\mathcal{C}$ is effectively nowhere dense then the algorithm can be computed from ϵ. In view of similar issues in circuit complexity, the second result is said to be uniform.

5.1.2 FO$^+$, q-rank, and queries in normal form

The first step of our algorithm is to transform the query into some normal form: we need the queries to be local. This is usually performed using Gaifman's Theorem [15]. After that, and as Example 1-C suggests, in order to enumerate a query we will have to enumerate several other queries within some bags of the neighborhood cover. This will be done by induction, and we make sure that the new queries have a quantifier-rank not exceeding the one given by Gaifman's Theorem. This will be presented in Lemma 5.5. Unfortunately, these new queries are no longer local. Furthermore, applying Gaifman's Theorem already blew up the original quantifier-rank of the input query. If at every step of the induction we apply Gaifman's Theorem (which blows up the quantifier-rank) and the mechanism of Lemma 5.5 (which blows up the locality), then the considered radius for the Splitter game is not fixed, and the induction (on the number of rounds) is wrecked. To overcome this issue, we reuse a stronger normal form presented in [18] that maintains some notion of quantifier-rank. This normal form uses the logic FO$^+$ and the notion of q-rank that we describe now.

Recall that in FO$^+$ we are allowed to use atoms of the form $\mathrm{dist}(x,y) \leq d$ for any constant d. Following [17, Section 7.2], we say that a FO$^+$ query φ has *q-rank at most ℓ*, where q and ℓ are in $\mathbb{N}$, if it has quantifier-rank at most ℓ and each distance-atom $\mathrm{dist}(x,y) \leq d$ in the scope of $i \leq \ell$ quantifiers satisfies $d \leq (4q)^{q+\ell-i}$. We also define (as in [17] and [18])

$$f_q(\ell) \quad := \quad (4q)^{q+\ell}.$$

The use of the FO$^+$ notation enables a particular normal form for queries. The goal is to decompose every query into local queries. Intuitively, it is similar to a Gaifman normal form. However, this decomposition has the advantage of controlling the quantifier-rank of the generated local queries.

For formulating the normal form result, we need the following notation. An *(r,q)-independence sentence* is an FO$^+$-sentence of the form

$$\exists z_1 \cdots \exists z_{k'} \Big(\bigwedge_{1 \leq i < j \leq k'} \mathrm{dist}_{>r'}(z_i, z_j) \ \wedge \ \bigwedge_{1 \leq i \leq k'} \psi(z_i) \Big)$$

where $k' \leq q$ and $r' \leq r$ and $\psi(z)$ is quantifier-free.

Given a colored graph G of domain V and a tuple $\overline{a} = (a_1, \ldots, a_k) \in V^k$, for all $r \in \mathbb{N}$ we define the *r-distance type* of $\overline{a}$ as the undirected graph $\tau_r^G(\overline{a})$ whose nodes are $[1,k]$, and where $\{i,j\}$ is an edge iff $G \models \mathrm{dist}_{\leq r}(a_i, a_j)$. The set of all possible distance types with k elements is denoted $\mathcal{T}_k$.

For a colored graph G, a neighborhood cover $\mathcal{X}$ of G, a number $r \in \mathbb{N}$, and a tuple $\overline{a}$ of nodes of G we say that a bag $X \in \mathcal{X}$ *r-covers* $\overline{a}$ if $N_r^G(\overline{a}) \subseteq X$.

Recall from Section 2 that σ_c is the schema of c-colored graphs, i.e., σ_c contains a binary relation symbol E and c unary relation symbols $C_1, \ldots, C_c$.

We now have provided all the notions necessary for stating the normal form of [18] that will help us overcome the issues explained at the beginning of Section 5.1.2. Roughly speaking, this normal form states the following. Given a c-colored graph G and a kr-neighborhood cover $\mathcal{X}$ of G, we can add a number of suitable colors to the nodes of G, turning G into a c'-colored graph $G^\star$. These colors help us to decompose an FO$^+$-formula $\varphi(\overline{x})$ that speaks about G into a set S of very basic sentences (namely, Boolean combinations of (r,q)-independence sentences) speaking about $G^\star$ and a set F of formulas speaking about the subgraphs $G^\star[X]$ induced by the bags $X \in \mathcal{X}$. Whenever given a k-tuple $\overline{a}$ of nodes of G, we can decide whether $G \models \varphi(\overline{a})$ by (1) determining the r-distance type τ of $\overline{a}$, (2) determining the particular sentence ξ of S that fits to τ and that is satisfied by $G^\star$, and (3) considering each connected component I of τ and (3.1) finding the particular formula ψ in F that fits to I, τ, and ξ, (3.2) finding a bag $X \in \mathcal{X}$ that r-covers $\overline{a}_I$, and (3.3) checking whether $G^\star[X] \models \psi(\overline{a}_I)$. Thus, checking whether $G \models \varphi(\overline{x})$ boils down to checking very basic sentences in $G^\star$ along with checking the formulas ψ of F "locally" in the subgraphs of $G^\star$ induced by the bags of $\mathcal{X}$. What is crucial for our application of this decomposition is that the sets S and F are independent of the particular graph G and, furthermore, if $\varphi(\overline{x})$ has q-rank at most ℓ, then all the formulas ψ in F also have q-rank at most ℓ. Let us now turn to the precise formal statement of this decomposition.

Theorem 5.4 (Rank-Preserving Normal Form [18, Theorem 7.1]). *Let $q, k \in \mathbb{N}$ be such that $k \leq q$, let $\ell := q-k$, $r := f_q(\ell)$. For every $c \in \mathbb{N}$ we can compute a $c' \geq c$ such that the following is true for $\sigma := \sigma_c$ and $\sigma^\star := \sigma_{c'}$. For every $\mathrm{FO}^+[\sigma]$-formula $\varphi(\overline{x})$ of q-rank at most ℓ where $\overline{x} = (x_1, \ldots, x_k)$, and for each distance type $\tau \in \mathcal{T}_k$ we can compute a number $m_\tau \in \mathbb{N}$ and, for each $i \leq m_\tau$,*

- *a Boolean combination ξ^i_τ of (r,q)-independence sentences of schema $\sigma^\star$ and*
- *for each connected component I of τ an $\mathrm{FO}^+[\sigma^\star]$-formula $\psi^i_{\tau,I}(\overline{x}_I)$ of q-rank at most ℓ*

such that the following holds:

For all c-colored graphs G and all kr-neighborhood covers $\mathcal{X}$ of G, there is a $\sigma^\star$-expansion $G^\star$ of G (which only depends on $G, \mathcal{X}, q, \ell$) with the following properties, where V denotes the domain of G and $G^\star$:

(a) For all $\overline{a} \in V^k$ and for $\tau := \tau^G_r(\overline{a})$ we have:
$G \models \varphi(\overline{a})$ *iff there is an $i \leq m_\tau$ that satisfies the following condition*

$(*)_i$: $G^\star \models \xi^i_\tau$ *and for every connected component I of τ there is an $X \in \mathcal{X}$ that r-covers $\overline{a}_I$ and* $G^\star[X] \models \psi^i_{\tau,I}(\overline{a}_I)$.

(b) For all $\overline{a} \in V^k$ and for $\tau := \tau^G_r(\overline{a})$, there is at most one $i \leq m_\tau$ such that the condition $(*)_i$ is satisfied.

(c) For all $\overline{a} \in V^k$, all connected components I of $\tau := \tau^G_r(\overline{a})$, all $X, X' \in \mathcal{X}$ that both r-cover $\overline{a}_I$, and all $i \leq m_\tau$,

$$G^\star[X] \models \psi^i_{\tau,I}(\overline{a}_I) \iff G^\star[X'] \models \psi^i_{\tau,I}(\overline{a}_I).$$

Furthermore, for every nowhere dense[3] class $\mathcal{C}$ of colored graphs, there is an algorithm which, when given as input a $G \in \mathcal{C}$, a kr-neighborhood cover $\mathcal{X}$ of G of degree at most $|V|^\epsilon$, and parameters $q, \ell \in \mathbb{N}$, computes $G^\star$ in time $O(|V|^{1+2\epsilon})$.

5.1.3 The Removal Lemma

The following lemma generalizes the idea of Step 4 of the preprocessing for the testing of distance queries in the proof of Proposition 4.2, where we introduced new relations in order to cope with the removal of one node (typically, the answer of Splitter in the Splitter game). The goal is to rewrite a query into an equivalent one when a node is removed from the colored graph.

The lemma is present in a similar form in [18] (see Lemma 7.8); its proof is straightforward. For a tuple $\overline{a}$ and a set I of indices, we denote by $\overline{a}_I$ the projection of $\overline{a}$ onto its components whose indices belong to I. By $\overline{a}_{\setminus I}$ we denote the projection of $\overline{a}$ onto its components whose indices are *not* in I.

Lemma 5.5 (Removal Lemma). *There is an algorithm which takes as input numbers $k, q, \ell, c \in \mathbb{N}$, a c-colored graph G of domain V, a k-ary query $\varphi(\overline{z}) \in \mathrm{FO}^+[\sigma_c]$ of q-rank at most ℓ, a set of free variables $\overline{y} \subseteq \overline{z}$, and a node $s \in V$, and which produces*

- *a number $c' \geq c$,*
- *a query $\varphi'(\overline{z} \setminus \overline{y}) \in \mathrm{FO}^+[\sigma_{c'}]$ of q-rank at most ℓ,*
- *a graph H that is a coloring of $G \setminus \{s\}$ using c' colors (i.e. a $\sigma_{c'}$-expansion of $G \setminus \{s\}$),*

such that for all tuples $\overline{b}$ over V where $\{i \leq k \mid b_i = s\} = \{i \leq k \mid z_i \in \overline{y}\} =: I$ we have

$$G \models \varphi(\overline{b}) \iff H \models \varphi'(\overline{b}_{\setminus I}).$$

Moreover, the running time of this algorithm is linear in the size of G, and

- *c' only depends on c, q, ℓ,*
- *φ' only depends on $c, q, \ell, \varphi, \overline{y}$,*
- *H only depends on c, q, ℓ, G, s.*

[3] Here the paper does not explicitly mention whether the result is uniform. A quick look at the proof clearly indicates that the constants are computable if $\mathcal{C}$ is effectively nowhere dense. This is the case because the proof is performed by induction on the number of rounds that Splitter needs to win the game.

5.1.4 Kernel

For technical reasons, we not only want to know whether a given vertex belongs to some bags. We also want to know whether its p-neighborhood (for some well chosen p) is included in a given bag. This is where the following definition comes into play.

Definition 5.6 (Kernel)**.** Let $\mathcal{X}$ be an r-neighborhood cover of a colored graph G with domain V. For all $X \in \mathcal{X}$ and $p \leq r$, the *p-kernel* of X is the set $K_p(X) := \{a \in V \mid N_p^G(a) \subseteq X\}$.

Lemma 5.7 ([13, Lemma 8.1])**.** *Assume $\mathcal{X}$ is an r-neighborhood cover of a colored graph G. Given a bag X and a number p, we can compute $K_p(X)$ in time $O(p \cdot \|G[X]\|)$.*

Together with the Storing Theorem 3.1 this lemma shows that, after a pseudo-linear time preprocessing we can, given a node a and a bag X, test in constant time whether a belongs to the p-kernel of X.

5.1.5 Shortcut pointers

Example (2)**.** *Our first query example (i.e., Example (1-A)) is limited in the sense that all its solutions satisfy the same distance type requiring that x is close to y. We therefore consider a slightly more complicated query:*

$$q(x, y) \quad := \quad \mathrm{dist}_{>2}(x, y) \ \wedge \ B(y)$$

where B is interpreted as the set of "blue" nodes of a colored graph. The goal is, given a node a, to enumerate all blue nodes that are at distance greater than 2 from a. As previously, we compute a $(2, 4)$-neighborhood cover, and given a node a, we consider the bag $X := \mathcal{X}(a)$.

We then start two concurrent enumeration processes: the first one only enumerates nodes that are within X; the second one enumerates those that are not in X. The first one uses ideas previously presented, diving into $G[X \setminus \{v\}]$ using the query constructed by Lemma 5.5 and induction on λ. We now explain how the second one works.

Note that for every node b outside of X, we have $\mathrm{dist}(a, b) > 2$ as $N_2^G(a) \subseteq X$. Therefore, it suffices to enumerate all blue nodes that don't belong to X. To do so, during the preprocessing phase we compute for all nodes c of G and all bags $X \in \mathcal{X}$ with $c \in X$, the smallest blue node bigger than c that is not in X; let us denote this node by $v(c, X)$. As the degree of our cover is pseudo-constant, the domain of the function $v(\cdot, \cdot)$ is pseudo-linear and the computation of the function can be done in pseudo-linear time using the Storing Theorem 3.1.

However, a naive extension of this idea for queries of larger arities does not work: consider the query

$$q(x, y, z) \ := \ \mathrm{dist}_{>2}(x, z) \ \wedge \ \mathrm{dist}_{>2}(y, z) \ \wedge \ B(z).$$

Assume that, given a pair (a, b) of nodes, we want to enumerate all nodes c such that $(a, b, c) \in q(G)$. Given a, b we consider the bags $X := \mathcal{X}(a)$ and $Y := \mathcal{X}(b)$. Now, we have three concurrent processes, one of them being in charge of enumerating all blue nodes that are neither in X nor in Y. The previous algorithm can enumerate all blue nodes that are not in X, but some of those may be in Y. Given c in $X \cup Y$, computing the smallest blue node c' bigger than c which falls out of $X \cup Y$, may require quadratic time and space. Therefore, we need another approach.

As previously let $v(c, X, Y)$ be the smallest blue node bigger than (or equal to) c that is neither in X nor in Y. With our time and space constraints, it is not possible to compute and store $v(c, X, Y)$ for every tuple c, X, Y. It turns out that we only need to know the result of v to a subdomain of small size. We will first only consider the cases where $c \in X$. For a given c our assumption on the degree of the cover implies that there would only be few such X, but the number of Y remains too big. We therefore further restrict ourselves to the cases where $c \in X$ and $v(c, X) \in Y$. Now, given c there are few possible X to consider and also few possible Y i.e. the domain of $v(\cdot, \cdot, \cdot)$ satisfying these conditions has a pseudo-linear size and the result of the function over this domain can be computed and stored in pseudo-linear time.

Finally, let us explain how to use what has been computed to retrieve $v(c, X, Y)$ for any tuple c, X, Y. First, we test whether $c \in X$ and $c \in Y$. If none are true, then $v(c, X, Y) = c$. Second, assume that

$c \in X$ (the case $c \in Y$ is symmetrical). Then, as for the previous query, we retrieve $v(c, X)$ and test whether $v(c, X) \in Y$. If not, we are done as $v(c, X) = v(c, X, Y)$. Otherwise we are in the specific case where $c \in X$ and $v(c, X) \in Y$, and $v(c, X, Y)$ as been computed and can be retrieved. Lemma 5.8 below generalizes these ideas.

The idea developed in the previous example is now made concrete is the following lemma.

Lemma 5.8 (Skip pointers [30]). *For every nowhere dense[4] class $\mathcal{C}$ of colored graphs, there is a preprocessing algorithm with input: $G \in \mathcal{C}$ of domain V, $r \in \mathbb{N}_{\geq 1}$, $\epsilon \in \mathbb{Q}_{>0}$, $k \in \mathbb{N}$, an r-neighborhood cover $\mathcal{X}$ of G of degree at most $|V|^\epsilon$, and $L \subseteq V$.*

The preprocessing works in time $O(|V|^{1+k\epsilon})$ and afterwards enables us, when given a node b and a set S of at most k bags of $\mathcal{X}$, to compute in constant time the node

$$\text{SKIP}(b, S) \;:=\; \min\Big\{b' \in L \,:\, b' \geq b \,\wedge\, b' \notin \bigcup_{X \in S} K_r(X)\Big\}$$

(recall from Definition 5.6 that $K_r(X)$ is the r-kernel of X).

Proof. The lemma was proved already in [30], but in order to make the current paper a bit more self-contained let us recapitulate the proof details here.

From now on we fix ϵ, r, G, $\mathcal{X}$ and L as in the statement of the lemma.

We assume that all kernels have already been computed. This is without loss of generality modulo a preprocessing of time $O(\|G\|^{1+\epsilon})$ using Lemma 5.7.

The domain of the SKIP$(\cdot, \cdot)$-function is too big (recall that there can be a linear number of bags) so we cannot compute it during the preprocessing phase. Fortunately, computing only a small part of it will be good enough for our needs. For each node b we define by induction a set $SC(b)$ of sets of at most k bags. We start with $SC(b) = \emptyset$ and then proceed as follows.

- For all nodes b of G and for all bags X in $\mathcal{X}$ with $b \in K_r(X)$, we add $\{X\}$ to $SC(b)$.
- For all nodes b of G, for all sets S of bags from $\mathcal{X}$, and all bags X of $\mathcal{X}$, if $|S| < k$ and $S \in SC(b)$ and SKIP$(b, S) \in K_r(X)$, then we add $\{S \cup \{X\}\}$ to $SC(b)$.

In the preprocessing phase we will compute SKIP(b, S) for all nodes b of G and all sets $S \in SC(b)$. Before explaining how this can be accomplished within the desired time constraints, we first show that this is sufficient for deriving SKIP(b, S) in constant time for all nodes b and all sets S consisting of at most k bags of $\mathcal{X}$.

Claim 5.9. *Given a node b of G, a set S of at most k bags of $\mathcal{X}$, and* SKIP(c, S') *for all nodes $c > b$ of G and all sets $S' \in SC(c)$, we can compute* SKIP(b, S) *in constant time.*

Proof. We consider two cases (testing in which case we fall can be done in constant time as the kernels have been computed and S has size bounded by k).

Case 1: $b \in L$ and $b \notin \bigcup_{X \in S} K_r(X)$. In this case, b is SKIP(b, S) and we are done.

Case 2: $b \notin L$ or $b \in \bigcup_{X \in S} K_r(X)$. In this case, let c be the smallest element of L strictly bigger than b. If there is no such c, then SKIP$(b, S) =$ Null and we are done. Otherwise, we proceed as follows.

If $c \notin \bigcup_{X \in S} K_r(X)$, then c is SKIP(b, S) and we are done. Otherwise, we know that $c \in K_r(X)$ for some $X \in S$. Therefore $\{X\} \in SC(c)$. Let S' be a maximal (w.r.t. inclusion) subset of S in $SC(c)$. Since $\{X\} \in SC(c)$, we know that S' is non-empty.

We claim that SKIP$(c, S') =$ SKIP(b, S). To prove this, let us first assume for contradiction that SKIP$(c, S') \in K_r(Y)$ for some $Y \in S$. By definition, this implies that Y is not in S'. Hence $|S'| < |S| \leq k$. Thus, by definition of $SC(c)$ we have $S' \cup \{Y\} \in SC(c)$ and S' was not maximal.

[4] Note that everything in this lemma is computable from the input, even when $\mathcal{C}$ is not effectively nowhere dense. While computing a neighborhood cover is in non-uniform FPT, here the neighborhood cover is part of the input.

Moreover, by definition of SKIP(c, S'), every point between c and SKIP(c, S') is either not in L or in some $K_r(Z)$ with $Z \in S'$ (and therefore $Z \in S$). As all nodes between b and c are not in L, the claim follows. □

We conclude by showing that $SC(b)$ is small for all nodes b of G and that we can compute efficiently SKIP(b, S) for all nodes b and all sets $S \in SC(b)$.

Claim 5.10. *For each node b of G, $|SC(b)|$ has size $O(|V|^{k\epsilon})$. Moreover, it is possible to compute* SKIP(b, S) *for all nodes b of G and all sets $S \in SC(b)$ in time $O(|V|^{1+k\epsilon})$.*

Proof. We start by proving the first statement, and afterwards we use Claim 5.9 to show that we can compute these pointers inductively.

By $SC_\ell(b)$ we denote the subset of $SC(b)$ of sets S with $|S| \leq \ell$. Let d be the degree of the cover $\mathcal{X}$, i.e., $d \leq |V|^\epsilon$. By definition of d, we know that $|SC_1(b)| \leq d$ for all nodes b of G. For the same reason, we have that $|SC_{\ell+1}(b)|$ is of size at most $O(d \cdot |SC_\ell(b)|)$. Therefore, for all $b \in V$, we have:

$$|SC(b)| \;=\; |SC_k(b)| \;\leq\; O(d^k).$$

We compute the pointers for b from b_{max} to b_{min} downwards, where b_{max} and b_{min} are, respectively, the biggest and the smallest element of V. Given a node b in V, assume we have computed SKIP(c, S') for all $c > b$ and $S' \in SC(c)$. We then compute SKIP(b, S) for $S \in SC(b)$ using Claim 5.9.

At each step, the pointer is computed in constant time. Since there are $O(|V|^{1+k\epsilon})$ of them, the time required to compute them is as desired. □

The combination of these two claims proves Lemma 5.8. □

5.2 The main algorithm

We now fix k and assume that Theorem 5.1 and Lemma 5.2 hold for $k-1$. Our goal is to show that Lemma 5.2 then holds for k.

Let us fix an arbitrary number $q \in \mathbb{N}$ with $q \geq k$, let $\ell := q-k$, and let $r := f_q(\ell) = 4q^{q+\ell}$. Note that this is the same choice of parameters as for the Rank-Preserving Normal Form Theorem 5.4. Our goal is to show that the statement of Lemma 5.2 is true for all k-ary queries φ of q-rank at most ℓ.

For every $\lambda \in \mathbb{N}_{\geq 1}$ let $\mathcal{C}_\lambda$ be the subclass of $\mathcal{C}$ consisting of all $G \in \mathcal{C}$ such that Splitter wins the $(\lambda, 2kr)$-Splitter game on G. Clearly, $\mathcal{C}_\lambda \subseteq \mathcal{C}_{\lambda+1}$ for every λ. Since $\mathcal{C}$ is nowhere dense, by Theorem 4.6 there exists a number $\Lambda := \lambda(2kr) \in \mathbb{N}$ such that $\mathcal{C} = \mathcal{C}_\Lambda$. Thus,

$$\mathcal{C}_1 \;\subseteq\; \mathcal{C}_2 \;\subseteq\; \cdots \;\subseteq\; \mathcal{C}_\Lambda \;=\; \mathcal{C}.$$

We proceed by induction on λ and prove the result for all $G \in \mathcal{C}_\lambda$. The induction base for $\lambda = 1$ follows immediately, since by definition of the Splitter game every $G \in \mathcal{C}_1$ has to be edgeless, and thus a naive algorithm works. For the induction step consider a $\lambda \geq 2$ and assume that the statement of Lemma 5.2 already holds for $\lambda-1$, i.e., for all colored graphs in $\mathcal{C}_{\lambda-1}$ and all k-ary queries of q-rank at most ℓ. Additionally, recall that we assume that the full statement of Theorem 5.1 and Lemma 5.2 already holds for all queries of arity $\leq k-1$. This also implies that we can use the statement of Corollary 2.4 for queries of arities $\leq k-1$.

We fix an $\epsilon \in \mathbb{Q}_{>0}$ and an FO$^+$-query $\varphi(\overline{x}, x_k)$ of arity k and q-rank at most ℓ. Here $\overline{x} = (x_1, \ldots, x_{k-1})$ is a $(k-1)$-ary tuple of variables and x_k is the k^{th} variable. Our goal throughout the rest of this section is to provide an algorithm which upon input of a $G \in \mathcal{C}_\lambda$ of domain V performs a preprocessing phase using time $O(|V|^{1+\epsilon})$, such that afterwards upon input of a tuple $\overline{a} \in V^{k-1}$ and an element $a_k \in V$, we can compute in constant time the element $a'_k \in V$ such that:

- $G \models \varphi(\overline{a}, a'_k)$,
- $a'_k \geq a_k$,
- and a'_k is minimal.

If no such element exists, we output Null.

The general idea is to build on the Rank-Preserving Normal Form Theorem (Theorem 5.4) and reduce the computation of the query φ on G to the evaluation of another query within $G[X]$ for the bags X of a neighborhood cover of G that contains the r-neighborhood of an element of $\overline{a}$. In order to do this we need to

1. compute a kr-neighborhood cover $\mathcal{X}$ of G. This can be done thanks to Theorem 4.4.
2. Be able to test distances up to r in order to compute the distance types τ compatible with $\overline{a}$. We have seen how to do this in Section 4.
3. Check for each such τ and each $i \leq m_\tau$ whether ξ^i_τ holds. This can be done thanks to the Model Checking Theorem (Theorem 5.3).
4. Let I be a connected component of τ and $i \leq m_\tau$. We need to evaluate the formulas $\psi^i_{\tau,I}$ given by the Rank-Preserving Normal Form Theorem. If k does not belong to I, then this is just a matter of testing whether $\psi_{\tau,I}(\overline{a}_I)$ holds and this can be done by induction on k.

   The difficulty is the case when $k \in I$ as a'_k is not known.

   In case that $J := I \setminus \{k\} \neq \emptyset$, we are looking for the smallest a'_k such that $\psi^i_{\tau,I}(\overline{a}_J, a'_k)$ holds.

   In this case we consider a suitable bag X whose kernel contains at least one element from $\overline{a}_J$, and compute the smallest suitable a'_k within $G[X]$. This is the difficult part, requiring an induction on λ and whose sketch is given in the next bullet.

   In case that $I = \{k\}$, we consider the bags $\mathcal{X}(a_1), \ldots, \mathcal{X}(a_{k-1})$, use the SKIP pointers provided by Section 5.1.5 to compute several answer candidates, and then return the smallest of these.
5. In order to compute the smallest suitable a'_k within $G[X]$ we have to make sure we consider only elements a'_k that are sufficiently far from all the elements of $\overline{a}$ not in $\overline{a}_I$ and sufficiently close to all the elements of $\overline{a}_I$. To do this we first transform the formula $\psi^i_{\tau,I}$ into $\Psi^i_{\tau,I,p}$ by adding sufficiently many free variables in order to include the elements of $\overline{a}$ that fall within X and adding clauses enforcing the necessary distance properties (note that we do not need to consider the elements of $\overline{a}$ not in X as those are necessarily far from a'_k. The resulting formula is described at Step 7 of the preprocessing phase).

   It remains to find the smallest a'_k for which $G[X] \models \Psi^i_{\tau,I,p}(\overline{a}, a'_k)$. Either this is s_X, the answer of Splitter in the Splitter game when Connector plays the center c_X of X, or we can evaluate the formula computed from $\Psi^i_{\tau,I,p}$ by the Removal Lemma and evaluate it by induction on λ on a component resulting from the Splitter game.

This is essentially what we do.

We first describe the preprocessing phase, then the answering procedure. While describing these, we also provide a runtime analysis and a correctness proof.

5.2.1 The preprocessing phase

Consider the query $\varphi(\overline{x}, x_k)$ with $\overline{x} = (x_1, \ldots, x_{k-1})$. We choose $\delta > 0$ such that $2\delta + \delta^2 + k\delta \leq \epsilon$.

Let $G \in \mathcal{C}_\lambda$ be the input and let $n := |V|$ be the size of the domain V of G. The preprocessing phase is composed of the following steps:

1. Let $f_{\mathcal{C}}(\cdot, \cdot)$ be the function provided by Theorem 4.4. If $n \leq f_{\mathcal{C}}(2kr, \delta)$, we use a naive algorithm to compute the query result $\varphi(G)$ and trivially provide the functionality claimed by Lemma 5.2.

   From now on, consider the case where $n > f_{\mathcal{C}}(2kr, \delta)$. Recall from Theorem 2.1 that this, without loss of generality, implies that $\|G\| \leq n^{1+\delta}$.
2. For every $k' < k$ and every distance type $\tau' \in \mathcal{T}_{k'}$, consider the k'-ary query $\rho_{\tau'}(x_1, \ldots, x_{k'})$ defined as the conjunction of the formulas $\mathrm{dist}_{\leq r}(x_i, x_j)$ for all edges $\{i, j\}$ of τ' and the conjunction of the formulas $\neg\, \mathrm{dist}_{\leq r}(x_i, x_j)$ for all $i, j \in [1, k']$ with $i \neq j$ for which τ' does not contain the edge $\{i, j\}$.

Note that for every $\overline{a} \in V^{k'}$ we have $G \models \rho_{\tau'}(\overline{a})$ iff $\tau' = \tau_r^G(\overline{a})$. We spend time $O(|V|^{1+\delta})$ to perform the preprocessing phase provided by the Proposition 4.2 for the distance query $\mathrm{dist}_{\leq r}(z_1, z_2)$ used in the query $\rho_{\tau'}$. Henceforth, for each $k' < k$ and each $\tau' \in \mathcal{T}_{k'}$, this will enable us upon input of a tuple $\overline{a} \in V^{k'}$, to test in constant time whether $G \models \rho_{\tau'}(\overline{a})$, i.e., whether $\tau_r^G(\overline{a}) = \tau'$.

3. Using the algorithm provided by Theorem 4.4, we compute a $(kr, 2kr)$-neighborhood cover $\mathcal{X}$ of G with degree at most n^δ.

   Furthermore, in the same way as in [17, Lemma 6.10], we also compute for each $X \in \mathcal{X}$ a list of all $b \in V$ satisfying $\mathcal{X}(b) = X$, and we compute a node c_X such that $X \subseteq N_{2kr}^G(c_X)$.

   In addition, we use Lemma 5.7 to compute for every bag $X \in \mathcal{X}$ the r- kernel of X, i.e., the set $K_r(X) = \{a \in X \mid N_r^G(a) \subseteq X\}$.

   All of this can be efficiently stored and retrieved with the Storing Theorem 3.1. This can be done in time $O(n^{1+\delta})$.

4. Let σ be the schema of G and use the algorithm provided by the Rank-Preserving Normal Form Theorem 5.4 upon input of $\varphi, G, \mathcal{X}$ to compute in time $O(n^{1+\delta})$ the schema $\sigma^\star$, the $\sigma^\star$-expansion $G^\star$ of G, and for each distance type $\tau \in \mathcal{T}_k$ the number m_τ, the $\mathrm{FO}^+[\sigma^\star]$-sentences ξ_τ^i and the $\mathrm{FO}^+[\sigma^\star]$-formulas $\psi_{\tau,I}^i(\overline{x}_I)$ of q-rank at most ℓ, for each $i \leq m_\tau$ and each connected component I of τ.

   Afterwards, we proceed in the same way as in [17, 18] to compute, within total time $O(n^{1+\delta})$, for every $X \in \mathcal{X}$ the structure $G^\star[X]$, and we let $G_X^\star$ be the expansion of $G^\star[X]$ where the new unary relation symbol K is interpreted by the r- kernel $K_r(X)$.

   Note that $G_X^\star$ has domain X and belongs to the class $\mathcal{C}_\lambda$.

5. By the Rank-Preserving Normal Form Theorem 5.4 for all $\overline{a} = (a_1, \ldots, a_{k-1})$ in V^{k-1} and all $a_k \in V$ we have that $G \models \varphi(\overline{a}, a_k)$ if and only if there is a distance type $\tau \in \mathcal{T}_k$ and an $i \leq m_\tau$ such that:

   (a) $\tau = \tau_r^G(\overline{a}, a_k)$

   (b) $G^\star \models \xi_\tau^i$

   (c) $G^\star[\mathcal{X}(a_k)] \models \psi_{\tau,J}^i(\overline{a}_J)$, where J is the connected component of τ with $k \in J$.
   Note that for a_k the bag $\mathcal{X}(a_k)$ r-covers the tuple $\overline{a}_J$, since $k \in J$, J is a connected component of $\tau = \tau_r^G(\overline{a}, a_k)$, $|J| \leq k$, and $\mathcal{X}$ is a kr-neighborhood cover of G.

   (d) For all connected components I of τ with $k \notin I$ we have $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi_{\tau,I}^i(\overline{a}_I)$, where $\mathcal{X}(\overline{a}_I)$ is defined to be $\mathcal{X}(a_{\min(I)})$; note that this bag r-covers $\overline{a}_I$.

   Using the Model Checking Theorem 5.3, we can test in time $O(n^{1+\delta})$ for every $\tau \in \mathcal{T}_k$ and $i \leq m_\tau$ whether $G^\star \models \xi_\tau^i$.

   We continue by performing the following preprocessing steps for every distance type $\tau \in \mathcal{T}_k$ and every number $i \leq m_\tau$ such that $G^\star \models \xi_\tau^i$. If there is no such τ and i then we can safely stop as there is no tuple $(\overline{a}, a_k)$ with $G \models \varphi(\overline{a}, a_k)$.

6. For every connected component I of τ with $k \notin I$, the query $\psi_{\tau,I}^i(\overline{x}_I)$ has arity $\leq k-1$. By our induction hypothesis, the statement of Theorem 5.1 (and its corollaries) already holds for this query. Thus, for each $X \in \mathcal{X}$ we can use time $O(|X|^{1+\delta})$ to perform the preprocessing phase provided by Theorem 5.1 for the query $\psi_{\tau,I}^i(\overline{x}_I)$ and the colored graph $G_X^\star$. Using Corollary 2.4 and having performed this preprocessing will henceforth enable us, upon input of a tuple $\overline{a}_I$ of elements in X, to test in constant time whether $G^\star[X] \models \psi_{\tau,I}^i(\overline{a}_I)$.

   The total time taken by these preprocessing steps is of order at most

$$\sum_{X \in \mathcal{X}} |X|^{1+\delta} \;\leq\; \Big(\sum_{X \in \mathcal{X}} |X| \Big)^{1+\delta} \;\leq\; (n^{1+\delta})^{1+\delta} \;\leq\; n^{1+\epsilon}$$

   (here, we use that $\mathcal{X}$ has degree $\leq n^\delta$, which implies that $\sum_{X \in \mathcal{X}} |X| \leq n^{1+\delta}$).

7. Let J be the connected component of τ with $k \in J$ and note that x_k is the last variable in the tuple $\overline{x}_J$. Let $z_1, \ldots, z_{k-|J|}$ be new variables and consider for each $p \in \{0, \ldots, k-|J|\}$ the query

$$\Psi^i_{\tau,J,p}(z_1, \ldots, z_p, \overline{x}_J) \;:=\; \psi^i_{\tau,J}(\overline{x}_J) \;\wedge\; K_r(x_k) \;\wedge\; \rho_\tau(\overline{x}_J) \;\wedge \bigwedge_{p' \in [1,p]} \mathrm{dist}(x_k, z_{p'}) > r.$$

   Note that the query $\Psi^i_{\tau,J,p}$ has q-rank at most ℓ.

   As explained in the sketch we aim at restricting the evaluation to the substructure induced by a bag X. The query then ensures that $\psi^i_{\tau,J}(\overline{x}_J)$ is satisfied and that the nodes of $\overline{x}$ that are not in $\overline{x}_J$ but fall in the bag X are sufficiently far away from x_k. Since we don't know in advance how many there will be, we anticipate all possibilities (by considering every $p \leq k-|J|$). Note that the arity of the query $\Psi^i_{\tau,J,p}$ is k for $p = k-|J|$, and it is smaller than k for smaller p.

   For every $X \in \mathcal{X}$ we would like to provide the following functionality: Upon input of a tuple of $p+|J|-1$ elements $c_1, \ldots, c_p, \overline{a}_{J\setminus\{k\}}$ and an element a_k, we want to be able to compute in constant time the smallest a'_k in X such that $G^\star_X \models \Psi^i_{\tau,J,p}(c_1, \ldots, c_p, \overline{a}_{J\setminus\{k\}}, a'_k)$ and $a'_k \geq a_k$.

   But as the query's arity $p+|J|$ might be as large as k, we do not have the statement of Lemma 5.2 available for this query. As a remedy, we perform the following steps 8–11 which make use of our second inductive assumption, stating that Lemma 5.2 already holds for the class $\mathcal{C}_{\lambda-1}$ and for queries of arity up to k.

8. Recall that in Step 3 we have already computed for every X in $\mathcal{X}$ a node c_X whose $2kr$-neighborhood contains X. Since $G \in \mathcal{C}_\lambda$, we know that Splitter wins the $(\lambda, 2kr)$-Splitter game on G. For every $X \in \mathcal{X}$ we now compute a node s_X that is Splitter's answer if Connector plays c_X in the first round of the $(\lambda, 2kr)$-Splitter game on G. From Remark 4.7 we know that the nodes $(s_X)_{X\in\mathcal{X}}$ can be computed within total time $O(n^{1+\delta})$.

9. Let $p \in \{0, \ldots, k-|J|\}$, let $k' := p+|J|$ and let $\overline{z} = (z_1, \ldots, z_{k'}) := (z_1, \ldots, z_p, \overline{x}_J)$. Recall that $k \in J$ and x_k is the last variable of the tuple $\overline{x}_J$, hence $x_k = z_{k'}$. For every set $\overline{y}$ of variables from $\overline{z}$, we proceed as follows. For every $X \in \mathcal{X}$ we apply the Removal Lemma 5.5 to the colored graph $G^\star_X$, the query $\Psi^i_{\tau,J,p}(\overline{z})$, the variables $\overline{y}$, and the node s_X. This yields a query $\Psi'^i_{\tau,J,p,\overline{y}}(\overline{z} \setminus \overline{y})$ of q-rank at most ℓ and an expansion $H^\star_X$ of $G^\star_X \setminus \{s_X\}$ by unary predicates, such that for all k'-tuples $\overline{b}$ over X where $\{i \leq k' \mid b_i = s_X\} = \{i \leq k' \mid z_i \in \overline{y}\} =: \Delta$ we have

$$G^\star_X \models \Psi^i_{\tau,J,p}(\overline{b}) \quad\Longleftrightarrow\quad H^\star_X \models \Psi'^i_{\tau,J,p,\overline{y}}(\overline{b}_{\setminus\Delta}).$$

   For every $X \in \mathcal{X}$, this takes time $O(\|G^\star_X\|)$. Hence, the total time taken by Step (9) is in $O(\sum_{X\in\mathcal{X}} \|G^\star_X\|)$. Since a given edge (or a node) can only be found in at most n^δ different bags *(due to the degree of our cover)*, we have $\sum_{X\in\mathcal{X}} \|G^\star_X\| \leq n^\delta \|G\|$. Moreover, after Step 1 we know that $\|G\| \leq n^{1+\delta}$. Hence, $n^\delta \|G\| \leq n^{1+2\delta} \leq n^{1+\epsilon}$.

10. By our choice of the node s_X we know that Splitter wins the $(\lambda-1, 2kr)$-Splitter game on $H^\star_X$. Hence, $H^\star_X$ belongs to $\mathcal{C}_{\lambda-1}$, for every $X \in \mathcal{X}$. Using our induction hypothesis, we thus spend for every X time at most $O(|X|^{1+\delta})$ to perform the preprocessing phase for Lemma 5.2 on $H^\star_X$, since the queries have arity at most k and q-rank at most ℓ. Here, we carry this out for the queries $\Psi'^i_{\tau,J,p,\overline{y}}(\overline{z} \setminus \overline{y})$, for all $\overline{y} \subseteq \overline{z}$.

    Note that henceforth, this will allow us to do the following for every $X \in \mathcal{X}$:

    For any $\overline{y}$ with $x_k \notin \overline{y}$, when given an assignment $\overline{a}'$ in $X \setminus \{s_X\}$ to the variables in $\overline{z} \setminus (\overline{y} \cup \{x_k\})$, and for any element b in $X \setminus \{s_X\}$, we can compute in constant time the smallest $b' \in X \setminus \{s_X\}$ such that $H^\star_X \models \Psi'^i_{\tau,J,p,\overline{y}}(\overline{a}', b')$ and $b' \geq b$.

    The total time taken for this preprocessing step is of order at most $\sum_{X\in\mathcal{X}} |X|^{1+\delta} \leq n^{1+\epsilon}$.

11. In addition of the previous step, we also spend, for every $X \in \mathcal{X}$, time at most $O(|X|^{1+\delta})$ to perform the preprocessing phase for Theorem 5.1 on $H^\star_X$ for every query $\Psi'^i_{\tau,J,p,\overline{y}}(\cdot)$, when $x_k \in \overline{y}$. Since $x_k \in \overline{y}$, the arity of the query is at most $k-1$.

    This allows us (using Corollary 2.4), given an assignment $\overline{a}'$ in $X \setminus \{s_X\}$ to the variables in $\overline{z} \setminus \overline{y}$, to test in constant time whether $H^\star_X \models \Psi'^i_{\tau,J,p,\overline{y}}(\overline{a}')$.

    Again, the total time taken for this preprocessing step is in $O(n^{1+\epsilon})$.

The next two steps are only performed when $J = \{k\}$; otherwise the preprocessing phase stops here. Note that if $J = \{k\}$, then the tuple $\overline{x}_J$ only consists of the variable x_k. Furthermore, for a tuple $\overline{a} = (a_1, \ldots, a_{k-1})$ and a node a_k, the tuple $\overline{a}_J$ consists of the single element a_k.

12. We compute the set
$$L^i_{\tau,J} \quad := \quad \{\, a_k \in V \mid G^\star[\mathcal{X}(a_k)] \models \psi^i_{\tau,J}(a_k) \,\}.$$
    This can be achieved as follows: For each $X \in \mathcal{X}$ use the algorithm provided by the Unary Theorem 5.3 to compute in time $O(|X|^{1+\delta})$ the result of the unary query $\psi^i_{\tau,J}$ on $G^\star_X$, and let L_X be the intersection of this query result with the list of all elements b with $\mathcal{X}(b) = X$ (recall that we already precomputed this list in Step 3).

    Furthermore, $L^i_{\tau,J}$ is the disjoint union of the sets L_X for all $X \in \mathcal{X}$. It can therefore be computed in time $O(\sum_{X \in \mathcal{X}} |X|^{1+\delta})$ and hence in time $O(n^{1+\epsilon})$.

13. We compute the skip pointers with respect to the set $L := L^i_{\tau,J}$ and $K_r(X)$ for all $X \in \mathcal{X}$ as in Lemma 5.8. By Lemma 5.8 this is done in time $O(n^{1+k\delta})$ and hence in time $O(n^{1+\epsilon})$.

This concludes the preprocessing phase, and we have argued that all preprocessing steps can be done in total time $O(n^{1+\epsilon})$.

5.2.2 The answering phase

We now describe how, upon input of a tuple $\overline{a} = (a_1, \ldots, a_{k-1}) \in V^{k-1}$ and an element $b := a_k \in V$ we can compute in constant time the minimal b' such that

- $G \models \varphi(\overline{a}, b')$,
- and $b' \geq a_k$,

or the value Null in case that such a b' does not exist.

What we actually do is the following: For all possible pairs (τ, i), we compute in constant time the minimal $b'_{\tau,i}$ such that:

(a) $\tau = \tau^G_r(\overline{a}, b'_{\tau,i})$,

(b) $G^\star \models \xi^i_\tau$,

(c) $G^\star[\mathcal{X}(b'_{\tau,i})] \models \psi^i_{\tau,J}(\overline{a}_{J\setminus\{k\}}, b'_{\tau,i})$, where J is the connected component of τ with $k \in J$,

(d) for all connected components I of τ with $k \notin I$ we have $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)$, where $\mathcal{X}(\overline{a}_I)$ denotes the bag $\mathcal{X}(a_{\min(I)})$, and

(e) $b'_{\tau,i} \geq a_k$,

As there are only a constant number of pairs (τ, i), we can compute all $b'_{\tau,i}$ and output the smallest of them. By the Rank-Preserving Normal Form Theorem 5.4, this is the correct answer. If all of them are equal to Null, we output Null.

From now on, we consider a fixed pair (τ, i), and we therefore omit the subscript (τ, i). Let τ' be the subgraph of τ induced on $\{1, \ldots, k-1\}$. By the functionality provided by Step 2 of the preprocessing

phase we can test in constant time whether $\tau_r^G(\overline{a}) = \tau'$. If this is not the case, we know that for this τ, the condition of item (a) cannot be satisfied by any $b' \in V$. Therefore, we can safely output Null.

Otherwise, i.e., if $\tau_r^G(\overline{a}) = \tau'$, we proceed as follows.

By Step 5 of the preprocessing phase, we can check in constant time whether $G^\star \models \xi_\tau^i$. We thus know if item (b) is satisfied. Furthermore, using the functionality provided in Step 6 of the preprocessing phase, we can test in constant time for all connected components I of τ with $k \notin I$, whether $G^\star[\mathcal{X}(\overline{a}_I)] \models \psi^i_{\tau,I}(\overline{a}_I)$. Afterwards, we know if the item (d) is satisfied.

If one of the items (b) or (d) is not satisfied, we know that there is no matching solution for this $\overline{a}$ and (τ, i), and we can therefore safely output Null.

Otherwise, i.e., if the items (b) and (d) are satisfied, we let J be the connected component of τ with $k \in J$, and we proceed with the two following cases.

Case I: $J = \{k\}$**.**

In this case, every matching solution b' for this $\overline{a}$, this $b := a_k$, and this (τ, i) has to be of distance greater than r to every element in $\overline{a}$. Consider the bags $\mathcal{X}(a_1), \ldots, \mathcal{X}(a_{k-1})$, let $k' := |\{\mathcal{X}(a_\nu) \mid \nu \in \{1, \ldots, k-1\}\}|$, and let $X_1, \ldots, X_{k'}$ be a list of these bags. Clearly, $k' \leq k-1$, and for each component a_ν of $\overline{a}$, there is exactly one κ such that $\mathcal{X}(a_\nu) = X_\kappa$.

For each $\kappa \leq k'$, we have the following definitions:

- Let p_κ be the number of elements in $\{a_1, \ldots, a_{k-1}\}$ that belong to X_κ, and let $\overline{c}_\kappa := (c_{\kappa,1}, \ldots, c_{\kappa,p_\kappa})$ be a list of all these elements.
- Let $\overline{y}$ consist of the variables x_ν for all $\nu \in \{1, \ldots, k-1\}$ such that $a_\nu = s_{X_\kappa}$.
- Let $\overline{c}'_\kappa = (c'_{\kappa,1}, \ldots, c'_{\kappa,p'_\kappa})$ be a list of all elements of $\overline{c}_\kappa$ that are not equal to s_{X_κ}.
- Let b_κ be the smallest element of $X_\kappa \setminus \{s_{X_\kappa}\}$ with $b_\kappa \geq b$, obtained using the data structure of the Storing Theorem 3.1 when computing the neighborhood cover.

We compute the following $2k' + 1$ answer candidates:

- For each $\kappa \in \{1, \ldots, k'\}$ we want to compute the smallest element in $X_\kappa \setminus \{s_{X_\kappa}\}$ that is far away from all the nodes in the list $\overline{c}_\kappa$ and that is $\geq b_\kappa$.

  More precisely, we define b'_κ as the smallest element of $X_\kappa \setminus \{s_{X_\kappa}\}$ such that $G^\star_{X_\kappa} \models \Psi^i_{\tau,J,p_\kappa}(\overline{c}_\kappa, b'_\kappa)$ and $b'_\kappa \geq b_\kappa$. Note that this is exactly the smallest node in $K_r(X_\kappa) \setminus \{s_{X_\kappa}\}$ that satisfies the items (a), (c) and (e) .

  From the statement made at the end of Step 9 of the preprocessing phase, we know that to compute this node b'_κ, we can use the functionality provided by Step 10 of the preprocessing phase: For all $\kappa \leq k'$, we compute b'_κ as the smallest element in $X_\kappa \setminus \{s_{X_\kappa}\}$ such that $H^\star_{X_\kappa} \models \Psi'^i_{\tau,J,p_\kappa,\overline{y}}(\overline{c}'_\kappa, b'_\kappa)$ and $b'_\kappa \geq b_\kappa$.

- For each $\kappa \in \{1, \ldots, k'\}$ we want to check if $G^\star_{X_\kappa} \models \Psi^i_{\tau,J,p_\kappa}(\overline{c}_\kappa, b)$ holds for the particular node $b := s_{X_\kappa}$. If this is the case we set b''_κ to s_{X_κ}, otherwise to Null.

  By the statement made at the end of Step 9 of the preprocessing phase, this check can be performed by using the functionality provided by Step 11 of the preprocessing phase: We simply check in constant time if $H^\star_{X_\kappa} \models \Psi'^i_{\tau,J,p_\kappa,\overline{y}\cup\{x_k\}}(\overline{c}'_\kappa)$.

- Using the functionality provided by Step 13 of the preprocessing phase, we compute in constant time b'_0, the smallest element of the set

$$\{\, b_0 \in L \mid b_0 \notin \bigcup_{\kappa \leq k'} K_r(X_\kappa) \wedge b_0 \geq b\}$$

  where $L := L^i_{\tau,J}$ is the set computed in Step 12 of the preprocessing phase.

It should be clear that every b'_κ, s_{X_κ} and b'_0 is a matching solution for $\overline{a}$ and (τ, i). Let us now argue that the smallest such matching solution for $\overline{a}$ and (τ, i) (that is $\geq b$) is one of them: Note that any matching solution b' for $\overline{a}$ and (τ, i) is either in the r-kernel of one of the canonical bags $\mathcal{X}(a_\nu)$, and then it must be one of the b'_κ or one of the b''_κ, or it must be b'_0. Therefore, we can safely output

$$b' := \min\Big(\{b'_\kappa \mid \kappa \leq k'\} \cup \{b''_\kappa \mid \kappa \leq k'\} \cup \{b'_0\}\Big).$$

Case II: $\{k\} \subsetneq J$

W.l.o.g. let us assume that $1 \in J$ and that $\{1, k\}$ is an edge in τ.

Regarding item (c), note that the Rank-Preserving Normal Form Theorem 5.4 tells us that instead of the bag $\mathcal{X}(b')$ we can use *any* bag X that r-covers $\overline{a}_J$.

We define:

- $X := \mathcal{X}(a_1)$. Note that every $b \in V$ that satisfies item (a) belongs to X and, moreover, X r-covers $\overline{a}_J$ for $\overline{a} = (a_1, \ldots, a_{k-1})$.
- Let p be the number of elements of $\{a_1, \ldots, a_{k-1}\}$ that belong to X but not to the tuple $\overline{a}_J$. Let $c_1, \ldots, c_p$ be the list of all these elements.
- Let $c'_1, \ldots, c'_{p'}$ be the elements of $c_1, \ldots, c_p$ that are not equal to s_X.
- Let $\Gamma := J \setminus \{k\}$.
- Let $\overline{a}'_\Gamma$ be the tuple obtained from $\overline{a}_\Gamma$ by removing all components whose entry is s_X.
- Let $\overline{y}$ consist of the variables x_ν for all $\nu \in \{1, \ldots, k-1\}$ such that $a_\nu = s_X$.
- Let b_X be the smallest element of $X \setminus \{s_X\}$ that is $\geq a_k$. It is derived from the data structure of the Storing Theorem 3.1 obtained when computing the neighborhood cover.

Since all matching b' must be close to a_1 in this case, it suffices to compute the following two elements:

- We want to compute the smallest element in $X \setminus \{s_X\}$ that is far from all the nodes $c_1, \ldots, c_p$. More precisely, we want to compute the smallest b' in X that is $\geq a_k$ (and therefore $\geq b_X$) such that
$$G^\star_X \models \Psi^i_{\tau,J,p}(c_1, \ldots, c_p, \overline{a}_\Gamma, b').$$
Note that such a node precisely satisfies the items (a), (c) and (e).

  From the statement made at the end of Step 9 of the preprocessing phase, we know that to compute such nodes b', we can use the functionality provided by Step 10 of the preprocessing phase: We compute the smallest b'_1 in $X \setminus \{s_X\}$ such that $b'_1 \geq b_X$ and
$$H^\star_X \models \Psi'^i_{\tau,J,p,\overline{y}}(c'_1, \ldots, c'_{p'}, \overline{a}'_\Gamma, b'_1).$$
- We also want to check if $G^\star_X \models \Psi^i_{\tau,J,p}(c_1, \ldots, c_p, \overline{a}_\Gamma, b'_2)$ holds for the particular node $b'_2 := s_X$, and if so, we want to output it.

  From the statement made at the end of Step 9 of the preprocessing phase, we know that this check can be performed by using the functionality provided by Step 11 of the preprocessing phase: We simply check in constant time if
$$H^\star_X \models \Psi'^i_{\tau,J,p,\overline{y}\cup\{x_k\}}(c'_1, \ldots, c'_{p'}, \overline{a}'_\Gamma).$$

The final output is $b' := \min\{b'_1, b'_2\}$. This concludes the description of the answering procedure. While describing this procedure, we have already verified that it outputs exactly the smallest $b' \in V$ that is $\geq b$ and satisfies $G \models \varphi(\overline{a}, b')$.

As we compute only a constant number of answer candidates (at most $2k+1$ for each pair (τ, i)) and then take the smallest of them, the correct solution is computed in constant time.

This completes the proof of Lemma 5.2 and hence also completes the proof of Theorem 5.1.

6 Conclusion

We have shown how to efficiently enumerate the results of first-order queries over any nowhere dense class of databases. We achieved constant delay enumeration after a pseudo-linear time preprocessing. We also showed that after a pseudo-linear preprocessing we can, on input of an arbitrary tuple, test in constant time whether it is a solution to the query.

We did not mention the size of the constant factor. Already for boolean queries the constant factor is at least a tower of exponentials whose height depends on the size of the query. Moreover, an elementary constant factor is not achievable if the class of structures contains all trees (unless FPT $=$ AW[$*$], cf. [14]).

Furthermore, when $\mathcal{C}$ is not *effectively* nowhere dense, the main algorithm is not even FPT. We carefully highlighted the steps that require $\mathcal{C}$ to be *effectively* nowhere dense in order to obtain computable constant factors.

An improvement of our work would be to extend the results to a dynamic setting that avoids recomputing from scratch the index built during the preprocessing phase. For instance, the index structure allowing for constant delay enumeration can be updated in constant time in the setting of FO-queries over classes of databases of bounded degree [7] and in the setting of q-hierarchical unions of conjunctive queries over arbitrary databases [6, 8]. In the nowhere dense case, constant update time seems unrealistic, as already for boolean queries over trees the best we can do so far are logarithmic time updates [5].

It seems plausible that there exists an index structure, computable in pseudo-linear time and allowing for constant delay enumeration and logarithmic time updates. Preliminary results were obtained in this direction for very simple structures such as words [28] and trees [2]. Generalizations to more complex structures remains for future work.

One could finally wonder whether a linear preprocessing time can be achieved. This would in particular imply that the model checking problem could be solved in time linear in the size of the database. Up to now, the best time complexity for the model checking problem over nowhere dense databases is pseudo-linear and it is an open problem whether this can be done in time linear in the size of the input database.

7 Appendix: Proofs from Section 3

This section is devoted to the proof Theorem 3.1. For ease of read, we recall the Theorem and some parts of Section 3.

Theorem (Storing Theorem)**.** *For every fixed $k \in \mathbb{N}$ and $\epsilon > 0$, there is an integer $c \in \mathbb{N}$ such that for every integer $n \in \mathbb{N}$ there is a data structure that stores the value of a k-ary function f of domain $\mathrm{Dom}(f) \subseteq [n]^k$ with:*

- *initialization time $c \cdot |\mathrm{Dom}(f)| \cdot n^\epsilon$,*
- *update time $c \cdot n^\epsilon$ whenever a pair[5] $(\overline{a}, b)$ is added to or removed from f,*
- *lookup time c,*
- *and at any point in time, the space used by the data structure is $c \cdot |\mathrm{Dom}(f)| \cdot n^\epsilon$.*

Here, lookup means that given a tuple $\overline{a} \in [n]^k$, the algorithm either answers b if $\overline{a} \in \mathrm{Dom}(f)$ and $f(\overline{a}) = b$; or $\overline{a}'$ if $\overline{a} \notin \mathrm{Dom}(f)$ and $\overline{a}' := \min\{\overline{x} \in \mathrm{Dom}(f) \ : \ \overline{x} > \overline{a}\}$; or Null if no such tuple exists.

7.1 Description of the data structure

Fix ϵ and n. Let $d := \lceil n^\epsilon \rceil$ and $h := \lceil \frac{1}{\epsilon} \rceil$. As usual, for $x \in \mathbb{Q}$, $\lceil x \rceil$ denotes the smallest integer y such that $x \leq y$.

[5] we identify f with its graph $\{(\overline{a}, b) \mid \overline{a} \in \mathrm{Dom}(f),\ f(\overline{a}) = b\}$

Every $i \in [n]$ can be uniquely decomposed in base d into a string of length h whose letters are from $[0, d-1]$ since $d^h \geq n$. We arbitrarily assume that the string starts with the higher powers of d and ends with the lowest ones. Given all this, every tuple in $[n]^k$ can be decomposed into a string of length kh whose letters are from $[0, d-1]$. We then associate to the function f a partial tree $T(f)$ of maximal depth kh and degree d, where each node has 0 or d children and each leaf at depth kh represents an element of the domain of f (by looking at the sequence of child numbers in the path from the root to that leaf). The size of $T(f)$ is then $O(n^\epsilon \cdot |\text{Dom}(f)|)$.

Our data structure is an encoding of $T(f)$ with extra information in order to navigate efficiently in the tree and to update it efficiently. As for leaves, to any node of $T(f)$ at depth i we can associate a string over $[0, d-1]$ of length i. Given a leaf of $T(f)$ we associate a tuple $\overline{b}$ as the smallest tuple (in lexicographical order) of the domain of f whose encoding has a prefix larger than the one of the current node.

Each inner node of the tree associated to f is represented by $d+1$ consecutive registers in our memory each containing a pair (δ, r) where δ is either $0, 1$ or -1 and r is a value that will help us navigating in the tree.

Consider an inner node x of $T(f)$ and assume that x is the i^{th} child of y. Let R be the i^{th} register representing y and R' be the first register representing x. Then the content of R is $(1, R')$ and the content of the last register representing x contains $(-1, R)$. This encodes the parent/child relation of $T(f)$. The rest of the encoding will help updating the structure efficiently.

If the j^{th} child of y is a leaf, for $j \leq d$, then the content of the j^{th} register representing y is $(0, \overline{b})$ where $\overline{b}$ is the tuple associated to that leaf.

In the case when x is at depth $kh-1$ (i.e. all its children are leaves), for $i \leq d$, we set the content of the i^{th} register representing x as $(1, f(\overline{a}))$ if the i^{th} leaf of x represents a tuple $\overline{a}$ in the domain of f, and as $(0, \overline{b})$ otherwise where $\overline{b}$ is the tuple associated to that leaf.

Finally, we have a register R_0 that contains the next available (unused) register.

Our data structure is illustrated in Figure 1.

7.2 Accessing the information.

7.2.1 Accessing the values of the function.

Given a k-tuple $\overline{a} \in [n]^k$, our goal is to test whether $\overline{a}$ is in the domain of f, and if so, to output $f(\overline{a})$.

The two following procedures enable us to perform this in time $O(kh)$, hence in constant time. Recall that $d := \lceil n^\epsilon \rceil$ and $h := \left\lceil \frac{1}{\epsilon} \right\rceil$.

We will use different registers names, R, S, and S' for a better readability. While the R registers store our functions, the S and S' registers can be seen as two working tapes of constant size (kh). The first procedure decomposes a k-tuple into a sequence of numbers in $[0, d-1]$ of length kh. This is a simple decomposition in base d using Euclidean division.

Algorithm 1 Decomposition$(a_1, \ldots, a_k)$

```
for i = 1 to k do                          ▷ basically an Euclidean division
    A ← a_i
    for j = h(i − 1) to hi − 1 do
        B ← ⌊A/d⌋                          ▷ the quotient of A/d
        S_j ← A − d · B                    ▷ the remainder of A/d
        A ← B
    end for
end for
```

The above procedure sets registers $S_0, \ldots, S_{kh-1}$ so that:

$$a_i = \sum_{j=h(i-1)}^{hi-1} S_j \cdot d^{j-h(i-1)}.$$

The next procedure returns the value of $f(\overline{a})$. It does so by navigating the tree structure downward from the root using the decomposition of $\overline{a}$ in base d.

Algorithm 2 Access($\overline{a}$)

1: **Decomposition**($\overline{a}$) ▷ decompose $\overline{a}$ using registers $S_0, \ldots, S_{kh-1}$
2: $l \leftarrow 1$ ▷ contains the working register
3: $bool \leftarrow 1$
4: $i \leftarrow 0$ ▷ the current depth
5: **while** $i \leq kh - 1$ & $bool = 1$ **do**
6: $\quad (bool, l) \leftarrow R_{(l+S_i)}$ ▷ follow the search path
7: $\quad i \leftarrow i + 1$
8: **end while**
9: **Return**$(bool, l)$

The procedure returns a pair $(bool, l)$ of the form $(1, b)$ or $(0, \overline{a}')$. If the first component is 1 then $b = f(\overline{a})$. Otherwise $\overline{a} \notin \mathrm{Dom}(f)$ and $\overline{a}'$ is the smallest tuple bigger than $\overline{a}$ in the domain of f.

7.2.2 Computing next and previous tuples.

In order to update our data structure, it will be useful to compute the smallest (resp. biggest) tuple that is within the domain of f and strictly bigger (resp. smaller) than a given $\overline{a}$. Given any k-tuple $\overline{a}$, we let $\overline{a}_> := \min\{\overline{b} \in \mathrm{Dom}(f) \mid \overline{b} > \overline{a}\}$ and $\overline{a}_< := \max\{\overline{b} \in \mathrm{Dom}(f) \mid \overline{b} < \overline{a}\}$. If there is no element bigger (resp. smaller) than $\overline{a}$ in the domain of the function, we set $\overline{a}_> := \mathsf{Null}$ (resp. $\overline{a}_< := \mathsf{Null}$).

Recall that **Access**$(\overline{a})$ returns $(0, \overline{a}_>)$ when $\overline{a}$ does not belong to the domain of f, and $(1, f(\overline{a}))$ otherwise. In the second case, **Access**$(\overline{a}+1)$ yields the desired result, where $\overline{a}+1$ is the tuple immediately following $\overline{a}$ in lexicographical order of $[n]^k$. Altogether the computation is performed in time $O(kh)$, i.e. in constant time.

The computation of $\overline{a}_<$ can be obtained similarly with a dual data structure using the reverse lexicographical order instead of the lexicographical order.

7.3 Initialization and insertions

It remains to compute and update our data structure. The computation will be done by inserting tuples in the domain of f one by one. If we can show that each insertion can be done in time $O(n^\epsilon)$, then the total time for computing the data structure is $O(|\mathrm{Dom}(f)| \cdot n^\epsilon)$ as desired. In this section we show how an insertion can be achieved. We first initialize the data structure with an empty f and then show how to extend its domain with one extra tuple.

The initialization is pretty straightforward. We build the root of the tree where everything points to Null, this means creating its d children.

Algorithm 3 Init( )

1: $R_0 \leftarrow d + 2$ ▷ update the total memory currently used
2: **for** $i = 1$ `to` d **do**
3: $\quad R_i \leftarrow (0, \mathsf{Null})$
4: **end for**
5: $R_{d+1} \leftarrow (-1, \mathsf{Null})$

Adding information is, however, a bit more challenging. It requires two things: find the correct subtree to add or remove the information, and update the content of the register with the appropriate values.

We now show that, given a pair $(\overline{a}, b)$, we can add to the data structure the information that $f(\overline{a}) = b$ and accordingly update the data structure in time $O(n^\epsilon)$.

The update procedure can be decomposed into two steps. The first one adds $(\overline{a}, b)$ to the structure using the **Insert** procedure described below. The second one updates the content of the relevant registers. The main goal (an difficulty) of the second step is to update every values of the form $(0, \overline{b})$ that are impacted by the addition (or removal) of the tuple. This is the purpose of the **Clean** subroutine, which is used both when we add and when we remove tuples. To do so, we need the two tuples $\overline{a}_<$ and $\overline{a}_>$, i.e., the biggest tuple in $\mathrm{Dom}(f)$ that is smaller than $\overline{a}$ and the smallest one that is bigger than $\overline{a}$. Recall that both $\overline{a}_<$ and $\overline{a}_>$ can be computed in time $O(kh)$ as explained in Section 7.2.2. The key observation is that the cells of the form $(0, \overline{b})$ that require an update must lie between the search paths for $\overline{a}_<$ and $\overline{a}_>$. There are few such cells: $O(dkh)$.

For example, consider the data structure of Figure 1, and the case where 19 must be removed from the domain. We first compute the surrounding elements of 19: 5 and 24. and look for the path leading to 19. We then conclude that the array stored in cells $R_{21} - -R_{24}$ is now irrelevant. We therefore move the content of the array $R_{25} - -R_{28}$ in place of $R_{21} - -R_{24}$. Immediately after that we update the content of R_{15} that should now contain $(1, 21)$ and R_0 that should contain 25. Finally, we look at each cell that lies between the paths going to 5 and 24, and replace the value $(0, 19)$ by $(0, 24)$ in cells R_7, R_2, R_{13}, and R_{14}.

To recapitulate, the next procedure updates the tree structure by adding $\overline{a}$ in the domain of f and setting $b = f(\overline{a})$. The subroutine **Decomposition** gives the path in the tree leading to the leaf coding $\overline{a}$. If some nodes along this path are missing they will be created in a top-down fashion when invoking the subroutine **Insert**. The **Clean** subroutines ensure that the leaves of the tree that do not correspond to a tuple in the domain of f do point to the closest larger tuple within the domain of f.

Algorithm 4 Add$(\overline{a}, b)$

Compute $\overline{a}_<$ and $\overline{a}_>$	▷ See Section 7.2.2
Decomposition$(\overline{a})$	▷ Decompose $\overline{a}$ using register $S_0, \ldots, S_{kh-1}$
Insert$(1, 0, b)$	▷ Insert the desired leaf and its ancestors at the right places, i.e. as specified by $S_0, \ldots, S_{kh-1}$
Clean($\overline{a}_<, \overline{a}$**)**	▷ the leaf nodes between $\overline{a}_<$ and $\overline{a}$ whose content is $(0, x)$ should be updated in order to replace x by $\overline{a}$
Clean($\overline{a}, \overline{a}_>$**)**	▷ the leaf nodes between $\overline{a}$ and $\overline{a}_>$ whose content is $(0, x)$ should be updated in order to replace x by $\overline{a}_>$

The main subroutine is **Insert**(l, i, b) whose goal is to insert if necessary a new node at depth i along the path specified by $S_0, \ldots, S_{kh-1}$ in order to eventually, when i is $kh - 1$, set the value b for $f(\overline{a})$. Initially it starts with the root, $i = 0$, and the first register representing the root i.e. R_1 and $l = 1$. This is done top-down in the obvious way. Recall that each time we create a new node we need to created it d siblings.

Algorithm 5 Insert(l, i, b)

if $i = kh - 1$ **then** ▷ we are at the leaves level,
  $R_{l+S_i} \leftarrow (1, b)$ ▷ the content of the register representing $\overline{a}$ is set to the value $b = f(\overline{a})$
else
  $(bool, l') \leftarrow R_{l+S_i}$ ▷ we look at the content of the S_i^{th} register of the current node. $bool$ says whether there is already an S_i^{th} child in the data structure.
  if $bool = 0$ **then** ▷ we need to create a new subtree
    $R_{l+S_i} \leftarrow (1, R_0)$ ▷ we use d new registers for that
    $l' \leftarrow R_0$
    for $j = 0$ `to` $d - 1$ **do**
      $R_{R_0+j} \leftarrow (0, 0)$ ▷ their content will get their correct value later during the Clean procedures
    end for
    $R_{R_0+d} \leftarrow (-1, l + S_i)$ ▷ the last register points to the parent
    $R_0 \leftarrow R_0 + d + 1$ ▷ R_0 contains the last available memory
  end if ▷ in any case, $R_{(l+S_i)}$ now contains $(1, l')$.
  Insert$(l', i + 1, b)$ ▷ we continue down within the correct subtree
end if

It remains to describe the cleaning subroutine. **Clean(**$\overline{a}, \overline{b}$**)** is expected to replace the content of all the leaf nodes between $\overline{a}$ and $\overline{b}$ of the form $(0, x)$ by $(0, \overline{b})$. This is done by a simple depth-first left-first traversal of the tree, starting from $\overline{a}$ and ending in $\overline{b}$. There are two special cases when $\overline{a} =$ Null and when $\overline{b} =$ Null. Note that it is called with either **Clean(**$\overline{a}_<, \overline{a}$**)** or **Clean(**$\overline{a}, \overline{a}_>$**)** hence at least one of its inputs, namely $\overline{a}$ is not Null. The other input may be Null if $\overline{a}$ is the first or last element in the domain of f. Notice also that by definition of $\overline{a}_<$ all nodes between $\overline{a}_<$ and $\overline{a}$ are leaves, same with $\overline{a}$ and $\overline{a}_>$.

Algorithm 6 Clean$(\overline{a}_1, \overline{a}_2)$

if $\overline{a}_1 \neq$ Null **Decomposition**$(\overline{a}_1)$ ▷ decompose $\overline{a}_1$ using registers $S_0, \ldots, S_{kh-1}$
if $\overline{a}_2 \neq$ Null **Decomposition'**$(\overline{a}_2)$ ▷ decompose $\overline{a}_2$ using registers $S'_0, \ldots, S'_{kh-1}$
if $\overline{a}_1 =$ Null **then** ▷ $\overline{a}_2$ is the first element in the domain of f
  Fill_Left$(1, 0, \overline{a}_2)$ ▷ sets to $(0, \overline{a}_2)$ the label of all nodes before the leaf corresponding to $\overline{a}_2$
else if $\overline{a}_2 =$ Null **then** ▷ $\overline{a}_1$ is the last element in the domain of f
  Fill_Right$(1, 0,$ **Null**$)$ ▷ sets to $(0,$ Null$)$ the label of all the nodes after the leaf corresponding to $\overline{a}_1$
else
  Fill$(1, 0, \overline{a}_2)$ ▷ sets to $(0, \overline{a}_2)$ the labels of all leaf nodes between the leaves corresponding to $\overline{a}_1$ and $\overline{a}_2$
end if

The procedures **Fill_Left**, **Fill_Right**, and **Fill** use the information present in the registers $S_0, \ldots, S_{kh-1}$ and $S'_0, \ldots, S'_{kh-1}$. The procedure **Fill_Right**$(l, i, \overline{a}_2)$, which is also invoked within **Fill**, assumes that $\overline{a}_1$ is in the domain of f and the path associated to $\overline{a}_1$ has been created in the data structure. It then sets to $(0, \overline{a}_2)$ the label of all nodes that are after the leaf decomposed as $S_0, \ldots, S_{kh-1}$ in the depth-first search order of the tree, starting from the node at depth i, pointed by R_l. It is only invoked in the context where all those nodes are leaves. Hence it is enough to go along the path specified by the S_i and to set the content of all the siblings to $(0, \overline{a}_2)$.

Algorithm 7 Fill_Right $(l, i, \overline{a}_2)$

1: **if** $i < kh$ **then** ▷ we are working with an inner node
2: **for** $l + S_i < l' < l + d$ **do**
3: $R_{l'} \leftarrow (0, \overline{a}_2)$ ▷ we set the appropriate value
4: **end for**
5: $(bool, l) \leftarrow R_{l+S_i}$ ▷ as $\overline{a}_1$ is in the domain, $bool = 1$ and l is a pointer
6: **Fill_Right**$(l, i + 1, \overline{a}_2)$ ▷ we continue down within the tree
7: **end if**

Similarly, the dual procedure **Fill_Left**$(l, i, \overline{a}_2)$ assumes that $\overline{a}_2$ is in the domain of f and the path associated to $\overline{a}_2$ has been created in the data structure. It then sets to $(0, \overline{a}_2)$ the label of all nodes that are before the leaf decomposed as $S'_0, \ldots, S'_{kh-1}$ in the depth-first search order of the tree, starting from the node at depth i, pointed by R_l.

Algorithm 8 Fill_Left $(l, i, \overline{a}_2)$

if $i < kh$ **then** ▷ we are working with an inner node
  for $l \leq l' < l + S'_i$ **do**
    $R_{l'} \leftarrow (0, \overline{a}_2)$ ▷ we set the appropriate value
  end for
  $(bool, l) \leftarrow R_{l+S'_i}$ ▷ as $\overline{a}_2$ is in the domain, $bool = 1$ and l is a pointer
  Fill_Left$(l, i+1, \overline{a}_2)$ ▷ we continue down within the tree
end if

Finally, we combine the above two procedures in the appropriate way. Here, **Fill**$(l, i, \overline{a})$ assumes that $\overline{a}$ is in the domain of f and that the path associated to $\overline{a}$ exists already. It is done by first finding the level i where S_i and S'_i disagree and then call **FillLeft** and **FillRight** starting from this level to clean the corresponding subtree.

Algorithm 9 Fill $(l, i, \overline{a}_2)$

if $S_i = S'_i$ **then** ▷ we spot the first level where S_i and S'_i disagree
  $(bool, l') \leftarrow R_{l+S_i}$
  Fill$(l', i+1, \overline{a}_2)$
else ▷ $S_i < S'_i$
  for $l + S_i < l' < l + S'_i$ **do**
    $R_{l'} \leftarrow (0, \overline{a}_2)$ ▷ we take care correctly of the current level
  end for
  if $i < kh$ **then** ▷ if we are not at a leaf level, we need to set the subtrees appropriately
    $(bool, l') \leftarrow R_{l+S_i}$
    Fill_Right$(l', i+1, \overline{a}_2)$
    $(bool, l') \leftarrow R_{l+S'_i}$
    Fill_Left$(l', i+1, \overline{a}_2)$
  end if
end if

Note that each subroutine works in time linear in khd, hence in $O(n^\epsilon)$. Given f, we can therefore create the data structure for f in time $O(n^\epsilon \cdot |\mathrm{Dom}(f)|)$ as required.

7.4 Removing information

When a tuple is removed, we start in the same way as for insertion, but we apply a further operation: deleting one or several unused subtrees to prevent the data structure to grow indefinitely.

We now show that, given a pair $(\overline{a}, b)$, we can remove from the data structure the fact that $\overline{a}$ is in the domain of f. This can require up to three steps. First, change the label of the leaf corresponding to $\overline{a}$ in the data structure. Secondly, possibly remove the subtree containing $\overline{a}$ off the data structure. And finally, clean the data structure between $\overline{a}_<$ and $\overline{a}_>$.

Algorithm 10 Remove$(\overline{a})$

Compute $\overline{a}_<$ and $\overline{a}_>$	▷ as explained above
Decomposition$(\overline{a})$	▷ decompose $\overline{a}$ using registers $S_0, \ldots, S_{kh-1}$
$l \leftarrow$ **Run**$(1, 0)$	▷ find the node representing $\overline{a}$ in the structure
Cut(l)	▷ remove possible subtrees
Clean($\overline{a}_<, \overline{a}_>$**)**	▷ ensure that all pairs $(0, x)$ of the data structure have the right value for x

The procedure **Run**(l, i) processes the tree structure downward returning by induction the register for the S_i^{th} child of R_l. It is initially invoked with $(1, 0)$ and eventually will return the register of the leaf corresponding to $\overline{a}$.

Algorithm 11 Run(l, i)

if $i < kh - 1$ **then**	▷ we are not at the leaves level
$(bool, l') \leftarrow R_{l+S_i}$	▷ we look at the content of the S_i^{th} register of the current node.
Run$(l', i + 1)$	▷ we continue in the correct subtree
else	▷ we are at the leaves level
Return(l)	
end if	

The procedure **Cut** removes the subtree of a node if it no longer contains an element in the domain of f. This is done bottom-up starting from a leaf corresponding to a tuple that has been removed from the domain of f. As we always enforce that a node has 0 or d children we need to check whether all siblings can be safely removed before removing the node and its siblings. We then reuse the newly freed memory in order to optimize space.

Algorithm 12 Cut(l)

$(bool, i) \leftarrow (0, 0)$
while $bool \neq -1$ **do** ▷ We go to the last child of the current node
    $(bool, l') \leftarrow R_{l+i}$
    $i \leftarrow i + 1$
end while
$l \leftarrow l + i - d - 1$ ▷ We can now go to the first child
$(bool, i) \leftarrow (0, 0)$
while $i < d \ \& \ bool = 0$ **do** ▷ we check whether any sibling of l contains an element in the domain of f, i.e. whether bool=1
    $(bool, l') \leftarrow R_{l+i}$
    $i \leftarrow i + 1$
end while
if $bool = 0$ **then** ▷ if no, the node and its siblings can be safely removed and their memory reused
    $(-1, l') \leftarrow R_{l+d}$ ▷ we save the address of the parent of the current node
    $R_{l'} \leftarrow (0, 0)$ ▷ the value will be corrected later
    for $0 \leq j \leq d$ **do** ▷ we now need to save memory, moving the nodes at the end of the memory in place of those just deleted
        $R_{l+j} \leftarrow R_{(R_0-(d+1)+j)}$
    end for ▷ it remains to change the child relation of the parent of those nodes
    $(-1, l'') \leftarrow R_{R_0-1}$ ▷ l'' now contains the address of their parents
    $R_{l''} \leftarrow (1, l)$ ▷ the child relation is updated
    $R_0 \leftarrow R_0 - (d + 1)$ ▷ update the last available memory
    Cut(l') ▷ we start again with the parent level
end if

All the procedures take time $O(khd)$ and are therefore in $O(n^\epsilon)$ as desired. This ends the appendix presenting proof details of the Storing Theorem 3.1.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In Dan Suciu, Sebastian Skritek, and Christoph Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 89–103. ACM, 2019.

[3] Guillaume Bagan. MSO queries on tree decomposable structures are computable with linear delay. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[4] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[5] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[6] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 303–318. ACM, 2017.

[7] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. *ACM Trans. Database Syst.*, 43(2):7:1–7:32, 2018.

[8] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering UCQs under updates and in the presence of integrity constraints. In Benny Kimelfeld and Yael Amsterdamer, editors, *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, volume 98 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[9] Rodney G. Downey, Michael R. Fellows, and Udayan Taylor. The parameterized complexity of relational database queries and an improved characterization of W[1]. In Douglas S. Bridges, Cristian S. Calude, Jeremy Gibbons, Steve Reeves, and Ian H. Witten, editors, *First Conference of the Centre for Discrete Mathematics and Theoretical Computer Science, DMTCS 1996, Auckland, New Zealand, December, 9-13, 1996*, pages 194–213. Springer-Verlag, Singapore, 1996.

[10] Arnaud Durand and Etienne Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *ACM Trans. Comput. Log.*, 8(4):21, 2007.

[11] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating answers to first-order queries over databases of low degree. In Richard Hull and Martin Grohe, editors, *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 121–131. ACM, 2014.

[12] Markus Frick. Generalized model-checking over locally tree-decomposable classes. *Theory Comput. Syst.*, 37(1):157–191, 2004.

[13] Markus Frick and Martin Grohe. Deciding first-order properties of locally tree-decomposable structures. *J. ACM*, 48(6):1184–1206, 2001.

[14] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Ann. Pure Appl. Logic*, 130(1-3):3–31, 2004.

[15] Haim Gaifman. On local and non-local properties. In *Proceedings of the Herbrand Symposium*, volume 107 of *Studies in Logic and the Foundations of Mathematics*, pages 105 – 135. Elsevier, 1982.

[16] Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. Deciding first-order properties of nowhere dense graphs. In David B. Shmoys, editor, *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 89–98. ACM, 2014.

[17] Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. Deciding first-order properties of nowhere dense graphs. *J. ACM*, 64(3):17:1–17:32, 2017.

[18] Martin Grohe and Nicole Schweikardt. First-order query evaluation with cardinality conditions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2018, Houston, TX, USA, June 10–15, 2018*. ACM, 2018. Full version available at CoRR, http://arxiv.org/abs/1707.05945, 2017.

[19] Wojciech Kazana. *Query evaluation with constant delay. (L'évaluation de requêtes avec un délai constant).* PhD thesis, École normale supérieure de Cachan, Paris, France, 2013.

[20] Wojciech Kazana and Luc Segoufin. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science*, 7(2), 2011.

[21] Wojciech Kazana and Luc Segoufin. Enumeration of first-order queries on classes of structures with bounded expansion. In Richard Hull and Wenfei Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 297–308. ACM, 2013.

[22] Wojciech Kazana and Luc Segoufin. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.*, 14(4):25:1–25:12, 2013.

[23] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley, 1998.

[24] Stephan Kreutzer and Anuj Dawar. Parameterized complexity of first-order logic. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:131, 2009.

[25] Leonid Libkin. *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[26] Jaroslav Nesetril and Patrice Ossona de Mendez. First order properties on nowhere dense structures. *J. Symb. Log.*, 75(3):868–887, 2010.

[27] Jaroslav Nesetril and Patrice Ossona de Mendez. On nowhere dense graphs. *Eur. J. Comb.*, 32(4):600–617, 2011.

[28] Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2018, Houston, TX, USA, June 10–15, 2018*. ACM, 2018.

[29] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO queries over nowhere dense graphs. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 151–163. ACM, 2018.

[30] Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, volume 68 of *LIPIcs*, pages 20:1–20:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[31] Sebastian Siebertz. *Nowhere Dense Classes of Graphs: Characterisations and Algorithmic Meta- Theorems.* PhD thesis, Technical University of Berlin, Germany, 2016.

[32] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.

[33] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982.

[34] Alexandre Vigny. *Query enumeration and nowhere dense graphs. (Énumération des requêtes et graphes nulle-part denses).* PhD thesis, Paris Diderot University, France, 2018.