Theoretical Computer Science 116 (1993) 117–149
Elsevier

A closed-form evaluation for Datalog queries with integer (gap)-order constraints*

Peter Z. Revesz

Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE, USA

Abstract

Revesz P.Z., A closed-form evaluation for datalog queries with integer (gap)-order constraints, Theoretical Computer Science 116 (1993) 117–149.

We provide a generalization of Datalog based on generalizing databases by adding integer order constraints to relational tuples. For Datalog queries with integer (gap)-order constraints (denoted as $\mathit{Datalog}^{<_{\mathbb{Z}}}$), we show that there is a closed-form evaluation. We also show that the tuple recognition problem can be done in PTIME in the size of the generalized database, assuming that the size of the constants in the query is logarithmic in the size of the database. Note that the absence of negation in critical, $\text{Datalog}^{\neg}$ queries with integer order constraints can express any Turing-computable function.

1. Introduction

In this paper we consider a generalization of Datalog based on the notion of a constraint tuple. The important idea of a constraint tuple comes from constraint logic programming systems, e.g. [4, 8, 14], and it generalizes the notion of a ground fact. This allows the declarative programming of new applications, including various combinatorial search problems (see [30] for a survey). Recently, Kanellakis et al. [17] extended this idea to database systems through the design of CQLs (*constraint query*

Correspondence to: Peter Z. Revesz, Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE 68588, USA.

* A preliminary version of the results in this paper appeared in [26]. This work was supported in part by NSF grant IRI-8617344, NSF-INRIA grant INT-8817874 and by ONR grant N00014-83-K-0146 ARPA Order No. 4786 while the author was at Brown University, and by the Institute for Robotics and Intelligent Systems in Canada.

0304-3975/93/$06.00 © 1993—Elsevier Science Publishers B.V. All rights reserved

languages). For example, in the relational database model, $R(3,4)$, or $R(x,y)$ with $x=3$, $y=4$, is a tuple of arity 2. In our framework, $R(x,y)$ with $x=y$, $x<2$ is a generalized tuple of arity 2 and so is $R(x,y)$ with $x-y\geqslant 25$, where x and y are any integers satisfying these constraints. An important feature of generalized tuples of arity k is the description, by a finite representation, of a possibly infinite set of standard relational database tuples with arity k.

A CQL can be considered to be a union of a database query language and a decidable logical theory together with a bottom-up evaluation in closed form. For a CQL to be feasible, the bottom-up evaluation has to be efficient. A generally accepted measure of performance of a bottom-up evaluation is *data complexity*, which was introduced by Chandra and Harel [5], and by Vardi [31]. Data complexity measures the complexity of answering a fixed query in terms of the size of the database. The rationale behind data complexity is that the size of the database dominates the query size by several orders of magnitude for most applications.

It is well known that any Datalog and any inflationary Datalog$^{\neg}$ [1,21] query evaluated on traditional relational databases have PTIME data complexity. In [17] it is also shown that any inflationary Datalog$^{\neg}$ with rational order and any inflationary Datalog$^{\neg}$ with equality over an infinite set of constants query can be also evaluated in PTIME data complexity. The latter case is also considered in [2, 11, 19, 25]. However, the case of Datalog queries combined with the theory of integer order [10] was left as an open problem.

There are other attempts to combine some form of integer constraint solving with existing relational database languages. For example, Kabanza et al. [15] examined a way of combining relational calculus with a limited form of recursion, called *linear repeating points*. (Linear repeating points are points of the form $c+kn$, for each integer n and fixed integer constants c and k.) As [15] shows, for unary and binary predicates, this gives exactly Presburger definability, which is a clear extension of relational calculus, but it still lacks expressibility of simple operations such as transitive closure. Another related paper by Chomicki and Imielinski [7] builds directly on Datalog by restricting the application of the successor function to one argument (say always the first argument) of relations. This restriction disallows the order predicate, which cannot be expressed in this language. Hence, both of these results leave open the natural case of Datalog combined with the theory of integer order. There are many problems that can be expressed in this language. The following *age-bounds* problem is one example.

Example 1.1. Alfred and Alice have children Bernard, Carl and Donald in this order. Bernard and Bernice have child Edward. Donald and Denise have children Elise and Fred. Edward and Elise have child Gerald. Fred and Felice have child Harold (see Fig. 1). Alfred is not yet 70, and Harold is already in school. Gerald was born just last month. Elise is over 4 years older than her younger brother Fred, and Bernard is over 25 years older than his son Edward. Assuming that between parents and their children

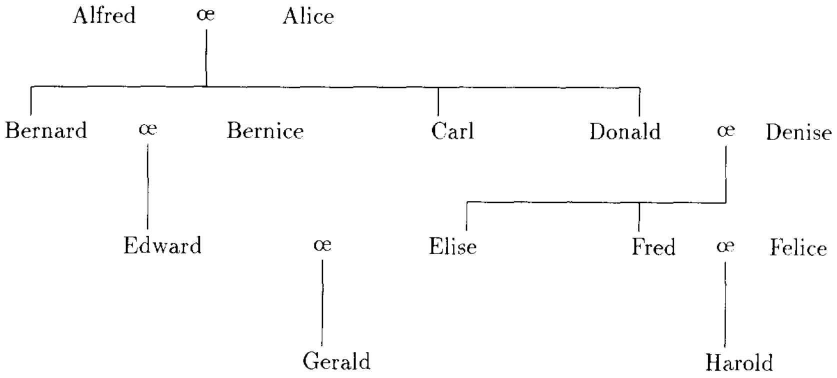


Fig. 1. Alfred and Alice's family.

there is over 17 years difference and that no siblings are twins or born in the same year, how old is Donald?

The *Datalog*$^{<z}$ program below finds for each person p the set of ages y that p can have. It will be the conjunction of the strictest upper and lower age limits. The upper bound will be the conjunction of another set of constraints: the upper bound implied by the age of the parents of p, the upper bound implied by the age of an older sibling of p, and the explicitly stated upper bound for the age of p. The upper bound that is calculated from the age of the parents is again a conjunction of constraints. Similarly for the other defined relations.

age(p, y)	:—	*upper_age(p, y), lower_age(p, y)*
upper_age(p, y)	:—	*upper_bound(p, y), upper_age_by_parents(p, y), upper_age_by_sibling(p, y)*
upper_age_by_parents(p, y_1)	:—	*father(p, f), upper_age*(f, y_2), *gen_diff*(y_1, y_2), *mother(p, m), upper_age*(m, y_3), *gen_diff*(y_1, y_3) *diff*(p, y_1, f, y_2), *diff*(p, y_1, m, y_3)
upper_age_by_parents(p, y)	:—	*no_known_parents(p)*
upper_age_by_sibling(p, y_1)	:—	*next_sibling(n, p), upper_age*(n, y_2), *sibl_diff*(y_1, y_2), *diff*(p, y_1, n, y_2)
upper_age_by_sibling(p, y)	:—	*youngestchild(p)*
lower_age(p, y)	:—	*lower_bound(p, y), lower_age_by_child(p, y), lower_age_by_sibling(p, y)*
lower_age_by_child(p, y_1)	:—	*eldestchild(p, e), lower_age*(e, y_2) *gen_diff*(y_2, y_1), *diff*(e, y_2, p, y_1)
lower_age_by_child(p, y)	:—	*childless(p)*
lower_age_by_sibling(p, y_1)	:—	*next_sibling(p, n), lower_age*(n, y_2), *sibl_diff*(y_2, y_1), *diff*(n, y_2, p, y_1)
lower_age_by_sibling(p, y)	:—	*youngestchild(p)*

Most of the input database relations can be represented in a standard Datalog style, including the *father*, *mother*, *no_known_parents*, *eldestchild*, *next_sibling*, *youngestchild*, and *childless* relations. We describe here only those database relations whose definitions need integer order constraints. We use here for integer variables x and y, abbreviations like $x <_{10} y$, that stands for the constraint $x+10<y$ (see the text for further explanation).

$gen\_diff(y_1, y_2)$:— $y_1 <_{17} y_2$ %minimum age difference between generations
$sibl\_diff(y_1, y_2)$:— $y_1 < y_2$ %minimum age difference between two siblings

$diff(Edward, y_1, Bernard, y_2)$:— $y_1 <_{25} y_2$
$diff(Fred, y_1, Elise, y_2)$:— $y_1 <_4 y_2$
$diff(x, y_1, z, y_2)$:— $x \neq Edward, x \neq Fred$

$upper\_bound(Alfred, y)$:— $y<70$
$upper\_bound(Gerald, y)$:— $y=0$
$upper\_bound(x, y)$:— $x \neq Alfred,\ x \neq Gerald$

$lower\_bound(Gerald, y)$:— $y=0$
$lower\_bound(Harold, y)$:— $5<y$
$lower\_bound(x, y)$:— $x \neq Gerald,\ x \neq Harold,\ 0 \leqslant y$

Using these inputs, the program can find the best bounds for each person's age. For example, since Harold is childless and a youngest child, the tuples *lower_age_by_child*(*Harold*, y) and *lower_age_by_sibling*(*Harold*, y) do not place any restrictions on his age (i.e., y can be any integer in these). Therefore, *lower_age*(*Herold*, y) will remain just the lower bound that is given initially, (i.e., $5<y$). Since Harold is also the eldest child of Fred, and there is no explicitly given difference between Harold and Fred's ages, *lower_bound_by_child*(Fred, y_1) will be $(5<y)\wedge(y <_{17} y_1)=5 <_{18} y_1 = 23<y_1$. Since Fred is also a youngest child, Fred's *lower_age* will be the same. Hence, Fred is more than 23 years old. By continuing this way, the program will find that Donald's age is between 47 and 49.

Our main difficulty in integrating relational database languages and integer-order constraints was developing the appropriate quantifier elimination procedure for the positive existential subset of the theory of integer order. The quantifier elimination procedure that we develop can be used recursively to evaluate $Datalog^{<z}$ queries. (This paper assumes that the variables range over the integers, but the techniques are also applicable to any discrete or (gap)-ordered domain.)

This paper describes for $Datalog^{<z}$ queries a bottom-up evaluation method that terminates in a closed form on any generalized relational database input (Theorem 3.17). The evaluation method always gives some (possibly non-unique) generalized relational database output. This output database is equivalent to the unique least-fixpoint model of the implicit unrestricted relational database input (Theorem 3.19).

In general, the output of our evaluation procedure is not unique. Note that this does not cause any problems because the least-fixpoint model of any query is a *unique* unrestricted relational database. The situation of having several possible generalized relational database outputs is only due to the fact that the same unrestricted relational database can be described finitely by several syntactically different generalized relational databases.

Let Q be any *Datalog*$^{<_\mathbb{Z}}$ query with program P and generalized database d. For any relational tuple $A(t)$, testing whether $A(t)$ belongs to the model of Q can be done in $O(n^{k+1}+(u-l)^{O(mk^2)})$ time (Theorem 4.7), where n is the size of d, k is the largest arity of a relation in P, u is the largest and l is the smallest constant in either t or Q, and m is the maximum number of relation symbols in a rule of P. For the case of linear recursive programs, the test has NLOGSPACE data complexity (Theorem 4.8).

The above result is quite intuitive. For example, one may ask "Could Donald be 48 years old?". This can be answered by testing whether the tuple *Age*(*Donald*, *48*) belongs to the model of the *Datalog*$^{<_\mathbb{Z}}$ program and the generalized database input in Example 1.1. Many generations' time may pass, but if the database is kept updated, then the same program can still test Donald's age, and the test will take longer because of his new descendants.

The existence of a closed form for *Datalog*$^{<_\mathbb{Z}}$ is pleasantly surprising. It is easy to see that stratified Datalog$^{\neg}$ with integer order is undecidable as this constraint query language can express all Turing-computable funtions (see Proposition 2.3). The key reason for the undecidability is the first-order expressibility of the successor function, which together with the recursive power of Datalog gives the expressibility of all μ-recursive functions.

Proposition 2.3 is worth comparing with the complexity results of Immerman [12] and Vardi [31]. They show that any query in any language with PTIME data complexity can be expressed in Datalog$^{\neg}$ with an integer order predicate, and conversely, any query in Datalog$^{\neg}$ with an integer order predicate has PTIME data complexity. Immerman and Vardi, however, use only traditional relational databases, whereas in Proposition 2.3 we use generalized relational databases.

An important technical contribution of this paper is the definition of a *gap-graph* on integer variables. Intuitively, k-variable gap-graphs partition the k-dimensional point space and capture the essential properties of each subset. A gap-graph is a graph with variables and two constants as vertices and some (undirected or directed) edges between distinct vertices together with a *gap-order labeling* for each edge. A gap-order is either $=$ on an undirected edge or $<_g$ on a directed edge for some nonnegative integer subscript g that denotes the minimum difference between the two ordered elements. (That is, the vertices can represent either integer constants or integer variables. In the latter case, the gap-order labels are meant to restrict the possible values that the variables can take.) While gap-graphs arise naturally, they potentially form an infinite partition of the point space (since there is no limit a priori on the gap values): the challenge is to show that a finite number of gap-graphs always suffice to describe any input and output generalized database. We show this to be true in a very general setting using a geometric argument.

We start by giving some definitions and propositions in Section 2. The closed-form evaluation method for Datalog with integer order queries is described in Section 3. Also in Section 3, we prove that the method terminates on any input and computes a well-defined unique model for each query. The analysis of the tuple recognition problem for $Datalog^{<z}$ queries in general and for the special case of linear recursive queries is given in Section 4. Finally, open problems are listed in Section 5.

2. Basic concepts

Our work will be a particular generalization of Codd's *relational database model* [3] and the language *Datalog*. We assume that the reader is familiar with these concepts and their relevance to databases. An introduction to these can be found for example in [16, 28]. In this section we will give only the necessary definitions for the new generalizations of relational databases and Datalog.

Generalized relational databases. Our database framework is set up as follows. Let $A(x_1, \ldots, x_k)$ be a relation symbol with arity k. Let $t \in \mathbb{Z}^k$ be any sequence of k integers. (We denote by $\mathbb{Z}$ the set of integer numbers.) We call t a *tuple* and $A(t)$ a *relational tuple*. In the relational model database relations are composed of a finite number of relational tuples. An *integer order constraint* is of the form $u = v$, $u \neq v$, $u \leqslant v$, or $u < v$, where u and v are variables or constants. Variables range over the *intensional domain* $\mathbb{Z}$. Constants and $=$, $\neq$, $\leqslant$, $<$ are interpreted as integer numbers and their ordering. Let $\phi(x_1, \ldots, x_k)$ be a conjunction of integer order constraints over distinct variables $x_1, \ldots, x_k$. We call ϕ a *constraint tuple*. We call an expression of the form $A(x_1, \ldots, x_k)$:— $\phi(x_1, \ldots, x_k)$ a *generalized relational tuple*, where $x_1, \ldots, x_k$ are distinct variables, and, when there is no confusion about substitutions, we write $A(\phi)$.

We view each generalized relational tuple $A(x_1, \ldots, x_k)$:— $\phi(x_1, \ldots, x_k)$ as a finite description for a possibly infinite number of relational tuples $A(t_1), \ldots, A(t_n)$, where each $t_i \in \mathbb{Z}^k$ and t_i satisfies ϕ, i.e., $t_i \models \phi$ in the standard sense. Therefore, we are dealing with special types of unrestricted (finite or infinite) relational databases. For each database and query program, we call the finite set of constants appearing in them the *active domain D*.

Closed-form bottom-up evaluation. We require that the evaluation of the query, given an input generalized relation, yield an output generalized relation which has the same type of constraints as the input generalized relation. This we call the *closed-form* requirement. One reason for this requirement is that a closed form allows the composition of queries.

We also require that the evaluation be bottom-up, i.e., that it evaluate the subgoals before the head of the rules. We make this requirement to allow the possibility of compiler and run-time optimizations. We know that many good optimization methods for relational database languages are based on bottom-up evaluation. For more about optimization and the importance of bottom-up evaluation, see [16, 28].

Safety. Recall that an assumption is Codd's relational model is that relations (both input and output) are always finite structures, that is, they are always composed of a finite number of relational tuples. This is called the *safety* requirement. Analogous to that requirement, in our extension of the relational database model, we require that relations be composed of a finite number of generalized relational tuples. This requirement assures that the queries are always evaluable in finite time. Guaranteed finite-time evaluation helps in testing and debugging programs, especially since the halting problem in general is undecidable, and makes automated programming possible. (Note that finiteness does not follow from closure. In general, the defined database relations may be describable by only an infinite number of constraints of the types that appear in the database input.)

An important point to note here is that in the relational database model, to guarantee safety (i.e., that the output database is finite), only a restricted subset of relational calculus, namely, *safe relational calculus* is allowed as a query language [3]. By contrast, our general evaluation method works for any *Datalog*$^{<z}$ query, that is, the evaluation method finds a finite generalized output database in finite time for any finite generalized input database. Let us now describe our query language.

Datalog with integer order. The syntax is that of traditional Datalog where the bodies of rules can also contain a conjunction of integer order constraints. A *Datalog* program P with integer order, is a finite set of rules of the form

$$A_0 :\!\!- A_1, A_2, \ldots, A_l.$$

The expression A_0 (the rule *head*) must be an atomic formula of the form $R(x_1, \ldots, x_n)$, and the expressions $A_1, \ldots, A_l$ (the rule *body*) must be atomic formulas of the form $x_i = x_j$, $x_i \neq x_j$, $x_i \leqslant x_j$, $x_i < x_j$, or $R(x_1, \ldots, x_n)$, where R is some predicate symbol.

The semantics of a Datalog program P on a generalized database $r_1, \ldots, r_n$ (representing unrestricted relational database $\rho_1, \ldots, \rho_n$) is the least fixpoint of the monotone mapping defined by a first-order formula ϕ_P and $\rho_1, \ldots, \rho_n$. We explain ϕ_P by an example right below. This is as in the case without constraints, the only difference being the use of unrestricted relational databases [16, 24]. The following example appears in [17], except that there the variables are interpreted to range over the rationals.

Example 2.1. Consider the Datalog with integer order query P:

$$R(x, y) :\!\!- R(x, z), R_0(z, y), x \leqslant y, y \leqslant z$$

$$R(x, y) :\!\!- R_0(x, y)$$

Let this query be applied to a generalized database r_0 representing the unrestricted relation ρ_0. Then ϕ_P is the following first-order formula, where R_0 is interpreted as ρ_0:

$$\phi_P(x,y) \equiv \psi(x,y;R) \equiv R_0(x,y) \vee \exists z(R(x,z) \wedge R_0(z,y) \wedge x \leqslant y \wedge y \leqslant z).$$

This formula defines a mapping from unrestricted relations ρ of arity 2 to unrestricted relations of arity 2. Here R is singled out because it is the predicate variable that is initialized to ρ and will receive the output of ϕ_P. This mapping is

$$\rho \rightarrow \{a, b \in \mathbb{Z}^2 \mid \langle \mathbb{Z}, \delta, \rho_0, \rho \rangle \models \phi_P(a,b)\},$$

where we denote by δ our interpretation of the symbols $=$, $\neq$, $\leqslant$ and $<$. This mapping is monotone with respect to set inclusion for ρ. By the Tarski fixpoint theorem, it has a least fixpoint, which is the output of the query program applied to input r_0.

Datalog with integer gap-order (*Datalog*$^{<_\mathbb{Z}}$). The syntax is that of Datalog with integer order constraints with one addition. In the rule bodies and the generalized database we allow atomic formulas called *gap-orders* of the form $x_i <_g x_j$ where $g \in \mathbb{N}$. (We use $\mathbb{N}$ to denote the set of natural numbers.) We interpret this as a limited form of addition, namely, as $x_i + g < x_j$. Otherwise, the semantics is the same as for Datalog with integer order constraints. Let us illustrate this language also with an example.

Example 2.2. The following *Datalog*$^{<_\mathbb{Z}}$ program expresses a variant of the shortest path problem:

$$P(x,y,s_1,s_2) \text{:--} P(x,z,s_1,s_3), E(z,y,s_3,s_2)$$

$$P(x,y,s_1,s_2) \text{:--} E(x,y,s_1,s_2)$$

Suppose that the input relation E describes distances between cities in miles using generalized database tuples. For example, the generalized database tuple

$$E(x,y,s_1,s_2) \text{:--} x = \text{Toronto},\ y = \text{Boston},\ s_1 <_{400} s_2$$

describes that there is a 400 miles long direct flight from Toronto to Boston. Note that the last two arguments represent by *variables* the endpoints of a stretched-out and *movable* string and the gap-value between the endpoints represents the distance between the two cities. Similarly, the generalized tuple

$$E(x,y,s_1,s_2) \text{:--} x = \text{Boston},\ y = \text{London},\ s_1 <_{2000} s_2$$

describes that there is a 2000 miles long direct flight from Boston to London.

Intuitively, the strings need to be concatenated to measure distances of flights with connections. The query will do exactly that, i.e., concatenate the strings along all possible paths. Then it becomes easy to check whether there is a path of length l between any two given cities c_1 and c_2 by testing whether in the generalized output database there is a generalized relational tuple $P(x_1,x_2,x_3,x_4)$:— $\phi(x_1,x_2,x_3,x_4)$ such that $t \models \phi(x_1,x_2,x_3,x_4)$, where $t = (c_1, c_2, 0, l)$. For example, we could test whether $t = (\text{Toronto, London, } 0, 2500)$ is such a tuple. The answer in this case would be "yes", indicating that there is a flight (with connections) of length 2500 from Toronto to London.

Remark. The type of test in Example 2.2 is called a *tuple recognition* test, i.e., we want to find whether a standard relational database tuple t is in the generalized output database. We know that l is the length of the shortest path between cities c_1 and c_2 if the recognition test succeeds for tuple $t = (c_1, c_2, 0, l)$ and fails for tuple $t' = (c_1, c_2, 0, l-1)$. Hence, using binary search or other standard search techniques, we can always find by a number of tuple recognition tests the length of the shortest path. Recognition tests like the one in Example 2.2 motivate our analysis of the problem in Section 4.

Generalized data complexity. The way Chandra and Harel [5] and Vardi [31] define data complexity is based on tuple recognition. That is, they actually phrase the query evaluation problem as a decision problem. They definc, for a fixed query Q, a language $L_Q = \{(t, d)\text{: } t \in Q(d)\}$. That is, the language consists of the set of strings which are pairs of a standard relational database tuple t and an input database d (also written as a string of standard relational database tuples) such that t is in the output database in a complete evaluation of Q on d. In Section 4 we use a modified definition of data complexity to allow d to be a generalized database. We define the language as $L_Q = \{(t, d)\text{: } t \models g \wedge g \in Q(d)\}$, where t is a standard relational tuple that satisfies some generalized relational tuple g that is in the generalized output database in our evaluation of Q on d.

Stratified Datalog$^{\neg}$ *with integer order constraints.* The syntax is that of Datalog with integer order constraints with one addition. We allow in rule bodies expressions of the form $\neg R(x_1, \ldots, x_n)$, where R is some predicate symbol. We give the language stratified semantics [6]. Recall that we call a relation an *intensional database* (or IDB for short) relation if it is defined in terms of other database relations, i.e., the relational predicate symbol occurs in the head of one or more rules of the program. The stratified semantics introduced by Chandra and Harel [5] means that we group IDB predicates (and rules in which they occur as head) into strata, or layers, and evaluate each layer in sequence. The only restriction that we make is that an IDB predicate should not occur negated before the computation passed its layer.

We give the following simple proposition to illustrate that dealing with integer order can be hard and to put our results on *Datalog*$^{<z}$ into a better context.

Proposition 2.3. *Any Turing-computable function is expressible by a query of Stratified Datalog*$^{\neg}$ *with integer order constraints.*

Proof. Since the class of Turing-computable and the class of μ-recursive functions are equivalent (see [20, 22]), it is enough to show that all μ-recursive functions can be expressed. Recall from [22] (see Definition 5.5.2 in [22]) that a function is μ-recursive if and only if it can be obtained from the initial functions of zero, projection, and successor by the operations of composition, primitive recursion, and unbounded minimalization applied to regular functions. (See for the other basic definitions also [22]. We repeat here for clarity only the definition of *regular*. A $(k+1)$-place function g is *regular* if and only if, for every $\bar{n}\in\mathbb{N}^k$, there is an m such that $g(\bar{n},m)=0$.)

In our Datalog$^\neg$ simulation we use the convention that the function values of the initial and all built-up functions are in the last argument of the predicates with the same name that the functions have. The zero function is immediate. We do it by simply letting *zero*(0) be a database fact. The projection functions are also straightforward. For each ith k-place projection function, with $1\leqslant i\leqslant k$, we let

$$project_i(x_1,\ldots,x_n,x_{n+1}):\!-\,x_{n+1}=x_i$$

be database facts. The successor function can be expressed as follows:

$$succ(x,y):\!-\,x<y,\neg\, distant(x,y)$$

$$distant(x,y):\!-\,x<z<y$$

In the above the bottom rule must be done before the top rule by stratification. Together the two rules say that y is the successor of x, if y is greater than x and there is no integer z between x and y.

Let $l>0$ and $k\geqslant 0$ and let g be an l-place function and f and $h_1,\ldots,h_l$ be k-place functions. Then the composition of f obtained from g and $h_1,\ldots,h_l$ can be simulated as follows:

$$f(\bar{n},y):\!-\,g(\bar{x},y),h_1(\bar{n},x_1),\ldots,h_l(\bar{n},x_l)$$

Primitive recursion is also straightforward:

$$f(\bar{n},0,y):\!-\,g(\bar{n},y)$$

$$f(\bar{n},m',y):\!-\,h(\bar{n},m,x,y),f(\bar{n},m,x),succ(m,m')$$

For the unbounded minimalization, stratified negation is again necessary. As in the definition of successor, the bottom rule is evaluated first:

$$f(\bar{n},m):\!-\,g(\bar{n},m,0),\neg\, smaller(m',m)$$

$$smaller(m',m):\!-\,g(\bar{n},m',0),m'<m$$

Finally, we note that the depth of the stratification need not be larger than two, because of a theorem of Kleene [20] which states that only one application of unbounded minimalization is sufficient to express any μ-recursive function. □

Remark. It follows from [17] that stratified Datalog$^\neg$ with rational order can be evaluated in PTIME in the size of any generalized extensional database. Clearly, the difficulty comes from having the integers as the domain and not from negation. In the theory of rational order [9] we cannot express many things that we can express in the theory of integer order; for example, we cannot express successor $(+1)$.

3. A closed-form evaluation for Datalog with integer gap-order

In this section we show that, for *Datalog*$^{<_z}$ programs and any generalized input database, there is a closed-form evaluation. This evaluation strategy will be similar to the naive evaluation algorithm used for Datalog only. That is, at each step, all possible rule applications are considered with all possible substitutions of database facts for subgoals. The complication results from the database facts being generalized relational tuples.

Our main goal is to prove that there is an evaluation algorithm for which the number of steps necessary is finite and that the algorithm computes a well-defined, unique output for each query. (For complexity results, see Section 4). We will proceed as follows.

In Section 3.1 we simplify the problem by transforming *Datalog*$^{<_z}$ queries into a special form, where each generalized tuple is represented as a *gap-graph* (Definitions 3.1 and 3.2 and Lemma 3.4). Second, we give some basic notions about gap-graphs, the most important of which is *consistency* (Definition 3.4–3.6), and show a simple way to test whether a gap-graph is consistent (Lemma 3.8).

In Section 3.2, we define *shortcut*, *merge* and *subsume* as the three basic operations on gap-graphs (Definition 3.8–3.10). We also show in several lemmas the semantic correctness of these operators. Essentially, this means that the operators are consistency preserving (Lemmas 3.11–3.13).

In Section 3.3 we define generalized rule applications and our evaluation method in terms of the four basic operations on gap-graphs. We prove in Theorem 3.17 that the evaluation method always terminates. Finally, we prove in Theorem 3.19 that the evaluation method always yields the desired unique least-fixpoint model (defined in Definition 3.18).

3.1. Gap-graphs

For this section, let D be any fixed finite set of integers. We start with the basic definitions of *gap-orders* and *gap-graphs.*

Definition 3.1. Let x and y be any two integer variables. Given some assignment to the variables, a *gap-order* $x <_g y$ for some gap-value $g \in \mathbb{N}$ holds if and only if $x + g < y$ holds in the given assignment. A *gap-order* $x = y$ holds if and only if x and y are equal in the given assignment.

Definition 3.2. Let $x_1, \ldots, x_n$ be a set of integer variables, and let l and u be any elements of $D \cup \{+\infty, -\infty\}$ such that $l < u$. Then a *gap-graph* is any graph that has $n+2$ vertices labeled $x_1, \ldots, x_n$ and l and u and has between any pair of distinct vertices at most one undirected edge labeled by $=$ or one directed edge labeled by a gap-order $<_g$ for some $g \in \mathbb{N}$.

In what follows, we will always assume when talking about edges that an edge labeled by $=$ is undirected and an edge labeled by a $<_g$ for some $g \in \mathbb{N}$ is directed. The direction in the latter case is necessary only to make it clear which vertex is less than the other. We will assume that if a directed edge from vertex v to another vertex u has label $<_g$ on it, then $v <_g u$ is the gap-order constraint that is represented within the gap-graph.

Two examples of gap-graphs are shown in Fig. 4(a) and (b). The first one has both undirected and directed edges, while the second one has only directed edges.

Definition 3.3. A gap-graph G is *consistent* if and only if there is an integer assignment $\mathscr{A}$ to the variables in the vertices that satisfies all the gap-order labels on the edges. We denote this as $\mathscr{A} \models G$.

Gap-graphs provide an alternative representation of generalized tuples which are conjunctions of integer gap-order constraints. This is spelled out in Lemma 3.4. Lemma 3.4 helps to transform the problem of evaluating a *Datalog*$^{<_\mathbb{Z}}$ query into a graph problem. We can assume later that the generalized input database has only gap-graphs as generalized tuples.

Lemma 3.4. *Let C be any conjunction of integer (gap)-order constraints over variables $x_1, \ldots, x_k$. Let l be the smallest and u the largest constant in C (if there are no constants, let $l = -\infty$ and $u = +\infty$, and if there is only one constant c, let $l = c$ and $u = +\infty$). Let n be the size of C (measured in the number of atomic constraints $=$, $\neq$, $\leqslant$, $<$, or $<_g$). Then*

(a) *C can be represented as a finite disjunction of gap-graphs over $x_1, \ldots, x_k, l, u$. Moreover, any assignment $\mathscr{A}$ satisfies C if and only if $\mathscr{A}$ is a consistent assignment to at least one of the gap-graphs in the representation.*

(b) *For any fixed k, the representation can be found in $\mathrm{O}(n^{k+1})$ time.*

Proof. See Appendix. □

Example 3.5. The algorithm in the proof of Lemma 3.4 is too long to illustrate in every detail, but suppose as a simple example that C is the formula

$$x \neq y \wedge y \leqslant 10 \wedge x <_3 z \wedge z \leqslant y.$$

Then rewrite it as

$$(x < y \vee y < x) \wedge (y < 10 \vee 10 = y) \wedge x <_3 z \wedge (z < y \vee z = y).$$

Put this into disjunctive normal form

$$(x < y \wedge y < 10 \wedge x <_3 z \wedge z < y) \vee \cdots \vee (y < x \wedge 10 = y \wedge x <_3 z \wedge z = y).$$

Each of the eight disjuncts can be represented by a gap-graph over vertices $x, y, z, 10, +\infty$. Note that the last disjunct is unsatisfiable, since $10 = y = z$ and x is supposed to be both greater and less than 10 (by more than 3). Its corresponding gap-graph will be inconsistent.

The following two definitions describe some concepts that are related to gap-graphs and are used repeatedly throughout Section 3.

Definition 3.6. For each gap-graph G, we call the graph obtained by deleting all gap-order labels in G the *underlying graph* of G. For each graph G, we call the graph obtained by merging all equal vertices and deleting all edges with $=$ labels the *compact graph* of G.

For example, Fig. 2 is the underlying graph and Fig. 3 is the compact graph of the gap-graph (a) in Fig. 4. Note that compact graphs have only directed edges. Also note that a compact graph may have multiple labels for vertices and multiple edges between vertices. Using Definition 3.6 simplifies our proofs when only the properties captured by the underlying graphs or the compact graphs are important. For instance, it is simpler to talk about path lengths within compact graphs than within gap-graphs.

Definition 3.7. A directed edge from vertex v to another vertex w with any gap-value in a gap-graph we denote as (v, w). A chain of directed edges $(v, w_1), (w_1, w_2), \ldots, (w_{n-1}, w_n)$ we call a *path*. If $v \equiv w_n$, then we call the path a *cycle*. The *length* of the path from v to w_n is the sum of the gap-values in labels of the directed edges plus $(n-1)$. If there is no directed edge leading to a vertex, then the vertex is called a *leaf*. A gap-graph is *acyclic* if and only if its compact graph is acyclic.

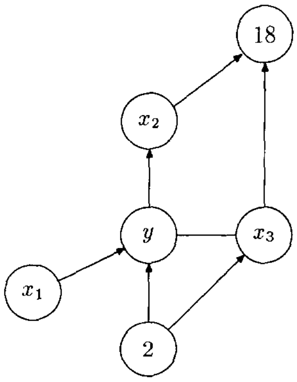


Fig. 2. Example of underlying graph.

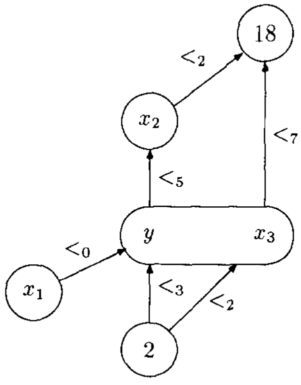


Fig. 3. Example of compact graph.

For example, $(x,y),(y,x_2),(x_2,18)$ is a path of the gap-graph (a) in Fig. 4. The length of the path is 9. The leaves of the gap-graph are x_1 and 2. The gap-graph is acyclic, because its compact graph in Fig. 3 is acyclic.

When we know the gap-value g of a directed edge (v,w), and there is no confusion with gap-orders, we will also use $v <_g w$ to refer to the directed edge.

It is clear from Lemma 3.4 that the alternative representation by gap-graphs is semantically equivalent in the usual sense to the conjunctive integer gap-order formula. Therefore, the notion of consistency of a gap-graph is naturally linked with the notion of satisfiability of a conjunctive integer order formula. The advantage of using the former is that it can be easily checked whether a gap-graph is consistent. That check is described in Lemma 3.8.

Lemma 3.8. *Let G be a gap-graph with vertices $v_1,\ldots,v_n,l,u$, where $l,u \in D \cup \{+\infty,-\infty\}$ and $l<u$. When $l,u \in D$, G is consistent if and only if it is acyclic and in the compact graph of G the longest path from l to u is less than $(u-l)$ and there is no path from u to l. When $l=-\infty$ or $u=+\infty$, G is consistent if and only if it is acyclic and there is no directed edge ending at l or starting from u.*

Proof. See Appendix. □

Note that when $l=-\infty$ or $u=+\infty$, we cannot have a directed edge ending at l or starting from u, because these would say that there is some variable v smaller than $-\infty$ or greater than $+\infty$, which is clearly impossible.

3.2. The operations on gap-graphs

In this section we describe the three basic operations on gap-graphs. These operations are called *shortcut* (Definition 3.9), *merge* (Definition 3.10), and *subsume* (Definition 3.11).

The three operations are defined on gap-graphs that serve as generalized tuples. The definitions of these operations may be broadened to generalized relations that are composed of a finite number of gap-graphs. Intuitively, in that case the shortcut would replace the projection, the subsume would replace in most cases the difference, and the merge would replace the natural join operation in relational algebra.

Definition 3.9. Let G be a gap-graph with vertices $y,v_1,\ldots,v_n,l,u$, where $l,u \in D \cup \{+\infty,-\infty\}$ and $l<u$. Then a *shortcut* operation over vertex y transforms G into an output gap-graph with vertices $v_1,\ldots,v_n$ and $v_{n+1}=l$ and $v_{n+2}=u$ as follows.

First, for each $0<i,j\leqslant n+2$, do the following:

If $v_i=y$ and $y=v_j$ are edges in G, then add $v_i=v_j$ as an undirected edge to G.

If $v_i=y$ and $y <_g v_j$ are edges in G, then add $v_i <_g v_j$ as a directed edge to G.

If $v_i <_g y$ and $y=v_j$ are edges in G, then add $v_i <_g v_j$ as a directed edge to G.

If $v_i <_{g_1} y$ and $y <_{g_2} v_j$ are edges in G, then add $v_i <_{g_1+g_2+1} v_j$ as a directed edge to G.

Second, for each g and h, if $v_i <_g v_j$ and $v_i <_h v_j$ and $g < h$, then delete first edge. If more than one edge remains between any two vertices, then the shortcut operation fails, returns an error message, and it does not produce a shortcut gap-graph as output. Otherwise, also delete y and all edges incident on y.

The intuition behind the shortcut operation is fairly straightforward. We want to eliminate a vertex y. Just erasing y and the edges incident on y is not enough, because they imply gap-order constraints about other vertices and that information would be lost. We need to explicitly preserve that information. The first part of the operation does exactly that. It is easy to see that it tests all possible cases in which two edges incident on y can imply a new gap-order constraint. Also note that the case $v_i <_{g_1} y$ and $y >_{g_2} v_j$ is left out, because these two edges do not imply a new gap-order constraint.

The first part only adds (undirected or directed) edges to the graph. As a result of the first part, it could happen that the graph will have multiple edges between some pair of vertices v and w. The second part cleans up these multiple edges. It deletes all directed edges from v to w except the one with the largest gap-order constraint. By symmetry, it does the same for all directed edges from w to v.

By Definition 3.2, there can be only one edge between each pair of vertices v and w in a gap-graph. Hence, if after the second step there remain several edges (e.g., an undirected and a directed edge, or two edges with opposite directions) between any pair of vertices v and w, then the shortcut operation cannot return a gap-graph. It is in this case that the shortcut operation fails and returns an error message saying "no gap-graph output can be produced".

Parenthetically, we remark that the gap-graph returned by the shortcut operation may be inconsistent by Definition 3.6. Note that consistency is not checked by the shortcut operation.

An example of the shortcut operation is shown in Fig. 4. The input gap-graph is shown in part (a) and the gap-graph obtained as a result of the shortcutting over vertex y is shown in part (b). There the shortcut creates three new directed edges, i.e., $x_1 <_6 x_2$, $x_1 <_1 x_3$ and $2 <_9 x_2$, and updates the gap value of one directed edge, i.e., edge $2 <_3 x_3$.

Definition 3.10. Let G_1 and G_2 be two-graphs over some (maybe different) subsets of the variables $v_1, \ldots, v_n$ and over the same constants l and u, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. Then a *merge* operation on G_1 and G_2 creates a gap-graph G with vertices $v_1, \ldots, v_n, v_{n+1} = l, v_{n+2} = u$ as follows. For each $0 < i, j \leqslant n+2$, do the following:

If there is no edge between v_i and v_j in G_1 and G_2, then do nothing.

If there is an edge between v_i and v_j in only one of G_1 or G_2, then add that edge to G.

If $v_i = v_j$ is an edge in both G_1 and G_2, then add $v_i = v_j$ as an edge to G.

If $v_i <_{g_1} v_j$ in G_1 and $v_i <_{g_2} v_j$ in G_2 are edges, then add $v_i <_{max(g_1, g_2)} v_j$ as an edge to G.

If $v_i >_{g_1} v_j$ in G_1 and $v_i >_{g_2} v_j$ in G_2 are edges, then add $v_i >_{max(g_1, g_2)} v_j$ as an edge to G.

In any other case the merge operation fails, and it does not return any graph.

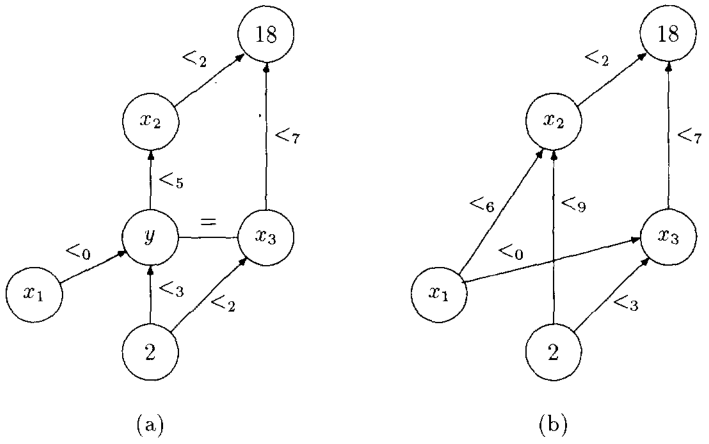


Fig. 4. Example of shortcut of a gap-graph.

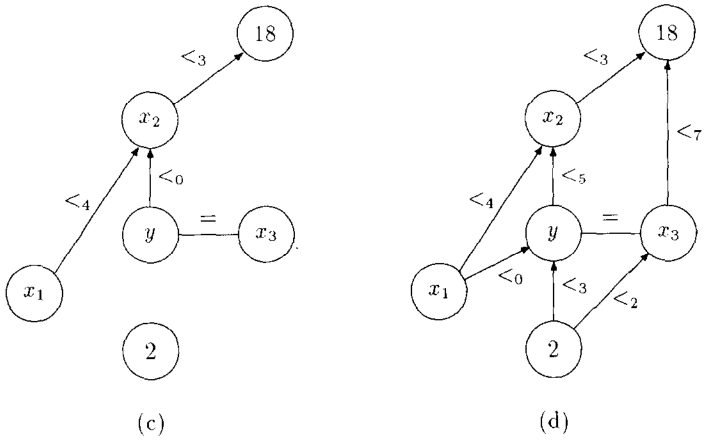
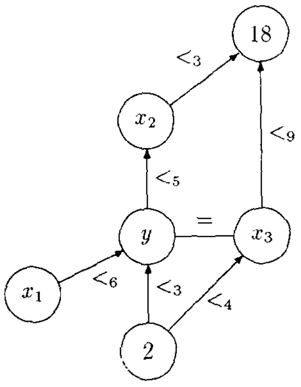


Fig. 5. Example of merge of gap-graphs.

The intuition behind the merge operation is also simple. We want any assignment that satisfies the output gap-graph to satisfy both of the input gap-graphs. The operation guarantees this by checking that the corresponding edges in the two input gap-graphs are compatible and by adding always the edge which has the stricter gap-order constraint to the output gap-graph. The last condition "in any other case" includes the cases when G_1 and G_2 are not compatible, for example, when, for some variable v and w, one specifies that v is less than w while the other says that v is greater than w. In these cases there is clearly no assignment that can satisfy both graphs; hence, the merge operation will fail.

An example of the merge operation is shown in Fig. 5. The input gap-graphs are (a) and (c), Figs. 4 and 5 and the result of the merge operation is in part (d), Fig. 5. There the merge operation creates a gap-graph such that on each edge the gap value is the



(e)

Fig. 6. Example of subsume of gap-graphs.

maximum of the corresponding gap values in the two input gap-graphs. For example, $x_2 <_2 18$ in (a) and $x_2 <_3 18$ in (c) yields $x_2 <_3 18$ in (d).

Definition 3.11. Let G_1 and G_2 be two gap-graphs over the same set of variables $v_1, \ldots, v_n$ and constants $v_{n+1} = l$ and $v_{n+2} = u$, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. We say that G_1 *subsumes* G_2 when $v_i <_{g_2} v_j$ (or $v_i = v_j$) is a directed (or undirected) edge in G_2 if and only if $v_i <_{g_1} v_j$ for some $g_1 \geqslant g_2$ (or $v_i = v_j$) is a directed (or undirected) edge in G_1.

The subsume operation serves to compare gap-graphs that have the same underlying graphs. For gap-graph A to subsume another gap-graph B, for each corresponding directed edge, A must have a larger gap-order constraint than B has. The importance of this comparison will become clear in Section 3.3, where we show a visual representation of the notion of subsumption.

Next we give an example of subsume using Fig. 6. The gap-graphs (a) and (e) in Figs. 4 and 6 have the same underlying graphs. By checking the conditions in Definition 3.11, we see that (e) subsumes (a). (Note that the "and only if" within Definition 3.11 ensures that the gap-graphs compared have the same underlying graphs.)

After describing how the three operations are performed, we now show the semantic correctness of the operations defined, that is, we show that the operations are consistency-preserving. We make this notion more precise for each of the cases discussed below. For the shorcut operation, we want to show that the input gap-graph is consistent if and only if the output gap-graph is consistent.

Lemma 3.12. *Let G be a gap-graph over variables $y, v_1, \ldots, v_n$ and constants $v_{n+1} = l$ and $v_{n+2} = u$, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. Let G' be the gap-graph obtained by shortcutting over y in G (if it exists). Let $a_0, a_1, \ldots, a_n$ be any sequence of integer numbers. Then $a_0, a_1, \ldots, a_n \models G$ if and only if G' exists and $a_1, \ldots, a_n \models G'$.*

Proof. See Appendix. □

For the merge operation, we want to show that the *and* of the input gap-graphs is consistent if and only if the output gap-graph is consistent.

Lemma 3.13. *Let G_1 and G_2 be two gap-graphs over some (maybe different) subsets of the variables $v_1, \ldots, v_n$ and over the same constants $v_{n+1} = l$ and $v_{n+2} = u$, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. Let G be the gap-graph obtained by merging G_1 and G_2 (if it exists). Then, for any assignment $\mathscr{A} = \{a_1, \ldots, a_n\}$, $\mathscr{A} \models G_1$ and $\mathscr{A} \models G_2$ if and only if G exists and $\mathscr{A} \models G$.*

Proof. See Appendix. □

For the subsume operation we want to show that if $\mathscr{A}$ is a consistent assignment to a gap-graph G, then $\mathscr{A}$ is also a consistent assignment to any gap-graph that G subsumes. Note that the reverse may not be true. Our evaluation method uses only the first direction.

Lemma 3.14. *Let G_1 and G_2 be two gap-graphs over variables $v_1, \ldots, v_n$ and constants l and u, where $l, u \in D \cup \{+\infty, -\infty\}$ and $l < u$. If G_1 subsumes G_2 then, for any assignment $\mathscr{A} = \{a_1, \ldots, a_n\}$, if $\mathscr{A} \models G_1$, then $\mathscr{A} \models G_2$.*

Proof. By Definition 3.3 $\mathscr{A} \models G_1$ if and only if $\mathscr{A}$ satisfies all gap-order constraints on the edges. Assume that $\mathscr{A} \models G_1$ and G_1 subsumes G_2. It is easy to see that $\mathscr{A}$ also satisfies each gap-order constraint in G_2. Each gap-order constraint $a_i <_{g_2} a_j$ (or $a_i = a_j$) in G_2 holds because $a_i <_{g_1} a_j$ for some $g_1 \geqslant g_2$ (or $a_i = a_j$) is true in G_1 by Definition 3.11. Hence, $\mathscr{A} \models G_2$. □

3.3. The query evaluation method

In Section 3.3.1 we present our query evaluation algorithm called EVAL for *Datalog*$^{<z}$ queries that are presented in *gap-graph form*. We also describe a simple procedure to find the gap-graph form of *Datalog*$^{<z}$ queries.

In Section 3.3.2 we show that EVAL terminates on every Datalog with integer order query that is input in gap-graph form. To prove termination we transform the problem of query evaluation to a geometric problem, by representing each gap-graph as a point in some finite-dimensional space. That enables the use of a geometric lemma (Lemma 3.16) to prove termination in Theorem 3.17.

In Section 3.3.3 we show that every *Datalog*$^{<z}$ query has a unique *full-model*, i.e., a unique fixpoint unrestricted relational database. We also show that what EVAL returns is always some generalized output database (in gap-graph form) that is a finite description of the full-model.

3.3.1. The query evaluation algorithm

At first we define what we mean by a *gap-graph form.*

Definition 3.15. Let Q be any $Datalog^{<z}$ query with program P and database d, and let u be the largest and l the smallest constant in Q. We call the *gap-graph form* of Q the query that is obtained by rewriting each generalized tuple of d into a semantically equivalent disjunction of gap-graphs, and each rule of P with a conjunction of integer order constraints into a semantically equivalent disjunction of rules with gap-graph constraints, such that all the gap-graphs use the constants l and u, and gap-graphs belonging to the same relation or rule also use the same set of vertices.

The gap-graph form described in Definition 3.15 can be obtained simply by using Lemma 3.4.

Next we describe the query evaluation algorithm called EVAL. The input of the query evaluation algorithm is a $Datalog^{<z}$ query in gap-graph form. The output of the query evaluation algorithm is a generalized output database in gap-graph form. That is, for each IDB predicate of the form $R(x_1, \ldots, x_k)$, the algorithm produces a set of gap-graphs over vertices $x_1, \ldots, x_k, l, u$.

```
Query evaluation algorithm EVAL
WHILE can add a gap-graph to the database DO
  BEGIN
    Add a gap-graph to the database using a rule application for some rule of the form
    $A_0(x_1, \ldots, x_m) :- A_1(\ldots), \ldots, A_k(\ldots), C$ in $P$, where $A_0, A_1, \ldots, A_k$ are relation
    symbols, $C$ is a gap-graph, and the set of variables in the rule is $S = \{x_1, \ldots, x_m, y_1, \ldots, y_n\}$.
    A rule application consists of the following steps:
    (1) Pick for each $A_i$, $1 \leqslant i \leqslant k$, a gap-graph from the database.
    (2) Merge the gap-graphs picked in the first step and gap-graph $C$.
    (3) Shortcut out vertices $y_1, \ldots, y_n$ to yield $G$.
    (4) Check that $G$ is consistent and does not subsume another gap-graph for $A_0$ in
        the database. If true, add $G$ for $A_0$ to the database.
  END
```

Let d be the output database. Suppose that G is a gap-graph over $x_1, \ldots, x_m, l, u$ that is derived for $A_0(x_1, \ldots, x_m)$ by some rule application during the query evaluation. Then G is a generalized tuple of the generalized output relation A_0 in d. We denote this as $A_0(G) \in d$.

3.3.2. A termination proof for EVAL

To analyze better the termination of EVAL, we provide first a visual interpretation of its operation. This visual interpretation maps each gap-graph in the database to a point in some fixed-dimensional space. The spatial relations among the points will provide important clues to the relations among the gap-graphs. We do this as follows.

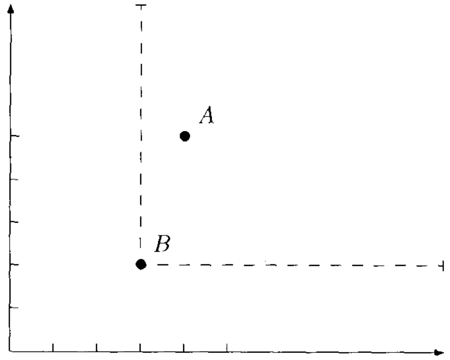


Fig. 7. Geometric interpretation of subsumption.

Let G be a gap-graph with m directed edges. Let σ be an ordering of the directed edges in G. Let $B=(b_1,\ldots,b_m)$ be the m-dimensional tuple obtained by listing the gap values in the labels of the directed edges in σ order. Then we can map G to a point in m-dimensional space.

Suppose that $G_1,\ldots,G_n$ have the same underlying graph with m directed edges. We can map each G_i as a point B_i in m-dimensional space, using the same fixed ordering of directed edges. We know that G_i subsumes G_j if and only if each coordinate of B_i is $\geqslant$ the same coordinate of B_j.

For example, looking at two gap-graphs A and B over x, l, u, where x is some variable and l and u are constants, assume that there are two directed edges in both A and B, namely, $l <_4 x$ and $x <_5 u$ are edges in A and $l <_3 x$ and $x <_2 u$ are edges in B. Then we can map these gap-graphs to points in two dimensions as shown in Fig. 7. There is a visual representation for the fact that A subsumes B, and so does any gap-graph that is mapped to a point in the upper rightmost region bounded by the dashed lines.

We will also use the following geometric lemma.

Lemma 3.16. *In any fixed dimension, any sequence of distinct points with only natural-number coordinates must be finite, if no point subsumes any earlier point in the sequence.*

Proof. We prove the theorem by induction on the dimension k of the space in which the points lie. For $k=0$, the whole space is only a single point; hence, the theorem holds. Now we assume that the theorem holds for k dimensions and show that it is true for $k+1$ dimensions.

Let S be any arbitrary sequence of points in which no point subsumes any earlier point Let $x_1,\ldots,x_{k+1}$ be the coordinate axis of the $(k+1)$-dimensional space, and let $(a_1,\ldots,a_{k+1})$ be the first point in S. By the requirement of nonsubsuming and distinctness, for any later point $(b_1,\ldots,b_{k+1})$ for some i, $0\leqslant i\leqslant k+1$, it must be true that $b_i<a_i$. This means that any point in the sequence after $(a_1,\ldots,a_{k+1})$ must be

within one or more of the k-dimensional regions $x_1=0$ or $x_1=1$ or $\ldots, x_1=a_1$ or $\ldots x_{k+1}=0$ or $x_{k+1}=1$ or $\ldots x_{k+1}=a_{k+1}$.

As we add points to each of these regions from S, no point can subsume any earlier one within these regions. Therefore, by the induction hypothesis, only a finite number of points from S can be placed into each of these k-dimensional regions. Since the number of these regions is finite, S also must be finite. □

Now we are ready to prove that EVAL terminates on all of its inputs, which means that there is always a gap-graph closed form.

Theorem 3.17. *Any Datalog$^{<z}$ query has a bottom-up evaluation that terminates in a gap-graph closed form.*

Proof. Suppose that we use algorithm EVAL to evaluate a query P. Let $R(x_1, \ldots, x_k)$ be any IDB predicate in P. All database facts for R are gap-graphs over $x_1, \ldots, x_k, l, u$. Consider all the gap-graphs over these vertices. There is an infinite number of these gap-graphs. However, the gap-graphs can have only 4^{k+2} underlying graphs. That is because, between each distinct pair of vertices v and w, we may have no edge, $v=w$, $v<w$, or $v>w$. For each of these underlying graphs, fix an ordering for the set of directed edges. Also, for each underlying graph with m directed edges, create an m-dimensional picture. We have only a finite number of pictures and each picture has a finite dimension. After each rule application, if a gap-graph G is added to R, then map G to a point in the picture that corresponds to the underlying graph of G, using the fixed ordering. By Lemma 3.16, the mapping to each picture terminates. Since there are a finite number of IDB predicates, reasoning similarly to R for each, we see that the bottom-up evaluation described in EVAL terminates. □

3.3.3 A model for Datalog queries with integer order

In the introduction we gave an unrestricted semantics to *Datalog*$^{<z}$ queries. According to the unrestricted semantics, a generalized database d_1 with a finite number of gap-graph tuples can be interpreted to be a finite description for some unrestricted relational database d_2 with a finite or infinite number of standard relational database tuples. We call d_2 the *full version* of d_1. We make this more precise in the following definition.

Definition 3.18. Let d_1 be a generalized relational database with gap-graph tuples, and let d_2 be an unrestricted relational database. Then d_2 is the *full version* of d_1 if, for all tuples $t, A(t) \in d_2$ if and only if there exists G such that $t \models G$ and $A(G) \in d_1$. □

The unrestricted semantics together with Tarski's fixpoint theorem can be used to show that *Datalog*$^{<z}$ queries have a unique unrestricted least model. However, the unrestricted least model using the *naive evaluation* implied by Tarski's fixpoint theorem would take an infinite number of iterations to evaluate. In this section we

show that our query evaluation algorithm finds a finite description for the unrestricted least model. By Theorem 3.17, we know that this description is always some generalized database with gap-graph tuples, and it is evaluable in finite time.

Our query evaluation algorithm is nondeterministic, and it may return different generalized databases with gap-graph tuples. That is no cause for alarm. It simply means that there may be several finite descriptions for the same unrestricted least model.

For example, think of an empty database input and a program with two rules: $Out(x)$:—$10 <_5 x$ and $Out(x)$:—$10 < x$. If we choose the first rule and then the second rule, we have two gap-graphs for the *Out* database relation. If we choose the second rule first, then we have only one, because the gap-graph $10 <_5 x$ subsumes $10 < x$ and will not be added. Since we have in both cases the gap-graph $10 < x$ in the output, the two different sets of gap-graph outputs have the same unrestricted least models.

Theorem 3.19. *Let P be a Datalog$^{<z}$ query with generalized database d_1 in gap-graph form. Let d_2 be the full version of d_1. If $M_P(d_1)$ is any output of EVAL on P and $L_P(d_2)$ is the output of the naive evaluation of $P[d_2/d_1]$ (with input d_2 instead of d_1) then $L_P(d_2)$ is the full version of $M_P(d_1)$.*

Proof. (*If*): Let $M_P^i(d_1)$ denote the database after the ith rule application in EVAL. We prove by induction in i that, for all tuples t, if there exists G such that $t \models G$ and $A(G) \in M_P^i(d_1)$, then $A(t) \in L_P(d_2)$. For $i=0$, $M_P^0(d_1)=d_1$ and $d_2 \subseteq L_P(d_2)$; hence, the claim holds. Now assume the claim for i and prove for $i+1$.

Suppose the $(i+1)$th rule application uses rule $A_0(x_1,\ldots,x_m)$:—$A_1(\ldots)$, $A_2(\ldots)$, $\ldots, A_k(\ldots), C$ of P, where C is a gap-graph, and let $S=\{x_1,\ldots,x_m,y_1,\ldots,y_n\}$ be the variables in the rule. Suppose the application picks from $M_P^i(d_1)$ gap-graphs $G_1,\ldots,G_k$ for $A_1,\ldots,A_k$. Let M be the merge of $G_1,\ldots,G_k$ and G the shortcut of M over $y_1,\ldots,y_n$. Assume that G is consistent and that $A_0(G)$ is added to $M_P^{i+1}(d_1)$. Note that $M_P^{i+1}(d_1)$ is also in gap-graph form.

Let $t=(a_0,\ldots,a_m)$ be any tuple such that $t \models G$. By Lemma 3.12, there exists a $\mathscr{B}=\{a_0,\ldots,a_m,b_1,\ldots,b_n\}$ such that $\mathscr{B} \models M$. By Lemma 3.13, $\mathscr{B} \models G_1,\ldots,\mathscr{B} \models G_k$, and $\mathscr{B} \models C$. Note that, for each $1 \leqslant j \leqslant k$, the jth projection of $\mathscr{B}$ generates a tuple t_j such that $t_j \models G_j$. By the induction hypothesis, $t_j \models G_j$ and $A_j(G_j) \in M_P^i(d_1)$ implies that $A_j(t_j) \in L_P(d_2)$. Then, by the naive evaluation, $A_0(t) \in L_P(d_2)$ must be true. Reasoning similarly for each t, we see that, for all t, if $t \models G$ then $A_0(t) \in L_P(d_2)$. Since $A_0(G)$ is the only fact added in the $(i+1)$th application, the claim holds.

(*Only if*): Let $L_P^i(d_2)$ denote the database after the ith rule application is the naive evaluation. We prove by induction on i that, for all tuple t, if $A(t) \in L_P^i(d_2)$, then there exists G such that $t \models G$ and $A(G) \in M_P(d_1)$. For $i=0$, $L_P^0(d_2)=d_2$ and $d_1 \subseteq M_P(d_1)$; hence, the claim holds. Now assume the claim for i and prove for $i+1$.

Suppose the $(i+1)$th rule application uses rule $A_0(x_1,\ldots,x_m)$:—$A_1(\ldots), A_2(\ldots)$, $\ldots, A_k(\ldots), C$ of P, where C is a gap-graph and let $S=\{x_1,\ldots,x_m,y_1,\ldots,y_n\}$ be the variables in the rule. To perform the $(i+1)$th rule application in the naive evaluation, assume that some assignment $\mathscr{B}=\{a_1,\dots,a_m,b_1,\dots,b_n\}$ is chosen for the values of the variables in the rule. Each substitution of $\mathscr{B}$ into $A_1,\dots,A_k$ yields a relational tuple $A_1(t_1),\dots,A_k(t_k)$. Suppose that these are all present in $L_P^i(d_2)$ and that $\mathscr{B}\models C$; hence, $A_0(t)$, where $t=(a_1,\dots,a_m)$, is added to $L_P^{i+1}(d_2)$.

By the induction hypothesis there must be gap-graphs $G_1,\dots,G_k$ such that $t_1\models G_1,\dots,t_k\models G_k$ and $A_1(G_1),\dots,A_k(G_k)\in M_P(d_1)$. Hence, with the necessary restrictions, $\mathscr{B}\models G_1,\dots,\mathscr{B}\models G_k$, and $\mathscr{B}\models C$. Let M be the gap-graph obtained by merging the extensions of $G_1,\dots,G_k$ and C. By Lemma 3.13, M exists and $\mathscr{B}\models M$. Let G be the gap-graph obtained by shortcutting over vertices $y_1,\dots,y_n$ in M. By Lemma 3.12, G exists and $\mathscr{B}\models G$. Clearly, G is consistent. If G does not subsume another gap-graph for A_0 in $M_P(d_1)$ then, by the generalized evaluation, $A_0(G)\in M_P(d_1)$ must be true. If G subsumes another gap-graph G' for A_0 in $M_P(d_1)$ then, by Lemma 3.14, $\mathscr{B}\models G'$. Hence, either $t\models G$ and $A_0(G)\in M_P(d_d)$ or $t\models G'$ and $A_0(G')\in M_P(d_1)$, as required. By reasoning similarly for each t, we see that the claim holds. □

Corollary 3.20. *Let P be any generalized database logic program with generalized database d. Let $M_1(d)$ and $M_2(d)$ be the outputs of two different sequences of executions of EVAL. Then for all tuple t, exists gap-graph G_1 such that $t\models G_1$ and $A(G_1)\in M_1(d)$ if and only if there exists gap-graph G_2 such that $t\models G_2$ and $A(G_2)\in M_2(d)$.*

We call each finite description output a *gap-model* of the *Datalog*$^{<z}$ query P. We also call the unrestricted least model the *full-model* $\mathscr{M}_P$ of P because it is the set of relational tuples that satisfy any generalized relational tuple in any gap-model. Corollary 3.20 is another way of seeing that $\mathscr{M}_P$ is unique even though the gap-models are not.

4. The recognition problem

In this section we present an efficient tuple recognition algorithm called TEST. By tuple recognition we mean checking whether a given standard relational database tuple is in $\mathscr{M}_P$, where $\mathscr{M}_P$ is the full-model of a given *Datalog*$^{<z}$ query (program with database) P. We saw in the previous section that $\mathscr{M}_P$ is a unique unrestricted (finite or infinite) set of standard relational tuples. Therefore, the tuple recognition problem is well-defined. For a fixed *Datalog*$^{<z}$ program P and any generalized input database d, the complexity of performing the tuple recognition check in terms of the size of d we call the *generalized data complexity* of P (see also the definition in Section 2).

The tuple recognition problem is motivated by several reasons. For example, if tuples in a database describe pairs of cities between which there is an airplane flight, a user may want to know only whether the tuple which describes a connection between two particular cities is in the model of the query.

Algorithm TEST is a modification of algorithm EVAL of Section 3. Suppose that we are given a tuple t to be tested. Then let l be the largest and u the smallest constant in t or the generalized database, which we assume is given in gap-graph form. The first observation is the following. Each gap-graph is composed of three main parts: first, those vertices which are less than l; second, those vertices which are between l and u; and third, those vertices which are greater than u. Of course, some gap-graphs may have parts other than the three just mentioned, for example, they may have isolated vertices that are indeterminate with respect to l and u. The definition of *(l, u)-graphs* (see Definition 4.1) is used to eliminate these other parts from consideration and to simplify the reasoning in this section.

The second observation is that the precise gap values are necessary to keep only for the vertices and edges in the second group. Note that t itself will have only vertices between l and u. Hence, for the tuple recognition problem it is enough to test whether t satisfies any gap-graph G in the full-model, which has only vertices between l and u. For all the other gap-graphs in the generalized input and output databases instead of the precise gap values for the edges in the first and the third groups, it is enough that we keep the directions only. We call these simplified gap-graphs the *partial graphs* (see Definition 4.2). Each of the partial graphs serves as a representative of a set of gap-graphs which are exchangeable as far as recognizing t is concerned.

Algorithm TEST will take as inputs and give as outputs partial graphs. This simplification yields a well-defined upper bound on the size of the output database in terms of the size of the query (see Definition 4.3 and Lemma 4.4). That is the main intuition behind Theorem 4.7.

Definition 4.1. Let G be a gap-graph over $x_1, \ldots, x_n, l, u$ with constants $l < u$. If, for each vertex x_i in G, $x_i <_g l, x_i = l, x_i = u, x_i >_g u$ is an edge, or $l <_g x$ and $x_i <_h u$ are both edges in G for some g, $h \geqslant 0$, then G is called an *(l, u)-graph.*

For example, the gap-graph (a) in Fig. 4, which has $l = 2$ and $u = 18$, is not a (2, 18)-graph because, for vertex x_1, none of the five conditions listed in Definition 4.1 hold. However, if we added any one of the edges $x_1 <_{99} 2$, $x_1 = 2$, or $2 <_1 x_1$, then we would have a (2, 18)-graph.

Definition 4.2. Let G be an (l, u)-graph over $x_1, \ldots, x_n, l, u$ with constants $l < u$. We call the graph obtained by deleting all gap values in G except on the edges that are on a path from l to u in the compact graph of G the *partial graph* of G.

Suppose we added to gap-graph (a) (Fig. 4) the edge $x_1 <_{99} 2$ to obtain a (2, 18)-graph. To obtain a partial graph, we now have to delete the gap-value 99 on the edge from x_1 to 2.

Definition 4.3. Let $A(x_1, \ldots, x_k)$ be a relational atom and G_p be the partial graph of some consistent (l, u)-graph over $x_1, \ldots, x_k, l, u$. Then $A(G_\text{p})$ is called a base.

We call *p-application* the rule application with G replaced by the partial graph G_p of G and the subsumption check skipped in step (4). We call TEST the algorithm EVAL with "gap-graph" replaced with "partial graph" and "application" replaced with "p-application".

Lemma 4.4. *Let P be a generalized database logic program with m IDB predicates, each with arity $\leqslant k$. Let l be the smallest and u the largest element of D. Then there are $m(u-l)^{O(k^2)}$ bases.* TEST *terminates after at most that many iterations.*

Proof. Let $A(G_p)$ be any base. Note that G_p is consistent by Definition 4.3. Assume that A has arity k. For $k+2$ vertices, there are $(k+2)$ $(k+1)/2$ distinct pairs. For each of these, we may have no edge, an $=$ edge, a $<_g$ or a $>_g$ gap-order for some g. If the edge does not lie on a path from l to u in the compact graph of G_p, then the gap value must be absent (or 0). It is does then, by consistency and Lemma 3.3, the gap value $g<u-l$. Therefore, for each $<_g$ and $>_g$ edge, there can be at most $u-l$ different gap values. This gives at most $(2+2(u-l))^{(k+2)(k+1)/2}$ many choices for base $A(G_p)$. We can reason similarly for each of the m relational atoms. Since TEST derives always bases, TEST terminates within that many iterations. □

The key reason that using bases works is the limited interaction among the three main parts of gap-graphs. As a result, we can show in Lemmas 4.5 and 4.6 that a gap-graph G is derived by EVAL if and only if the partial graph G' of G is derived by TEST.

Lemma 4.5. *Let $A_0 :\!- A_1, \ldots, A_k, C$ be a rule in a generalized database logic program P, and let $G_1, \ldots, G_k$, C be (l,u)-graphs and $G'_1, \ldots, G'_k, C'$ be their partial graphs. Then a rule application derives $A_0(G)$ using $A_1(G_1), \ldots, A_k(G_k)$ and C if and only if a p-application derives $A_0(G')$ using $A_1(G'_1), \ldots, A_k(G'_k)$ and C', where G' is the partial graph of G.*

Proof. See Appendix. □

Lemma 4.6. *Let P be a generalized database logic program with a database d_1 that has only (l,u)-graph constraints in the database. Let d_2 be the database which has exactly the partial form of each (l,u)-graph in d_1. Then algorithm* EVAL *derives $A_0(G)$ using d_1 as input if and only if algorithm* TEST *derives $A_0(G')$ using d_2 as input, where G' is the partial graph of G.*

Proof. We can define generalized derivation trees. Structurally, the derivation tree for any $A_0(G)$ in d_1 is the same as for any $A_0(G')$ in d_2, assuming that EVAL is run subsumption-test-free. Then the proof is by induction on the depth of the derivation trees using Lemma 4.5.

Suppose now that $A_0(G)$ is derived but not added to the database by EVAL because there is already an $A_0(G'')$ such that G subsumes G''. By Lemma 3.14, there is no t such that $t \models G$ and $t \not\models G''$. By Theorem 3.19, P has the same full-model whether we add (just this time) G to the database or not. Hence, by induction on the derivation trees, EVAL finds the same full-model in a subsumption-test-free run as in a regular run. □

Theorem 4.7. *Let t be any tuple. For any fixed generalized database logic program P with a generalized database d, we can test whether $A_0(t) \in \mathcal{M}_P$ in $O(n^{k+1} + (u-l)^{O(mk^2)})$ time, where n is the size of d, k is the largest arity of a relation in P, u is the largest and l is the smallest constant in either t or the query, and m is the maximum number of relation symbols in a rule of P.*

Proof. Transform the query into gap-graph form using Lemma 3.4. This takes $O(n^{k+1})$ time and creates at most that many gap-graphs. Next transform each gap-graph G over $x_1, \ldots, x_k, l, u$ in the query into a partial, (l, u)-graph.

First, transform G into an (l, u)-graph. For each x_i, select one of the constraints $x_i < l$, $x_i = l$, $x_i = u, x_i > u$, or $l < x_i$ and $x_i < u$, and add that to G, unless there are equal or stronger gap-order constraints already present in G. After the selected constraints are added, we obtain an (l, u)-graph by Definition 4.1. We can make 5^k selections. For these (l, u)-graphs the set of consistent assignments is disjoint. Since the size of G is $O(k^2)$, the total size of the set of (l, u)-graphs is $O(5^k k^2)$. Second, make all (l, u)-graphs partial by deleting unnecessary gap values. Since many (l, u)-graphs may have the same partial graph, this makes the size of the database only smaller. This step takes $O(k^2)$ time for each (l, u)-graph and $O(5^k k^4)$ time for each gap-graph G. Hence, the initialization requires $O(n^{k+1})$ time.

Now run algorithm TEST. For gap-graphs with k vertices, each merge, shortcut and rule application takes $O(k^2)$ time, i.e., each takes constant time. To do one iteration, we may need to pick from the database m bases, which is another constant. Since there are, by Lemma 4.4, $m(u-l)^{O(k^2)}$ many bases, the maximum number of substitutions that need be tried is $O((u-l)^{O(mk^2)})$. Each iteration will take that much time.

Lemma 4.4 also implies that TEST will terminate after $m(u-l)^{O(k^2)}$ iterations because that is the maximum number of times we may derive a new base. Since the total running time of algorithm TEST is the product of the time to do one iteration and the number of iterations, we have $O((u-l)^{O(mk^2)}) * m(u-l)^{O(k^2)} = O((u-l)^{O(mk^2)})$ total running time, not counting the initialization. When TEST terminates, test, for each base $A(G')$, whether $t \models G'$. These tests require $O(k^2)$, that is, a constant time for each base and, hence, a total of $m(u-l)^{O(k^2)}$ time. Therefore, the total running time including the initialization and the model tests at the end is $O(n^{k+1} + (u-l)^{O(mk^2)})$.

The correctness of the algorithm can be checked as follows. By Lemma 4.6, if $A(G')$ is a base in the database, then G' is the partial graph of some G such that $A(G)$ would be in the database by running EVAL subsumption-test-free instead of TEST. Since G' and G are (partial) (l, u)-graphs, for each vertex x of both G and G', one of $x <_g l$, $x = l$, $l <_g x <_h u$, $x = u$, or $x >_g u$ must be true for some g and h. Suppose that $t \models G'$. Since t satisfies all gap-order constraints in G' and all values within t are $\geqslant l$ or $\leqslant u$, the first and the last cases cannot happen. Hence, every vertex is equal to l or u or lies on a path from l to u in the compact graph of G'. Hence, every edge will lie on a path from l to u; hence, G' must be the same as G by Definition 4.2. Hence, $t \models A(G)$ and, by Theorem 3.19, $t \in \mathscr{M}_P$. □

4.1. *Piecewise linear programs*

We can find another subset of queries for which the recognition test can be done efficiently, namely, in NLOGSPACE data complexity. These are the piecewise linear Datalog programs of Ullman and Van Gelder [29]. These queries have only one recursive predicate in each rule (for the exact definition, see [29]).

Theorem 4.8. *The recognition test for piecewise linear Datalog and generalized databases with gap-graph tuples can be done in* NLOGSPACE.

Proof. As in Lemma 4.6, it is enough to consider only subsumption-test-free derivations of algorithm EVAL. In piecewise linear recursive programs all derivations are chains with possible sideway branches of length one. That is because at every rule application we use a rule that has only one recursive predicate. The rule application will find one IDB gap-graph for the predicate in the head of the rule, using only the previously derived IDB gap-graph for the recursive predicate in the subgoals (the main line of the chain) and one gap-graph from the input database for each of the nonrecursive predicates in the subgoals (the length one branches).

It follows that, during any nondeterministic derivation, we need to store only as many gap-graphs as is necessary for one rule evaluation. We store the last derived IDB gap-graph and the gap-graphs form the database that we are considering for the rule application. Since the program is fixed, the number of gap-graphs that we need to store has a fixed constant upper bound which is exactly the maximum number of predicates in any one of the rules of the program.

It is also easy to see that each gap-graph requires only $O(\log u)$ space to store, since there are only a constant number of edges in any gap-graph – more precisely there are $O(k^2)$ edges, where k is the maximum arity of a relation in the program – and the size of the gap value on each of the edges is at most $u - l$.

Therefore, by guessing always correctly the rule to use and the substitutions from the database, we can derive in NSPACE($\log n$) any given relational tuple $A(t)$ that is derivable. (Or derive a gap-graph G of size $O(\log u)$ for the relation A and verify that $t \models G$). As shown by Immerman [13] and Szelepcsènyi [27], NSPACE($\log n$) is closed under complement. Hence, we can test in NSPACE($\log n$) whether $A(t)$ is derivable. □

5. Open problems

We list some open problems in conclusion:

(1) Establish a good bound on the maximum gap size used by EVAL.

(2) Develop more and better pruning methods for rule applications.

(3) Find complexity class lower bounds. For example, we could consider, instead of tuple recognition, the problem of gap-graph recognition, i.e., tell whether all the standard relational tuples implied by a gap-graph are in the model of a query. Would the lower bound be NP-hard in this case?

(4) Is it possible to combine $Datalog^{<_z}$ queries with the results in [7, 15, 18]?

(5) Is it possible to add negation to $Datalog^{<_z}$ in a safe way? Proposition 2.3 shows only that stratified negation is not feasible.

Appendix

Proof of Lemma 3.4. At first, eliminate from C all constraints between two constants. If any of these contraints is *false*, then C is also *false*. In this case we stop and do not return any gap-graph. If all of these constraints are *true*, then simply delete them from C. This simplification can be done in linear time in the size of C. If $k=0$, that is, C has no variables and is now empty, then we return the gap-graph with the single edge $l<u$ representing *true*. If $k\geqslant 1$, let s be the number of constants in the simplified formula and continue as follows.

(1) Eliminate from C the $\neq$ and $\leqslant$ constraints that have only variables in them. Replace in C all constraints of the form $v\neq w$ by $(v<w)\vee(w<v)$, and of the form $v\leqslant w$ by $(v<w)\vee(v=w)$, where v and w are both variables. Also eliminate from C the $\leqslant$constraints that have one constant in them. For each variable v and constants c_1 and c_2 such that $c_1<c_2$, if $v\leqslant c_1$ and $v\leqslant c_2$, then delete $v\leqslant c_2$. Similarly, if $c_1\leqslant v$ and $c_2\leqslant v$, then delete $c_1\leqslant v$. After the deletions, each variable will have at most one "$\leqslant$ upper bound" and at most one "$\leqslant$ lower bound". Replace each of the upper bounds $v\leqslant c$ by $(v<c)\vee(v=c)$ and each of the lower bounds $c\leqslant v$ by $(c<v)\vee(c=v)$.

Between pairs of variables, there can be at most $2k^2$ number of $\neq$ or $\leqslant$ constraints and replacements for them. Also, for k variables there can be only at most $2k$ number of upper and lower bounds and replacements for them. Altogether, there are at most $2k^2+2k$ replacements, each containing one $\vee$ connective and two atomic constraints.

Next put C into disjunctive normal form (i.e., disjunction of conjunctions). Since we choose always one atomic constraint from each replacement, the size of C will be at most $2^{2k^2+2k}n$, and the size of each disjunct (i.e., conjunction of constraints) of C will be at most n.

(2) Eliminate from each disjunct C' of C the $\neq$ constraints that have one constant in them. For each variable v do the following. Order the constants that occur within constraints of the form $v\neq c$ (or $c\neq v$), where c is any constant. Let these constants be

$c_1, c_2, \ldots, c_p$ in increasing order. Then replace these $\neq$ constraints by $(v<c_1 \vee (c_1<v \wedge v<c_2) \vee \cdots \vee c_p<v)$.

Replacing the $\neq$ constraints doubles the size of C', because we use two $<$ constraints for each $\neq$. There are k variables and k replacements. Since $p \leqslant s$, there are at most s new $\vee$ connectives in each replacement. Put C' into disjunctive normal form. The size of the disjunctive normal form of C' will be at most $(s+1)^k n$. The size of each disjunct C'' of C' will be at most n. Note that this last step puts C again into disjunctive normal form.

(3) After steps (1) and (2), the constraint C will be in disjunctive normal form, will have only $=$, $<$, and $<_g$ constraints, and will have at most $2^{2k^2+2k}(s+1)^k n$ size. We create a gap-graph for each disjunct C' of C as follows.

First replace constraints of the form $v=c$ by $((c-1<v) \wedge (v<c+1))$, where v is a variable and c is a constant, with $l<c<u$. Next replace each $<$ constraint by a $<_0$ gap-order constraint. We check now whether there are any constants or variables v and w such that more than one of $v=w$, $v <_g w$, and $w <_h v$ occur as constraints in C' for some g and h. If there are, then C' is clearly unsatisfiable; hence, we need not create a gap-graph G for it. Otherwise, take each constraint in C' in order.

If the constraint is $x_i = x_j$, then add $x_i = x_j$ as edge to G.

If the constraint is $x_i = l$ (or $x_i = u$), then add $x_i = l$ (or $x_i = u$) as edge to G.

If the constraint is $x_i <_g x_j$ for some g and it is the largest gap-value that occurs between x_i and x_j, then add $x_i <_g x_j$ as edge to G.

If the constraint is $c <_g x_i$ for some g and c, with $l \leqslant c \leqslant u$, and this is the largest gap-order lower bound on x_i within C', then add $l <_{(c-l+g)} x_i$ as edge to G.

If the constraint is $x_i <_g c$ for some g and c, with $l \leqslant c \leqslant u$, and this is the smallest gap-order upper bound on x_i within C', then add $x_i <_{(u-c+g)} u$ as edge to G.

It is easy to see that, for each C', if this algorithm yields a gap-graph, then the conjunction of the gap-order constraints within that gap-graph is equivalent to C'. This proves condition (a) of the lemma. Clearly, steps (1) and (3) require only a linear time in the size of the output. Sorting in step (2) requires $O(n \log n)$ time. As a preprocessing, we may produce a sorted list of the constants in C. Then the rest of step (2) also requires a linear time in the size of the output. Hence, the total time to produce the gap-graphs is $O(n \log n + 2^{2k^2+2k}(s+1)^k n)$. Since $s \leqslant n$ and $k \geqslant 1$ is fixed, the gap-graphs can be created in $O(n^{k+1})$ time. This proves condition (b). □

Proof of Lemma 3.8. In this proof we use G_c to denote the compact graph of G and the function $llp(v, w)$ for vertices v and w to denote the length of the longest path from v to w in G_c. We measure all path lengths in G_c. At first assume that $l, u \in D$ and that there is no path from u to l.

(*If*): Assume that the gap-graph G is acyclic and the longest path from l to u is less than $u-l$. Let m be the length of the longest path in G_c. Then we will make a consistent assignment to $v_1, \ldots, v_n$ as follows.

For each vertex v, with a path from l or u to v in G_c, assign the maximum of values $l+llp(l,v)+1$ and $u+llp(u,v)+1$. For each leaf vertex (other than possibly l or u) assign the value $(l-m-1)$. For each nonleaf vertex w, if there is no path from l or u to w, assign the maximum of the values $(l-m-1)+\max_x llp(x, w)+1$, where x is any leaf node in the gap-graph.

This assignment satisfies all the gap-order constraints in G. The only potential difficulty is in showing that all upper bounds are satisfied, involving u during the first set of assignments and involving all first-group vertices during the second set of assignments.

It is easy to see that the first group of vertices are assigned as low values as possible. Let v be any vertex in the first group. If $v <_g u$ is an edge in G, then $value(v) = l+llp(l,v)+1$ by our assignment and by acyclicity. Also, since $llp(l,u)<u-1$, $value(v)+g=l+llp(l,v)+1+g\leqslant l+llp(l,u)<l+(u-l)=u$. Hence, $value(v)+g<u$, as required.

Similarly, the second group of nonleaf vertices are also assigned as low values as possible. Let w be any vertex in the second group. If $w <_g v$ is an edge in G, with w in the second group and v in the first group then, by our assignment, $value(w)=(l-m-1)+\max_x llp(x,w)+1$, where x is any leaf node. Since, for each leaf node x, $llp(x,v)\leqslant m$, $value(w)+g=(l-m-1)+\max_x llp(x,w)+1+g\leqslant(l-m-1)+m=l-1<value(v)$. Hence, $value(w)+g<value(v)$, as required.

(*Only if*): If G_c has a cycle, the length of the longest path from l to u is $\geqslant(u-l)$, or there is path from u to l, then clearly there is no consistent assignment.

When $l=-\infty$ or $u=+\infty$ and there is a directed edge $(v,l)=(v,-\infty)$ or $(u,v)=(+\infty,v)$, then clearly there is no consistent assignment. Otherwise, we simply treat G as having no l or u in it, and carry out the above proof with the necessary simplifications. □

Proof of Lemma 3.12. (*If*): Assume that G' exists and assignment $\mathscr{A}'=\{a_1,\ldots,a_n\}\models G'$. Within G, let $u_1 <_{g_1} y, u_2 <_{g_2} y, \ldots, u_m <_{g_m} y$ be the edges incident on y with a gap-order label that constraints y from below, where the u's are vertices. Also, within G, let $y <_{f_1} w_1$, $y <_{f_2} w_2, \ldots, y <_{f_n} w_n$ be the edges incident on y with a gap-order label that constraints y from above, where the w's are vertices.

We want to find an assignment $\mathscr{A}=\{a_0,\ldots,a_n\}$ for the vertices in G that satisfies all the gap-order constraints in the edges. Take assignment $\mathscr{A}'$ as a substitution for all the vertices save for y. Then find k for which u_k+g_k is maximum and p for which w_p-f_p is minimum.

To obtain G' from G by a shortcut operation over y, we added in case 4 a constraint between each u_i and w_j, $1\leqslant i\leqslant m$ and $1\leqslant j\leqslant n$. In particular, we added the constraint $u_k <_{g_k+f_p+1} w_p$. Clearly, it is sufficient and necessary to choose a value a_0 for y such that $u_k+g_k<a_0<w_p-f_p$ to satisfy all lower and upper bounds on y within G.

Since $\mathscr{A}'\models G'$, there are at least g_k+f_p+1 integers between u_k and w_p after the assignment. Therefore, it is always possible to pick an integer a_0 such that there are at least g_k integers between u_k and a_0 and at least f_p integers between a_0 and w_p. If there is no v_i such that $y = v_i$ is a constraint in G, then pick any such a_0. Then $\mathscr{A} \models G$.

If $y = v_i$ for some v_i in G, then *let* $a_0 = a_i$. To obtain G' from G by a shortcut operation over y, we added in cases 2 and 3 as constraints $u_1 <_{g_1} v_i, u_2 <_{g_2} v_i, \ldots, u_m <_{g_m} v_i$ and $v_i <_{f_1} w_1, v_i <_{f_2} w_2, \ldots, v_i <_{f_n} w_n$. Therefore, a_0 satisfies all the lower and upper bounds on y. If any other $v_j = y$, then by case 1 of the shortcut, we know that $v_i = v_j$ is a constraint in G'; hence, $a_i = a_j$. Therefore, our choice for a_0 satisfies all constraints on y. Hence, $\mathscr{A} \models G$.

(*Only if*): Let $\mathscr{A} = \{a_0, a_1, \ldots, a_n\}$ be an integer assignment to the variables of G, such that $\mathscr{A} \models G$. Clearly, for all $0 < i, j \leqslant n+2$, if $v_i = y$ and $y = v_j$, then $v_i = v_j$ holds, if $v_i = y$ and $y <_g v_j$, then $v_i <_g v_j$ holds, if $v_i <_g y$ and $y = v_j$, then $v_i <_g v_j$ holds, and if $v_i <_{g_i} y$ and $y <_{g_j} v_j$, then $v_i <_{g_i + g_j + 1} v_j$ holds. Note that the shortcut operation adds only these types of constraints in the first step. Therefore, $\mathscr{A}$ is a consistent assignment to the graph after the first step. The deletions in the second step leave between each v_i and v_j possibly one gap-order $v_i = v_j$ (if there was one), possibly one gap-order $v_i <_{g_1} v_j$ (if there was any gap-order of the form $v_i <_{g_2} v_j$), and possibly one gap-order $v_i >_{h_1} v_j$ (if there was any gap-order of the form $v_i >_{h_2} v_j$). If there were more than one edge between any v_i and v_j, then the graph would be inconsistent and the shortcut operation would fail. However, deleting constraints must leave the graph consistent; hence, the shortcut operation cannot fail and it will return a gap-graph G'. Therefore, $\mathscr{A}' = \{a_1, \ldots, a_n\}$ is an integer assignment to the variables in G' such that $\mathscr{A}' \models G'$. □

Proof of Lemma 3.13. (*If*): Assume that G exists and $\mathscr{A} \models G$. By Definition 3.3, $\mathscr{A}$ satisfies all gap-order constraints on the edges in G. We show that $\mathscr{A}$ also satisfies all gap-order constraints on the edges in G_1 and G_2.

Let v_i and v_j be any pair of vertices. If $v_i = v_j$ in G_1 then, by either case 2 or 3, the merge operation adds $v_i = v_j$ to G. If $v_i <_g v_j$ in G_1 then, by either case 2 or 4, the merge operation adds $v_i <_h v_j$ to G for some $h \geqslant g$. If $v_i >_g v_j$ in G_1 then, by either case 2 or 5, the merge operation adds $v_i >_h v_j$ to G for some $h \geqslant g$. Hence, if there is a gap-order constraint between any pair of vertices in G_1, then there is an equal or stronger constraint between that pair of vertices in G. Therefore, $\mathscr{A} \models G_1$ must be true. By symmetry, $\mathscr{A} \models G_2$ must also be true.

(*Only if*): Assume that $\mathscr{A} \models G_1$ and $\mathscr{A} \models G_2$. Clearly if $v_i = v_j$ or $v_i <_g v_j$ or $v_i >_h v_j$ in either G_1 or G_2, then $v_i = v_j$ or $v_i <_g v_j$ or $v_i >_h v_j$ holds, if $v_i = v_j$ in G_1 and $v_i = v_j$ in G_2, then $v_i = v_j$ holds, if $v_i <_{g_1} v_j$ in G_1 and $v_i <_{g_2} v_j$ in G_2, then $v_i <_{\max(g_1, g_2)} v_j$ holds, if $v_i >_{g_1} v_j$ in G_1 and $v_i >_{g_2} v_j$ in G_2, then $v_i >_{\max(g_1, g_2)} v_j$ holds. Note that cases 2–5 of the merge operation add only these types of constraints to G. No $\mathscr{A}$ can satisfy more than one of the gap-order constraints $v_i = v_j$, $v_i <_g v_j$, and $v_i >_h v_j$ for any $g, h \geqslant 0$. Therefore, the last case in Definition 3.10 cannot occur and the merge cannot fail. Therefore, G exists and $\mathscr{A} \models G$ must be true. □

Proof of Lemma 4.5. Suppose that $A_0(G)$ is derived by a rule application. Then G is consistent by the check in step 4 of the rule application. Let M be the merge of the extensions of $G_1, \ldots, G_k$, and C. By Lemma 3.12. M is consistent and, by Lemma 3.13, $G_1, \ldots, G_k$ and C are also consistent. By Lemma 3.3, they are all acyclic.

Since $G_1, \ldots, G_k$ and C are (l, u)-graphs and every v_i appears in at least one of them, $v_i <_g l$, $v_i = l$, $v_i = u$, $v_i >_g u$, or $l <_g v_i$ and $v_i <_h u$ must be a constraint in at least one of them, for some g and h. Therefore, M will also have one of those constraints for each v_i. Hence, M will be an (l, u)-graph. Moreover, whenever v_i appears in any G_j, then v_i must have the same constraint on it as in M, ignoring gap values, because the five cases are completely disjoint.

Let $v_i <_g v_j$ for some g be an edge on a path from l to u in the compact graph of M. By acyclicity, $v_i <_g l$, $v_i = u$, and $u <_g v_i$ cannot be constraints in M for any g. Since M is an (l, u)-graph, either $l = v_i$ or $l <_g v_i <_h u$ must hold for some g and h. Similarly, either $l <_g v_j <_h u$ or $v_j = u$ must hold for some g and h. Hence, ($l = v_i$ or $l <_g v_i <_h u$) and ($l <_{g'} v_j <_{h'} u$ or $v_j = u$) must be in each G_j that contains v_i and v_j. Therefore, if G_j has a directed edge (v_i, v_j), then it lies on a path from l to u in the compact graph of G_j. By reasoning similarly for each edge on a path from l to u, we sec that they all preserve their gap values in G_j' by Definition 4.2. Every other edge in G_j' loses its gap-value.

The merge operation takes the maximum of the gap values on each edge. Clearly, M and the merge M' have the same set of edges, ignoring gap values. The merge operation takes the maximum of the gap values of each edge (v_i, v_j) from each G_j and C for M and for each G_j' and C' for M'. Therefore, for each edge on a path from l to u in M the gap value will be the same as for the same edge in M', while all other edges in M' will have no gap value. Therefore, M' is the partial graph of M.

The shortcut operation over a vertex y can only shorten paths between any two other vertices, and it can neither create nor destroy a path between them. Hence, any edge (v_i, v_j) that lies on a path from l to u in G must lie on a path from l to u in M or there must be edges (v_i, y) or $v_i = y$ and (y, v_j) or $v_j = y$ in M that lie on a path from l to u. Then the same edges with the same gap values are also in M'. Hence, (v_i, v_j) with the same gap value will be in the shortcut G'' of M' as in G. Therefore, the longest path from l to u is less than $(u - l)$ in G if and only if it is true in G'' if and only if it is true in G' the partial graph of G''. By Lemma 3.3, G is consistent if and only if G'' is consistent if and only if G' is consistent. Hence, $A_0(G)$ is derived by a rule application if and only if $A_0(G')$ is derived by a p-application. □

Acknowledgment

I thank Paris Kanellakis and Pascal Van Hentenryck for helpful comments on an earlier draft of this paper.

References

[1] S. Abiteboul and V. Vianu, Procedural and declarative Database update languages, in: *Proc. 7th ACM Symp. Principles of Database Systems* (1988) 240–250.

[2] A.K. Aylamazyan, M.M. Gilula, A.P. Stolboushkin and G.F. Schwartz, Reduction of the relational model with infinite domain to the case of finite domains, *Proc. USSR Acad. of Science (Doklady)* **286**(2) (1986) 308–311.

[3] E.F. Codd, A relational model for large shared data banks, *Comm. ACM* **13** (1970) 377–387.

[4] A. Colmerauer, An introduction to Prolog III, *Comm. ACM* **28** (1990) 412–418.

[5] A.K. Chandra and D. Harel, Computable queries for relational data bases, *J. Comput. System Sci.* **21** (1980) 156–178.

[6] A.K. Chandra and D. Harel, Horn clause queries and generalizations, *J. Logic Prog.* **2** (1985) 1–15.

[7] J. Chomicki and T. Imielinski, Relational specifications of infinite query answers, in: *Proc. ACM SIGMOD Internat. Conf. on Management of Data* (1989) 174–183.

[8] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier, The Constraint logic programming language CHIP, in: *Proc. 5th Generation Computer Systems*, Tokyo, Japan (1988).

[9] J. Ferrante and J.R. Geiser, An efficient decision procedure for the theory of rational order, *Theoret. Comput. Sci.* **4** (1977) 227–233.

[10] J. Ferrante and C.W. Rackoff, *The Computational Complexity of Logical Theories* (Springer, Berlin, 1979).

[11] R. Hull and J. Su, Domain independence and the relational calculus. Tech. Report 88-64, University of Southern California; *Acta Inform.*, submitted.

[12] N. Immerman, Relational queries computable in polynomial time. *Inform. and Control* **68** (1986) 86–104.

[13] N. Immerman, Nondeterministic space is closed under complement, *SIAM J. Comput.* **17** (1988) 935–938.

[14] J. Jaffar, J.L. Lassez, Constraint logic programming, in: *Proc. 14th ACM Symp. Principles of Programming Languages* (1987) 111–119.

[15] F. Kabanza, J.M. Stevenne and P. Wolper, Handling infinite temporal data, in: *Proc. 9th ACM Symp. on Principles of Database Systems* (1990) 392–403.

[16] P.C. Kanellakis, Elements of relational Database theory, in: J. van Leeuwen, A.R. Meyer, N. Nivat, M.S. Paterson and D. Perrin, eds., *Handbook of Theoretical Computer Science, Vol. B*, Chapter 17 (North-Holland, Amsterdam, 1990).

[17] P.C. Kanellakis, G.M. Kuper and P.Z. Revesz, Constraint query languages, in: *Proc. 9th ACM Symp. on Principles of Database Systems* (1990) 299–313.

[18] P.C. Kanellakis and P.Z. Revesz, On the relationship of congruence closure and unification, *J. Symbolic Comput.* **7** (1989) 427–444.

[19] M. Kifer, On safety, domain independence, and capturability of database queries, in *Proc. Internat. Conf. on Databases and Knowledge Bases* (1988).

[20] S.C. Kleene, General recursive functions on natural numbers, *Math. Ann.* **112** (1936) 727–742.

[21] P. Kolaitis and C.H. Papadimitriou, Why not negation by fixpoint? in: *Proc. 7th ACM Symp. on Principles of Database Systems* (1988) 231–239.

[22] H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation* (Prentice-Hall, Englewood Cliffs, NJ, 1981).

[23] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, 1984).

[24] Y.N. Moschovakis, *Elementary Induction on Abstract Structures* (North-Holland, Amsterdam, 1974).

[25] R. Ramakrishnan, Magic templates: A spellbinding approach to logic programs, in: *Proc. 5th Internat. Conf. on Logic Programming* (1988) 141–159.

[26] P.Z. Revesz, A closed form for Datalog queries with integer order, in: *Proc. 3rd Internat. Conf. on Database Theory* (1990) 187–201.

[27] R. Szelepcsènyi, The method of forced enumeration for nondeterministic automata, *Acta Inform.* **26** (1988) 279–284.

[28] J.D. Ullman, *Principles of Database Systems* (Computer Science press, Rockville, MD, 2nd edn., 1982).

[29] J.D. Ullman and A. Van Gelder, Parallel complexity of logical query programs, *Algorithmica* **3** (1988) 5–42.

[30] P. Van Hentenryck, *Constraint Satisfaction in Logic Programming* (MIT Press, Cambridge, 1989).

[31] M.Y. Vardi, The complexity of relational query languages, in: *Proc. 14th ACM Symp. on Theory of Computing* (1982) 137–146.