PERGAMON

Information Systems 27 (2002) 321–343

www.elsevier.com/locate/infosys

Pushing extrema aggregates to optimize logic queries[☆]

Filippo Furfaro[a,*], Sergio Greco[a] , Sumit Ganguly[b] , Carlo Zaniolo[c]

[a] *Dipto Elettronica Informatica e Sistemistica, Università della Calabria, ISI-CNR, Via P. Bucci 41 C, 87030 Rende-Cosenza, Italy*

[b] *Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur 208016, India*

[c] *Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90095, USA*

Received 5 March 2001; received in revised form 31 July 2001; accepted 5 December 2001

Abstract

In this paper, we explore the possibility of transforming queries with *minimum* and *maximum* predicates into equivalent queries that can be computed more efficiently. The main contribution of the paper is an algorithm for propagating min and max predicates while preserving query equivalence under certain monotonicity constraints. We show that the algorithm is correct and that the transformed query is often safe when the original one is not. Although in this paper we use logic rules, the technique presented can be used to optimize (possibly recursive) queries expressed using SQL3. © 2002 Published by Elsevier Science Ltd.

Keywords: Databases; Query optimization; Aggregates

1. Introduction

The rising demand for applications which support the decisional process and perform reporting has led to increasing interest in the problem of efficiently computing aggregate queries, which are widely used in such systems [1,2]. Thus, due to its practical importance, the study of logics and declarative languages with aggregates has received significant attention in the literature [3–8]. Most of the research has concentrated on the study of declarative programs containing recursive predicates with monotonic aggregates [9–11]. These works pursue the general objective of ensuring the existence of formal semantics and achieving an efficient computation for such programs [9,10,12–15].

The predicates *min* and *max* allow a declarative specification of queries but their computation is often not efficient. Here we consider queries containing *min* or *max* constructs, such as the one which searches for the minimum path connecting two nodes in a weighted graph, and propose a technique for transforming them

☆A preliminary version of this paper appeared in [4]. This work has been partially supported by MURST grants under the projects 'Data-X' and 'D2I', and by NSF under grant: IIS 0070135.

*Corresponding author.

E-mail addresses: furfaro@si.deis.unical.it (F. Furfaro), greco@si.deis.unical.it (S. Greco), sganguly@iitk.ac.in (S. Ganguly), zaniolo@cs.ucla.edu (C. Zaniolo).

into equivalent queries that can be computed more efficiently. Example 1, below, presents a query containing a *minimum* predicate.

Example 1. *Stratified shortest path.* Given a weighted directed graph represented as a base relation arc, where all edge weights are nonnegative, the predicate path computes the set of all triples (x, y, c) such that there is a path from node x to node y whose cost is c. The predicate sh_path yields all the triples (x, y, c) such that c is the least cost among all paths from node x to node y.

```
sh_path(X, Y, C) ← min(C, (X, Y), path(X, Y, C)).
path(X, Y, C) ← arc(X, Y, C).
path(X, Y, C) ← path(X, Z, C1), arc(Z, Y, C2), C = C1 + C2.
```

In the above example, the meaning of the min predicate min(C,(X,Y),path(X,Y,C)) is to select among the tuples of path with the same values of X and Y the ones with the least value for the attribute C. This predicate has second order syntax but its semantics is first order, and a precise semantics can be assigned to our program, by simply viewing the rules containing the min predicate as a shortcut for rules with negated body literals. For instance, the first rule is a shorthand for the following rules:

```
sh_path(X, Y, C) ←          path(X, Y, C),   ¬a_lesser_path(X, Y, C).
a_lesser_path(X, Y, C) ←    path(X, Y, C),   path(X, Y, C′),C′ < C.
```

where a_lesser_path is a new predicate symbol not appearing elsewhere in the program. This has formal semantics because, by rewriting the min predicates by means of negation, we get a stratified program [16,17]. However, a straightforward evaluation of such a stratified program would materialize the predicate path and then choose the smallest cost tuple for every pair of nodes x and y. There are two problems with this approach: first, it is very inefficient, and second, the computation could be non-terminating if the relation arc is cyclic and the domain of the cost attribute is infinite (if the cost domain is finite the complexity of the computation depends on the dimension of the domain).

To solve these problems we begin by observing that all minimum paths of length $n+1$ can be generated from the minimum paths of length n, since the weights are nonnegative. Thus, the min predicate can be pushed into recursion, in such a way that the generation of new paths is interwoven with the computation of shortest paths, yielding the program of Example 2.

Example 2. *Unstratified shortest path.*

```
path(X, Y, C) ← arc(X, Y, C).
path(X, Y, C) ← sh_path(X, Z, C1), arc(Z, Y, C2), C = C1+C2.
sh_path(X, Y, C) ← min(C, (X, Y), path(X, Y, C)).
```

Unfortunately, as we attempt to give a meaning to Example 2, by rewriting it using negation, as in the previous example, we obtain a non-stratified program, with all the accompanying semantics and computational problems: the answer (under the well-founded semantics) could contain undefined atoms and the computation could be inefficient. [3,8,9,11,13,18–24].

In this paper, we consider the problem of taking a program such as that of Example 1, where a *min* predicate is given as a post-condition on a recursive graph computation, and transform the program into an equivalent one where the *min* predicate is pushed into the recursive rules. Once a query is rewritten, as in

Example 2, the greedy fixpoint procedure [9] can be used to compute the transformed program efficiently. Thus, we allow the user to write stratified queries which have clear and intuitive semantics [25] but are expensive to answer, and leave the system the task of efficiently computing the queries by rewriting them. We concentrate on the class of cost-monotonic programs which, because of their syntactic structure and stratification conditions induced by the underlying cost domain, have a total well-founded model. Although our discussion deals explicitly with min programs only, the symmetric properties of *max* programs follow by duality.

We point out that we use Datalog to present how min and max aggregates are propagated down into queries for the sake of simplicity of presentation. As shown in the following example, the technique presented here can also be used to optimize (recursive) queries expressed by means of SQL3 [26,27]. For instance, assuming that the schema for the relation `arc` is `Arc(Source,Dest,Cost)`, the query of Example 1 can be expressed in SQL3 as follows:

```
CREATE RECURSIVE VIEW Path(Source,Dest,Cost) AS
( SELECT Source, Dest, Cost
  FROM   Arc
  UNION
  SELECT Path.Source, Arc.Dest, Path.Cost + Arc.Cost AS Cost
  FROM   Path, Arc
  WHERE  Path.Dest = Arc.Source
)
SELECT Source, Dest, MIN(Cost)
FROM   Path
GROUP BY Source, Dest
```

whereas an SQL3-like[1] query equivalent to the query of Example 2 can be expressed as follows:

```
CREATE RECURSIVE VIEW Path(Source,Dest,Cost) AS

( SELECT Source, Dest, MIN(Cost) AS Cost
  FROM     Arc
  GROUP BY Source, Dest
  UNION
  SELECT   Path.Source, Arc.Dest, MIN(Path.Cost + Arc.Cost) AS Cost
  FROM     Path, Arc
  WHERE    Path.Dest = Arc.Source
  GROUP BY Path.Source, Arc.Dest
)
SELECT     Source, Dest, MIN(Cost)
FROM       Path
GROUP BY Source, Dest.
```

The rest of the paper is organized as follows. In the next section, we review the syntax, the semantics of min programs and the notion of monotonic min queries. In Section 3, we review the computation of min queries. In Section 4, we present a technique for the rewriting of queries into equivalent ones where the min predicate is pushed into the recursive rules. In Section 5, we present our conclusions and suggestions to extend the technique to larger classes of queries.

[1] SQL3 queries must be aggregate stratified.

2. Basic definitions

2.1. Datalog

We assume finite countable sets of constants, variables and predicate symbols. A (simple) *term* is either a constant or a variable. A (standard) *atom* is of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. A *literal* is an atom A or its negation $\neg A$. A *rule* is of the form $A \leftarrow B_1, \ldots, B_n$, where A (called head) is an atom and $B_1, \ldots, B_n$ (called body) is a conjunction of literals. A ground rule with an empty body is called a *fact*. A (Datalog) program is a finite set of rules.

Given a program P, the Herbrand universe for P, denoted H_P, is the set of all constants occurring in P. The Herbrand Base of P, denoted B_P, is the set of all ground atoms whose predicate symbols occur in P and whose arguments are elements from the Herbrand universe. A *ground instance* of a rule r in P is a rule obtained from r by replacing every variable X in r by a constant in H_P. The set of ground instances of r is denoted by *ground*(r); accordingly, *ground*(P) denotes $\bigcup_{r \in P}$ *ground*(r). A ground atom is called tuple of fact.

Given a literal A, $\neg\neg A$ denotes A. Let I be a set of ground literals; then $\neg I$ denotes the set $\{\neg A \mid A \in I\}$, and I^+ (resp., I^-) denotes the set of all literals (resp., negated atoms) in I. Given a Datalog program P, we denote with $\bar{I} = B_P - (I^+ \cup \neg I^-)$ the set of facts in the Herbrand base which are undefined in the interpretation I. I is a (*partial*) *interpretation* of P if it is *consistent*, i.e., $I^+ \cap \neg I^- = \emptyset$. Moreover, if $I^+ \cup \neg I^- = B_P$, the interpretation I is called *total*.

Let I be an interpretation for a program P, then the truth value of an atom $A \in B_P$ with respect to interpretation I, denoted by $I(A)$, is equal to (i) *true* if $A \in I$, (ii) *false* if $\neg A \in I$ and *undefined* otherwise, i.e., $A \in \bar{I}$. We assume the linear order $\mathit{false} < \mathit{undefined} < \mathit{true}$ and $\neg \mathit{undefined} = \mathit{undefined}$.

A rule $A \leftarrow A_1, \ldots, A_m$ in *ground*(P) is satisfied w.r.t. an interpretation I if $I(A) \geqslant \mathit{min}\{I(A_i) \mid 1 \leqslant i \leqslant m\}$. An interpretation I is a model if all rules in *ground*(P) are satisfied. The semantics of logic programs is given in terms of partial stable model semantics [28] which we briefly recall next.

An interpretation M of P is a *P-stable* (*partial stable*) model if it is the minimal model of the positive program P^M obtained from *ground*(P) by replacing each negated body literal $\neg A$ with the complement of the truth value of A w.r.t. M. A P-stable model M of P is (i) *T-stable* (*total stable*) if it is a total interpretation of P, and (ii) *well-founded* if it is the intersection of all P-stable models of P. T-stable model was the first notion of stable model and was defined in [28]; existence of a T-stable model for any program is not guaranteed. It is well known that every program admits a unique well-founded model which can be computed in polynomial time. Positive programs have a total well founded model (and consequently a unique total stable model) which coincides with the minimum model [28].

Let I be an interpretation for a program P. The *immediate consequence operator* $T_P(I)$ is defined as the set containing the heads of each rule $r \in$ *ground*(P) s.t. the body of r is true in I. The semantics of a *positive* (i.e. negation-free) program P is given by the unique minimal model; this minimum model coincides with the least fixpoint $T_P^{\infty}(\emptyset)$ of T_P [29].

The *dependency graph* G_P of a program P is a directed graph whose nodes are predicate symbols in P. There is an arc from a node q to a node p if there is a rule r in P such that a q-atom appears in the body and a p-atom appears in the head. Moreover, an arc from q to p is labelled with $\neg$ if q appears negated in r.

Given two predicate symbols p and q in a program P, we say that p *depends on* q if there is a path from q to p in G_P. The maximal strong components of G_P will be called *recursive components*. Predicates in the same recursive component are *mutually recursive*. A rule is *recursive* if its head predicate symbol is mutually recursive with some predicate symbol occurring in the rule body. A maximal set of rules of a program P whose head predicate symbols belong to the same recursive component of the graph G_P is called sub-program of P. Sub-programs can be (partially) ordered on the base of the dependencies among predicate symbols.

A program P is *stratified* if G_P does not contain cycles with marked arcs [16]. Stratified programs have a total well-founded model which coincides with the unique stable model; this model is also called *perfect model* or *stratified model* [30]. The perfect model of a stratified program can be computed by partitioning the program into an ordered number of suitable subprograms (called 'strata') and computing the fixpoints of every stratum from the lowest one up [16,31,32].

Locally stratified programs are defined in an analogous way by considering the dependency graph $G_{ground(P)}$ of $ground(P)$. $G_{ground(P)}$ is constructed as follows: for each ground atom in $ground(P)$ there is a node in $G_{ground(P)}$ and there is an arc from the atom q to the atom p if there is a rule r in $ground(P)$ with p as head and q appear in the body; moreover, an arc from q to p is labelled with $\neg$ if q appears negated in r [30]. Locally stratified programs also have a total well-founded model (also called perfect model [30]) which coincides with the unique total stable model.

In general, the predicate symbols of a program can be partitioned into *extensional* and *intensional* predicates (called EDB and IDB predicates, respectively). EDB predicates never occur in the rule heads as they are assumed to be defined by a number of ground facts stored in some database D. A (Datalog) *query* Q is a pair $\langle g(X), P\rangle$ where P is a (Datalog) program, g is a predicate symbol in P, with arity $n \geqslant 0$, and X is a list of n variables; the atom $g(X)$ is said to be the query-goal. Given a database D, the *answer* to Q on D, denoted by $Q(D)$, is the set of relations on g denoted as $A_g = \{M(g) \mid M$ is a model defining the semantics of $P \cup D\}$.[2] We say that two queries Q_1 and Q_2 are equivalent ($Q_1 \equiv Q_2$) if for each database D the answers to Q_1 and Q_2 on D are the same.

2.2. Min queries

The notion of a minimum naturally assumes the existence of a domain of constants over which a total order is defined. Formally, we assume the existence of an alphabet of constants, functions and predicate symbols. We assume also that there are a built-in set of constants K, called *cost domain*, two built-in binary predicates $<$ and $\leqslant$, and built-in functions whose arguments are elements of K. In our examples, we will use the cost domain of real numbers, where the built-in predicates $<$ and $\leqslant$ are the usual operators defined for real numbers and the built-in functions are the usual arithmetic operators ($+, -, *$, etc.).

Definition 1. A *special atom* is of the form $min(C, S, Q)$ where: (1) Q is a standard atom of first order logic, called *minimized atom*, (2) S is a set of variables, appearing as arguments in Q, called *grouping variables*, (3) $C \notin S$ is a variable, appearing as an argument of Q, called the *cost variable*. Cost variables can only take values from the cost domain.

A *min atom* is either a special atom or a standard atom. A *min rule* is of the form $A \leftarrow B_1, \ldots, B_n$ where A is a standard atom and $B_1, \ldots, B_n$ are min atoms. A (*min*) *program* is defined by a set of min rules P, its underlying cost domain K_P plus an additional constraint defining the subrange of the cost domain whose values can be assigned to cost arguments. We will call such a subrange the *valid cost sub-domain*. Both examples 1 and 2 are min programs. A (*min*) *query* Q is a pair $\langle g(X), P\rangle$ where P is a min program, $g(X)$ is a standard atom.

For the sake of simplicity, we consider min programs without negation. The extension to programs with stratified negation is straightforward. Without loss of generality, we assume that min rules are either standard Horn rules or non-Horn rules of the form $H \leftarrow min(C, S, Q)$ where H and Q are standard atoms.

The semantics of a min program is defined by taking a min program P and defining a first order formula *foe*(P), called the *first order extension* of P, obtained by replacing every occurrence of special atoms $min(C, S, q(\bar{Y}, C))$ by the pair of goals $q(\bar{Y}, C)$, $\neg a\_lesser\_q(\bar{Y}, C)$. The distinguished *a_lesser_q* predicate is

[2] With a little abuse of notation D denotes here the set $\{p(t) \mid t \in \text{relation } p \text{ of } D\}$.

defined by the additional rule of the form:

$$a\_lesser\_q(\bar{Y}, C) \leftarrow q(\bar{Y}, C),\quad q(\bar{X}, C'),\ \ C' < C,$$

where an argument variable in $\bar{X}$ coincides with the corresponding variable in $\bar{Y}$ if this variable appears in the grouping set S and is a new variable otherwise. The new variable C' corresponds to the variable C.

Given a min program P and an interpretation M for $foe(P)$, M_P denotes the set of atoms in M whose predicate symbols appear in P, i.e. M_P is derived from M by deleting atoms defining predicate symbols introduced in the rewriting of P into $foe(P)$. Since we view a min program simply as a shorthand of $foe(P)$, we will say that if a set of atoms M is an interpretation (resp. a (minimal) model) for $foe(P)$, then M_P is an interpretation (resp. a (minimal) model) for P. Therefore, we can define the semantics of min programs in terms of rewritten programs, using concepts such as stratification, well founded model and stable models, developed for Datalog with negation.

In order to guarantee query equivalence between the source and the rewritten programs, we consider classes of min programs such that the well-founded model of the rewritten program is total (and, consequently, there is a unique stable model) [32,33]. In this paper we concentrate on the class of cost monotonic programs which, because of their syntactic structure, define a (local) stratification of the ground program and, therefore, have a total well-founded model. This model can be computed efficiently, using the greedy fixpoint procedure introduced in [9] (see Section 3). However, our technique can also be used for classes of weakly stratified programs [31] such as the modularly stratified class of [34] or the XY-stratified class [35].

We will assume that certain database predicates arguments called *cost arguments* can only take values from the cost domain, or a subset of this domain called a *valid cost sub-domain*. In most examples of this paper, we will assume that the set of non-negative numbers is our valid cost sub-domain. The presence of database goals with cost arguments, and the fact that built-in predicates can be true only if their arguments are from the cost domain, imply that certain arguments in the derived predicates also belong to the cost domain. We point out that the introduction of the valid cost domain is motivated by the fact that programs satisfy some particular properties only if we consider a subset of the cost domain.

A *valid instantiation* of a rule r in a program $foe(P)$ is a ground instance of r that is obtained by replacing all the variables of r with ground terms from $H_{foe(P)} \cup K'_P$ where K'_P is a valid cost sub-domain, and such that each goal corresponding to a built-in predicate is true. Hence, valid instantiations will not contain a goal such as $3 > 4 + 1$ which is always false and thus inconsequential. The valid instantiation of program $foe(P)$ is simply the set of all valid instantiations of all rules of $foe(P)$. The *valid dependency graph* of a (min) program P, denoted $G_{ground(P)}$, is a directed graph whose nodes are ground atoms of $foe(P)$. G_P contains an arc from a node A to another node B if there is a valid instantiation of some rule r in $foe(P)$ such that A appears in the body and B appears in the head of r; moreover if A appears negated then the arc is marked, and will be said to be negative.

We assume that predicates symbols are partitioned into two sets: *standard predicates* and *cost predicates*; cost predicates have a particular argument, called *cost argument*, which takes values from the valid cost sub-domain. Atoms whose predicate symbol is a cost predicate are called cost atoms. In the following, the last argument of a cost atom denotes the cost argument. Thus, given a ground cost atom $A = p(t_1, \ldots, t_n, c)$, then the cost value of A, denoted by $cost(A)$, is c. Moreover, we say that two ground cost atoms $p(t_1, \ldots, t_n, c_1)$ and $p(u_1, \ldots, u_n, c_2)$ are *contradicting* on the cost attribute if $t_i = u_i$ for all $i \in [1..n]$ and $c_1 \neq c_2$.

Thus, given a min program P it is possible to divide the predicate symbols appearing in P into the two distinguished sets of cost predicates and standard predicates. We will only consider programs where each recursive set of predicates consists of cost predicates (called *min sub-program*) or standard predicates (called *standard sub-program*). Predicates that appear inside special atoms will be called *minimized predicates*.

3. Cost monotonic programs and fixpoint computation

As mentioned in the previous section, we concentrate on the class of cost monotonic programs which are locally stratified w.r.t. the cost argument and which can be computed very efficiently using the greedy fixpoint algorithm defined in [9].

3.1. Cost monotonic programs

The following definitions characterize such a class of programs.

Definition 2. Let P be a min program with valid dependency graph G_P. P is said to be *cost monotonic* if for every recursive cost predicate symbol q and for each pair of ground atoms $\langle A = q(t_1, \ldots, t_n),\ B = q(u_1, \ldots, u_n) \rangle$, the following two conditions hold:

(1) if there exists a path from A to B in $G_{ground(P)}$, then there exists $i \in [1..n]$ such that $t_i \leqslant u_i$,
(2) if such a path contains a negated arc, then $t_i < u_i$.

It follows from the above definition that in order to determine whether a program is cost monotonic it suffices to find only one argument satisfying the expected property among the several possible ones of a recursive predicate. Consider for instance the following example:

Example 3. *Modified shortest path.*

```
p(X,Y,C) ← arc(X,Y,C).
p(X,Y,C) ← sp(X,Z,C₁), arc(Z,Y,C₂), C = C₁+C₂−2.

sp(X,Y,K) ← min(C,(X,Y), p(X,Y,C)), K = C + 1.
```

The database predicate `arc(X,Y,C)` describes a set of arcs and their costs. The third argument of `arc` is a number representing the edge weight, and so the third argument of `p` and the third argument of `sp` are cost arguments. Such a program is cost monotonic if we restrict the valid cost sub-domain to real numbers > 1. Indeed, by assuming that the weight of arcs is > 1, for each pair of tuples $t_1 = \mathtt{p}(x_1, y_1, c_1)$ and $t_2 = \mathtt{p}(x_2, y_2, c_2)$ (resp. $t_1 = \mathtt{sp}(x_1, y_1, c_1)$ and $t_2 = \mathtt{sp}(x_2, y_2, c_2)$) such that t_1 depends on t_2, we have that $c_1 > c_2$ whereas for weights $\leqslant 1$ the program is not monotonic since we have $c_1 \leqslant c_2$.

Theorem 1 (Ganguly et al. [9]). *Every cost monotonic min program is locally stratified.*

The main consequence of the above theorem is that cost monotonic programs always have a total well-founded model which coincides with the perfect model. Although the problem of determining whether a program is monotonic is undecidable [9], simple sufficient conditions can be given that are general enough to deal with the common situations of practical interest. We now define the notion of *uniformly cost monotonic* program using cost graphs.

The *cost graph* CG_P of a program P is a directed graph whose nodes are cost predicate symbols in P. There is an arc from a node q to a node p with label r if there is a rule r in P such that a q-atom appears in the body and a p-atom appears in the head. Moreover, an arc from q to p labelled with rule r is marked with $\geqslant$ if the arc is derived from a Horn rule and the value of the cost argument of p is $\geqslant$ the value of the cost argument of q for every ground interpretation of r. If the arc is derived from a non-Horn rule then it is marked with *min* and is called *min arc*. The cost graph for the program of Example 2 contains an arc from *path* to *sh_path* marked with r_3/min and an arc from *sh_path* to *path* marked with $r_2/\geqslant$.

Definition 3. Let P be a min program with cost graph CG_P and let SC be a strong component in CG_P containing cost predicates and let SP be the min sub-program whose head predicate symbols are in SC. We say that SP is *uniformly cost monotonic* if

(1) all arcs connecting nodes of SC are marked with *min* or $\geqslant$, and
(2) each cycle contains at least one arc marked with $\geqslant$.

Moreover, we say that P is *uniformly cost monotonic* if all min sub-programs are uniformly cost monotonic.

Observe that for a given min program P, the cost monotonic property is defined on the (dependency graph of the) rewritten program *foe*(P) whereas the uniformly cost monotonic property is defined directly on P.

Theorem 2 (Ganguly et al. [9]). *Uniformly cost monotonic min programs are cost monotonic.*

The program of Example 2 under the cost domain of positive real numbers is uniformly cost monotonic. The program of Example 3 is monotonic but not uniformly cost monotonic if we restrict the valid cost sub-domain to real numbers greater than 1 (for numbers <2, the condition $C \geqslant C_1$ does not hold); however, it becomes uniformly cost monotonic if we restrict the valid cost domain to real numbers $\geqslant 2$.

Proposition 1. *Let P be a min program. Checking if P is uniformly cost monotonic can be done in polynomial time.*

Proof. The check can be made by (i) constructing the cost graph CG_P of P, and (ii) checking if there is a strong component of CG_P containing a cycle without arcs marked with $\geqslant$. Clearly, the construction of the cost graph is polynomial. To verify condition (ii) it is sufficient to remove from the graph CG_P the arcs marked with $\geqslant$ and check if the graph is still cyclic.

In practice, the construction of the cost graph would begin by drawing the dependencies between recursive predicate names in strong components involving cost predicates. In our example, we have two nodes, namely *path* and *sh_path*. From the second rule we derive the arc from *sh_path* to *path*, that is labelled with $r_2/\geqslant$ since the we have $cost(path(X, Y, C)) \geqslant cost(sh\_path(X, Y, C))$ for all possible instantiations of the rule. From the third rule we derive the arc from *path* to *sh_path* labelled with r_3/min since it is derived from a special rule. It is reasonable to expect that an intelligent compiler will be able to perform this simple analysis on arithmetic expressions. But, the introduction of an explicit additional goal $C \geqslant C1$ would make the monotonic cost constraint detectable by even a very unsophisticated compiler.

3.2. Fixpoint computation

We now recall how the intended model M_P of a min program P is computed. We assume that P is partitioned into a partially ordered set of sub-programs which are computed following the topological order among predicates symbols: standard sub-programs are computed using the standard fixpoint operator T_P whereas min sub-programs are computed using the greedy fixpoint operator U_P below defined.

The clauses of a min (sub-)program P can be partitioned into (1) the set of Horn Clauses H and (2) the set of non-Horn Clauses N. The immediate consequence operators for H, N and P denoted, respectively, by T_H, T_N and T_P are defined in the usual way [31,32].

Under the restriction of uniform monotonic cost, any arbitrary min program can be evaluated efficiently using a procedure derived by combining the procedure described in [36] with the no-backtracking improvement which follows from Dijkstra's algorithm as shown in [9]. This 'new' procedure, called *greedy fixpoint*, consists of applying a modified immediate consequence operator U_P until saturation to produce the unique stable model of the program [28]. The operator U_P takes two arguments: the first argument I is used to build up the stable model, and the second argument L is an auxiliary set that contains the set of least elements computed so far. The operator U_P, basically, alternates between two phases: in the first phase all the Horn clauses are fired (V_P), while, in the second phase the new minima are collected and the non-Horn clauses are fired (G_P). The operator G_P uses a $least(I, L)$ operator which returns the set of facts in I that are minimal in cost and do not contradict any fact that is already in L w.r.t. the arguments specified by the grouping variables.

Definition 4. *Greedy alternating operator.* Let P be a min program, the three operators G_P, U_P and V_P from $2^{B_P} \times 2^{B_P} \rightarrow 2^{B_P} \times 2^{B_P}$ are defined as follows:

$$V_P(I, L) = (I \cup T_H(I), L),$$

$$G_P(I, L) = (I \cup T_N(A), L \cup A),$$

$$U_P(I, L) = G_P(V_P(I, L)),$$

where $A = least(I, L)$

Intuitively, the set L contains the set of 'smallest' tuples that were produced so far. The basic step in the greedy operator is to choose a tuple from I that (1) does not contradict existing minima in L and (2) is the smallest among all such elements. This tuple is then used to fire the non-Horn rules (see Example 4). U_P^n is defined for every finite n as follows: $U_P^0 = (\emptyset, \emptyset)$ and $U_P^n = U_P^n(\emptyset, \emptyset) = U_P(U_P^{n-1})$ if $n > 0$. For $n = \infty$, U_P^∞ is defined as $\bigcup_{n\, finite} U_P^n$. If A is a ground atom with a cost attribute then $cost(A)$ denotes the value of this cost attribute. If S is a set of ground atoms then $cost(S)$ denotes the largest value of the set $\{cost(A) \mid A \in S\}$.

The following theorem state the soundness and completeness of greedy fixpoint procedure.

Theorem 3 (Ganguly et al. [9]). *Let P be a uniformly cost monotonic min program such that $U_P^\infty(\emptyset, \emptyset) = (M, L)$ then M is the unique stable model of $foe(P)$.*

The U_P operator generates tuples by alternating between the Horn clauses and the non-Horn clauses. It fires the Horn rules once using T_H and uses the greedy idea to obtain tuples using T_N. The strict alternation between V_P and G_P is not entirely necessary, since the following operator U'_P also computes the intended model for any countable ordinal $n > 0$ [9]:

$$U'_P(I, L) = G_P(V_P^n(I, L)).$$

In particular, if $n = \infty$ then the operator reduces to the operator used to construct the stable model in [36] (except that backtracking is never needed). In the shortest path query of Example 2 the above observation is quite trivial because $T_H = T_H^\infty$ implying that $V_P^n = V_P$ for all countable ordinals $n > 0$. The following example presents a trace of the greedy fixpoint procedure on an example program.

Example 4. Consider the following program computing the shortest distance of the nodes in a given graph from the source node a:

```
arc(a, b, 1).
arc(b, c, 1).
arc(a, c, 3).
arc(c, d, 2).
```

$\mathtt{path(a,0).}$

$\mathtt{path(Y,C) \leftarrow spath(X,C_1),\ arc(X,Y,C_2),\ C = C_1{+}C_2.}$

$\mathtt{spath(X,C) \leftarrow min(C,(X),path(X,C)).}$

Here the graph is represented by the database D consisting of the single relation *arc*. Let us step through the computations of the U_P operator. Initially $I = \emptyset$ and $L = \emptyset$. In the first step, after firing the Horn rules we obtain $I = D \cup \{path(a,0)\}$ with L unchanged. We now wish to find the set of tuples of I that do not contradict minima in L and also do not belong to L. Since L is empty, clearly this is $\{path(a,0)\}$. We now choose the least element from this set, include it in L and use it to fire the non-Horn rule to obtain the tuple $spath(a,0)$. Thus $I = D \cup \{path(a,0), spath(a,0)\}$ and $L = \{path(a,0)\}$ after the completion of the first iteration.

We now return to firing the Horn rules using the V_P operator and obtain $L = \{path(a,0)\}$ and $I = D \cup \{path(a,0), path(b,1), path(c,3), spath(a,0)\}$. At this point we have to find the set of tuples whose minima are not contradicted by tuples in L. These tuples are $spath(a,0)$, $path(b,1)$ and $path(c,3)$. We choose the least of these tuples, and obtain $A = \{spath(a,0)\}$, then add A to L. This set is also used to fire the non-Horn rule. Hence at the end of the second iteration, the values for I and L are as follows: $I = D \cup \{path(a,0), path(b,1), path(c,3), spath(a,0)\}$ and $L = \{path(a,0), spath(a,0)\}$.

In the next step we fire the Horn rules again but do not obtain new tuples. Then we find the set of tuples in I whose minima are not contradicted by tuples in L and this set is $A = \{path(b,1)\}$. Therefore, the tuple $path(b,1)$ is added to L and the non-Horn rule is fired. Hence, at the end of the third iteration, we have $I = D \cup \{path(a,0), path(b,1), path(c,3), spath(a,0), spath(b,1)\}$ and $L = \{path(a,0), spath(a,0), path(b,1)\}$.

In the fourth step, from the firing of the Horn rules, the tuple $path(c,2)$ is inferred and the tuple $spath(b,1)$ is added to L; from the firing of the non-Horn rule no new tuples are derived. Therefore, at the end of the step we have $I = D \cup \{path(a,0), path(b,1), path(c,3), path(c,2), spath(a,0), spath(b,1)\}$ and $L = \{path(a,0), spath(a,0), path(b,1), spath(b,1)\}$.

In the next step from the firing of the Horn rule no new tuples are derived, the tuple $path(c,2)$ is added to L and the tuple $spath(c,2)$ is derived from the firing of the non-Horn rule.

At the end of the process we get the following sets: $I = D \cup \{path(a,0),\ path(b,1),\ path(c,3),\ path(c,2),\ path(d,4),\ path(d,5),\ spath(a,0),\ spath(b,1),\ spath(c,2),\ spath(d,4)\}$ and $L = \{path(a,0),\ path(b,1),\ path(c,2),\ path(d,4),\ spath(a,0),\ spath(b,1),\ spath(c,2),\ spath(d,4)\}$. The tuples $path(c,3)$ and $path(d,5)$ are not added to L because they contradict, respectively, the tuples $path(c,2)$ and $path(d,4)$ already present in L.

The Greedy fixpoint algorithm can be improved in a way which is similar to the improvement of the semi-naive over the naive computation [37]. Moreover, the computation of finding the set of tuples that do not contradict the minima already in L can be made much more efficient by using appropriate data structures (i.e., the data structure heap can be used to store the elements in $I - L$). After making these optimizations it can be shown that the evaluation of the above program mimics Dijkstra's algorithm for shortest paths (having a as source node) and therefore has the same complexity.

4. Propagation of min predicates

In this section, we present an algorithm for transforming monotonic min programs into *query-equivalent* programs which can be more efficiently implemented using simple variations of the semi-naive algorithm such as the semi-naive greedy operator G_P.

Let S and T be sets of natural numbers. Then, $min(S \cup T) = min(\{min(S), min(T)\})$. If S and T are extension of predicates p and q then we have $min(p \vee q) = min(min(p) \vee min(q))$. The minimum of a disjunction of predicates is the minimum of the disjunction of the minimum of each of the individual predicates. This is the basic idea behind propagation. In turn, if p and q are defined in terms of other predicates then the above step could be carried further.

Therefore, in this section we design an algorithm for propagating min predicates while preserving query equivalence under certain monotonicity constraints. We first present an example and then develop the theory for propagation.

Example 5. Consider a program P for evaluating the shortest path.

```
sp(X, Y, C) ← min(C, (X), p(X, Y, C)).

p(X, Y, C) ← arc(X, Y, C).
p(X, Y, C) ← p(X, Z, C1),  p(X, Y, C2),  C = C1 + C2.
```

Suppose the query goal is of the form `sp(X,Y,C)`. If we try to evaluate this using the greedy algorithm we face the problem of having to compute the predicate `p(X,Y,C)` first. However, it is clear that tuples in `sp` are members of the predicate `cp` which is defined as below:

```
cp(X, Y, C) ← min(C, (X), arc(X, Y, C)).
cp(X, Y, C) ← min(C1, (X), p(X, Z, C1)),  min(C2, (X), p(X, Y, C2)),  C = C1 + C2.
```

The second rule of `cp` can be replaced by

```
cp(X, Y, C) ← sp(X, Z, C1),  sp(X, Y, C2)),  C = C1 + C2.
```

Moreover, in order to rewrite our programs, instead of propagating min predicates (as in the above example), we associate adornments to minimized atoms and propagate such adornments in a similar way to the magic-set rewriting technique [38].

As in the magic-set method, adornments specify the role played by variables. In particular, we associate adornments to cost predicates by assigning to each of their arguments one of the three labels u, e, or m as follows:

(1) the cost argument may be labelled m, denoting that the predicate is 'minimized';
(2) an argument not contained in the set of grouping variables is labelled e, denoting that the corresponding variable is existentially quantified;
(3) an argument appearing in the set of grouping variables is labelled u; denoting that the corresponding variable is universally quantified.

Consider, for instance, the min atom $min(C, (X, Y), p(X, Z, Y, C))$. The adorned atom associated with the minimized atom is $p^{ueum}(X, Z, Y, C)$. Predicates that are not minimized will then be assigned the default adornment ε. For instance, in Example 1, the adornment for *sh_path* is ε whereas the adornment for *path* is *uum*.

Thus, we use the concept of *adornments* to abbreviate minimized predicates. The basic component of the transformation algorithm is the propagation of an adornment from the head of a rule into the body of the rule. We discuss this for Horn rules and non-Horn rules next.

4.1. Propagating adornments into horn rules

Consider the following rule for which we wish to propagate the adornment *uum* for `p` into the body.

$$p(X, Z, C) \leftarrow q(X, Y, C_1),\ s(Y, Z, C_2),\ C = C_1 + C_2.$$

In this case, a particular tuple p(X,Z,C) is a *possible* candidate member of $\mathrm{p}^{\mathrm{uum}}$ provided that the conjunction $\mathrm{q}^{\mathrm{uum}}(\mathrm{X},\mathrm{Y},\mathrm{C}_1)$, $\mathrm{s}^{\mathrm{uum}}(\mathrm{Y},\mathrm{Z},\mathrm{C}_2)$ is true. Clearly, this condition is necessary but not sufficient for two reasons: (i) the combination of two tuples from $\mathrm{q}^{\mathrm{uum}}$ and $\mathrm{s}^{\mathrm{uum}}$ may give a tuple whose cost could not be minimum with respect to a fixed adornment, and (ii) the predicate p can appear in the head of some other rule. This motivates us to introduce a new *surrogate* predicate $\mathrm{cp}^{\mathrm{uum}}$ (the name for the surrogate predicate symbol is obtained by prefixing *c* to the name of the predicate and by superscripting the adornment) whose definition is intended to collect the set of candidate minimum tuples for the given adornment. Thus, the propagation of the adornment *uum* in the above rule results in the following rule:

$$\mathrm{cp}^{\mathrm{uum}}(\mathrm{X},\mathrm{Z},\mathrm{C}) \leftarrow \mathrm{q}^{\mathrm{uum}}(\mathrm{X},\mathrm{Y},\mathrm{C}_1),\ \mathrm{s}^{\mathrm{uum}}(\mathrm{Y},\mathrm{Z},\mathrm{C}_2),\ \mathrm{C} = \mathrm{C}_1 + \mathrm{C}_2.$$

If there are multiple rules that define the predicate p, we generate one of such adorned rules for each rule of the 'original' program. Consequently, we define $\mathrm{p}^{\mathrm{uum}}$ as the minimum over $\mathrm{cp}^{\mathrm{uum}}$ as follows:

$$\mathrm{p}^{\mathrm{uum}}(\mathrm{X},\mathrm{Z},\mathrm{C}) \leftarrow \mathrm{min}(\mathrm{C},(\mathrm{X},\mathrm{Z}),\ \mathrm{cp}^{\mathrm{uum}}(\mathrm{X},\mathrm{Z},\mathrm{C})).$$

The above rule is called the minima definition of p, in terms of the surrogate predicate $\mathrm{cp}^{\mathrm{uum}}$ for the adornment *uum*. In this manner, we generate the adorned definition of a predicate for a given adornment, which is the union of the adorned rules for the predicate and the minima definition of the predicate in terms of the surrogate predicate for that adornment.

4.1.1. Adorning variables by u

In the above example, the variable Y is shared by both $\mathrm{q}(\mathrm{X},\mathrm{Y},\mathrm{C}_1)$ and $\mathrm{s}(\mathrm{Y},\mathrm{Z},\mathrm{C}_2)$ in the body of the rule. In this case, and in all the cases analogous to this one, it is safe to adorn Y as *u* in both the q and s predicates in the body. It is not safe to adorn Y as *e* in either or both the predicates. Suppose that, for instance, we adorn Y by *e* in $\mathrm{q}(\mathrm{X},\mathrm{Y},\mathrm{C}_1)$ and by *u* in $\mathrm{s}(\mathrm{Y},\mathrm{Z},\mathrm{C}_2)$. This would give the following adorned rule:

$$\overline{\mathrm{cp}}^{\mathrm{uum}}(\mathrm{X},\mathrm{Z},\mathrm{C}) \leftarrow \mathrm{q}^{\mathrm{uem}}(\mathrm{X},\mathrm{Y},\mathrm{C}_1),\ \mathrm{s}^{\mathrm{uum}}(\mathrm{Y},\mathrm{Z},\mathrm{C}_2),\ \mathrm{C} = \mathrm{C}_1+\mathrm{C}_2.$$

It should be clear that $\overline{\mathrm{cp}}^{\mathrm{uum}}$ may be a proper subset of $\mathrm{cp}^{\mathrm{uum}}$ since there may not be any Y value in $\mathrm{s}(\mathrm{Y},\mathrm{Z},\mathrm{C}_2)$ (and therefore in $\mathrm{s}^{\mathrm{uum}}(\mathrm{Y},\mathrm{Z},\mathrm{C}_2)$)[3] corresponding to the Y value in $\mathrm{q}^{\mathrm{uem}}(\mathrm{X},\mathrm{Y},\mathrm{C}_1)$. Hence, *shared variables in the body of a rule are adorned as u.*

Furthermore, consider the variables X and Z in the body which are not shared and whose adornments are prescribed as *u* in the head. These variables are also adorned as *u*: *variables that are adorned as u in the head are adorned as u in the body*, irrespective of whether they are shared or not.

4.1.2. Adorning variables by e

Consider the following rule and the adornment *uem* to be propagated to the body:

$$\mathrm{p}(\mathrm{X},\mathrm{Z},\mathrm{C}) \leftarrow \mathrm{q}(\mathrm{X},\mathrm{U},\mathrm{C}_1),\ \mathrm{s}(\mathrm{W},\mathrm{Z},\mathrm{C}_2),\ \mathrm{C} = \mathrm{C}_1 + \mathrm{C}_2.$$

Clearly, X must be adorned as *u* since it is adorned as *u* in the head. Let us now consider the variables U and W. These variables are neither shared nor do they appear in the head. We adorn these variables by *e*, since their values are inconsequential. The variable Z in the body is also adorned as *e*, since it appears only once in the body and is adorned as *e* in the head. Thus, *variables that appear only once in the body and do not appear in the head are adorned as e. Variables that appear only once in the head as e and appear only once in the body are also adorned as e.*

[3] Unless s is total in its first argument.

4.2. Monotonicity conditions

The program of Section 4.1, assuming a cost domain of not negative numbers, is uniformly cost monotonic. This property ensures that the resulting rewritten program has a well-defined formal meaning and can be implemented efficiently with the greedy fixpoint algorithm. However, the propagation of min predicates can be applied to a larger class of queries satisfying monotonic conditions defined below. Clearly, the pushing of min predicates depends on the properties of the functions used to compute the cost arguments—not every function is as well-behaved as the addition used to compute $\mathtt{C} = \mathtt{C}_1+\mathtt{C}_2$.

4.2.1. Monotonicity of the cost functions used in the body

We now consider the constraints that cost predicates and functions computing cost values must satisfy in order to make the propagation of adornments allowed. Consider the following rule (which is identical to the one considered previously, except for the equation computing the head cost argument) and the adornment uum (same as before).

$$\mathtt{p(X,Z,C)} \leftarrow \mathtt{q(X,Y,C_1)},\ \mathtt{s(Y,Z,C_2)},\ \mathtt{C} = \mathtt{C_1}+(\mathtt{C_2}\ \mathtt{mod}\ 5).$$

In this case, the adornment cannot be propagated into the body predicate $\mathtt{s(Y,Z,C_2)}$ since the minimum value of $\mathtt{C_2}$ (as an integer) may not necessarily imply the minimum contribution of $(\mathtt{C_2}\ \mathtt{mod}\ 5)$ to C. However, the cost variable for the head is a non-decreasing monotonic function of the cost variable for the q-predicate (see Definition 5) and hence the adornment can be propagated into the q-predicate. The adorned rule is

$$\mathtt{cp^{uum}(X,Z,C)} \leftarrow \mathtt{q^{uum}(X,Y,C_1)},\ \mathtt{s(Y,Z,C_2)},\ \mathtt{C} = \mathtt{C_1}+(\mathtt{C_2}\ \mathtt{mod}\ 5).$$

This example shows that some body predicates may stay unadorned and are called unadornable predicates. All adornments to unadornable predicates resulting from the propagation process in the previous rules are ignored.

4.2.2. Totality of the cost predicate in the body

We now consider the issue of totality of the cost predicate that is raised by the following example:

$$\mathtt{p(X,Z,C)} \leftarrow \mathtt{q(X,Y,C_1)},\ \mathtt{s(Y,Z,C_2)},\ \mathtt{C} = \mathtt{log}(\mathtt{C_1} + \mathtt{C_2} - 5).$$

Suppose that (as before) we wish to propagate the adornment *uum* into the body. In this case, however, it is not possible to propagate the adornment into any of the body predicates. Suppose that for a certain X, Y and Z, the minimum value for both $\mathtt{C_1}$ and $\mathtt{C_2}$ is 2. Since $\mathtt{C} = \mathtt{log}(-1)$ is undefined, the adorned rule for this example is:

$$\mathtt{cp^{uum}(X,Z,C)} \leftarrow \mathtt{q(X,Y,C_1)},\ \mathtt{s(Y,Z,C_2)},\ \mathtt{C} = \mathtt{log}(\mathtt{C_1}+\mathtt{C_2}-5).$$

We formally define total monotonic mappings below.

Definition 5. Let P be a min program with cost domain C_P and valid cost sub-domain K_P. Let $f(X_1, \ldots, X_n)$ be an n-ary function on the cost domains. We say that the equality goal $Y = f(X_1, \ldots, X_n)$ defines a *total monotonic* mapping from X_i, $(1 \leqslant i \leqslant n)$ to Y when the following two conditions are satisfied:

totality: if $X_1, \ldots, X_n \in C_P$ then $f(X_1, \ldots, X_n) \in K_P$, and
monotonicity: if $X_1, \ldots, X_n, U, V \in C_P$ and $V \leqslant U$ then

$$f(X_1, \ldots, X_{i-1}, V, X_{i+1}, \ldots, X_n) \leqslant f(X_1, \ldots, X_{i-1}, U, X_{i+1}, \ldots, X_n).$$

The rules for propagating adornments into the body of Horn rules are summarized below. We use the following running example:

Example 6.

$$\mathtt{cp^{uem}(X,Y,C)} \leftarrow \mathtt{q(X,Z,N_1,C_1),\ r(Z,Y,N_2,C_2),\ s(Z,W,C_3),}$$
$$\mathtt{N_1 = N_2+1,\ C = C_1+C_2+(C_3\, mod\, 5).}$$

where the last goal in the body of the rule defines a total monotonic mapping from $\mathtt{C_1}$ and $\mathtt{C_2}$ to $\mathtt{C}$; the mapping from $\mathtt{C_3}$ to $\mathtt{C}$ is not monotonic.

4.2.3. Propagation of adornments

Given a rule r with head predicate $p(\bar{X}, C)$, we denote with $B(r)$ the set of cost predicates in the body of r and with $TC(r)$ the set of cost predicates in $B(r)$ such that for each $q(\bar{Y}, C') \in B(r)$ there is a total-monotonic mapping from C' to C and C' is a variable not appearing in any other predicate in $B(r)$. In our example $TC(r) = \{\mathtt{q(X,Z,N_1,C_1), r(Z,Y,N_2,C_2)}\}$ and $B(r) = TC(r) \cup \{\mathtt{s(Z,W,C_3)}\}$.

Given a variable X we denote with $\varphi(X)$ the set of variables containing X plus all variables whose values depend on the value of X, i.e., all variables which are connected to X through some equational condition. For instance, in Example 6 we have $\varphi(\mathtt{N_1}) = \{\mathtt{N_1, N_2}\}$ and $\varphi(\mathtt{C_1}) = \{\mathtt{C, C_1, C_2, C_3}\}$.

(1) Predicates in $B(r) - TC(r)$ are adorned with the empty adornment ε. We call such predicates *non-adorned predicates*.
(2) Predicates in $TC(r)$ are adorned with the adornment $\alpha \neq \varepsilon$. We call such predicates *adorned predicates*. Each predicate $q(X_1, \ldots, X_n, C) \in TC(r)$, where C is the variable denoting the cost argument, is adorned as follows:
    (a) The cost argument C is labelled with m ($\mathtt{C_1}$ and $\mathtt{C_2}$ in Example 6);
    (b) Each variable labelled as u in the head is labelled as u in the body (X in Example 6).
    (c) Each variable X_i ($i \leqslant 1 \leqslant n$) is labelled as u if there is a variable in $\varphi(X_i)$ appearing in some other non built-in goal ($\mathtt{Z}$, $\mathtt{N_1}$ and $\mathtt{N_2}$ in Example 6).
    (d) Each variable X_i ($i \leqslant 1 \leqslant n$) which has not be labelled by m or u in steps $(a) - (c)$ is labelled as e (This labels Y as e in Example 6.)

The adorned rule for the Example 6 is:

$$\mathtt{cp^{uem}(X,Y,C)} \leftarrow \mathtt{q^{uuum}(X,Z,N_1,C_1),\ r^{ueum}(Z,Y,N_2,C_2),\ s(Z,W,C_3),}$$
$$\mathtt{N_1 = N_2+1,\ C = C_1+C_2+(C_3\, mod\, 5).}$$

4.2.4. Minima definition of a predicate in terms of its surrogate predicate

Given a Horn clause with head predicate symbol p, the above rules generate the adorned version of each rule defining p, for a given adornment α. The adorned version of each rule defines a surrogate predicate cp^{α}. We complete the minima definition of p in terms of cp^{α} for the adornment α by introducing the following rule in the transformed program.

$$\mathtt{p^{\alpha}(X_1, \ldots, X_n)} \leftarrow \mathtt{min(X_i, W, cp^{\alpha}(X_1, \ldots, X_n))},$$

where X_i is the cost argument and W is the set of variables adorned as u in α. The above rule is denoted by $mindef(p, cp^{\alpha}, \alpha)$ and is read as the minima definition of p in terms of cp^{α} as determined by the adornment α.

4.3. Propagating adornments into non-Horn rules

The propagation of an adornment β into a min rule $p(X) \leftarrow min(C, W, q(Y))$ is performed by rewriting the rule as $p^{\beta}(X) \leftarrow q^{\alpha}(Y)$ where α is the adornment associated to the min atom.

In the following example we illustrate how an adornment is propagated into a minimized predicate.

Example 7. Consider the following rules where q is a base predicate symbol and the query goal is $s(X, Y, C)$.

$$\mathtt{s(X,Y,C)} \leftarrow \mathtt{min(C,(X),p(X,Y,C))},$$

$$\mathtt{p(X,Y,C)} \leftarrow \mathtt{min(C,(Y),q(X,Y,C))}.$$

The adornment associated to s is ε. By propagating ε, the adornment *uem* for the predicate p is derived. By propagating *uem* in the second rule we derive the adornment *eum* for q.

Moreover, in the propagation of the adornment α into the rule $p(X) \leftarrow min(C, W, q(Y))$ if every variable adorned as u in α appears in W, all remaining variables in W can also be adorned as e. For instance, the propagation of the adornment *uem* into the rule

$$p(X, Y, C) \leftarrow min(C, (X, Y, Z), q(X, Y, Z, V, C))$$

produces the adornment *ueeem* for q.

4.4. Adorning predicates and rules

So far, we have discussed how to propagate an adornment from the head of a rule into its body to generate the adorned version of the same rule.

For derived predicates, the adorned definition of a predicate p w.r.t. an adornment α is the set of rules containing all the rules which define the surrogate predicate cp^{α} and the rule which expresses the minima definition of p in terms of cp^{α} w.r.t. the same adornment α, i.e.,

$$p^{\alpha}(X_1, \ldots, X_n) \leftarrow min(X_i, W, cp^{\alpha}(X_1, \ldots, X_n)),$$

where X_i is the variable adorned as m in α and W is the set of variables adorned as u in α.

If p is a base predicate, then the adorned definition of p w.r.t. the adornment α is the minima definition p in terms of p w.r.t. the adornment α, i.e.,

$$p^{\alpha}(X_1, \ldots, X_n) \leftarrow min(X_i, W, p(X_1, \ldots, X_n)).$$

where X_i is the variable adorned as m in α and W is the set of variables adorned as u in α.

The minima definition of any predicate p in terms of another predicate q w.r.t. ε is the empty set of rules, i.e. no minima definition for p is produced. Thus, if p is defined by Horn rules, every rule defining p is rewritten by replacing every predicate symbol q with q^{ε}.

We now formally define the adorned definition of a predicate p w.r.t. a given adornment α. In order to do so, we use the following terminology:

(1) *adorn-rule*(r, α) is the adorned rule obtained by propagating the adornment α for the head of r into the body of r.
(2) *mindef*(p, q, α) denotes the rule that defines p as the minima of q w.r.t. the adornment α.

The adorned definition of a predicate p w.r.t. an adornment α, denoted by *adorn-pred*(p, α) is equal to

- *mindef*(p, p, α), if p is a base predicate
- $\{$*adorn-rule*$(r, \alpha) \mid r$ defines $p\} \cup \{$*mindef*$(p, cp^{\alpha}, \alpha)\}$, otherwise.

Example 8. Consider the program of Example 1. The adorned definition of the predicate sh_path w.r.t. the adornment ε is:

$$adorn\text{-}pred(\mathtt{sh\_path}, \varepsilon) = \{\mathtt{sh\_path}^{\varepsilon}\mathtt{(X,Y,C)} \leftarrow \mathtt{path}^{\mathtt{uum}}\mathtt{(X,Y,C)}\}.$$

Thus, the adorned definition of the predicate path appearing in the program with the adornment *uum* is

$$\mathit{adorn\text{-}pred}(\mathtt{path}, \mathit{uum}) = \begin{cases} \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow \mathtt{arc}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}), \\ \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow \mathtt{path}^{\mathtt{uum}}(\mathtt{X},\mathtt{Z},\mathtt{C}_1),\mathtt{arc}^{\mathtt{uum}}(\mathtt{Z},\mathtt{Y},\mathtt{C}_2),\ \mathtt{C} = \mathtt{C}_1 + \mathtt{C}_2, \\ \mathtt{path}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow \mathtt{min}(\mathtt{C},(\mathtt{X},\mathtt{Y}),\ \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C})), \end{cases}$$

whereas the adorned definition of the predicate arc appearing in the program with the adornment *uum* is

$$\mathit{adorn\text{-}pred}(\mathtt{arc}, \mathit{uum}) = \{\mathtt{arc}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow \mathtt{min}(\mathtt{C},(\mathtt{X},\mathtt{Y}),\mathtt{arc}(\mathtt{X},\mathtt{Y},\mathtt{C}))\}.$$

The resulting program corresponds to the program of Example 2.

4.5. Algorithm for adorning a program

In this subsection, we discuss an algorithm that generates an adorned program from a given min program and a query predicate.

The algorithm begins by generating the adorned definition of the query predicate, whose symbol we assume to be q. This may generate new adorned predicate symbols. The adorned definitions of these predicate symbols are (recursively) included in the adorned program until all definitions of generated adorned predicates have been included.

General min-programs may have predicates without cost arguments or predicates whose cost arguments are either non-monotonic or are not total. These predicates may not yield any adorned definitions during the propagation algorithm. Hence, the definition of such predicates must be included in the adorned program.

A simple way of achieving this purpose is to initiate the adornment propagation algorithm with the query predicate symbol q and the adornment ε. After all adorned definitions are generated, the set of rules reachable from q^ε form the adorned program. Finally, in the adorned program, all occurrences of q^ε are replaced by q. The algorithm which propagates *min* predicates is shown in Fig. 1. The variables S and T contain, respectively, the set of predicate symbols to be adorned and the set of predicate symbols already adorned, and *body*(r) denotes the set of atoms in the body of a rule r.

The following example shows how the algorithm works.

Example 9. Consider the program of Example 1 and the query sh_path(X,Y,C). In the first step, the rule immediately below is produced:

$$\mathtt{r}_1 : \mathtt{sh\_path}^{\varepsilon}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow \mathtt{path}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}).$$

At step 2, the algorithm generates the rules

$$\mathtt{r}_2 : \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow \mathtt{arc}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}).$$

$$\mathtt{r}_3 : \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftrightarrow \mathtt{path}^{\mathtt{uum}}(\mathtt{X},\mathtt{Z},\mathtt{C}_1),\ \mathtt{arc}^{\mathtt{uum}}(\mathtt{Z},\mathtt{Y},\mathtt{C}_2),\ \mathtt{C} = \mathtt{C}_1 + \mathtt{C}_2.$$

$$\mathtt{r}_4 : \mathtt{path}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \rightarrow \mathtt{min}(\mathtt{C},(\mathtt{X},\mathtt{Y}),\ \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C})).$$

Finally, at step 3, the rule

$$\mathtt{r}_5 : \mathtt{arc}^{\mathtt{uum}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow \mathtt{min}(\mathtt{C},(\mathtt{X},\mathtt{Y}),\mathtt{arc}(\mathtt{X},\mathtt{Y},\mathtt{C})).$$

is produced.

```
**Algorithm 1.** *Propagation of Min Predicates*
Input: A min query $\langle g(X), P\rangle$
Output: Adorned min program;
var $S, T, new\text{-}pred$ : set of predicate symbols;
    $p$ : predicate symbol;
    $\alpha$: adornment;
begin
    $S := \{q^\epsilon\}$;  $T := \emptyset$;
    while $S \neq \emptyset$ do
        Choose an arbitrary predicate $p^\alpha \in S$;
        Include $adorn\text{-}pred(p, \alpha)$ in $P$;
        $new\text{-}pred = \{s^\alpha \mid s^\alpha \in body(r)$ for some rule $r \in adorn\text{-}pred(p, \alpha)\} - T$;
        $S := S \cup new\text{-}pred$;
        $S := S - \{p^\alpha\}$;  $T := T \cup \{p^\alpha\}$;
    end while
    $Q :=$ set of rules reachable from $q^\epsilon$;
    return $Q$ after replacing $q^\epsilon$ in $Q$ by $q$;
end.
```

Fig. 1. Propagation of min predicates.

4.6. *Query equivalence of original and adorned programs*

In this section, we discuss the proof of query equivalence between a given query and the adorned query obtained as the output of Algorithm 1.

Note that Algorithm 1 includes the adorned definition of predicates into the source program P at each iteration. Hence, until termination, there is at least one adorned predicate symbol in P which is not defined in P, making it 'incomplete'. In order to prove the equivalence of the adorned program and the original program, we begin by completing the definition of such incomplete programs.

The *adorn-completion*(P), for a given set of rules P, is P union with *mindef*(p, p, α), for every p^α which appears in the body of some rule in P but is undefined in P. That is, for each undefined n-ary predicate symbol p^α a rule of the form

$$p^\alpha(X_1, \dots, X_n) \leftarrow min(X_i, W, p(X_1, \dots, X_n)),$$

where X_i is the cost argument and W is the set of variables adorned as u in α, is added to *adorn-completion*(P).

Lemma 1. *Let $Q = \langle q, P_0 \rangle$ be the input query of the Algorithm* 1 *and let $P_1 = P_0 \cup$ adorn-pred(q, ε) be the set of rules contained in P at the end of the first step of the Algorithm* 1. *Then*:

(1) $\langle q, P_0 \rangle \equiv \langle q, adorn\text{-}completion(P_1) \rangle$;
(2) $\langle q, adorn\text{-}completion(P_1) \rangle \equiv \langle q^\varepsilon, adorn\text{-}completion(P_1) \rangle$.

Proof. In the first step of the algorithm and in the adorn-completion process no rule is removed from P: only rules which are not reachable from q are added to P. Hence, $\langle q, P_0 \rangle \equiv \langle q, \text{adorn-completion}(P_1) \rangle$.

Moreover, the rules added in the adorn-completion process make all the rules of P_1 that can be reached from q reachable from q^ε: no rule which is 'inherited' from the original set P_0 and which cannot be reached from q can be reached from q^ε in adorn-completion(P_1). Hence $\langle q, \text{adorn-completion}(P_1) \rangle \equiv \langle q^\varepsilon, \text{adorn-completion}(P_1) \rangle$. □

Lemma 2. *Let $\langle q, P_0 \rangle$ be the input query of the Algorithm* 1 *and let P_i and P_{i+1} be the set of rules contained in P, respectively, at the end of the ith and $(i+1)$th steps. Then, $\langle q^{\varepsilon}, adorn\text{-}completion(P_i) \rangle \equiv \langle q^{\varepsilon}, adorn\text{-}completion(P_{i+1}) \rangle$.*

Proof. At the $(i+1)$th step of the Algorithm 1, one of the not yet 'expanded' symbols p^{α} is chosen and the set *adorn-pred*(p, α) is added to P. Hence, we have that:

$$adorn\text{-}completion(P_{i+1}) = (adorn\text{-}completion(P_i) - mindef(p,p,\alpha)) \cup adorn\text{-}pred(p,\alpha) \cup minset,$$

where $minset = \{mindef(q,q,\alpha) \mid q^{\alpha}$ appears in the body of some rule r such that $r \in adorn\text{-}pred(p,\alpha)$ and $r \notin adorn\text{-}completion(P_i)\}$.

If p is a base predicate $adorn\text{-}pred(p,\alpha) = mindef(p,p,\alpha)$, and so $adorn\text{-}completion(P_{i+1}) = adorn\text{-}completion(P_i)$. Otherwise, the set of rules contained in $adorn\text{-}pred(p,\alpha) \cup minset$ computes the same minimum calculus defined by $mindef(p,p,\alpha)$ in terms of the adorned predicates obtained from the predicates appearing in the body of the rules defining p. Indeed, if p is defined by the rules $r_1, \ldots, r_m$, where r_i is of the form $r_i : p \leftarrow q_{i_1}, \ldots, q_{i_n}$, the rule $mindef(p,p,\alpha)$ searches among all p-atoms which satisfy some r_i (where $i \in [1..m]$) for the ones having the minimum cost argument (according to the adornment α) and returns them. The adorned rule corresponding to r_i and contained in $adorn\text{-}pred(p,\alpha)$ is of the form: $r'_i : cp^{\alpha} \leftarrow q_{i_1}^{\alpha_1}, \ldots, q_{i_n}^{\alpha_n}$ and *minset* contains the rules $mindef(q_1, q_1, \alpha_1), \ldots, mindef(q_n, q_n, \alpha_n)$. According to the definition of propagated adornments, if a p-atom obtained from the rule r_i is returned by $mindef(p,p,\alpha)$, for each q_{i_j}-atom occurring in the body of r_i there exists a $q_{i_j}^{\sigma}$-atom (where σ is the propagation of α) with the same values for the corresponding arguments.

Thus, in both cases (either p is a base predicate or a derived one) we have that $\langle q^{\varepsilon}, adorn\text{-}completion(P_i) \rangle \equiv \langle q^{\varepsilon}, adorn\text{-}completion(P_{i+1}) \rangle$. □

Essentially, Lemma 1 states that query equivalence is preserved at the first iteration step of Algorithm 1 whereas Lemma 2 states that query equivalence is preserved at each subsequent iteration step of Algorithm 1. The following theorem formally states the query equivalence of the original and the adorned program.

Theorem 4. *Let Q be the program output by Algorithm* 1 *applied to the input query $\langle q, P \rangle$. Then $\langle q, P \rangle \equiv \langle q, Q \rangle$.*

Proof. Algorithm 1 terminates in a finite number k of steps, since P contains a finite number of rules and of predicates, and a predicate that is adorned in a certain step of the algorithm cannot be re-elaborated, for the same adornment, in any of the successive steps (see Proposition 2). Let P_0 be the set of rules contained in P before the execution of the first step of the algorithm, and let P_i (for $1 \leqslant i \leqslant k$) denote the set of rules contained in P at the end of the (i)-th iteration. It follows from Lemma 1 that $\langle q, P_0 \rangle \equiv \langle q^{\varepsilon}, adorn\text{-}completion(P_1) \rangle$. Lemma 2 states that, for any $1 \leqslant i < k$, $\langle q^{\varepsilon}, adorn\text{-}completion(P_i) \rangle \equiv \langle q^{\varepsilon}, adorn\text{-}completion(P_{i+1}) \rangle$, so we have that $\langle q, P_0 \rangle \equiv \langle q^{\varepsilon}, adorn\text{-}completion(P_k) \rangle$. Since the definition of all adorned predicates of P_k is in P_k, it holds that $adorn\text{-}completion(P_k) = P_k$. Hence, since Q is obtained extracting from P_k the set of rules reachable from q^{ε}, and then by replacing q^{ε} with q, we have that $\langle q, P_0 \rangle \equiv \langle q, Q \rangle$. □

In the following example we present the completion of the adorned program at the various steps.

Example 10. Consider again the program of Example 1 and the query *sh_path*(X, Y, C). In the first step the program P_1 consists of the rules in P plus the adorned rule.

$\mathtt{r_1 : sh\_path^{\varepsilon}(X, Y, C) \leftarrow path^{uum}(X, Y, C).}$

The adorn-completion of P_1 consists of P_1 plus the rule

$path^{\mathtt{uum}}(X, Y, C) \leftarrow min(C, (X, Y), path(X, Y, C))$.

At step 2 the algorithm generates the program P_2 (*adorn-completion*(P_1)) obtained by adding to P_1 the rules

$\mathtt{r}_2 : \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X}, \mathtt{Y}, \mathtt{C}) \leftarrow \mathtt{arc}^{\mathtt{uum}}(\mathtt{X}, \mathtt{Y}, \mathtt{C})$.

$\mathtt{r}_3 : \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X}, \mathtt{Y}, \mathtt{C}) \leftarrow \mathtt{path}^{\mathtt{uum}}(\mathtt{X}, \mathtt{Z}, \mathtt{C}_1),\ \mathtt{arc}^{\mathtt{uum}}(\mathtt{Z}, \mathtt{Y}, \mathtt{C}_2),\ \mathtt{C} = \mathtt{C}_1 + \mathtt{C}_2$.

$\mathtt{r}_4 : \mathtt{path}^{\mathtt{uum}}(\mathtt{X}, \mathtt{Y}, \mathtt{C}) \leftarrow \mathtt{min}(\mathtt{C}, (\mathtt{X}, \mathtt{Y}),\ \mathtt{cpath}^{\mathtt{uum}}(\mathtt{X}, \mathtt{Y}, \mathtt{C}))$.

The program *adorn-completion*(P_2) is obtained by adding to P_2 the rules

$arc^{\mathtt{uum}}(X, Y, C) \leftarrow min(C, (X, Y), arc(X, Y, C))$.

At the third and final step the algorithm generates the new program P_3 by adding to P_2 the rule

$\mathtt{r}_5 : \mathtt{arc}^{\mathtt{uum}}(\mathtt{X}, \mathtt{Y}, \mathtt{C}) \leftarrow \mathtt{min}(\mathtt{C}, (\mathtt{X}, \mathtt{Y}), \mathtt{arc}(\mathtt{X}, \mathtt{Y}, \mathtt{C}))$.

The adorn-completion of P_3 coincides with P_3 since all adorned predicates are defined. Observe that at each step the queries ($\mathtt{sh\_path}^{\varepsilon}(\mathtt{X}, \mathtt{Y}, \mathtt{C}), P_i$) and ($\mathtt{sh\_path}(\mathtt{X}, \mathtt{Y}, \mathtt{C}), P$) are equivalent.

4.7. Complexity

We conclude by presenting the complexity of the rewriting algorithm and showing how the computational complexity improves for the single source shortest path.

Proposition 2. *The complexity of Algorithm* 1 *applied to a min query* $\langle g(X), P \rangle$ *is bounded by* $O(\sum_p 2^{n_p} \times size(def(p)) + size(P))$ *where* i) *p is a cost predicate in P*, ii) $n_p + 1$ *is the arity of p*, iii) *def(p) is the set of rules in P defining p*, iv) *size(def(p)) measures the size of def(p), and* v) *size(P) measures the size of P.*

Proof. The maximum number of steps, equal to the number of possible adornments for the cost predicates, is $\sum_p 2^{n_p}$ where p is a cost predicate with arity $n_p + 1$. (p can be adorned in $2^{n_p} + 1$ different manners since the cost argument can be adorned only with the symbol m, whereas the other arguments with either u or e; if no adornment is propagated on p, it is adorned with ε.) At each step of the algorithm, a predicate p^{α} is chosen from the set S containing all the adorned predicates which have not been defined in any of the previous steps and *adorn-pred*(p, α) is computed. The construction of *adorn-pred*(p, α) consists in the rewriting of all rules defining p by propagating α in their body. The cost of this operation is linear with respect to the dimension of the set of rules defining p (here denoted by *size*(*def*(p))).

Standard predicates relevant for computing the (rewritten) query are copied in the rewritten program with cost bounded by *size*(P). □

It is important to note that although the size of the rewritten program could be exponential in the arity of predicates (as for the well-known magic-set technique), in practical case, for each cost predicate there are only a few different adornments (often only one).

Observe that, for an input min program P rewritten into P^{α} by Algorithm 1, for each predicate q in P and for each adorned predicate q^{β} in P^{α} derived from q, the number of q^{β}-atoms obtained from the computation of P^{α} (using the operator G_P) cannot be greater than the number of q-atoms obtained by computing P (using the operator T_P). The same result also holds for the surrogate predicate cq^{β}: the number of cq^{β}-atoms obtained from the computation of P^{α} (using G_P) cannot be greater than the number of q-atoms obtained by computing P (using T_P).

Thus, if for each predicate q appearing in P there is only a constant number of adorned predicates q^β in P^α, the size of $G^\infty_{P^\alpha}(\emptyset, \emptyset)$ is bounded by $O(T^\infty_P(\emptyset))$; the constant factor depends on the number of possible adornments.

The following example shows how the complexity of the single source shortest path is improved by propagating minimum.

Example 11. Assume we are given a weighted directed graph $G = (N, E)$ stored by means of the relation *arc* and a cost domain D. The query $d(Y, C)$ over the following program P

$$\mathtt{d(Y,C)} \leftarrow \mathtt{min(C,(Y),p(Y,C))}.$$

$$\mathtt{p(a,0)}.$$

$$\mathtt{p(Y,C)} \leftarrow \mathtt{p(X,C_1),\ arc(X,Y,C_2),\ C = C_1{+}C_2}.$$

computes the minimum distance of nodes in G from the source node a. The rewritten program P' is

$$\mathtt{d(Y,C)} \leftarrow \mathtt{p^{um}(Y,C)}.$$

$$\mathtt{p^{um}(Y,C)} \leftarrow \mathtt{min(C,(Y),cp^{um}(Y,C))}.$$

$$\mathtt{cp^{um}(a,0)}.$$

$$\mathtt{cp^{um}(Y,C)} \leftarrow \mathtt{p^{um}(X,C_1),\ arc^{uum}(X,Y,C_2),\ C = C_1{+}C_2}.$$

$$\mathtt{arc^{uum}(Y,C)} \leftarrow \mathtt{min(C,(X,Y),arc(Y,C))}.$$

which can be simplified to the program P''

$$\mathtt{d(Y,C)} \leftarrow \mathtt{min(C,(Y),cp^{um}(Y,C))}.$$

$$\mathtt{cp^{um}(a,0)}.$$

$$\mathtt{cp^{um}(Y,C)} \leftarrow \mathtt{d(X,C_1),\ arc(X,Y,C_2),\ C = C_1{+}C_2}.$$

since the definitions of d and p^{um} coincide.

For the computation of the complexity we assume that the cost of accessing and storing a tuple is constant and that at each step only the tuples computed in the previous step are used to compute new tuples (semi-naive optimization [37]).

For the evaluation of P we first compute the rules defining p and next the rule defining d. The number of tuples in p is bounded by $O(|N| \times |D|)$ and for each tuple $p(x, c_1)$ we select the set of arcs with source node x at cost $O(|N|)$ since each node may have $O(|N|)$ arcs starting in the node x. The rule defining d can be computed at cost $O(|N| \times |D|)$ (the cardinality of p). Therefore, the global cost is $O(|N|^2 \times |D|)$.

For the evaluation of program P'', observe that the number of d-tuples is bounded by $O(|N|)$ whereas the number of cp^{um}-tuples is bounded by $O(|N|^2)$. Assuming that at each step only one tuple with least cost is selected from cp^{um} and stored into d, the computation terminates in $O(|N|)$ steps.[4] At each step, a tuple $d(x, c_1)$ is stored into d and we select the set of arcs with source node x at cost $O(|N|)$. Therefore, the global cost is $O(|N|^2)$. □

The above example shows that the complexity is independent from the size of the cost domain. Therefore, for a large domain D, the complexity improves dramatically. Similar results can be obtained from other

[4]The complexity does not change if we relax the assumption of selecting only one tuple with least cost.

greedy or dynamic programming problems where the early application of minima cuts the domain of evaluation.

5. Conclusion

In this paper, we have presented a technique for the propagation of extrema predicates into possibly recursive queries. The propagation of such meta-level predicates assures an efficient computation, by using simple variations of the seminaive fixpoint, such as the greedy fixpoint operator [9]. It has been shown that by using appropriate data structures, the computation of programs with extrema by means of specialized algorithms has complexity comparable to that of procedural languages [19]. For instance, the computation of the shortest paths program of Example 1, after the propagation of the min predicates, has the same complexity as Floyd's algorithm; furthermore, the following single source shortest path program,

```
sh_path(Y, C) ← min(C, (Y), path(X, Y, C)).
path(Y, C) ← arc(a, Y, C).

path(Y, C) ← path(Z, C1), arc(Z, Y, C2), C = C1+C2.
```

after the propagation of the min predicates, has the same complexity as the celebrated Dijkstra's algorithm.

The technique introduced here can also be applied to larger classes of queries such as XY-stratified and modularly stratified queries. We present here two examples of programs which are not uniform cost monotonic but present such forms of stratification that propagation of extrema can be performed and computation can be done by means of specialized algorithms [35,10].

Example 12. *Transportation problem*. An airplane can carry on a scheduled flight a cargo with maximum weight of 10 000 lb. There are n different items that could be transported, with item i weighing a_i lb and providing a profit of c_i dollars if transported. The problem consists in maximizing the global profit of the item transported.

```
max − cargo(Value) ← max(Value, (), cargo(I, Weight, Value)).
cargo(0, 0, 0).

cargo(I + 1, Weight, Value) ← cargo(I, Old_Weight, Old_Value),
                              item(I + 1, W, V), elements(I, X),
                              Weight = Old_Weight + X * W, Weight < 10000,
                              Value = Old_Value + X * V.
```

The recursive rules defining the predicate cargo are XY-stratified (i.e. there is a stratification induced by the first argument of cargo) and, therefore, we could safely propagate the *max* aggregate.

Example 13. We are given a number of basic components stored by means of a relation *basic-part* and a relation *fix* which stores for each basic component the time to fix it (e.g. the time to substitute the component). A complex component is constituted by a set of components (either basic or complex). The maximum time to fix a component is 0 if the component has been tested or if it has no suspected parts. A part is suspected if some of its components are not working. Moreover, if a component has some suspected part then the time to make it work depends on the time to fix its basic components.

```
max − time(X, C) ← max(C, (X), working(X, C)).

working(X, 0) ← tested(X).
working(X, 0) ← part(X, Y), ¬has − suspect − part(X).
working(X, C) ← part(X, Y), has − suspect − part(X), fix − time(X, C).
has − suspect − part(X) ← part(X, Y), ¬working(Y, 0).

fix − time(X, C) ← basic − part(X), fix(X, C).
fix − time(X, C) ← part(X, Y), fix(Y, C).
```

The above program is not stratified, but, if the relation *part* is 'acyclic' it is modularly stratified [34]. Also in this case the *max* predicate can be safely propagated.

Acknowledgements

The authors are grateful to the anonymous referees for their useful comments.

References

[1] Y.K. Ng, N. Qaraeen, Data retrieval and aggregates in SQL∗/NR, Proceedings of the Conference on Information Systems and Management of Data, 1995, pp. 283–301.
[2] H. Wang, C. Zaniolo, User defined aggregates in object-relational systems, Proceedings of the International Conference on Data Engineering, 2000, pp. 135–144.
[3] S. Greco, Dynamic programming in datalog with aggregates, IEEE Trans. Knowledge Data Eng. 1999, pp. 265–283.
[4] S. Greco, C. Zaniolo, S. Ganguly, Optimization of logic queries with MIN and MAX predicates, International Conference on Flexible Query Answering, 1998, pp. 188–202.
[5] S. Grumbach, M. Rafanelli, L. Tininini, Querying aggregate data, Proceedings of the International Symposium on Principles of Database Systems, 1999, pp. 174–184.
[6] L. Hella, L. Libkin, J. Nurmonen, L. Wong, Logics with aggregate operators, Proceedings of the International Conference on Logic in Computer Science, 1999, pp. 35–44.
[7] L. Libkin, L. Wong, New techniques for studying set languages, bag languages and aggregate functions, Proceedings of the International Symposium on Principles of Database Systems, 1994, pp. 155–166.
[8] A. Van Gelder, Foundation of aggregation in deductive databases, Proceedings of the International Conference on Deductive and Object Oriented Databases, 1993, pp. 13–34.
[9] S. Ganguly, S. Greco, C. Zaniolo, Extrema predicates in deductive databases, J. Comput. System Sci. 51 (2) (1995) 244–259.
[10] R. Ramakrishnan, K.A. Ross, D. Srivastava, S. Sudarshan, Efficient incremental evaluation of queries with aggregation, Proceedings of the International Symposium on Logic Programming, 1994, pp. 204–218.
[11] K. Ross, Y. Sagiv, Monotonic aggregation in deductive databases, J. Comput. System Sci. 54 (1) (1997) 79–97.
[12] F. Buccafurri, N. Leone, P. Rullo, Enhancing disjunctive datalog by constraints, IEEE Trans. Knowledge Data Eng. 12 (5) (2000) 845–860.
[13] M.P. Consens, A.O. Mendelzon, Low complexity aggregation in graphlog and datalog, Theoret. Comput. Sci. 116 (1–2) (1993) 95–116.
[14] A. Lefebvre, Towards an efficient evaluation of recursive aggregates in deductive databases, Proceedings of the International Conference on Fifth Generation Computer Systems, 1992, pp. 915–925.
[15] C. Liu, A. Ursu, A framework for global optimization of aggregate queries, Proceedings of the International Conference on Knowledge Management, 1997, pp. 262–269.
[16] K. Apt, H. Blair, A. Walker, Towards a theory of declarative programming, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, 1988, pp. 89–148.
[17] A. Van Gelder, Negation as failure using tight derivations for general logic programs, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, San Francisico, 1988, pp. 149–176.
[18] S.W. Dietrich, Shortest path by approximation in logic programs, ACM Lett. Program. Lang. Systems 1 (1992) 119–137.

[19] S. Greco, C. Zaniolo, Greedy algorithms in datalog with choice and negation, Proceedings of the International Joint Conference and Symposium on Logic Programming, 1998, pp. 294–309.
[20] D. Kemp, P. Stuckey, Semantics of logic programs with aggregates, Proceedings of the 1991, International Symposium on Logic Programming, 1991, pp. 387–341.
[21] I.S. Mumick, H. Pirahesh, R. Ramakrishnan, The magic of duplicates and aggregates, Proceedings of the 16th Conference on Very Large Data Bases, 1990, pp. 264–277.
[22] K. Ross, D. Srivastava, P.J. Stuckey, S. Sudarshan, Foundations of aggregation constraints, Theoret. Comput. Sci. 193 (1–2) (1998) 149–179.
[23] S. Sudarshan, R. Ramakrishnan, Aggregation and relevance in deductive databases, Proceedings of the 17th Conference on Very Large Data Bases, 1991, pp. 501–511;
R. Ramakrishnan, D. Srivastava, S. Sudarshan, CORAL: Control, Relations and Logic Proceedings of International Conference on Very Large Data Bases (1992) 238–250.
[24] S. Sudarshan, D. Srivastava, R. Ramakrishnan, C. Beeri, Extending well-founded and valid semantics for aggregation, Proceedings of the International Logic Programming Symposium, 1992, pp. 591–608.
[25] I.S. Mumick, O. Shmueli, How expressive is stratified aggregation, Ann. Math. Artif. Intell. 15 (3–4) (1995) 407–434.
[26] ANSI/ISO/IEC 9075, *SQL standard* at http://www.ansi.org/.
[27] S.J. Finkelstein, N. Mattos, I.S. Mumick, H. Pirahesh, Expressing recursive queries in SQL, ISO-IEC JTC1/SC21 WG3 DBL MCI Technical Report, March 1996.
[28] M. Gelfond, V. Lifschitz, The stable model semantics of logic programming, Proceedings of the Fifth International Conference on Logic Programming, 1988, pp. 1070–1080.
[29] J. Lloyd, Foundation of Logic Programming, 2nd Edition, Springer, Berlin, 1987.
[30] T. Przymusinski, On the declarative semantics of deductive databases and logic programming, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, San Francisico, 1988, pp. 193–216 (Chapter 5).
[31] H. Przymusinska, T.C. Przymusinski, Weakly perfect model semantics for logic programs, Proceedings of the Fifth International Conference on Logic Programming, 1988, pp. 1106–1120.
[32] A. Van Gelder, K.A. Ross, J.S. Schlipf, The well-founded semantics for general logic programs, J. ACM 38 (3) (1991) 620–650.
[33] D. Saccà, The expressive powers of stable models for bound and unbound DATALOG queries, J. Comput. System Sci. 54 (3) (1997) 441–464.
[34] K. Ross, Modular stratification and magic sets for datalog programs with negation, J. ACM 41 (6) (1994) 1216–1266.
[35] C. Zaniolo, N. Arni, K. Ong, Negation and aggregation in the logic language LDL++, Proceedings of the International Conference on Deductive and Object-Oriented Databases, 1993, pp. 204–221.
[36] D. Saccà, C. Zaniolo, Stable models and non-determinism in logic programs with negation, Proceedings of the Ninth ACM Conference on Principles of Database Systems, 1990, pp. 205–217.
[37] J. Ullman, Principles of Data and Knowledge-Base Systems, Computer Science Press, New York, 1988.
[38] C. Beeri, R. Ramakrishnan, On the power of magic, J. Logic Program. 10 (3&4) (1991) 255–299.