# Extrema Predicates in Deductive Databases

**Sumit Ganguly**
Department of Computer Science
Rutgers University
New Brunswick, NJ 08904
sumit@cs.utexas.edu

**Sergio Greco***
Dip. Elettronica Informatica e Sistemistica
Università della Calabria
87030 Rende, Italy
greco@deis21.deis.unical.it

**Carlo Zaniolo**
Computer Science Department
University of California at Los Angeles
Los Angeles, CA, 90024
zaniolo@cs.ucla.edu

## Abstract

*A novel approach is proposed for expressing and computing efficiently a large class of problems, including finding the shortest path in a graph, that were previously considered impervious to an efficient treatment in the declarative framework of logic-based languages. Our approach is based on the use of* min *and* max *predicates having a first-order semantics defined using rules with negation in their bodies. We show that, under certain monotonicity conditions, (1) there exists a total well-founded model for these programs expressed using negation, (2) this model can be computed efficiently using a procedure called* greedy fixpoint*, and (3) programs with* `min/max` *goals on recursively-defined cost predicates can often be rewritten into more efficient ones by pushing* min *and* max *predicates into recursion. The greedy fixpoint evaluation of the program expressing the shortest path problem coincides with Dijkstra's algorithm, once the finite differencing techniques of seminaive fixpoint are applied.*

*Work partially supported by the project "Sistemi Informatici e Calcolo Parallelo" obiettivo "Logidata+" of C.N.R. Italy.

# 1 Introduction

Current research on deductive databases and logic programming strives to support a declarative high-level problem formulation without surrendering the performance obtainable by careful programming in an imperative language. In this respect, an interesting challenge is posed by graph problems, such as finding the shortest path in a graph, that are common in many applications. What makes this issue intriguing is the fact that, while deductive databases deal very well with closure problems, they cannot even approach the elegance and efficiency of, say, Dijkstra's algorithm on the shortest path problem, which can be viewed as a simple generalization of the graph-closure problem.

In this paper, we introduce *min* and *max* meta-level constructs with a first order semantics, allowing a declarative specification of such programs, and we develop techniques for their efficient implementation.

Example 1, below, gives a declarative formulation of the shortest path problem. Given a weighted directed graph represented as a base relation *arc*, where all edge weights are non-negative, the predicate *path* computes the set of all triples *(X, Y, C)* such that there is a path from node $X$ to node $Y$ with cost $C$. The predicate *sh_path(X, Y, C)* is intended to yield all the triples *(X, Y, C)* such that $C$ is the least cost among all paths from node $X$ to node $Y$.

**Example 1:** Shortest path.

$$
\begin{array}{ll}
\mathtt{sh\_path(X,Y,C)} \leftarrow & \mathtt{min(C,(X,Y),\ path(X,Y,C))}. \\
\mathtt{path(X,Y,C)} \leftarrow & \mathtt{arc(X,Y,C)}. \\
\mathtt{path(X,Y,C)} \leftarrow & \mathtt{path(X,Z,C1),\ arc(Z,Y,C2),\ C = C1 + C2}.
\end{array}
$$

□

A precise semantics can be assigned to our program, by simply viewing the first rule containing the *min* predicate as a short hand for the following rule,

$$
\begin{array}{ll}
\mathtt{sh\_path(X,Y,C)} \leftarrow & \mathtt{path(X,Y,C),\ \neg a\_lesser\_path(X,Y,C)}. \\
\mathtt{a\_lesser\_path(X,Y,C)} \leftarrow & \mathtt{path(X,Y,C'),C' < C}.
\end{array}
$$

where $C'$ is a new variable not appearing elsewhere in the rule. Our program has formal semantics, inasmuch as this is a stratified program [1, 15, 14]. However, a straightforward evaluation of such a stratified program would materialize the predicate *path(X, Y, C)* and then choose the smallest cost tuple for every $X$ 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. To solve these problems, one begins by observing that, since the weights are non-negative, all minimum paths of length $n + 1$ can be generated from the minimum paths of length $n$. Thus, the *min* predicate can be pushed into recursion, in such a way that the generation of new paths is interleaved with the computation of shortest paths, yielding the program of Example 2.

**Example 2:** Shortest paths between pairs of nodes

$$
\begin{aligned}
&\texttt{path(X,Y,C)} \leftarrow && \texttt{arc(X,Y,C).}\\
&\texttt{path(X,Y,C)} \leftarrow && \texttt{sh\_path(X,Z,C1),\ arc(Z,Y,C2),\ C = C1 + C2.}\\
&\texttt{sh\_path(X,Y,C)} \leftarrow && \texttt{min(C,(X,Y),\ path(X,Y,C)).}
\end{aligned}
$$

□

Unfortunately, as we attempt to give a meaning to Example 2 by its rewriting using negation, as in the previous example, we obtain a non-stratified program, with all the open semantic and computational issues therewith. These problems, widely recognized by deductive database researchers, have been the subject of recent works [4, 13, 7, 20, 10, 17]. These works however, consider the general problem of aggregates, e.g., including average, sum, etc. Our paper focuses only on extrema aggregates and proposes specialized, and thus very efficient, techniques for handling them. Although our discussion deals explicitly only with *min* programs, the symmetric properties of *max* programs follow by duality.

**Example 3:** Given a database of part-subpart relationship in `partof`, the following program demonstrates the use of the *max* predicate to compute `earliest(I,T)` that gives the earliest assembly time, `T`, of an item `I`, from the waiting times `wait(I, T)` of the children. This program assumes that once all the subparts of a part are available, assembling the compound part takes negligible time.

$$
\begin{aligned}
&\texttt{earliest(I,T)} \leftarrow && \texttt{wait(I,T).}\\
&\texttt{not\_before(I,T)} \leftarrow && \texttt{partof(J,I),\ earliest(J,T).}\\
&\texttt{earliest(I,T)} \leftarrow && \texttt{max(T,(I),\ not\_before(I,T)).}
\end{aligned}
$$

□

The paper is organized as follows. In the next section, we introduce the notion of monotonic *min* programs, and prove that these programs are weakly stratified [15], and therefore have a total well-founded model [26, 24], which coincides with their unique stable model [9]. In Section 3, we introduce the *greedy fixpoint* procedure used for the efficient computation of monotonic min-programs and show that it computes the unique minimal model. In Section 4, we pursue optimized implementations for the greedy fixpoint approach and show that the seminaive improvement yields the complexity of Dijkstra's algorithm on the classical shortest-path-in-a-graph problem [6].

In Section 5, we turn to 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 this is done, the greedy fixpoint procedure can be used to compute efficiently the transformed program. In Section 6, we consider the problem of pushing constants into these programs and show how this is accomplished by a simple adaptation of the magic set method. In the last section we compare our approach with related work and propose issues for further research.

## 2 Syntax and Semantics

Given a program $P$, we assume the existence of a totally ordered domain $D_P$, over which the notion of extrema, and built-in predicates symbols have been pre-defined. $D_P$ will be called the *cost domain.* In this paper, our cost domain is the set of real numbers and the built-in predicates over this domain are $=$, $<$ and $\leq$, $>$, $\geq$ and $\neq$. Following [14], we assume that instances of these built-in binary predicates are true iff (1) they have the form $e_1 \theta e_2$, where $e_1$ and $e_2$ are arithmetic expressions on the cost domain, and (2) $e_1$ and $e_2$ evaluate to values, say $val(e_1)$ and $val(e_2)$ such that $val(e_1)\ \theta\ val(e_2)$. Thus, $5 > 1 + 3$ is predefined to be true, while $5+1 < 3-2$ and $a = 2+1$ are defined to be false (respectively, for failing the first and the second condition). We will consider arithmetic expressions built using arithmetic operators that are total on the domain of real numbers, such as $+$, $-$, $*$, $mod$, etc.

We assume the basic concepts of terms, atoms, Horn clauses, Horn rules, (Herbrand) interpretations and (Herbrand) models [12] and define the syntax of our min-programs as follows:

**Definition 1:** A *special atom* is of the form $min(C, S, Q)$ where:

1. $Q$ is an atom of first order logic, and will be called a *minimized atom* (also called *minimized predicate*).
2. $S$ is a set of variables of $Q$ called *grouping variables.* The variables of $Q$ which are not in $S$ will be called *existential.*
3. $C \notin S$ is a variable, appearing as an argument of $Q$; $C$ will be called the *minimized argument* of $Q$. Minimized variables can only take values from the cost domain. □

**Definition 2:** A *literal* is either an atom (a positive literal) or its negation (a negative literal). A *min-literal* is either a literal, a special atom or the negation of a special atom. □

Following first order logic, we define *min-formulas* (open, closed and universal), and *min-clauses* inductively using min-literals as the base case. We say that a *min-clause* is *definite* if it contains exactly one positive literal which is not a special atom. A *min-program* is a finite set of definite min-clauses. The set of predicates in a program $P$ that are mutually recursive with a minimized predicate will be called a *min-clique* for $P$.

We define the semantics of a min-program, $P$, by its rewriting into a program with negated goals, denoted $foe(P)$. The rewritten program is obtained by replacing each special goal

$$\leftarrow ..., min(C, S, q(\bar{Y}, C)), ...$$

by a pair of goals

$$\leftarrow ..., q(\bar{Y}, C), \quad \neg a\_lesser\_q(\bar{Y}, C), ...$$

The distinguished $a\_lesser\_q$ predicate is defined by the additional rule:

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

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 variable $C'$ is new and corresponds to the variable $C$. (Thus a new variable is introduced for each existential variable in the min-atom.) Therefore, Example 2 is viewed as a shorthand of:

**Example 4:** The foe rewriting of Example 2.

$$
\begin{array}{ll}
\mathtt{path(X,Y,C)} \leftarrow & \mathtt{arc(X,Y,C).} \\
\mathtt{path(X,Y,C)} \leftarrow & \mathtt{sh\_path(X,Z,C1),\ \ arc(Z,Y,C2),\ \ C = C1 + C2.} \\
\mathtt{sh\_path(X,Y,C)} \leftarrow & \mathtt{path(X,Y,C),\ \ \neg a\_lesser\_path(X,Y,C).} \\
\mathtt{a\_lesser\_path(X,Y,C)} \leftarrow & \mathtt{path(X,Y',C'), C' < C.}
\end{array}
$$

□

Since we will view a min-program simply as a shorthand of $foe(P)$, we will say that $M$ is an interpretation (resp. a (minimal) model) for $P$, when $M$ is an interpretation (resp. a (minimal) model) for $foe(P)$. Therefore, we can define the semantics of min-programs by using concepts such as stratification, well founded models and stable models, developed for logic programs with negation.

In particular we will use the concept of *stable set* and *stable model* [9]. Let $Q$ be a program and let $ground(Q)$ denote the ground instantiation of $Q$ obtained by replacing each rule of $Q$ containing variables with all its ground instances. Let $M$ be a set of atoms from the Herbrand base of $P$. Let $ground_M(Q)$ be the program obtained from $ground(Q)$ by the following transformation, called *stability transformation*: [9].

1. delete each rule that has a negative literal $\neg B$ in its body with $B \in M$, and

2. delete all negative literals in the body of the remaining rules.

Since $ground_M(Q)$ is a positive program, it has a unique minimal model. A model $M$ of $Q$ is said to be *stable* when $M$ is also the minimum model of $ground_M(Q)$ [9]. All stable models for $Q$ are also minimal models, but not vice versa. A given program can have one or more stable (total) models, or possibly none. Positive and stratified programs are among those that have exactly one stable model [9].

We can now proceed and define classes of min-programs, which because of their syntactic structure and stratification conditions induced by the underlying cost-domain, have a unique stable model that can be computed efficiently using the greedy fixpoint procedure described later.

We will assume that certain database predicate 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 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. More formally, let $P$ be a min-program and let $T_P$ denote its *immediate consequence operator* [12].

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 $U_{foe(P)} \cup D'_P$ where $D'_P$ is a valid cost subdomain, and such that each goal corresponding to a built-in predicate is true. Thus 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 program $foe(P)$, denoted $G_P$, is a directed graph whose nodes are ground atoms of $foe(P)$. In $G_P$, there is an arc from a node $A$ to another node $B$ if there is an 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.

Following practice in lattice theory, we define $T_P^n$ for every finite $n$ as follows: $T_P^0(I, L) = (I, L)$ and $T_P^n(I, L) = T_P(T_P^{n-1}(I, L))$ if $n > 0$. For $n = \infty$, $T_P^\infty(I, L)$ is defined to be $\cup_{n\ finite} T_P^n(I, L)$. We can now formalize the notion of cost domain as follows:

**Definition 3:** Let $P$ be a min-program, and let $det(P)$ denote the program obtained from $foe(P)$ by (i) removing all the `a_lesser` rules and (ii) removing all `a_lesser` goals from the bodies of the rules. The i-th argument of a predicate $q$ of $P$, is said to be a *cost argument* if the i-th argument of every $q$-atom in $T^\infty_{det(P)}(\emptyset)$ belongs to the cost domain. In this case, $q$ will be called a cost predicate and its instances will be called cost atoms. □

When, a given predicate $p$ has several cost arguments, we arbitrarily select one as it main cost argument: the value of the main cost argument for a $p$-atom $p_i$ will be denoted $cost(p_i)$.

**Definition 4:** Let $Q$ be a *min*-clique in a program $P$ with valid dependency graph $G_P$. Then $Q$ will be said to be *cost monotonic* when for every path from $q_1$ to $q_2$ in $G_P$, if $q_1$ and $q_2$ are two instances of the same predicate in $Q$, then

- $cost(q_1) \geq cost(q_2)$, and
- $cost(q_1) > cost(q_2)$ if the path contains a negated arc. □

A program $P$ will be said to be cost-monotonic when all its min-cliques are cost-monotonic. □

Thus (out of the possibly several cost argument of a recursive predicate), one argument can be selected such that the monotonicity property holds for the dependency graph of $P$. Take for instance, the following example:

**Example 5:**

$$
\begin{array}{lll}
\mathtt{p(X,Y,C)} \leftarrow & \mathtt{arc(X,Y,C).} \\
\mathtt{p(X,Y,C)} \leftarrow & \mathtt{sp(X,Z,C1),\ arc(Z,Y,C2),\ C = C1 + C2 - 2.} \\
\mathtt{sp(X,Y,K)} \leftarrow & \mathtt{min(C,(X,Y),\ p(X,Y,C)),\ K = C + 1.}
\end{array}
$$

□

The database predicate `arc(X,Y,C)` describes a set of arcs and their costs. Thus, the third argument of *arc* must be a number, which implies that the third arguments of `p` and `sp` are cost arguments. Furthermore, this program is cost-monotonic if we restrict the valid sub-domain to numbers greater than 1.

**Theorem 1:** Every cost-monotonic min-program is weakly stratified.

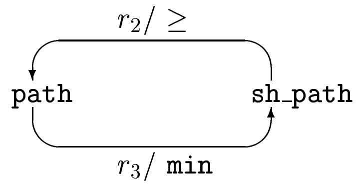
**Proof:** Let $P$ be a min-program. Let $P_H$ be the ground instantiation of $foe(P)$. The dependency relations $\prec$ and $\preceq$ between ground atoms of $P_H$ are defined by 1) $A \preceq B$ if there is a path from $A$ to $B$; 2) $A \prec B$ if there is a path from $A$ to $B$ passing through a negative arc. The built-in predicate $<$ in $foe(P)$, can be considered to belong to level 0. The model for the built-in predicate $<$ (denoted $M_0$) is the set of ground literals $C1 < C2$ such that $C1 < C2$ is true. By application of the Davis-Putnan reduction [3] we obtain the program $P_H^1$. The priority relation $\prec$ on the program $P_H^1$ is transitive, and a proof of this fact follows directly from the definitions. To prove that it is anti-symmetric, let $A \prec B$ ($B$ depends on $A$ through negation). Then there are two connected nodes $A'$ and $B'$ (not necessarily distinct from $A$ and $B$) such that $A' \prec B'$ ($(A', B')$ denotes the negative arc in the path $(A, B)$). Obviously, $cost(A') < cost(B')$ because we have now only rules with negated literals in the body of the form: $p(X, C) \leftarrow ..., q(X, C), \neg q(Y, C')$ iff $C' < C$. Now, if the condition $B \preceq A$ also holds, then there is path from $B'$ to $A'$. But since $P$ is a monotonic program, $cost(B') \leq cost(A')$—a contradiction. Therefore $\preceq$ is partial order and the valid instantiation of our program $P$ (i.e., $P_H^1$) is locally stratified. Therefore $P$ is weakly stratified. □

Thus, cost-monotonic programs always have a unique stable model. Observe that this class is quite broad and e.g., includes programs containing negation and arbitrary set-aggregate operators in their recursive rules. In this paper we focus on a very useful subclass of cost-monotonic min-programs that uses extrema predicates, and provide an efficient procedure for computing the stable models for programs in this class.

**Definition 5:** A min-program $P$ is said to be *uniformly cost-monotonic* if

1. the minimized argument of the minimized predicate is a cost argument—this will be assumed to be the main cost argument for the predicate.
2. for every valid instance of a recursive rule, the cost of the cost predicate in the head is $\geq$ than that of every cost predicate in the body. □

The program of Example 2 is is uniformly cost-monotonic, while that of Example 5 is not.




— **Fig. 1.** Cost graph for Example 1 —

**Observation 1:** Uniformly cost-monotonic min-programs are cost-monotonic.

In the rest of this section, we discuss the compilation-related problem of recognizing monotonic min-programs by their syntactic structure. While in general, the problem is untractable, sufficient conditions can be given that deal with the situations of practical interest, such as the examples given in this paper. Therefore, we now introduce the notion of cost graph to help detect programs that are uniformly cost-monotonic.

**Definition 6:** 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 $p$ labeled with rule $r$ is marked with $\geq$ if the arc is derived from a Horn rule and the value of the cost argument of $p$ is $\geq$ the value of to the cost argument of $q$ for every valid ground interpretation of $r$. If the arc is derived from a non-Horn rule then it is marked with $min$ and is called a *min-arc.* □

The cost graph for the program of Example 2 is given in Figure 1. The construction of such a cost graph begins 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 labeled with $r_2/\geq$ since we have $cost(path(X,Y,C1)) \geq cost(sh\_path(X,Y,C))$ for all possible instantiations of the rule. From the third rule we derive the arc from *path* to *sh_path* labeled 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 $C \geq C1$ would make the monotonic cost constraint detectable by even a very unsophisticated compiler.

## 3 The Greedy Evaluation Technique

In this section, we consider the problem of computing the intended model $M_P$ of a min-program $P$. We will compute one strong component (set of mutually recursive predicates) at a time following the topological order defined by the dependencies among the predicates symbols of the program. The clauses of a min-program $P$ can be partitioned into two sets, namely the set of Horn Clauses $H$ and the set of non-Horn Clauses $N$. $T_H$ and $T_N$

denote immediate consequence operators for the Horn clauses and the non-Horn clauses in $P$ respectively. Then, the greedy evaluation can be viewed as a specialization of the *Backtracking Fixpoint* procedure for computing stable models presented in [19] which can be summarized as follows:

Step 1. Assume some negative facts and, using these, fire the non-Horn clauses, giving some new positive facts,

Step 2. Use the newly derived facts to fire all the Horn rules until saturation (i.e., compute $T_H^\infty$),

Step 3. If there is no contradiction between the negative facts assumed in Step 1 and the positive ones derived in Step 2, then repeat Step 1, otherwise undo (backtrack) Step 2 and Step 1; then resume Step 1 with different assumptions.

The first source of inefficiency in the above procedure is the backtracking required in Step 3. This could be avoided if it were possible to make a judicious decision on which negative facts to assume at Step 1, such that these assumptions are never contradicted by positive facts later generated in Step 2. A second source of inefficiency is the need compute $T_H^\infty$ in Step 2—thus yielding a transfinite computation. In this paper we show that in many situations both inefficiencies can be removed. Thus, using a greedy approach minima can be selected at Step 1 that will never be contradicted at later steps. Moreover at Step 2, one only need to compute $T_H$ rather than $T_H^\infty$. We illustrate this next by using the program of Example 2.

### 3.1 Adornments

We associate *adornments* with atoms and their arguments in recursive rules of the program. Thus, if $g(t_1, ..., t_k)$ denotes either an atom in a rule or a term in such an atom, we will write $g^{a_1,...,a_k}(t_1, ..., t_k)$ and say that $a_j$ is the label of $t_j$ in $g(t_1, ..., t_k)$. Then, $a_j$ can have one of the following three values $u$, $e$, or $m$. These are (recursively) assigned as follows:

1. The label of a goal for its main cost argument is $m$.
2. Let $t$ be a goal or term in a goal. Then, $t$ is assigned a label $e$ for each argument which is (i) a constant, or (ii) an existential variable not labeled with $m$, or (iii) a term with all of its arguments labeled $e$. Such an argument of $t$ will be called *existential*.[1]

[1]The name is justified by the following *existential expansion* of special atoms: using De Morgan's law, $min(C, S, Q)$ is expanded into

$$Q \wedge \nexists \mathbf{w}(Q' \wedge C' < C).$$

where $Q'$ is the clause obtained from $Q$ by replacing every occurrence of a variable $X$ not belonging to $S$ by a new variable $X'$, and $\mathbf{w}$ denotes the set of new variables so constructed. The variables in $\mathbf{w}$ are the *existential variables* (as per Definition 1), while all the remaining ones are called *universal variables.* Thus for Example 2 we obtain the rule:

3. All the remaining arguments in a rule goal are labeled $u$ and called *universal*.

4. A term or a predicate $t$ appearing in the head of a rule is labeled with $\ell_1, ..., \ell_k$ where $\ell_i$ $(1 \leq i \leq k)$ is as follows: (i) $\ell_i = m$ if all occurrences of the $i$-th argument of $t$ in the recursive predicates of the body are labeled $m$, (ii) $\ell_i = e$ if it is a constant or if all occurrences of the $i$-th argument of $t$ in the recursive predicates of the body are labeled $e$, (iii) $\ell_i = u$ otherwise.

Consider, for instance, the min-goal: $min(C, (X, Y), p(f(X, Z), g(Y), C))$. Applying the definitions inductively, we obtain the following adornments $f^{ue}(X, Z), g^{u}(Y)$. Thus, the adorned predicate is: $p^{uum}(f^{ue}(X, Z), g^{u}(Y), C))$. Also observe that by the previous definitions, arguments of non-cost predicates are always adorned $\mathbf{u}$.

Fron now on, to simplify our discussion, we will assume that that at most one adornment exists for each predicate symbol in our min-cliques, even when this contains several min-rules. There is no loss of generality in this assumption, since every program can be put into this form by adding simple renaming rules. We will also assume, that each rule containing special atoms is of the following form ($\bar{X}$ and $\bar{Y}$ denote lists of terms):

$$p(\bar{X}, C) \leftarrow min(C, S, q(\bar{Y}, C))$$

where $S \subseteq var(\bar{Y})$ and $var(\bar{X}) \subseteq var(\bar{Y})$, with $var(\bar{X})$ denoting the set of variables appearing in $\bar{X}$. We further assume that each variable appears only once inside a predicate and that each predicate is defined either by some Horn rules (we call such a predicate a *Horn predicate*), or by one special rule (we call such predicates *non-Horn predicates*). Again, there is no loss of generality in this assumption, since every program can be put in this form by simple rewriting— this rewriting will be shown in Section 5. A programs satisfying the above conditions will be said to be in *standard form*. For programs in standard form, there is a unique adornment associated with each predicate in $P$, and thus, each argument of each predicate is labeled with either $e$, $m$ or $u$.

We can now give the inductive definition of congruence and the definition of set minimality with respect to the universally quantified arguments.

**Definition 7:** Let $P$ be a min-program. Two ground terms $t_1$ and $t_2$ will be said to be *congruent*, w.r.t. $P$, and we write $t_1 \cong t_2$, if (i) they coincide, or (ii) they have the same function symbol, the same number of arguments, and their respective arguments which are labeled $u$ are pairwise congruent.

Two ground atoms (facts) $f_1$ and $f_2$ are said to be congruent ($f_1 \cong f_2$) if they share the same predicate symbol, the same adornments, and their arguments labeled with $u$ are pairwise congruent. □

For instance, in Example 2, the adornment for $path$ is $uum$. Then, $path(a, b, 10) \cong path(a, b, 3)$ and $path(a, b, 10) \ncong path(a, c, 3)$ since their second arguments are different. However, if the adornment of $path$ were $uem$, then $path(a, b, 10) \cong path(a, b, 3) \cong path(a, c, 3)$.

$$sh\_path(X, Y, C) \leftarrow path(X, Y, C) \wedge \nexists(C') \; (path(X, Y', C') \wedge C' < C)$$

Since each predicate in $P$ has exactly one adornment, $\cong$ is indeed an equivalence relation (reflexive, symmetric, and transitive).

**Definition 8:** Let $P$ be a min-program, $B_P$ be the Herbrand base of $P$ and $I$ be a subset of $B_P$. If $I$ only contains cost atoms, and does not contain two distinct atoms that are congruent, then $I$ is said to be *cost-reduced.*

**Definition 9:** Let $L_1$ and $L_2$ be cost-reduced subsets of $B_P$. $L_1 \preceq L_2$ if for every $p_1 \in L_1$ there exists $p_2 \in L_2$ such that $p_1 = p_2$ or $p_1 \cong p_2$ and $cost(p_1) \leq cost(p_2)$. □

**Definition 10:** Let $P$ be a program with Herbrand base $B_P$. Let $I_1$, $I_2$, $L_1$ and $L_2$ be subsets of $B_P$. Then we say that $(I_1, L_1)$ is subsumed by $(I_2, L_2)$, and write $(I_1, L_1) \sqsubseteq (I_2, L_2)$, if $I_1 \subseteq I_2$ and $L_1 \preceq L_2$. □

The relationship $\preceq$ defined above is reflexive, transitive and antisymmetric for cost-reduced subsets, for which it defines a partial order. Therefore, the subsumption relation $\sqsubseteq$, defines a partial order on pairs, where the first component is a subset of Herbrand base $B_P$ and the second is a cost-reduced subset of the same. Thus, we say that $(I_1, L_1) \in (2^{B_P} \times 2^{B_P})$ is a *minimal* element of $(2^{B_P} \times 2^{B_P})$ when there is no $(I_2, L_2) \in (2^{B_P} \times 2^{B_P})$, with $(I_2, L_2) \sqsubseteq (I_1, L_1)$.

With $I$ and $L$ subsets of $B_P$, we now define:

$$I \setminus L \;=\; \{\; x \in I \mid \;\; \not\exists y \in L, x \cong y \;\}.$$

**Definition 11:**

Given a set of atoms $S$ we denote with $S_{cost}$ the set of cost-atoms of $S$. Moreover:

$$min(S) = \{x \in S_{cost} \mid \not\exists y \in S_{cost} \; s.t. \; x \cong y \; and \; cost(y) < cost(x)\}$$

$$least(S) = \{x \in S_{cost} \mid \not\exists y \in S_{cost} \; s.t. \;\; cost(y) < cost(x)\}$$

□

Thus $min(S)$ consists of the minimal-cost atoms among congruent ones in $S$, while $least(I)$ contains those having least cost in $S$. By definition, $least(S)$, and $min(S)$ are cost-reduced interpretations. If $S$ does not contain any cost atom, then $least(S)$ and $min(S)$ are empty.

### 3.2 Greedy Alternating Operator

We can now show that uniformly cost-monotonic min-programs can be evaluated efficiently using a procedure derived by combining the procedure described in [19] with the no-backtracking improvement which follows from Dijkstra's algorithm. 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 [9]. The operator $U_P$

takes two arguments, $I$ and $L$, which are interpretations of $P$ with $L \subseteq I$. $I$ is used to build up the stable model while $L$ holds the least cost 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 ($G_P$).

**Definition 12:** *Greedy Alternating Operator:*

Let $P$ be a min-program. We define three operators $G_P, U_P$ and $V_P$ from $2^{B_P} \times 2^{B_P} \rightarrow 2^{B_P} \times 2^{B_P}$ as follows:

$$\begin{aligned} V_P(I,L) &= (I \cup T_H(I), L) \\ G_P(I,L) &= (I \cup least(B), L \cup least(B)) \\ U_P(I,L) &= G_P(V_P(I,L)) \end{aligned}$$

where $B = (I \cup T_N(I)) \setminus L$ □

Intuitively, $L$ contains the set of minimal cost atoms produced so far. The basic step in the greedy operator chooses from $I \cup T_N(I)$ those atoms that (1) are not congruent to elements in $L$ and (2) have the least cost among all such elements (more than one facts could share such a cost).

There is a one-to-one correspondence between the minimal fixpoints of $U_P$ and the minimal models of $P$:

**Theorem 2:** *Let $P$ be a cost-monotonic min-program. Then, $M$ is a minimal model for $P$ if and only if $(M, min(M))$ is a minimal fixpoint for $U_P$.*

**Proof**: We begin by observing that $(M, L) = U_P(M, L)$ iff (i) $T_H(M) \subseteq M$ and (ii) $M_{cost} \setminus L = \emptyset$, with $M_{cost}$ denoting the set of cost atoms in $M$. Moreover if $(M, L)$ is a minimal fixpoint for $U_P$, then $L = min(M)$. (Indeed, if $(M, L)$ satisfies both (i) and (ii), so does $(M, min(M))$. Since, $M_{cost} \setminus L$ is empty, for each atom $x \in min(M)$ there exists an atom $y \in L \subseteq M$ such that $y \cong x$ and $cost(y) \geq cost(x)$—thus $min(M) \preceq L$, a contradiction).

Now, if $M$ is a model for $P$, then $T_H(M) \subseteq M$; so $(M, min(M))$ is a fixpoint for $U_P$. Vice versa, let $(M, min(M))$ be a fixpoint for $U_P$. Then, $M \subseteq T_H(M)$, and $T_N(M) \subseteq min(M) \subseteq M$. Thus $M$ is a model. It then follows that $M$ is a minimal model for $P$ iff $(M, min(M))$ is a minimal fixpoint for $U_P$. □

Thus, Theorem 2 characterizes the semantics of cost-monotonic min-programs, as the minimal fixpoints of the greedy operator $U_P$, much in the same way as the semantics of Horn clauses is characterized using the least fixpoint of $T_P$. We can now turn to the operational semantics and show how the successive powers of this operator compute the stable model of the original program.

**Theorem 3:** (Soundness.) *Let $P$ be a uniformly cost-monotonic min-program with stable model $M$. If $U_P^{\infty}(\emptyset, \emptyset) = (I, min(I))$, then $I \subseteq M$.*

**Proof:** Let $M$ be the unique stable model for the weakly stratified program $foe(P)$, and let $ground_M(foe(P))$ be the program constructed from the stability transformation. Say that $(I^n, L^n) = U_P^n(\emptyset, \emptyset)$ with $I^n \subseteq M$. But then, $T_H(I^n) \subseteq M$ since all the instance the Horn rules used to compute this step are in $ground_M(foe(P))$; also, $T_N(least(I^n \setminus L^n)) \subseteq M$ since all the instances of the non-Horn rules of $foe(P)$, used to compute this step are in $ground_M(foe(P))$ (negative goals deleted). If $least(I \cup T_N(I)) \setminus L) \in M$, our inductive proof is completed, since $I^0 \subset M$. Otherwise, suppose that there exists a $q \in least(I \cup T_N(I)) \setminus L)$ and $q \notin M$. Since $q \notin M$, $q \notin min(M)$ and $M$ must contain a $q' \cong q$ such that $cost(q') \leq cost(q)$. Now, in the connection graph of $ground_M(foe(P))$ there must be some path leading from $I^n$ to $q'$: consider an edge of this path with source node in $I^n$ and end node $p \notin I^n$. Because of the uniform min-monotonicity assumption $cost(p) \leq cost(q') < cost(q)$. This contradicts the assumption that $q$ is the least-cost atom among those which have no congruent atom in $L$. □

In general, a transfinite iteration on $U_P$ is needed to compute the stable model. For most programs of practical interest, however, a computation to the first ordinal suffices. In particular, this is the case of programs, such as those we have considered so far, where the cost function is computed by adding the costs of the arcs in a graph, where such costs are assumed to be positive. These programs have the unbounded cost property that can be defined as follows.

**Definition 13:** A uniformly cost-monotonic min-program $P$ is said to have the *unbound cost property*, when there is no upperbound to the cost of the nodes in every infinite path, in the dependency graph of $ground_M(foe(P))$. □

In practice, one can simply check that the unbounded cost property holds for the interpreted connection graph of the given program $P$ (this is a subgraph of the connection graph of $ground_M(foe(P))$).

**Theorem 4:** *(* Completeness*) Let $M$ be the stable model of a uniformly cost-monotonic min-program $P$ having the cost-unboundness property. Then, $U_P^\infty(\emptyset, \emptyset) = (M, min(M))$.*

**Proof:** Let $U_P^n(\emptyset, \emptyset) = (I^n, L^n)$ for all finite $n$ and $n = \infty$. We first show that $U_P(U_P^\infty(\emptyset, \emptyset)) = U_P^\infty(\emptyset, \emptyset)$. By contradiction, if $least(I^\infty \setminus L^\infty) \neq \phi$, then there exists an $A \in least(I^\infty \setminus L^\infty)$. Hence, $A \in I^m$ for some finite $m$. Since $A$ does not contradict minima in $L^\infty$, we conclude that $A$ does not contradict minima in $L^n$ for every finite $n$. However, $A \notin L^n$ for every finite $n$ and hence for each $x \in L^n$, $cost(x) < cost(A)$. Hence the sequence of sets $\{L^{m+k}\}$ is a strictly growing sequence of sets w.r.t. $\subset$. $L^{m+k} \subseteq I^{m+k} \subseteq M$, by Theorem 3. Hence we have an infinite subset of $M$ that is bounded in cost, contradicting the assumption.

Thus $(I^\infty, L^\infty) = U_P^\infty(\emptyset, \emptyset)$ is a fixpoint for $U_P$. Now, let $M$ be the stable model for $P$. Then, $(M, min(M))$ is a minimal fixpoint, by Theorem 2, and $I^\infty \subseteq M$ by Theorem 3. Thus, $I^\infty = M$ must hold, otherwise $(M, min(M))$ is not a minimal fixpoint. □

The following example presents a trace of the fixpoint computation of a uniformly cost-monotonic min-program.

**Example 6:**

```
r(a,b).
p(a,0).
s(X,C) ← q(X,C).
p(Y,D) ← s(X,C),  r(X,Y),  D = C + 1.
p(Y,D) ← q(X,C),  r(X,Y),  D = C + 2.
q(X,C) ← min(C,(X),p(X,C)).
```

Let us step through the computations of the $U_P$ operator. The program consists of two cliques containing respectively the predicate $r$ and the predicates $p$, $q$ and $s$. The predicate $r$ is computed first. The min-clique containing the mutually recursive predicates $p$, $q$, $s$ is then computed as follows. Initially $I = \{r(a,b)\}$ and $L = \phi$. In the first step, after firing the Horn rules we obtain $I = \{p(a,0), r(a,b)\}$ with $L$ unchanged. We now fire the non-Horn rules and we obtain $T_N(I) = \{q(a,0)\}$. The set of cost tuples of $I \cup T_N(I)$ that do not conflict with minima in $L$ with least cost is added to both the sets $I$ and $L$. Thus, after the completion of the first iteration of $U_P$ we have

$I = \{\ r(a,b), p(a,0), q(a,0)\ \}$ and
$L = \{\ p(a,0), q(a,0)\ \}$

We now return to firing the Horn rules using the $V_P$ operator and obtain $L = \{p(a,0),$ $q(a,0)\}$ and $I = \{r(a,b),\ p(a,0),\ q(a,0),\ s(a,0),\ p(b,2)\}$. At this point, firing the non-Horn rules gives $T_N(I)$ and we select the set of least cost tuple from $(I \cup T_N(I)) \setminus L$ and obtain the set $\{s(a,0)\}$ which is added to both the sets $I$ and $L$. Note that $s(a,0)$ belongs to $I$ and then it is added only the set $L$. Hence at the end of the second iteration, the values for $I$ and $L$ are as follows:

$I = \{\ r(a,b), p(a,0), q(a,0), s(a,0), p(b,2)\ \}$ and
$L = \{\ p(a,0), q(a,0), s(a,0)\ \}$.

Thus, we fire the Horn rules again to obtain the new tuple $p(b,1)$ which is added to $I$. Next, we find the set in $least((I \cup T_N(I)) \setminus L) = \{p(b,1), q(b,1)\}$ which is included in $I$ and $L$. Hence at the end of the third iteration, we have

$I = \{r(a,b), p(a,0), q(a,0), s(a,0), p(b,2), p(b,1), q(b,1)\}$, and
$L = \{p(a,0), q(a,0), s(a,0), p(b,1), q(b,1)\}$.

The reader may verify that the algorithm goes through one more iteration to include the element $s(b,1)$ in both $I$ and $L$, and then, it reaches fixpoint. □

### 3.3 Alternating Programs

The greedy evaluation technique is quite general, and applies to programs with arbitrary functions symbols and interpreted functions. Programs of practical interest, such as those of our examples, obey more restrictive syntactic structures, which ensure that the greedy fixpoint computation terminates always.

**Definition 14:** A min-program $P$ is said to be *alternating* if $T_H^\infty = T_H^k$ where $k \leq$ the maximum number of Horn predicates in a strong component. If $k = 1$, then $P$ is said to be *strictly alternating.* □

This definition is clearly satisfied when the sub-program obtained by deletion of non-Horn rules is not recursive, or equivalently, if the cost graph obtained by deletion of the min-arcs is acyclic. Both examples 2 and 3 are strictly alternating.

In Definition 3, we have introduced the notion of cost arguments of a program $P$, by considering the fixpoint of the program $det(P)$ obtained by removing the `a_lesser` goals and rules. Now, consider a goal $val(e_1)\theta val(e_2)$ in a rule $r$; this will be called an *evaluable cost goal* when all the variables in $e_1$ and $e_2$ are cost variables—i.e., variables that occupy a cost argument of some predicate of $r$. All the equality goals in our examples are evaluable cost goals (e.g., goal `C = C1 + C2` of Example 4).

We now define the program $nocost(P)$ as the one obtained by removing from $det(P)$ all (i) evaluable cost goals and (ii) all cost arguments in the remaining goals. (The notion of $nocost(P)$ is similar to that of "reduced programs" in [11].)

**Definition 15:** A uniformly min cost-monotonic program, $P$, will be called a min-Datalog program, when $nocost(P)$ contains no interpreted function or function symbol. □

Thus, a *min-Datalog* programs is a min-program that is free of function symbols except for the interpreted functions used in the computation of the cost arguments.

All the examples considered so far, are min-Datalog programs. The notion of safe rules and safe programs is well-known in deductive databases [23]. For the purpose of this discussion, it can be characterized as a functional dependency that must hold from the the instantiated body of a rule to its head, as follows:

**Definition 16:** Let $P$ a program and $ground(P)$ denote its Herbrand instantiation. Then, each rule $r \in P$ such that $ground(P)$ does not contain two instances of $r$ with identical body will be called *safe*. A program is safe if all its rules are safe. □

While each stable model of a non-recursive safe program is finite, this not the case for recursive programs with function symbols. However:

**Theorem 5:** *Let $P$ be a uniformly cost-monotonic min-Datalog program with stable model $M_P$. If $P$ is safe and alternating $M_P$ is finite.*

**Proof:** Let $P$ be a uniformly cost-monotonic min-Datalog program with stable model $M$. Consider a min-rule, such as, $r : q(X, C) \leftarrow min(C, S, p(X, C))$. Since $P$ is min-Datalog,

the first argument of $q$, must belong to the database or the constants of $nocost(P)$: a finite set by the min-Datalog assumption. But, since this rule is safe we also conclude that there is only one $X$-value for value of $C$. Thus, the set of q-facts in $M$ is finite. Now, to compute the remaining predicates in $M$, one can fire the the remaining (Horn) rules till saturation. But since the program is alternating, saturation is reached in a finite number of steps. $\Box$
Therefore, all the programs considered so far have a finite stable model.

As a corollary of the last theorem we have that:

**Corollary 1:** *Let $P$ be a uniformly cost-monotonic min-Datalog program. If $P$ is alternating, and $M$ is its unique stable model, then the complexity of computing $M$ is polynomial in the size of the database.* $\Box$

For the programs of interest the greedy fixpoint algorithm computes in polynomial time. Furthermore, the performance of the algorithm can be further improved using symbolic finite differentiation techniques similar to those used to obtain the well-known semi-naive computation of deductive databases [23]. Toward that goal we now introduce some simplifications of the greedy alternating fixpoint operator.

**Lemma 1:** *If $A, B$ and $C$ denote interpretations of a min-program, then the following equalities hold:*

$$\begin{array}{rcl} (A \cup B) \setminus C & = & (A \setminus C) \cup (B \setminus C), \\ least(A \cup B) & = & least(least(A) \cup least(B)), \\ min(A \setminus B) & = & min(A) \setminus B, \\ A \setminus (B \cup C) & = & (A \setminus B) \setminus C. \end{array}$$

**Proof:** Follows directly from the definitions. $\Box$

Consider now the operator $G_P$ whose definition is

$$G_P(I, L) \ = \ (I \cup least((I \cup T_N(I)) \setminus L), \ L \cup least((I \cup T_N(I)) \setminus L))$$

The application of the above lemma gives the following equalities:

$$\begin{array}{rcl} least((I \cup T_N(I)) \setminus L) & = & least((I \setminus L) \cup (T_N(I) \setminus L)) = \\ & = & least(least(I \setminus L) \cup least(T_N(I) \setminus L)) \end{array}$$

For each tuple $q \in T_N(I)$ there is a tuple $p \in I$ such that $p$ and $q$ differ only in the predicate symbol and $q$ is derived from $p$ by firing a non-Horn rule, then we have

$$least(T_N(I) \setminus L) = T_N(least(I \setminus L))$$

and

$$least(least(I \setminus L) \cup least(T_N(I) \setminus L)) = least(I \setminus L) \cup T_N(least(I \setminus L))$$

Since we have also that $I \cup (least(I) \setminus L) = I$, we can, finally, rewrite our greedy alternating operator as follows:

$$\begin{array}{rcl} V_P(I, L) & = & (I \cup T_H(I), L) \\ G_P(I, L) & = & (I \cup T_N(A), L \cup A \cup T_N(A)) \\ U_P(I, L) & = & G_P(V_P(I, L)) \end{array}$$

where $A = least(I \setminus L)$

This simplified version, was also used in [7], and is the basis for the efficient implementation discussed next.

# 4 The Greedy Semi-Naive Implementation

An efficient implementation for the greedy algorithm can be achieved by a symbolic differentiation technique similar to that used in the computation of the semi-naive fixpoint [22]. In this section we present the semi-naive implementation of the greedy fixpoint algorithm and the *alternating semi-naive greedy fixpoint algorithm.* We show that the alternating semi-naive greedy fixpoint algorithm applied to (strictly) alternating programs mimics Dijkstra's algorithm and computes the single source shortest paths with the same complexity of the classical algorithm.

In order to perform a finite differentiation in the fixpoint computation, the semi-naive transformation draws a distinction between the facts derived in the last iteration from those derived previously. Thus, the $I$-facts and the $L$-facts will be represented as follows:

$$I \cup \delta_H I \cup \delta_N I \quad \text{and} \quad L \cup \delta L$$

where:

- $\delta_H I$ and $\delta_N I$ denote the set of $I$-facts produced in the last iteration, respectively, by $V_P$ and $G_P$; $I$ contains all those produced before that, and
- $\delta L$ denotes the set of $L$-facts produced in the last iteration (by $G_P$), and $L$ denotes those produced before that.

We can now write out the fixpoint computation of $U_P$ into the following procedure:

**Algorithm 1: Greedy Fixpoint**

$I := J := L := \phi$;
$\delta L := \delta_H I := \delta_N I := \phi$;
**repeat**
  **step 1:** $I := I \cup \delta_H I \cup \delta_N I$;
    $\delta_H I := T_H(I) - I$;
  **step 2:** $J := (I \cup \delta_H I) \setminus (L \cup \delta L)$;

$L := L \cup \delta L;$
$A := least(J);$
**step 3:** $\delta_N I := T_N(A);$
$\delta L := A \cup \delta_N I;$
**until** $\delta_H I \cup \delta_N I \cup \delta_L I = \phi.$

Thus, **step 1** performs the computation $V_P$, while the computation of $G_P$ has been broken into **step 2** and **step 3**.

## 4.1 Semi-Naive Greedy Fixpoint

The semi-naive approach starts by separating recursive rules from non-recursive ones (called *exit rules*) defining our recursive predicates. We assume that the predicates appearing in the body of the exit rules have already been computed. (The program is first divided into strongly connected components and computed, one component at time, following the partial order defined by the dependencies between the predicates appearing in it.) Let $T_E$, denote the immediate consequence operator for the exit rules; it easy to see that $T_E$ can produce new tuples only at the very first step of the iteration. Thus, it can be moved out of the iteration loop. We can now symbolically differentiate $T_H(I \cup \delta_H I \cup \delta_N I)$, according to the well-documented transformations, applied either directly on the recursive Horn Clauses [18], or to their relational algebra equivalent [22]. Therefore, let $T_H^\delta(\delta_H I \cup \delta_N I, I)$ denote the result of the differentiation; for linear rules, we have: $T_H^\delta(\delta_H I \cup \delta_N I, I) = T_H(\delta_H I \cup \delta_N I)$.

Current seminave fixpoint techniques, however, do not address the differentiation of the non-monotonic operator at **step 2**. This can be performed using the equalities between expressions of set of tuples defined in Lemma 1.

In particular, from Lemma 1 we have (let the operator $\setminus$ take precedence over $\cup$):

$$(I \cup \delta_H I) \setminus (L \cup \delta L) = I \setminus (L \cup \delta L) \ \cup \delta_H I \setminus (L \cup \delta L) = (I \setminus L) \setminus \delta L \cup \delta_H I \setminus (L \cup \delta L)$$

But, observe that $I \setminus L$ is equal to the old value of $J$, i.e., the value of $J$ before **step 2** is executed. Thus **step 2** can be re-written as follows:

**step 2:** $J := (J \setminus \delta L) \cup (\delta_H I \setminus (L \cup \delta L));$
$L := L \cup \delta L;$
$A := least(J);$

A further improvement is possible, after observing that since $J$ is only used in the computation of *least* it is more efficient to keep only the set $K = min(J)$. Thus, $K$ can be computed by constructing the partition of $J$ induced by the equivalence relation $\cong$ and taking the least elements in the various classes of such a partition. It now follows immediately that $least(J) = least(K)$ and for any two sets $A$ and $B$, $min(A \setminus B) = min(A) \setminus B$. The value of $K$ at the end of step 2 is given by

$$min((J \setminus \delta L) \cup (\delta_H I \setminus (L \cup \delta L)))$$

Since $min$ distributes over union, the above is equal to:

$$min(min(J \setminus \delta L) \cup \delta_H I \setminus (L \cup \delta L))$$

$$= min(min(J) \setminus \delta L \cup \delta_H I \setminus (L \cup \delta L))$$

Observe that $min(J)$ in the above expression coincides with the old value of $K$. Hence the following expression relates the old value of $K$ (before step 2) to the new value of $K$ (after step 2):

$$min(K \setminus \delta L \cup \delta_H I \setminus (L \cup \delta L))$$

From the algorithm, we note that $\delta L = least(K) \cup T_N(least(K))$. Hence, $K \setminus \delta L = K - \delta L$. Hence **step 2** can now be rewritten as:

**step 2:** $K := min((K - \delta L) \cup (\delta_H I \setminus (L \cup \delta L))$;
$L := L \cup \delta L$;
$A := least(K)$;

The semi-naive greedy fixpoint procedure thus derived is conducive to efficient implementation, within the architectural framework of deductive databases [27]. We can assume that sets of facts are implemented via index-based structures, whereby operations such as variable assignment and set membership can be performed in constant time.

To ensure the efficiency of **step 2**, both the operations, $\setminus$ and $least$ must be performed efficiently. The first operation can be supported efficiently using an hash index on the universally quantified arguments, per predicate symbol; the $least$ operation can be implemented using a priority queue. Thus, $K$ denotes the set of cost tuples stored as a priority queue. The priority queue can be implemented as a binary heap or as a Fibonacci heap. We implement the priority queue by means of a binary heap, although in many cases the use of Fibonacci heaps gives better worst case complexity. This is because operations on Fibonacci heaps have a large constant factor that, in practice, make binary heaps preferable to Fibonacci heaps. The cost of inserting an element or of extracting the least element from a binary heap $K$ is $O(log\ |K|)$. The operation

$$K := min((K - \delta L) \cup (\delta_H I \setminus (L \cup \delta L));$$

can be performed, by first computing $K := K - \delta L$, in time $O(|\delta L| \times log|K|)$. Then, for each $f \in \delta_H I$ the following two operations are performed:

1. check if there is a $f_1 \in (L \cup \delta L)$ with $f_1 \cong f$;
2. if no such fact exists, then insert $f$ into the queue $K$.

Assuming the availability of indices, (1) is a constant time operation. The merge operation in (2), begins with searching $K$ for a fact $f_1 \cong f$. If such a fact does not exist, then $f$ is inserted into $K$; otherwise, if $cost(f) < cost(f_1)$, then $f_1$ is deleted from $K$ and stored as a regular tuple in $I$ (remember that $K$ contains a subset of the tuples of $I$) and $f$ is inserted into $K$. Otherwise, $cost(f) \geq cost(f_1)$ and $f$ is stored as a regular tuple in $I$.

Computationally, inserting into $K$ has cost $O(log(|K|)$ (both extraction and insertion of an element into a priority queue have cost $O(log(|K|))$ and in **step 2**, this operation is performed $O(|\delta_H I|)$ times. The operations $L \cup \delta L$ and $least(K)$ have costs respectively $O(|\delta L|)$ and $O(|least(K)| \times log|K|)$ where $|least(K)|$ is the number of element of $K$ with least cost. Finally, the operation $T_N(A)$ has cost has $O(|least(K)|)$.

The cost of **step 1**, is problem-specific. Thus, we will now concentrate on linear strictly alternating programs, which are prevalent in applications.

### 4.2 Alternating Linear Programs

We now show that the greedy semi-naive fixpoint algorithm specialized for (strictly) alternating linear programs emulates Dijkstra's celebrated algorithm.

Each predicate $p$ defined by a non-Horn rule in terms of a predicate $cp$ contains only a subset of tuples of $cp$ and, thus, need not be stored explicitly. In particular $p$ contains the tuples of $cp$ that are in $L$. Furthermore, since the recursive Horn rules are linear, $I$ is no longer needed inside the $T_H^{\delta}$ operator and, since the program in alternating, the new value of $\delta_H I$ in step 1 can be computed by using only $\delta_N I$. The tuples of the minimized predicates not stored into the priority queue $K$ are no longer relevant for the computation of the answer. Hence, explicitly maintaining the set $I$ is no longer necessary. The resulting algorithm, where $T_H$ denotes the operator $T$ applied only to the recursive Horn rules, is the following:

**Algorithm 2: Semi-naive Greedy Fixpoint for Linear Alternating Programs**

$L := K := \phi$;
$\delta L := \delta_N I := \phi$;
$\delta_H I := T_E(\phi)$;
**repeat**
  **step 1:** $\delta_H I := T_H(\delta_N I) - I$;
  **step 2:** $K := min((K - \delta L) \cup (\delta_H I \setminus (L \cup \delta L)))$;
    $L := L \cup \delta L$;
    $A := least(K)$;
  **step 3:** $\delta_N I := T_N(A)$;
    $\delta L := A \cup \delta_N I$;
**until** $\delta_H I \cup \delta_N I \cup \delta L = \phi$

Consider now, for instance, the single-source shortest-path problem on a graph with $n$ nodes and $e$ edges:

**Example 7:** Single-source Shortest-Path: Dijkstra Algorithm (for query goal: `?ssh_p(X,C)`).

```
ssp(a, 0).
ssp(Y, C) ←      sssh_p(Z, C1),  arc(Z, Y, C2),  C = C1 + C2.
sssh_p(Y, C) ←  min(C, (Y),  ssp(Y, C)).
```

□

The above program is linear and strictly alternating and hence it can be computed using Algorithm 2.

We now study the complexity of Algorithm 3 on Example 7. For each fact, $path(a, -) \in \delta_N$ generated in **step 3**, a set of corresponding facts is generated at **step 1** of the successive iteration where it is included in $\delta_H I$: this set has cardinality $O(\sum_{path(a,-) \in \delta_N I} outdegree(a))$, and the cost of producing this set is also proportional to this cardinality. Thus, the complexity of **step 2** is proportional to

$$O(\textstyle\sum_{path(a,-) \in \delta_N I} (outdegree(a) + 1) \times log(n))).$$

At the termination of the algorithm, $n$ nodes such as $path(a, -)$ have been added to $I$. Hence, the complexity of the whole algorithm is

$$O(\textstyle\sum_{path(a,\_) \in \delta_N I} O((outdegree(a) + 1) \times log(n))) = O((e + n) \times log(n)).$$

But since $O((e+n) \times log(n)) = O(e \times log(n))$, the complexity of the semi-naive greedy fixpoint coincides with Dijkstra's algorithm for the single-source shortest path problem [6]. Our algorithm, however, is general and supports an efficient treatment of variety of graph problems, in the rigorous framework of non-monotonic logic, and with a greater flexibility of underlying storage structures. In particularly they are implementable on database relations, possibly using relational-algebra oriented operators. In this context, for example, a $B^+$-tree could be used instead of the priority queue.

The complexity of all pairs shortest paths of Example 2 can be computed in analogous way and is equal to $O(n^4)$ where $n$ is the number of nodes in the graph. Assuming that at each iteration only one tuple is moved from $K$ to $L$ then the algorithm terminates in $O(n^2)$ iterations and *step 1* is executed in time $O(n)$ and the number of new paths computed is $O(n)$. The number of elements in the priority queue is $O(n^2)$ and then the cost to store these tuples in the priority queue is $O(n \times log\ n^2) = O(n^2)$. The global cost is then $O(n^2(n^2 + n)) = O(n^4)$.

Note that if we repeat the classical Dijkstra's algorithm, with each node in the graph becoming a source node, we obtain a complexity of $O(n \times e \times log\ n)$. The difference is due to the fact that the we have a 'global' priority queue and the cost of inserting one element is $O(n)$ instead of $O(log\ n)$. The complexity of using Algorithm 2 on Example 7 using Fibonacci heap implementations gives a complexity of $O(n(n \times log\ n + e))$. This is the same as the complexity of the classical algorithm if we consider $O(e) = O(n^2)$.

We conclude this section by stressing that although the alternating condition may appear to be restrictive, most programs can be rewritten into strictly alternating programs. The technique for the rewriting of programs with min-predicates into equivalent strictly alternating programs is based on the propagation of the min-predicates into the recursive rules and is presented in the next section.

## 5 Propagation of Min-Predicates

The availability of efficient procedures for evaluating min-programs opens interesting opportunities for further research, applications and extensions of practical interest. In this section, we briefly explore the automatic transformation of "naive" programs such as the (potentially unsafe) one of of Example 1 into programs such as Example 2, which are query equivalent to the original one, but can instead be safely and efficiently computed using the greedy seminaive fixpoint algorithm previously discussed. Here, we provide only a brief and informal discussion of these transformations, whereas a detailed discussion would take several pages and is given in [8].

We use *adornments* to abbreviate minimized predicates. The algorithm for propagating adornments into recursive rules works in a way similar to that of magic sets, i.e., by taking as input a rule with an adorned head, and returning a new rule where the recursive body predicates are adorned too. However, the generation of modified rules is slightly more complex than in the magic sets method, since an adorned rule must also be split into a Horn portion and a non-Horn portion, thus producing a pair of adorned rules. Let us illustrate this by means of the following example that involves several occurrences of recursive predicates in the body of a recursive rule (non-linear rule).

**Example 8:** Shortest path using Floyd algorithm

$$\begin{array}{ll}
\mathtt{sh_nlp(X,Y,C)} \leftarrow & \mathtt{min(C,(X,Y),nlp(X,Y,C))}.\\
\mathtt{nlp(X,Y,C)} \leftarrow & \mathtt{arc(X,Y,C)}.\\
\mathtt{nlp(X,Y,C)} \leftarrow & \mathtt{nlp(X,Z,C1),\ \ nlp(Z,Y,C2),\ \ C = C1 + C2}.
\end{array}$$

Suppose that a query *?sh_nlp(X,Y,C)* is given. This query computes the shortest paths by post-selecting among the `nlp` facts those tuples with the least cost for each pair $X, Y$. But since $C1' \leq C1$ and $C2' \leq C2$ implies that $C1' + C2' \leq C1 + C2$, we conclude that a path $p$ which is the combination of two paths $p1$ and $p2$ has minimum cost only if $p1$ and $p2$ have minimum costs. Therefore, we can (1) propagate the minimality conditions into the recursive predicates using adornments, and (2) then use these adornments to rewrite the original program into an optimized version.

During phase (1) adornments are passed from the head of a rule to its goals, as in the magic set method. For instance the propagation of adornments for the last example can be depicted as follows:

$$\begin{array}{l}
sh_nlp(X,Y,C) \Rightarrow nlp^{uum}(X,Y,C).\\
nlp^{uum}(X,Y,C) \Rightarrow arc^{uum}(X,Y,C).\\
nlp^{uum}(X,Y,C) \Rightarrow nlp^{uum}(X,Z,C1),\quad nlp^{uum}(Z,Y,C2),\quad C = C1{+}C2.
\end{array}$$

The propagation of adornments has thus generated an adorned recursive predicate $nlp^{uum}$. We can now rewrite the original recursive program according to adornment `uum`, that prescribes that the computation of the recursive rule should be followed by the minimization of the third argument of the head. Given the syntactic restrictions we have assumed for min-expressions, the operation just described requires two rules which are cascaded through the introduction of a new cnlp predicate, as follows:

$$
\begin{array}{ll}
\mathtt{sh\_nlp(X,Y,C)} \leftarrow & \mathtt{nlp^{uum}(X,Y,C).} \\
\mathtt{cnlp^{uum}(X,Y,C)} \leftarrow & \mathtt{arc^{uum}(X,Y,C).} \\
\mathtt{cnlp^{uum}(X,Y,C)} \leftarrow & \mathtt{nlp^{uum}(X,Z,C1),\ \ nlp^{uum}(Z,Y,C2),\ \ C = C1 + C2.} \\
\mathtt{nlp^{uum}(X,Y,C)} \leftarrow & \mathtt{min(C,(X,Y),cnlp^{uum}(X,Y,C)).}
\end{array}
$$

The following min-rule for $\mathtt{arc}^{uum}$ would also be needed in general:

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

In this particular example however, $\mathtt{arc}^{uum}$ only contributes to $\mathtt{cnlp}^{uum}$ (second rule) which is minimized anyway. Thus, we can remove this last rule along with the adornment to arc in the second rule.

In general, it is possible to have different *minimizations* for the same predicate, i.e., different adornments, each denoting different quantifications for its arguments. Consider the following non-linear shortest path program (the third argument of *nlp* is intended as an additional label):

$$
\begin{array}{ll}
\mathtt{sh\_nlp(X,Y,Z,C)} \leftarrow & \mathtt{min(C,(X,Y),nlp(X,Y,Z,C)).} \\
\mathtt{nlp(X,Y,Z,C)} \leftarrow & \mathtt{arc(X,Y,Z,C).} \\
\mathtt{nlp(X,Y,Z,C)} \leftarrow & \mathtt{nlp(X,U,W,C1),\ \ nlp(W,Y,Z,C2),\ \ C = C1 + C2.}
\end{array}
$$

The first rule produces the adornment: $\mathtt{nlp}^{uuem}$. Then this adornment is propagated as follows from the the third rule:

$$
nlp^{uuem}(X,Y,Z,C) \Rightarrow nlp^{ueum}(X,U,W,C1),\ nlp^{uuem}(W,Y,Z,C2),\ C = C1+C2.
$$

Thus, we now have generated a rule where the first occurrence of *nlp* in the body is adorned with *ueum* while the second one will be adorned with *uuem*. In general, variables that are labeled $u$ in the head, or are shared by different cost goals, are labeled with $u$, while variables that are labeled $e$ in the head or are local to a goal are labeled $e$.

The original programs will thus be rewritten into two sets of the rules, the first set defining $nlp^{ueum}$ and the second one defining $nlp^{uuem}$:

$$
\begin{array}{ll}
\mathtt{sh\_nlp(X,Y,Z,C)} \leftarrow & \mathtt{nlp^{uuem}(X,Y,Z,C).} \\
\\
\mathtt{nlp^{uuem}(X,Y,Z,C)} \leftarrow & \mathtt{min(C,(X,Y),cnlp^{uuem}(X,Y,Z,C)).} \\
\mathtt{cnlp^{uuem}(X,Y,Z,C)} \leftarrow & \mathtt{arc^{uuem}(X,Y,Z,C).} \\
\mathtt{cnlp^{uuem}(X,Y,Z,C)} \leftarrow & \mathtt{nlp^{ueum}(X,U,W,C1),\ \ nlp^{uuem}(W,Y,Z,C2),\ \ C=C1+C2.} \\
\mathtt{arc^{uuem}(X,Y,Z,C)} \leftarrow & \mathtt{min(C,(X,Y),arc(X,Y,Z,C)).} \\
\\
\mathtt{nlp^{ueum}(X,Y,Z,C)} \leftarrow & \mathtt{min(C,(X,Y),cnlp^{ueum}(X,Y,Z,C)).} \\
\mathtt{cnlp^{ueum}(X,Y,Z,C)} \leftarrow & \mathtt{arc^{ueum}(X,Y,Z,C).} \\
\mathtt{cnlp^{ueum}(X,Y,Z,C)} \leftarrow & \mathtt{nlp^{ueum}(X,U,W,C1),\ \ nlp^{ueum}(W,Y,Z,C2),\ \ C=C1+C2.} \\
\mathtt{arc^{ueum}(X,Y,Z,C)} \leftarrow & \mathtt{min(C,(X,Z),arc(X,Y,Z,C)).}
\end{array}
$$

In general, observe that adornments can be propagated only when the function yielding the cost arguments has certain monotonicity properties. Consider, for instance, the adorned predicate $\mathtt{p}^{uum}$ defined by the following rule:

$$\mathtt{p(X,Y,C)} \leftarrow \ \mathtt{q(X,Z,C1),\ \ r(Z,Y,C2),\ \ C = C1 + (C2\ mod\ 5).}$$

In this case, $C$ is monotonic w.r.t. $C1$ but not $C2$. Thus, it is possible to propagate the adornment into the definition of $q$ but not into the definition of $r$. The rewritten program thus contains the adorned $q^{uum}$ and the unadorned $r$.

We present now a sketch of the algorithm for the propagation of adornments. The complete algorithm is lengthy and involves many details described in [8]. We first discuss the propagation of adornments inside rules and we then sketch the algorithm.

## 5.1 Propagation of Adornments

The algorithm for generating adornments starts from $q^\epsilon$, where $q$ denotes the query goal and $\epsilon$ denotes the empty adornment, and propagates the adornment into all the rules that define $q$. When new adorned predicates are obtained in this process, these become candidates for further propagation. This step is continued until no new adorned predicates are found. We use the two sets $S$ and $T$ storing respectively the adorned predicates whose (adorned) definition must be added to the program and the adorned predicates whose definition has been generated.

Consider, for instance, the program of Example 5 with the query $?sh\_path(X,Y,Z,C)$. Initially we have the following set of adorned predicate symbols

$$S = \{sh\_path^\epsilon\}$$

The predicate $sh\_path^\epsilon$ is now propagated, by generating the adorned definition of its defining rules, and the new adorned predicate $path^{uuem}$ is generated. We thus have the following set of adornments:

$$S = \{path^{uuem}\} \qquad T = \{sh\_path^\epsilon\}$$

The predicate $path^{uuem}$ is moved into $T$ and the adorned definition of $path$ w.r.t. $uuem$ is generated. The two new adorned predicates $arc^{uuem}$ and $path^{ueum}$ are generated. We have then:

$$S = \{arc^{uuem}, path^{ueum}\} \qquad T = \{sh\_path^{\epsilon}, path^{uuem}\}$$

Suppose to move $arc^{uuem}$ from $S$ to $T$, then the adorned definition of $arc$ w.r.t. $uuem$ is generated (it consists of only the special rule that defines $arc^{uuem}$ in terms of $arc$) and no new predicates are added to $S$.

Further optimizations are possible to reduce the number of recursive adorned definitions. Suppose that adornments $p^{um}$ and $p^{em}$ are both reachable from $q^{\epsilon}$. Hence, instead of introducing adorned rules for $p^{em}$ and for $p^{um}$, it is equivalent to introduce adorned rules for $p^{um}$ and define $p^{em}$, by using a special rule, as follows:

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

In general, more constrained predicates can be defined in terms of less constrained ones; this heuristic helps in reducing the number of recursively defined predicates [8].

### 5.2 Generation of Modified Rules

Let $p$ be a predicate in the program, defined by a set of rules $P$, and let $\alpha$ be an adornment of $p$. The case where either $P$ is empty (i.e., $p$ is a database predicate) or $P$ contains only one special min-rule will be discussed later. For now we assume that $P$ is not empty and contains several Horn rules. Then every rule $r$ in $P$ is replaced by a modified rule $r^{\alpha}$; furthermore a min rule with head $p^{\alpha}$ is also added to the modified program. $r^{\alpha}$ is obtained from $r$ by replacing each predicate in the body with the corresponding adorned predicate and by replacing the head predicate symbol $p$ with the surrogate predicate $cp^{\alpha}$, as illustrated by the previous examples. Finally, the additional min-rule is of the form:

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

where $X_i$ is the cost argument and $W$ is the set of variables specified as universally quantified by $\alpha$.

We can now discuss the two special cases. If $p$ is a database predicate we cannot propagate the adornment, and then, the set of adorned rules contains only the rule

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

The second special case occur when the predicate $p$ is defined by exactly one min-rule $r$, of the form

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

The adorned rule $r^{\alpha}$ is

$$\mathtt{p}^{\alpha}(..) \leftarrow \quad \mathtt{q}^{\beta}(..)$$

where variables in the body of $r^{\alpha}$ are adorned according to the grouped variables $W$. For example if the rule is:

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

and the adornment for $p$ is $uum$ then the adorned rule is:

$$\mathtt{p^{uum}(X,Y,C)} \leftarrow \quad \mathtt{q^{ueem}(X,Y,Z,C).}$$

The rewriting described in the last section can turn a non-terminating program into a safe one. For Example 1, for instance, a fixpoint procedure cannot terminate if the relation *arc* is cyclic, since for positive values of `C2`, there is an infinite number *path* facts. However, upon propagation, we get the following program.

**Example 9:** The Program of Example 1 transformed by the propagation of `min`.

$$\begin{array}{ll}
\mathtt{sh_path^{\epsilon}(X,Y,C)} \leftarrow & \mathtt{path^{uum}(X,Y,C).} \\
\mathtt{cpath^{uum}(X,Y,C)} \leftarrow & \mathtt{arc^{uum}(X,Y,C).} \\
\mathtt{cpath^{uum}(X,Y,C)} \leftarrow & \mathtt{path^{uum}(X,Z,C1),\ \ arc^{uum}(Z,Y,C2),\ \ C = C1 + C2.} \\
\mathtt{path^{uum}(X,Y,C)} \leftarrow & \mathtt{min(C,(X,Y),\ \ cpath^{uum}(X,Y,C)).} \\
\mathtt{arc^{uum}(X,Y,C)} \leftarrow & \mathtt{min(C,(X,Y),\ \ arc(X,Y,C)).}
\end{array}$$

□

Now, the greedy procedure applied to this program terminates even if there are cycles in the data. Hence we would like to determine which programs are effectively computable by using the greedy fixpoint procedure, after propagation of min.

# 6 Propagation of Constants

The magic sets method provides an efficient fixpoint based computation of Horn clauses in the presence of bound arguments in recursive predicates. This key technique can still be used for recursive rules containing min predicates, once the special binding-passing properties of non-Horn rules are taken into account. For example, consider the following program, defining a variation of the same-generation problem:

**Example 10:** Same generation program

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

$$\begin{array}{ll}
\mathtt{sg(X,Y,C)} \leftarrow & \mathtt{flat(X,Y,C).} \\
\mathtt{sg(X,Y,C)} \leftarrow & \mathtt{up(X,X1,C1),\ \ sg(X1,Y1,C2),\ \ down(Y1,Y,C3),\ \ C = C1 + C2 + C3.}
\end{array}$$

□

Here, a query $?q(a, b, 5)$ evaluates to true, iff $a$ and $b$ are of the same generation, have cost 5, and there does not exist a node $c$ of the same generation as $a$ that has cost less than 5. This query can be answered by computing all facts of $sg$ that have $a$ as their first argument. Thus the binding, on the first argument of $sg$ (i.e., $X$) can be propagated. However, the bindings on $Y$ and $C$ cannot be propagated, since the minimum of $C$ must be computed over all possible pairs $(Y, C)$ associated with the given $X$. This can be seen by rewriting our rule using negation,

$$\mathtt{q(X,Y,C)} \leftarrow \quad \mathtt{sg(X,Y,C)},\ \neg(\mathtt{sg(X,Y',C')},\ \mathtt{C'} < \mathtt{C}).$$

Then, $X$ is the only binding in the negated literal.

Hence the propagation of bindings for min-predicates is as follows: an argument $\theta$ is adorned as $b$ in the body only if it is adorned as $b$ in the head and all variables of $\theta$ appear as grouping variables in the minimized predicate. All other variables in the body are adorned as $f$. The adorned version of the previous program is:

$$\mathtt{q^{bbb}(X,Y,C)} \leftarrow \quad \mathtt{min(C,(X),\ \ sg^{bff}(X,Y,C))}.$$

$$\begin{array}{ll} \mathtt{sg^{bff}(X,Y,C)} \leftarrow & \mathtt{flat(X,Y,C)}. \\ \mathtt{sg^{bff}(X,Y,C)} \leftarrow & \mathtt{up(X,X1,C1), sg^{bff}(X1,Y1,C2), down(Y1,Y,C3),\ \ C = C1 + C2 + C3}. \end{array}$$

The magic-set rewritten program is obtained by application of the standard rules with the query $q^{bbb}(a, b, 5)$.

Consider now the program of Example 1 with the query $sh\_path(a, Y, C)$. In programs, such as this, there is a choice of either propagating first the bound arguments or the min-adornments. It can be shown that it is always better to do the former before the latter [8]. For the program in Example 1, for instance, the application of the magic-set method followed by the propagation of $min$ predicates gives the following program where we have associated two different adornments to the cost predicates:

$$\begin{array}{ll} \mathtt{magic\_sh\_path^{bff}(a)}. & \\ \mathtt{magic\_path^{bff}(X)} \leftarrow & \mathtt{magic\_sh\_path^{bff}(X)}. \\ & \\ \mathtt{sh\_path^{bff}(X,Y,C)} \leftarrow & \mathtt{magic\_sh\_path^{bff}(X),\ \ path^{bff/uum}(X,Y,C)}. \\ \mathtt{path^{bff/uum}(X,Y,C)} \leftarrow & \mathtt{min(C,(X,Y),cpath^{bff/uum}(X,Y,C)}. \\ & \\ \mathtt{cpath^{bff/uum}(X,Y,C)} \leftarrow & \mathtt{magic\_path^{bff}(X),\ \ arc^{uum}(X,Y,C)}. \\ \mathtt{cpath^{bff/uum}(X,Y,C)} \leftarrow & \mathtt{magic\_path^{bff}(X),} \\ & \mathtt{path^{bff/uum}(X,Z,C1),\ \ arc^{uum}(Z,Y,C2),\ \ C = C1 + C2}. \\ \mathtt{arc^{uum}(X,Y,C)} \leftarrow & \mathtt{min(C,(X,Y),arc(X,Y,C))}. \end{array}$$

Since the original program is linear, the magic-set rules are not recursive. Therefore, it still possible to push a constant $a$ into the rules defining the predicates $sh\_path^{bff}$ and $cpath^{bff/uum}$. The use of the well-known re-writing techniques for left-linear programs [23] yield the following revised definitions for the two predicates:

$$\mathtt{sh\_path}^{\mathtt{bff}}(\mathtt{a},\mathtt{Y},\mathtt{C}) \leftarrow \ \mathtt{magic\_sh\_path}^{\mathtt{bff}}(\mathtt{X}), \mathtt{path}^{\mathtt{bff/uum}}(\mathtt{a},\mathtt{Y},\mathtt{C}).$$

$$\mathtt{cpath}^{\mathtt{bff/uum}}(\mathtt{a},\mathtt{Y},\mathtt{C}) \leftarrow \ \mathtt{arc}^{\mathtt{uum}}(\mathtt{a},\mathtt{Y},\mathtt{C}).$$
$$\mathtt{cpath}^{\mathtt{bff/uum}}(\mathtt{a},\mathtt{Y},\mathtt{C}) \leftarrow \ \mathtt{path}^{\mathtt{bff/uum}}(\mathtt{a},\mathtt{Z},\mathtt{C1}), \mathtt{arc}^{\mathtt{uum}}(\mathtt{Z},\mathtt{Y},\mathtt{C2}), \mathtt{C} = \mathtt{C1} + \mathtt{C2}.$$

The complexity of this program for a graph with $n$ nodes and $e$ arcs is $O(e \times log\ n)$. By propagating the binding in the query we have obtained the same complexity of Dijkstra's algorithm.

The technique presented here applies only to linear programs (i.e., those where rules contain at most one recursive predicate in the body). It does not apply to non-linear programs since the generalized magic sets method [2] applied to non-linear programs could produce programs where the magic predicates have no cost attribute but are mutually recursive with the original cost predicates. These programs can not be treated directly by the techniques discussed in this paper. However, the basic algorithm can be modified to cover this case as well [8].

# 7 Conclusion

The study of logic-based constructs and program classes that combine declarative semantics with efficient computation is increasingly receiving an amount of attention that is commensurate to its importance. While a discussion of the many interesting contributions in this area would take us beyond the scope of this paper, we will briefly mention approaches that use aggregates in recursion [4, 13, 20, 5, 11, 17, 16, 21, 25]. In particular, programs with min and max predicates can be computed efficiently using the top-down approach with extensions tables [5]: the algorithms presented in [5] take advantage of the dynamic programming property of shortest path and the ability of extension tables to store global information to provide an approximating evaluation that prunes the search space by ignoring suboptimal goals.

Within the bottom-up framework, the techniques presented in [4, 13] address general aggregate predicates (i.e., including cardinality and sum). For extrema predicates, an approach for evaluating min-programs is proposed in [20] using a special operator that does not require monotonicity w.r.t. cost. This operator, however, requires that the program must be stratified with respect to the aggregate predicates. The designers of CORAL have paid much attention to the efficient execution of programs with aggregates, and proposed the use of control annotations [16] to ensure efficient execution of programs such as the shortest path computation. An alternative approach proposed in the literature is that of achieving monotonicity for some aggregates (including extrema) by using lattices more general than that defined by set-containment [17]. Even with the extensions and improvements proposed in [25], though, several open issues must be solved before this promising approach can be used in the actual execution and compilation of logic programs [25]. On the other hand, several of these programs become containment-monotonic, and thus expressible efficiently in CORAL with control annotations, if multisets are used along with valid semantics for aggregation [21].

In this paper, we have restricted our attention to extrema predicates, and derived a greedy fixpoint algorithm that are computationally very efficiently. In fact, the greedy fixpoint supports the seminaive fixpoint improvement, and yields Dijkstra's algorithm on shortest path problems. Thus we have presented a method where the design of certain greedy algorithms follow from the Datalog formulation of the problem and the re-writing techniques commonly used in the implementation of Datalog [23]. This leads to solutions that are different from other bottom-up approaches previously discussed, and quite different from the top-down method for approximating effective evaluation with extension-tables discussed in [5]. The cost-monotonicity conditions used here are more general than the existence of semirings, and detection of cost-monotonicity appears easier than detection of semirings. While we have only considered programs with one cost attribute and containing either *min* or *max* predicates, a generalization to programs containing both *min* and *max* predicates and more than one cost attribute requires only simple extensions of the theory here presented.

This paper leaves many interesting problems open for further research and refinement. For instance, a precise characterization is needed for the computational complexity of the techniques so far proposed on well-known graph problems (for alternative formulations using *min* and *max* predicates vis a vis the complexity of classical procedural formulations, such as the Prim or Floyd algorithms). Another problem deserving investigation beyond the brief discussion of this paper is that of migrating *min* and *max* predicates into recursion—and the problem of combining said migration with constant-pushing methods, such as the generalized magic set method, while addressing issuses of possible loss of monotonicity and stratification caused by these transformations.

**Acknowledgments.** The authors are very grateful to the referees for their suggestions that produced significant improvements in the original manuscript.

## References

[1] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan-Kaufman, Los Altos, CA, 1988.

[2] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings Sixth ACM Symposium on Principles of Database Systems*, pages 269–283, 1987.

[3] C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[4] M.P. Consens and A.O. Mendelzon. Low complexity aggregation in graphlog and datalog. In *Proceedings of the third International Conference on Database Theory*, 1990.

[5] S. W. Dietrich. Shortest Path by Approximation in Logic Programs. *ACM Letters on Programming Languages and Systems* Vol 1, No, 2, pages 119–137, June 1992.

[6] E.W. Dijkstra. A note on two problems in connection with graph. *Numerische Mathematik*, 1:269–271, 1959.

[7] S. Ganguly, S. Greco, and C. Zaniolo. Minimum and Maximum Predicates in Logic Programming. *Proceedings Tenth ACM Symposium on Principles of Database Systems*, pages 154–163, 1991.

[8] S. Ganguly, S. Greco, and C. Zaniolo. On the Propagation of Extrema into Recursive Predicates UCLA Technical Report, CS Department, 1993.

[9] M. Gelfond and V. Lifschitz. The stable model semantics of logic programming. In *Proceedings of the Fifth Intern. Conference on Logic Programming*, pages 1070–1080, 1988.

[10] D. Kemp and P. Stuckey. Semantics of Logic Programs with Aggregates. In *Proceedings of the 1991 Intern. Symposium on Logic Programming*, pages 387–341, 1991.

[11] A.Lefebvre Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases, In *Proceedings of Int. Conference on Fifth Generation Computer Systems (FGCS)*, pages 915–925, Tokio, Japan, June 1992.

[12] J.W. Lloyd. *Foundation of Logic programming.* Springer-Verlag, 2nd ed. 1987.

[13] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the 16th Conference on Very Large Data Bases*, pages 264–277, 1990.

[14] S. A. Naqvi, S. Tsur *"A Logical Language for Data and Knowledge Bases"*, W. H. Freeman, 1989.

[15] H. Przymusinska and T.C. Przymusinski. Weakly perfect model semantics for logic programs. In *Proceedings of the Fifth Intern. Conference on Logic Programming*, pages 1106–1120, 1988.

[16] R. Ramakrishnan and S. Sudarshan CORAL: Control, Relations and Logic. In *Proceedings of the 18th Conference on Very Large Data Bases*, pages 283-250, Vancouver, British Columbia, 1992.

[17] K. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *Proceedings Eleventh ACM Symposium on Principles of Database Systems*, pages 161-171, 1992.

[18] D. Saccà and C. Zaniolo. Differential fixpoint methods and stratification of logic programs. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1988.

[19] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217, 1990.

[20] S. Sudarshan and R. Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the 17th Conference on Very Large Data Bases*, 1991.

[21] S. Sudarshan, D. Srivastava, R. Ramakrishnan and C. Beeri, Extending Well-Founded and Valid Semantics for Aggregation, In *Proceedings of the International Logic Programming Symposium*, pages 591-608, Vancouver, British Columbia, 1993.

[22] J. Ullman. *Principles of Data and Knowledge-Base Systems*, volume 1. Computer Science Press, New York, 1988.

[23] J. Ullman. *Principles of Data and Knowledge-Base Systems*, volume 2. Computer Science Press, New York, 1989.

[24] A. Van Gelder. The alternating fixpoint of logic programming with negation. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.

[25] A. Van Gelder. Foundation of Aggregation in Deductive Databases In *Proceedings of the 3rd International Conference on Deductive and O-0 Databases*, Phoenix, AZ, 1993.

[26] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[27] C. Zaniolo. Design and implementation of a logic based language for data intensive applications. In *Proceedings of the International Conference on Logic Programming*, 1988.