Minimum and Maximum Predicates in Logic Programming

Sumit Ganguly
Department of Computer Science
University of Texas at Austin
Austin, TX 78712
sumit@mcc.com

Sergio Greco*
Dipartimento di Sistemi
Universita' della Calabria
87030 Rende, Italy
sergio@mcc.com

Carlo Zaniolo
MCC
Balcones Center Drive, 3500
Austin, TX 78759
carlo@mcc.com

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 when certain monotonicity conditions hold then (1) there exists a total well-founded model for these programs containing negation, (2) this model can be computed efficiently using a procedure called greedy fixpoint, and (3) the original program can be rewritten into a more efficient one 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.*

1 Introduction

Current research on deductive databases and logic programming strives to support a declarative high-level problem formulation without losing the levels of efficiency obtainable by careful programming in an imperative language. In this respect, a particularly difficult challenge is posed by problems, such as finding the shortest path in a graph, that can be viewed as a generalization of the graph-closure problem. While current deductive databases deal very well with closures, they cannot even approach the elegance and efficiency of, say, Dijkstra's algorithm on the shortest path problem. In this paper we explicitly introduce *min* and *max* predicates —meta-level constructs with a first order semantics— allowing a declarative specification of such programs, and develop efficient techniques for implementing these programs.

Example 1, below, gives a declarative formulation of the shortest path problem. Given a directed graph represented as a base relation arc, 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(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:

```
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. □
```

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:

```
sh_path(X,Y,C) ← path(X,Y,C),
                 ¬(path(X,Y,C1),C1 < C)).
```

This has formal semantics, inasmuch as the negated conjunct in parenthesis can be defined by a new predicate, yielding a stratified program [1,11]. 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. These problems can be solved by observing that all minimum paths of length $n+1$ can be generated from the minimum paths of length n. Thus the min predicate can be pushed

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-430-9/91/0005/0154 $1.50

into recursion, in such a way that the generation of new paths is interleaved with the computation of shortest paths, yielding the following program:

Example 2:

```
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 Example 2 using negation, i.e., by rewriting it 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 two recent works [2,5]. 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.

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 [6], and therefore have a total well-founded model [12,10] which coincides with their unique stable model [4]. In Section 3 we introduce the *greedy fixpoint* procedure, which is based on Dijktra's shortest path algorithm [3], and can be used for the efficient computation of monotonic programs—we prove that it is sound and, if not transfinite, complete. In Section 4 we examine the problem of taking a program 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 done, the greedy fixpoint procedure can then be used to efficiently compute the transformed program. In Section 5 we consider the problem of pushing the min predicate in combination with the magic set method (constant pushing). The last section presents an overview of related approaches and issues for further research.

2 Syntax and Semantics

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 including two predicate symbols d(unary) and $<_d$(binary). We also assume that there is a Herbrand interpretation of this alphabet such that the interpretation of d coincides with the Herbrand Universe of the alphabet and the interpretation of $<_d$ is a total order on d. This alphabet together with the Herbrand interpretation is called a *cost domain*. The predicate symbols of this alphabet cannot be redefined by other means.

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

1. Q is an atom of first order logic.
2. S is a set of variables, called grouping variables, and Y is a variable such that $Y \notin S$. All variables in $S \cup \{Y\}$ appear in Q.
3. The variable Y is constrained to take values only from the cost domain d. □

Definition 2: A *min atom* is either a special atom or an atom of first order logic. A *min literal* is either a min atom or its negation. □

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 there is exactly one positive literal in the clause and this literal is an atom of first order logic i.e., is not a special literal. A *ruleset* R is a finite set of definite min clauses. A *min program* is a pair (R, C) where R is a ruleset and C is a cost domain. Both examples 1 and 2 are min program.

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 $min(Y, S, Q)$ by the following clause in first order logic:

$$Q \wedge d(Y) \wedge \neg(Q' \wedge d(Y') \wedge Y' <_d Y))$$

where

- Q' is the clause obtained from Q by replacing every occurrence of a variable W not belonging to S by a new variable W'.
- The predicates d and $<_d$ are assumed to be preinterpreted in the cost domain.

Example 3: Let F be the min atom $min(C,(X),path(X,Y,C))$. Then $foe(F)$ is:

$$path(X,Y,C) \wedge d(C) \wedge \neg(path(X,Y',C') \wedge d(C') \wedge (C' <_d C))$$ □

Definition 3: Given a min program P we will say that I is an interpretation of P if and only if it is an interpretation of $foe(P)$. □

Observe that such an interpretation will contain totally ordered domains, an ordering predicate, and possibly interpreted functions such as arithmetic operators. In addition we assume the existence of constraints on cost values of certain predicates. For instance, for Example 2 we will assume that arc is a database predicate whose third argument cannot be negative. Also, by definition, a rule of P is true iff the corresponding rule of $foe(P)$ is true, and M is a model for P iff it is a model for $foe(P)$.

For simplicity of discussion we will assume that each rule containing special atoms is of the following form:

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

There is no loss of generality in this assumption, since every program can be put in this form by simple rewriting.

An *instantiation* of a rule r in P is a ground instance of r that is obtained by replacing the variables of r with ground terms from the Herbrand Universe of P. An *interpreted instantiation* of r is an instantiation in which each goal corresponding to an interpreted predicate is true, and all the constraints on cost values of predicates hold. The interpreted instantiation of program P, denoted P_I, is simply the set of all interpreted instantiations of all rules of P. For instance, no interpreted instantiation of the second rule in Example 2 can contain a goal, say `1 = 1+2`, nor one `arc(a,b, -5)`.

The *interpreted dependency graph* of a program P, denoted G_P, is a directed graph whose nodes are ground atoms of $foe(P)$. There is an arc from a node A to another node B if there is an interpreted instantiation of some rule r in P such that A appears in the body of the instantiation and B appears in the head. We restrict our attention to the class of min programs where each predicate has at most one distinguished argument called the *cost argument*. Predicates with cost arguments are called *cost predicates*.

Definition 4: Let P be a min program with interpreted dependency graph G_P. P is said to be *(cost) monotonic* if for every pair (A, B) of ground cost predicates sharing the same predicate symbol, there is a path from A to B in G only if $cost(B) \geq cost(A)$ and if the path contains a negated arc then the condition $cost(B) > cost(A)$ holds. □

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

Proof: Let P be a min program. Let P_H be the Herbrand instantiation of $foe(P)$. The dependency relations $\prec$ and $\preceq$ between ground atoms of P_H are defined by a) $A \preceq B$ if there is a path from A to B; b) $A \prec B$ if there is a path from A to B passing through a negative edge. The predicate $<$ in $foe(P)$ is preinterpreted, and we can consider to belong to the level 0. The model for the preinterpreted predicate $<$ (denoted M_0) is the set of ground predicates $A < B$ such that A is less than B. By application of the Davis-Putnan reduction we obtain the program P_H^1 where each (ground) rule of the form $p(X,C) \leftarrow q(X,C), \neg(q(Y,C'), C' < C)$ is substituted by the rule $p(X,C) \leftarrow q(X,C)$ if $C' \geq C$ or by the rule $p(X,C) \leftarrow q(X,C), \neg q(Y,C')$ if $C' < C$.
The priority relation $\prec$ on the program P_H^1 is transitive. To prove that is anti-symmetric, let $A \prec B$. Then there exist a A' and a B' (not necessarily distinct from A and B) such that $A \preceq A' \prec B' \preceq B$ and $cost(A') < cost(B')$. 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 interpreted instantiation of our program P (i.e., P_H^1) is locally stratified. Therefore P is weakly stratified. □

Although the problem of determining whether a program is monotonic is undecidable, simple sufficient conditions can be found. We now define the notion of *uniformly monotonic* program using cost graphs. Predicates which appear inside min predicates are called min predicates.

The *cost graph* CG_P of a program P is a directed graph whose nodes are pairs p/k where p is a predicate symbol in P and k is the position of the cost argument of p ($p/0$ denotes a predicates that do not have associated a cost attribute). There is an arc from a node q/h to a node p/k with label r if there is an interpreted instantiation of 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/h to p/k labeled with rule r is marked with $\geq$ if the arc is derived from an Horn rule and the cost argument (k-th argument of) p is $\geq$ to the cost argument (h-argument of q) for every instance of r in G_P. If the arc is derived from a non Horn rule then it is marked with `min`.

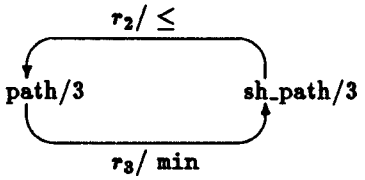
Definition 5: Let SC be a strong component in CG_P. We say that SC is uniformly monotonic if a cost argument can be identified for each node in SC and such that all arcs in SC are marked with $\geq$ or *min*. A program is said to be *uniformly monotonic* if all its strong components that contain min predicates are uniformly monotonic. □

Theorem 2: *Uniformly monotonic min programs are monotonic.*

Proof: Let P be a uniformly monotonic min program. The interpreted dependency graph for $foe(P)$ contains negated arcs from a predicate A to a predicate B only if the condition $cost(A) < cost(B)$ holds. This implies that P is also monotonic. □

Thus, uniformly monotonic programs are weakly stratified and hence their well-founded models are total. Moreover, this well founded model also coincides with the unique stable model and defines the formal semantics of such programs.

Sufficient conditions for uniform monotonicity are easily defined for most situations of practical interest, with the help of a cost graph. The cost graph for the program of Example 2 is as follows:



In practice, the construction of the cost graph would begin by drawing the dependencies between recursive predicate names in strong components involving min predicates. In our example, we have two rules and two arcs, one from *path* to *sh_path* and the other vice versa. The next task is that of identifying the cost arguments in all the predicates. This can be done by propagating the cost arguments from the predicates appearing inside a min predicate into the remaining ones. In Example 2, the third rule identifies the third argument in *path* as a cost argument; furthermore the value of this argument is copied into the third argument of *sh_path*. We depict the mapping defined by the third rule by a directed arc from *path*/3 to *sh_path*/3. Then, the second rule closes the loop by propagating *sh_path*/3 back to *path*/3, via an arithmetic expression. Having thus identified the cost arguments, one can now compare their values. The third rule is a non Horn rule and then the arc is marked with *min*. Notice that in the dependency graph associated to the program for each instantiation of the rule there are one positive arc from a predicate $path(a,b,c)$ to $sh\_path(a,b,c)$ and a negated arc from $path(a,b,c')$ to $sh\_path(a,b,c)$ such that $c > c'$. For the second rule, the fact that constraint $C2 \geq 0$ holds, implies that $C \geq C1$. (It is reasonable to expect that an intelligent compiler will be able to perform this simple analysis on arithmetic. But, the introduction of an explicit additional goal $C \geq C1$ would also be possible to make the monotonic cost constraint detectable by even an 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. The clauses of a min 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 [6,12].

The *Backtracking Fixpoint* procedure for computing stable models presented in [7] can be summarized as follows:

Step1. Assume some negative facts and, using these, fire the non Horn clauses, giving some new positive facts,

Step2. Use the newly derived facts to fire all the Horn rules until saturation (i.e. compute T_H^∞),

Step3. If there is no contradiction between the negative facts assumed in Step 1 and the positive ones derived in Step 2, then go back to Step 1, otherwise undo (backtrack) Step 2 and Step 1, then resume from Step 1 with different assumptions.

One of the chief sources of inefficiency in the above procedure is the backtracking step required in Step 3. This backtracking step 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 will never be contradicted by positive facts later generated in Step 2. The contribution of this paper is to show that such a decision can be made judiciously using criteria similar to those used by Dijkstra's algorithm. We illustrate this below by using Example 2:

```
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) ← path(X,Y,C),
                 ¬(path(X,Y,C1),C1 < C)).
```

Note that for this example $T_H = T_H^\infty$. In each iteration of Dijkstra's algorithm, we do the following two things. First, we use the Horn rules to obtain new facts $path(a,b,c)$. Second, we add a new tuple of the form $sh\_path(a,b,c)$ to a set L provided that (1) L does not contain a tuple $sh\_path(a,b,\_)$, and (2) that this tuple has the smallest cost among all the facts satisfying (1). The fact $sh\_path(a,b,c)$ can be added correctly only under the assumption that the final model, does not literals in the following set: $\{path(a,b,X) | X < c\}$. The basic idea behind Dijkstra's algorithm is that this is a safe assumption, provided that $sh\_path(a,b,c)$ denotes a path with current minimal cost. Therefore, Dijkstra's algorithm computes the stable model of our logic program using the backtracking fixpoint procedure—however, the need for backtracking is avoided since only the *least cost paths are added* at each step.

In this paper we show that 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 [7] 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 [4]. 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 which do not *dominate* facts that are already in L.

For example let p(X,C) ← q(X,C), ¬(q(X,D), D<C) be our min rule, and consider the atoms $p(a,0), p(a,1), p(b,2)$. Then, we say that $p(a,1)$ dominates $p(a,0)$ but $p(b,2)$ doesn't dominate any tuple. However, if the rule was instead p(X,C) ← p(X,C), ¬(p(Y,D),D<C), then $p(a,1)$ still dominates $p(a,0)$, but $p(b,2)$ dominates both $p(a,0)$ and $p(a,1)$. More formally, given a set I, we will denote by $Dom(L)$ the largest subset W of I containing L such that $T_N(W) = L$. Every member of $Dom(L)$ either belongs to L or dominates some tuple in L. Hence, the set of I facts that do not contradict minima (or do not dominate elements) already in L is $I - Dom(L)$. We define $least(I, L)$ to be the set of least-cost elements in $I - Dom(L)$.

Definition 6: *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:

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

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).

We can now state the soundness and completeness of our greedy fixpoint procedure. Following practice in lattice theory we define U_P^n for every finite n as follows: $U_P^0 = (\phi, \phi)$ and $U_P^n = U_P^n(\phi,\phi) = U_P(U_P^{n-1})$ if $n > 0$. For $n = \infty$, U_P^∞ is defined to be $\cup_{n\ finite} U_P^n$. 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 $\{cost(A) | A \in S\}$.

Theorem 3: *Let P be a uniformly monotonic min program with unique stable model M_P and $U_P^\infty = (I^\infty, L^\infty)$. Then $I^\infty \subseteq M_P$.*

Proof: Let $U_P^n = (I^n, L^n)$. We prove by induction that (i) $I^n \subset M_P$ and (ii) $\forall A \in M_P (cost(A) < cost(L^n) \Rightarrow A \in I^n)$. The base case for $n = 0$ is trivially true. Suppose the proposition holds for n. Clearly $T_H(I^n) \subset M_P$. Let $J = I^n \cup T_H(I^n)$. Let $S = least(J, L^n)$. Suppose that $cost(S) > cost(L^n)$. Let $T = min\{B \in M_P | cost(B) > cost(L^n)\}$. Hence there exists an element of T that is derived using ground atoms from I^n and a Horn rule from P. Thus, $T \cap S \neq \phi$. Hence, in all cases, (i.e., $cost(S) \geq cost(L^n)$) we have that for each A in M_P such that $cost(A) < cost(S)$ implies $A \in I^n$. Hence if S satisfies a formula $min(c, W, p(\bar{b}, c))$ then so does M_P. Thus $T_N(S) \subset M_P$. $U_P^{n+1} = (I^n \cup T_H(I^n) \cup T_N(S), L^n \cup S)$. Hence $I^n \cup T_H(I^n) \cup T_N(S) \subset M_P$ and condition (i) is satisfied. Since $cost(L^n \cup S) = cost(S) = cost(T_N(S))$ hence (ii) is satisfied as well. Let $U_P^\infty = (I^\infty, L^\infty)$. If $A \in I^\infty$ then $A \in I^n$ for some finite n. By the inductive proof above, $A \in M_P$. □

The completeness of the U_P operator can be proved under the assumption of the *unbounded cost property*. A program P with stable model M_P is said to be unbounded in cost if for every infinite subset S of M_P, $cost(S)$ is unbounded. Note that if M_P is finite then there are no infinite subsets and hence P is trivially unbounded in cost.

Theorem 4: *Let P be a uniformly monotonic min program which is cost unbounded. Let the stable model of P be M_P. Let $U_P^\infty = (I^\infty, L^\infty)$. Then $I^\infty = M_P$.*

Proof: Let $U_P^n = (I^n, L^n)$ for all finite n and $n = \infty$. We first show that $U_P(U_P^\infty) = U_P^\infty$. Suppose not. Then $least(I^\infty, L^\infty) \neq \phi$ and hence let $A \in least(I^\infty, L^\infty)$. Hence $A \in I^m$ for some finite m. Since A does not contradict minima in L^∞, 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 $cost(L^n) < cost(A)$. Hence the sequence of sets $\{L^{m+k}\}$ is a strictly growing sequence of sets w.r.t. $\subset$. $L^{m+k} \subset I^{m+k} \subset M_P$, by Theorem 3. Hence we have an infinite subset of M_P that is bounded in cost, contradicting the assumption.

We have established that $U_P(U_P^\infty) = U_P^\infty$. Hence $T_H(I^\infty) \subset I^\infty$. Since $least(I^\infty, L^\infty) = \phi$ and hence $T_N(I^\infty) \subset I^\infty$). Hence $T_P(I^\infty) = T_H(I^\infty) \cup T_N(I^\infty) \subset I^\infty$. Thus, I^∞ is a model of P. From Theorem 1, I^∞ is a subset of M_P which is a stable model and therefore minimal. Hence $I^\infty = M_P$. □

The U_P operator generates tuples by alternating be-

tween 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. Hence, U_P is computable. The strict alternation between V_P and G_P is not entirely necessary, in general, for any countable ordinal $n > 0$, the following operator U'_P also computes the intended model:

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

In particular, if $n = \infty$ then the operator reduces to the operator used to construct the stable model in [7] (except that backtracking is never needed). In the shortest path example we considered earlier 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:

```
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) ← p(X,C), ¬(p(X,D), D < C).
```

Let us step through the computations of the U_P operator. Initially $I = \phi$ 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 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 $\{p(a,0)\}$. We now choose the least elements from this set which is $\{p(a,0)\}$, include it in L and use it to fire the non Horn rule to obtain the tuple $q(a,0)$. Thus $I = \{p(a,0), r(a,b), q(a,0)\}$ after the completion of the first iteration of U_P.

We now return to firing the Horn rules using the V_P operator and obtain $L = \{p(a,0)\}$ and $I = \{p(a,0), r(a,b), q(a,0), s(a,0), p(b,2)\}$.
At this point we have to find the set of tuples whose minima are not contradicted by tuples in L. These tuples are $\{q(a,0), s(a,0)\}$. We choose the least of these tuples, and obtain $A = \{q(a,0), s(a,0)\}$, then add A to L. This set is also used to fire the non Horn rule but, since this rule uses p-atoms in the body, $T_N(A) = \phi$. Hence at the end of the second iteration, the values for I and L are as follows: $I = \{p(a,0), r(a,b), q(a,0), s(a,0), p(b,2)\}$ and $L = \{p(a,0), q(a,0), s(a,0)\}$.
We fire the Horn rules again to obtain the new tuple $\{p(b,1)\}$ in I. Next we find the set of tuples in I whose minima are not contradicted by tuples in L and this set is $A = \{p(b,1)\}$. The minima of this set is A itself and we include it in L and use it to fire the non Horn rule to obtain the tuple $q(b,1)$. Hence at the end of the third iteration, we have $I = \{p(a,0), r(a,b), q(a,0), s(a,0), p(b,2), p(b,1), q(b,1)\}$ and $L = \{p(a,0), q(a,0), s(a,0), p(b,1)\}$.
The reader may verify that the algorithm goes through one more iteration including the element $q(b,1)$ in L and then reaches a fixpoint. □

Similar improvement to the semi-naive over the naive computation can be incorporated into the greedy fixpoint algorithm. 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 element in $I - L$). After making these optimizations it can be shown that the evaluation of the following program mimics Dijkstra's algorithm for shortest paths (having a as source node) and therefore has the same complexity.

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

sh_path(Y,C) ←  path(Y,C),
                ¬(path(Y,D),D < C).
```

4 Propagation of min predicates

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.

In this section we design an algorithm for propagating min predicates while preserving query equivalence under certain monotonicity constraints. The intuitive notion of depth of propagation is the following. Let P be a min program satisfying the monotonicity constraints and let G be the And-Or graph of P. Then we consider P to be optimal if the rule nodes in any path in G whose last rule node is a non horn rule does not include two consecutive rule nodes which are Horn nodes. This property guarantees that the output of every Horn rule defining such a predicate is subject to the application of at least one min filter before it is used to fire another rule. We first present an example and then develop the theory for propagation.

Consider a non-linear program P for evaluating the shortest path.

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

Suppose the query is of the form $?sh\_path(X, Y, C)$. If we try to evaluate this using the greedy algorithm we face the problem of having to compute the predicate $path(X,Y,C)$ first. However, it is clear that tuples in *sh_path* are members of the predicate *cpath* which is defined as below.

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

We use *adornments* to abbreviate the min predicates which is defined as follows. An adornment is a (possibly empty) string over the set $\{u,e,m\}$ satisfying the following condition: If $|\alpha| > 0$ then there is a unique occurrence of m in α. Let α and β be two adornments such that $|\alpha| = |\beta|$ and the position of m in α and β are the same. We say that α is *more constrained* than β if for every i, $1 \leq i \leq |\alpha|$, $\alpha_i = u \Rightarrow \beta_i = u$ and there exists a j such that $\alpha_j = e$ and $\beta_j = u$.

For example a min predicate $min(C, (X,Y), path(X,Y,C))$ will be rewritten as $path^{uum}(X,Y,C)$ whereas a predicate $min(C, (X), path(X,Y,C))$ is rewritten as $path^{uem}(X,Y,C)$. Our non-linear rules above can be rewritten as follows:

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

Undoubtedly, this set of rules is query equivalent to the original program since all that we have done is abbreviate some of the definitions. Observe that we can replace the last rule above by the following rule:

```
path^uum(X,Y,C) ← min(C,(X,Y),cpath(X,Y,C))
```

and still retain program query equivalence. Now given a query of the form $?sh\_path$ we note that the rules defining *path* are not reachable from the *sh_path* and hence are not needed for evaluating the query. Thus for a query $?sh\_path(X,Y,C)$ the *query equivalent* program for P is the following:

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

The reader may note the property in the And-Or graph mentioned earlier for this program. We now formally present the theory for propagating the min predicate in a given min program P while preserving query equivalence.

Definition 7: Let f be an n-ary function. We say that f is *total-monotonic* in its i^{th} argument if the existence of $f(X_1,\ldots,X_{i-1},U,X_{i+1},\ldots,X_n)$ and the fact that $V \leq U$, implies that $f(X_1,\ldots,X_{i-1},V,X_{i+1},\ldots,X_n)$ exists and that $f(X_1,\ldots,X_{i-1},V,X_{i+1},\ldots,X_n) \leq f(X_1,\ldots,X_{i-1},U,X_{i+1},\ldots,X_n)$.

Let P be a program and A be a pre-interpreted predicate in P. Let S be a set of variables appearing in A. Then, A is said to be *total* w.r.t. S if every ground assignment of S can be extended to an assignment for the remaining variables such that A is true in the predefined interpretation.

The Horn clauses of P are required to be of the following form:

$$A \leftarrow B,\ldots,C,X = f(X_1,\ldots,X_n).$$

where

1. $A,B,\ldots,C$ are atoms. The variable X appears in the cost attribute position of A, and $X_1,\ldots,X_n$ are all the cost variables that appear in the body. There does not exist any other predicate in the body that contains more than one cost argument.
2. The predicate $X = f(X_1,\ldots,X_n)$ is total w.r.t. $\{X_1,\ldots,X_n\}$.

Rules for propagating adornments in Horn Clauses: We assume that for every predicate symbol p that appears in a min program P, the predicate symbol cp does not appear in P and may be used by the propagation algorithm.

We illustrate this by considering the following rule as a running example:

```
p(X,Y,C) ← q(X,Z,C1,N1),
           r(Z,Y,C2,N2),s(Z,W,C3),
           C = C1 + C2 + C3 + C3 mod 5,
           N1 = N2 + 1.
```

The adornment for p is *uem*. The following steps first assign adornments to variables in the body of the

rule and then converts it to adornments of argument positions in goals.

We first propagate the m adornment as follows. Let $W = \{X_j | f$ is total-monotonic in the j^{th} argument and X_j appears exactly twice in the body$\}$. We define disjoint sets of body predicates Q and R as follows. R is the set of predicates whose cost attributes appear in W. The set Q is the remaining set of predicates. The cost attributes of every member of R is adorned by m. In the example rule above, f is total-monotonic in $C1$ and $C2$ but not in $C3$. Hence $W = \{C1, C2\}$. Hence, $R = \{q(X, Z, C1, N1), r(Z, Y, C2, N2)\}$ and $Q = \{N1 = N2 + 1, s(Z, W, C3)\}$.

We now propagate the u and the e adornments. Variables in R that are adorned as u in the head are adorned as u in the body. Variables appearing in more than one predicate in R are also adorned as u. In our example rule X and Z are adorned as u. Variables in R that appear only once in $Q \cup R$ and do not appear in the head are assigned e. This rule is not applicable in the example above. Variables in R that appear only once in $Q \cup R$ and are adorned as e in the head are adorned as e. This rule adorns Y as e in our example.

At this stage some of the variables in R may have been left unadorned. It is safe to adorn all such variables with u. However, the evaluation is more efficient to adorn as many of them by e as possible. This improvement is expressed as follows. The remaining variables appearing in R are adorned as u or e such that after the adornment the following condition is satisfied: For every predicate A in Q, A is total w.r.t. the set of variables that are adorned as e. In our example, the remaining variables in R are $N1$ and $N2$. The predicate $N1 = N2 + 1$ is total w.r.t. $\{N1\}$ and w.r.t. $\{N2\}$ but is not total w.r.t. $\{N1, N2\}$. Thus, we can satisfy the above condition by adorning one of $N1, N2$ as u and the other as e. Let us choose to adorn $N1$ as e and $N2$ as u.

At this stage, all variables in R are adorned as u, e or m. Thus, argument positions of predicates in R are adorned with the same adornment as that of the variable occupying that position. Uninterpreted predicates in Q are adorned by ϵ.

The procedure above defines the body of the adorned rule. Let p be the predicate symbol appearing in the head and let α be the adornment for the head. If $\alpha \neq \epsilon$ then the head of the adorned rule is obtained by replacing p in the original rule by cp^{α}. If $\alpha = \epsilon$ then the head of the adorned rule is obtained by replacing p in the original rule by p^{ϵ}. The adorned rule for the example is:

$$\begin{aligned} \mathrm{cp}^{\mathtt{uem}}(\mathtt{X},\mathtt{Y},\mathtt{C}) \leftarrow\ & \mathrm{q}^{\mathtt{uume}}(\mathtt{X},\mathtt{Z},\mathtt{C1},\mathtt{N1}), \\ & \mathrm{r}^{\mathtt{uemu}}((\mathtt{Z},\mathtt{Y},\mathtt{C2},\mathtt{N2}),\mathtt{s}(\mathtt{Z},\mathtt{W},\mathtt{C3}), \\ & \mathtt{C} = \mathtt{C1} + \mathtt{C2} + \mathtt{C3} + \mathtt{C3} \bmod 5, \\ & \mathtt{N1} = \mathtt{N2} + 1. \end{aligned}$$

Rule for propagating adornment in Non Horn rules.

1. The variable denoting the cost attribute in the body is adorned as m.
2. Variables which are not adorned as e in the head and are grouped w.r.t. min in the body are adorned as u. All other variables are adorned as e.

The body of the adorned rule is obtained by replacing the min predicate by the adorned predicate. If the predicate symbol in the head is p and the adornment for the head is α then the head of the adorned rule is obtained by replacing the head of the original rule by p^{α}. For example if the rule is:

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

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

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

The above steps generate an adorned rule given a rule and an adornment for the head. We use this to define the adorned definition of a predicate p in a program w.r.t. a given adornment α. An example of the adorned definition may be found in the beginning of this section where an adorned definition of *path* was introduced w.r.t. the adornment uum.

Let p be an n-ary predicate with cost argument k and let α be a non-empty adornment for p. Let $W_1, \ldots, W_n$ be a sequence of n distinct variables. Let $S = \{W_i | \alpha_i = u\}$. Then $mindef(p, \alpha)$ denotes the singleton set containing the following min rule: $p^{\alpha}(W_1, \ldots, W_n) \leftarrow min(W_k, S, p(W_1, \ldots, W_n))$. $rmindef(p, \alpha)$ denotes the singleton set containing the following min rule: $p^{\alpha}(W_1, \ldots, W_n) \leftarrow min(W_k, S, cp(W_1, \ldots, W_n))$. If α is the empty adornment then $mindef(p, \alpha)$ and $rmindef(p, \alpha)$ are empty sets. Let P be a min program and p be a derived predicate symbol appearing in P. Given an adornment α for p, $adorn(p, \alpha)$ is defined to be $\{r^{\alpha} | p$ appears in the head of $r\} \cup rmindef(p, \alpha)$.

The algorithm for generating a query equivalent program is as follows. Starting from the query predicate symbol q with adornment ϵ we propagate the adornment in all the rules that define q. In this process, we obtain possibly new adorned predicates. These adorned predicates are then candidates for further propagation and this step is continued until no new adorned predicates are found. At the end of this step, the set of adorned predicate symbols which are reachable from q^{ϵ} are known. A query equivalent program can now be obtained by including the adorned definition of every adorned predicate. We introduce a minor optimization at this point. Suppose that adornments p^{um} and p^{em} are both reachable from q^{ϵ}. 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} as follows:

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

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. This notion is formalized below and the algorithm is given in Algorithm 1.

Given a set of adorned predicates T, a predicate symbol p^α belonging to T is said to be minimally constrained if there does not exist a p^β in T such that β is less constrained than α. Given a min program P and a set T of adorned predicates of P, we define $Reduce(P,T)$ as follows:

1. The set of rules $\{adorn(p,\alpha) | p^\alpha$ is minimally constrained in $T\}$ is included in $Reduce(P,T)$.
2. For every predicate symbol p^β in T which is not minimally constrained choose a p^α in T which is less constrained than β. Include $mindef(p^\alpha,\beta)$ in $Reduce(P,T)$.

Algorithm 1 *Propagate min predicates*
Input: *Min program P and a query $q(..)$*
Output: *Rewritten min program query equivalent to P*
var R,S,T : *set of predicate symbols*
var p,q : *predicate symbol*
var α,β: *adornment*
begin
  $S := \{q^\epsilon\}$; $T := \emptyset$;
  while $S \neq \emptyset$ **do**
    Choose an arbitrary predicate $p^\alpha \in S$;
    $R :=$ {predicate symbols appearing in bodies of rules in $adorn(p,\alpha)$};
    $S := S \cup (R - T)$;
    $S := S - \{p^\alpha\}$; $T := T \cup \{p^\alpha\}$;
  endwhile
  $P := Reduce(P,T)$;
end .

Theorem 5: *Let P be a min program and P^α the rewritten program obtained from P by application of algorithm 1, then P^α is query equivalent to P.* □

We have seen that the greedy fixpoint procedure may terminate on transformed programs although not on the original program. Hence it is an interesting problem to determine the class of programs which can be effectively computed by using the greedy fixpoint procedure, after propagation of the min. A Sufficient condition that guarantees the effective evaluation of an important subclass of min programs is presented below.

We will define *Min-Datalog* programs to be Datalog programs extended to allow interpreted functions (e.g., arithmetic) on cost arguments. All the examples considered so far, are Min Datalog programs.

Definition 8: Let P be a monotonic Min-Datalog program and Q be a query on P. Consider the program, say R, obtained from P by adding the following rule: $query(\bar{X}) \leftarrow Q$. We say that P is *safe* with respect to the query Q if the cost graph of R, CG_R, satisfies the following property:
For every node p in CG_R that lies on a cycle, every path from p to the node *query* contains an arc whose rule's label denotes a non horn rule. □

Then, the following theorem states that the size of all intermediate relations during the greedy fixpoint computation of a Min-Datalog program are finite.

Theorem 6: *Let P be Min-Datalog program and Q be the query such that P is safe with respect to Q. Let R be the program generated from P via the propagation of the min predicate. Then, M_R is finite, and the query Q is effectively computable.* □

5 Propagation of constants

The only departure from the standard magic sets algorithm arises due to the presence of the min rule s. The following example illustrates how binding information should be propagated inside min rules.

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

Suppose the binding for p is bbb. The binding for Y cannot be propagated since the minimum is desired over all possible values of Y. Similarly, the binding for C cannot be propagated, since, by definition, we choose the minimum ove r all possible values of C. In general, the cost attributes of every predicate is adorned as f irrespective of whether the query binds its cost attribute. Hence the adorned rule is:

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

Hence the rule for propagating binding information in a min rule is: A variable is adorned as b in the body only if it is adorned as b in the head and is grouped w.r.t. min. All other variables in the body are adorned as f. The magic rule corresponding to the above rule is:

$$\mathtt{mq^{bff}(X)} \leftarrow \quad \mathtt{mp^{bbb}(X,Y,C)}.$$

The remainder of the basic magic sets algorithm is not required to change. The generalized magic sets algorithm when applied to monotonic min programs could result in non monotonic min programs. Hence we restrict constant propagation using the basic magic sets algorithm [9].

6 Conclusion

In this paper we have presented techniques for the efficient computation of a large class of min and max programs. For example, given a database of part subpart relationship in *partof*, the following program demonstrates the use of the max predicate to express the earliest assembly time of items, *earliest*(I,T), given the minimum waiting times *wait* of the children. This program assumes that once all the subparts of a part are available, the overhead of assembly for the larger part is negligible.

```
earliest(I,T) ←     wait(I,T).
not_before(I,T) ←   partof(J,I), earliest(J,T).
earliest(I,T) ←     max(T,(I),not_before(I,T)).
```

The greedy fixpoint computation can be applied here, enabling the usage of Dijkstra's algorithm on this classical inventory problem. (Also, the max predicate could have been pushed inside recursion, if the programmer had chosen the non-recursive formulation of this problem).

The techniques presented here are more specialized than those presented in [2,5], which address generic aggregate predicates (i.e., including cardinality and sum). By restricting our attention to extrema predicates, we have obtained techniques that are significantly more efficient. In fact, the greedy fixpoint computation is supports the seminaive fixpoint improvement, not available in [2] and thus performs as Dijkstra's algorithm on shortest path problems. Furthermore, the monotonicity conditions introduced here are more general than the existence of semirings, and detection of monotonicity appears easier than detection of semirings. An alternative approach for evaluating min programs is proposed in [8] which designs another operator. The major distinction of their operator from ours is that it is is non monotonic w.r.t set inclusion, applies only when P is syntactically stratified w.r.t negation and does not require monotonicity w.r.t. cost. The non monotonicity (w.r.t. $\subset$) of their operator makes it less efficient than U_P in cases where both apply.

Several interesting problems have been left for further research. 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 Kleene or Floyd algorithms). We plan to study how the efficiency of the U_P operator may be enhanced using ideas of differential evaluation and using efficient data structures. Another open problem is combining the pushing of min and max predicates with the generalized magic set methods, and the related problems of dealing with the lack of monotonicity or stratification.

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] 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.

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

[4] 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.

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

[6] 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.

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

[8] R. Sudarshan, S. Ramakrishnan. Aggregation and relevance in deductive databases. *Submitted for publication*, 1991.

[9] J. Ullman. *Principles of Data and Knowledge-Base Systems*. Volume 2, Computer Science Press, New York, 1988.

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

[11] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176, Morgan-Kaufman, Los Altos, CA, 1988.

[12] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM (to appear)*, 1991.