How to Forget the Past Without Repeating It

Jeffrey F. Naughton* and Raghu Ramakrishnan†
Computer Sciences Department
University of Wisconsin-Madison, WI 53706, U.S.A.

Those who cannot remember the past are condemned to repeat it.

— *George Santayana*

Abstract

Bottom-up evaluation of deductive database programs has the advantage that it avoids repeated computation by storing all intermediate results and replacing recomputation by table lookup. However, in general, storing all intermediate results for the duration of a computation wastes space. In this paper we propose an evaluation scheme that avoids recomputation, yet under fairly general conditions at any given time stores only a small subset of the facts generated. The results constitute a significant first step in compile-time garbage collection for bottom-up evaluation of deductive database programs.

1 Introduction

A fundamental advance in deductive database technology has been the invention of bottom-up query evaluation strategies that retain the "focussing" properties of top-down evaluation strategies. One of the key advantages of these bottom-up strategies over common top-down approaches (such as Prolog) is that by storing all intermediate results, bottom-up evaluation is able to replace the recomputation of a fact by a simple lookup. In many cases this makes bottom-up evaluation polynomial time when the corresponding top-down computation is exponential time. However, this efficiency in the time domain is often accompanied by inefficiency in the space domain.

In this paper we propose an evaluation scheme, called *Sliding Window Tabulation*, that retains the time efficiency of bottom-up computation while improving its space efficiency. Intuitively, Sliding Window Tabulation stores a fact as long as the fact could still be used to produce answers or avoid redundant computation, but no longer. The need for such an evaluation scheme is perhaps best demonstrated by an example.

Example 1.1 Consider the problem of finding the longest common subsequence in a pair of strings. This problem is significant because it is paradigmatic of a number of problems that arise in DNA sequence analysis, an area that has been identified as a promising application for deductive database technology.

The standard algorithm of Hirchberg [Hir75] for the longest common subsequence can be expressed simply and elegantly using Prolog notation (the program appears in Section 5 of this paper). Unfortunately, on strings of length n, the time complexity of Prolog is both $O(\binom{2n}{n})$ and $\Omega(\binom{2n}{n})$. The function $\binom{2n}{n}$ is exponential in n and grows extremely quickly. For example, if $n = 20$, we have that $\binom{2n}{n} > 275 \times 10^9$; if $n = 100$, we have $\binom{2n}{n} > 1.8 \times 10^{59}$. Clearly, the Prolog evaluation strategy cannot be used on this program for any but the shortest of strings.

If we take the bottom-up approach of rewriting by Magic Templates [Ram88] followed by Seminaive bottom-up evaluation [Ban85], the running time is reduced to $O(n^2)$. This is a dramatic improvement; unfortunately, the space requirement is also $O(n^2)$. In DNA sequence analysis, comparison of strings of over 10^6 bases will be routine. (The human genome is estimated to contain over 10^9 base pairs.) Even if each fact to be stored fits in a single byte, on strings of this size, the standard bottom-up approach will require over a terabyte (10^{12} bytes) of storage.

The evaluation algorithm presented in this paper, "Sliding Window Tabulation," evaluates the LCS program in $O(n^2)$ time and $O(n)$ space. To our knowledge this is the first evaluation algorithm for deductive database queries that can feasibly be applied to evaluate queries on this program over large databases. □

*Supported by NSF grant IRI-8909795.

†Supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, an IBM Faculty Development Award and NSF grant IRI-8804319.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

The tradeoffs between memoing, recomputation, and efficient space management have been explored in the functional programming literature [Bir80, Coh83, Hil76], but to our knowledge have never been explored in the context of bottom-up evaluation of logic programs. A main contribution of this paper is to identify the problem of improving memory utilization in bottom-up evaluation of logic programs.

Sliding Window Tabulation, presented in Section 4, differs significantly from techniques suggested for functional programs in [Coh83, Bir80], and presents an interesting contrast: Rather than explore program schemas and schema transformations, we develop a uniform approach based on optimizing the structure of programs generated by the Magic Templates algorithm.

Our algorithm for tabulation of a program and query consists of three phases: 1) First, the program is analyzed to determine if tabulation can be profitably applied. 2) Next, the program is rewritten using the Magic Templates rewriting algorithm. 3) Finally, the resulting rewritten program is evaluated by an algorithm that attempts to store facts only as long as necessary, using information derived from the analysis of the program done in step 1.

The remainder of this paper is organized as follows. We give a brief overview of bottom-up evaluation in Section 2. We give an overview of Sliding Window Tabulation in Section 3, and consider the algorithm in more detail in Section 4. We present testable conditions for applicability of this algorithm and several results about its performance. We present a detailed discussion of the LCS problem introduced in Example 1.1 in Section 5. We present conclusions and directions for future work in Section 6.

2 The Bottom-Up Approach

To provide context for the discussion of Sliding Window Tabulation, and to make this paper self-contained, in this section we present an overview of the bottom-up approach to deductive database query evaluation. This approach has been developed without consideration for space utilization.

The first step in bottom-up evaluation is to rewrite the program using the Magic Templates transformation [Ram88]. Magic Templates extends the Magic Sets rewriting algorithm [BMSU86, BR87] to deal with non-ground facts. The important properties of the algorithm for the purposes of this paper is that given a program P and a query q, Magic Templates produces a new program P^{mg} such that evaluating P^{mg} bottom-up produces no irrelevant facts.

After applying the Magic Templates transformation, the resulting program P^{mg} is evaluated bottom-up using a *Seminaive* algorithm. Seminaive fixpoint evaluation [Ban85] ensures that derivations are not repeated in subsequent iterations, by considering in each iteration only rule instantiations that utilize at least one new fact generated in the previous iteration. (This is considered in more detail in Section 4.2.)

We now give a brief description of the Magic Templates algorithm. For details, consult [Ram88]. The Magic Templates algorithm [BMSU86, BR87, Ram88] transforms a program, for a given query form, in such a way that the Seminaive evaluation of the transformed program generates no irrelevant facts. The initial rewriting of a program and query is guided by a choice of *sideways information passing strategies*, or sips. For each rule, the associated sip is the (partial) order in which the body literals are to be evaluated.

The idea is to compute a set of auxiliary predicates that contain the goals. The rules in the program are then modified by attaching additional literals that act as filters and prevent the rule from generating irrelevant tuples. As a first step, however, we produce an *adorned* program in which predicates are adorned with an annotation that indicates which arguments are bound and which are free. Adornments for the program are determined from the query and the choice of sips. The Magic Templates algorithm is a two-step transformation in which we first obtain the adorned version of P and then apply the following transformation:

Definition 2.1 [The Magic Transformation] We construct a new program P^{mg}. Initially, P^{mg} is empty.

1. Create a new predicate *magic_p* for each predicate p in P. The arity is that of p.
2. For each rule in P, add the *modified version* of the rule to P^{mg}. If rule r has head, say, $p(\bar{t})$, the modified version is obtained by adding the literal $magic\_p(\bar{t})$ to the body.
3. For each rule r in P with head, say, $p(\bar{t})$, and for each literal $q_i(\bar{t}_i)$ in its body, add a *magic rule* to P^{mg}. The head is $magic\_q_i(\bar{t}_i)$. The body contains all literals that precede q_i in the sip associated with this rule, and the literal $magic\_p(\bar{t})$.
4. Create a *seed* fact $magic\_q(\langle\bar{c}\rangle)$ from the query.

□

A simple optimization is to delete all argument positions corresponding to free arguments from the magic predicates, since these positions always contain distinct variables. In the sequel, we will refer to the above two step transformation with this optimization as the Magic Templates algorithm. We now illustrate the bottom-up approach on a program to compute the Fibonacci numbers.

Example 2.1 The following program computes the Fibonacci numbers.

$fib(0,1).$
$fib(1,1).$
$fib(N, X1 + X2)$:- $N > 1, fib(N-1, X1), fib(N-2, X2).$

Magic Templates applied to this program and the query $fib(n, X)$? produces the "magic" rules

$m\_fib(n).$
$m\_fib(N-1)$:- $N > 1, m\_fib(N).$
$m\_fib(N-2)$:- $N > 1, m\_fib(N).$

and the modified original rules

$$
\begin{aligned}
fib(0,1) &:- \quad m\_fib(0).\\
fib(1,1) &:- \quad m\_fib(1).\\
fib(N, X1+X2) &:- m\_fib(N), N>1, fib(N-1, X1),\\
&\qquad fib(N-2, X2).
\end{aligned}
$$

One may verify that this program, when evaluated using Seminaive bottom-up, computes the answer to the query in linear time. □

3 Overview of Sliding Window Tabulation

Sliding window tabulation of a program/query pair $\langle P, q\rangle$ consists of two phases. In phase one, the magic rules of P^{mg} are repeatedly applied in order to determine the set of all basis facts relevant to the query. In phase two, the modified program rules of P^{mg} and the inverted magic rules of P^{mg} (inverted magic rules are defined in Definition 3.3 of Section 4) are repeatedly applied to work back up towards the query goal in order to generate answers. Only relevant basis facts may be used in this second phase of sliding window tabulation. Often we will refer to phase one as the "down" phase, since phase one works "downwards" toward basis facts. Similarly, we will refer to phase two as the "up" phase, since it works "upwards" toward answers.

Both the "down" phase and the the "up" phase are performed using a modification of Seminaive evaluation. The main idea behind this modification is to partition the facts produced during the fixpoint into possibly overlapping sets called *windows*. We emphasize that the partitioning of facts into windows is conceptual; as detailed in Section 4, in practice the algorithm accomplishes the partitioning by deriving a "windowing" function that maps facts to windows as they are produced during the query evaluation.

The computation of the answer to $\langle P, q\rangle$ begins with the window containing the "seed" magic fact, and proceeds by sliding the window downward, using the magic rules of P^{mg} to compute new magic facts as it goes, deleting the facts in a given window after they have been processed. The downward phase terminates when the evaluation of the window containing the "lowest" magic fact reachable from the query goal has been completed.

The upward phase starts with the lowest window that could contain a relevant basis fact, and proceeds by sliding the window upwards, computing program facts by applying the original rules and inverted magic rules of P along the way, again deleting the facts from a given window after it has been processed. The upward computation terminates when the current window has slid up to the window containing the highest answer to the query goal.

The key property of Sliding Window Tabulation with respect to efficient space utilization is that 1) during the "down" phase, no magic fact m is stored after the current window has slid below the lowest window that contains m, and 2) during the "up" phase, no program fact f or magic fact m is stored after the current window has slid above the uppermost window that contains f.

A variant of Sliding Window Tabulation stores all magic facts encountered in the down phase, and uses these stored magic facts instead of inverted magic rules in the up phase. This requires more storage than basic Sliding Window Tabulation, but can be more time-efficient if basic sliding window "overtabulates" on $\langle P, q\rangle$. (Overtabulation is described in Subsection 4.4.)

We will use the program to compute Fibonacci numbers, given in Example 2.1, as a running example. While the program is simple, it illustrates some important aspects of sliding window tabulation. A more complex example of sliding window tabulation is given in Section 5.

3.1 Detecting Applicability

In this subsection we develop sufficient conditions for Sliding Window Tabulation to apply, and give an algorithm that tests for these conditions.

A useful property of many programs is that the facts for magic predicates in P^{mg} can be computed independent of the facts for derived predicates of the original program. This allows us to separate the evaluation of P^{mg} into two phases: first, compute magic facts, and then compute program facts by applying the modified rules. To formalize this idea, we use the following definition:

Definition 3.1 Define the relation $p \rightarrow q$ such $p \rightarrow q$ holds if there is a rule with a q literal in the head and a p literal in the body. We let $\xrightarrow{*}$ denote the transitive closure of $\rightarrow$. □

Intuitively, if $p \xrightarrow{*} q$, then p is used to define q.

Definition 3.2 Let $\langle P, q\rangle$ be a program-query pair, and let P^{mg} be the result of the Magic Templates rewriting of $\langle P, q\rangle$. Then $\langle P, q\rangle$ has the *sliding window property* if:

1. If p_m is a derived predicate in P^{mg}, and m_p_n is a magic predicate, then $p_m \xrightarrow{*} m\_p_n$ does not hold.
2. There are constants u and l, where $l \leq u$, and a set of windows $\mathcal{W}_i$, where $l \leq i \leq u$, such that the following two conditions hold:
   (a) Consider a rule instantiation p :- m_p, p_1, p_2, ..., p_k in the fixpoint evaluation of P^{mg}. Suppose that p appears in window $\mathcal{W}_i$. Then there must be a window $\mathcal{W}_j$, where $j \leq i$, that contains every fact that appears in the body.
   (b) Consider an instantiation of a magic rule m_p_l :- m_p, p_1, p_2, ..., p_k in the fixpoint evaluation of P^{mg}. Suppose that m_p_l appears in window $\mathcal{W}_i$. Then there must be a window, say $\mathcal{W}_j$, where $j \geq i$, that contains every fact that appears in the body.

□

We now define inverted magic rules.

Definition 3.3 [Inverted Magic Rules] Let r be a magic rule of the form m_p :- $m\_q, e_1, \ldots, e_n$, where m_p and m_q are magic predicates and the e_i are EDB predicates. Then the *inversion* of r, denoted r', is the rule m_q :- $m\_p, e_1, \ldots, e_n$. The *inverted magic rules* for P^{mg} consist of the inversion of each magic rule of P^{mg}. □

Example 3.1 For the Fibonacci program of Example 2.1, the inverted magic rules are

$m\_fib(N)$:- $N > 1, m\_fib(N-1)$.
$m\_fib(N)$:- $N > 1, m\_fib(N-2)$.

Note that the fact $m\_fib(n)$ does not have a corresponding inverted version. □

To discover whether or not a program has the sliding window property, we first need to provide a mechanism for determining to which window or set of windows a given fact or goal belongs. We will assume that EDB facts cannot be discarded, and thus consider these facts to be in every window. Recall that the EDB relations are by definition those relations that are defined by tuples stored in the system rather than defined by rules. A user might be justifiably upset if a query evaluation algorithm deleted these relations as a side effect of answering a query. For this reason, we can restrict our attention to the set of magic facts and derived program facts in determining windows.

We simplify the search for windows by defining the set $\mathcal{W}_i$ in terms of the bound arguments of program facts and all arguments of magic facts. This uses the property that the set of bound arguments in the goals for a given predicate is exactly the set of bound arguments of the facts for that predicate whenever the bound arguments contain only ground terms.

We determine the windows $\mathcal{W}_i$ by defining a function that maps goals and facts to integers. The intent is that a given window contains all goals and facts that map to some specified subrange of integers. In more detail, if the IDB predicates in the program under consideration are $p_1, p_2, \ldots, p_k$, we require a set of k functions ϕ_i, for $1 \leq i \leq k$, where the function ϕ_i maps p_i facts and m_p_i goals to the integers.

For notational convenience, we will often use the generic function ϕ to represent all the ϕ_i. That is, if x is either m_p_i or p_i, the meaning of $\phi(x)$ is $\phi_i(x)$. If a window $\mathcal{W}_i$ consists of $\{x \mid c \leq \phi(x) \leq c + h\}$, we call c the *base* of the window and h the *height*.

A high-level description of an algorithm to detect when a program has the sliding window property follows. The algorithm takes as input the program P^{mg} corresponding to a program/query pair $\langle P, q\rangle$. If the algorithm can verify that P^{mg} has the sliding window property, it returns

- A function ϕ that determines to which set of windows a given fact or goal belongs.
- Two integers c_u and c_l, where c_l is the base of the lowest window that needs to be considered in the evaluation of $\langle P, q\rangle$, and c_u is the base of the highest such window.
- An integer h such that the windows are of height h.

With this terminology, the above parameters can be used to specify the windows $\mathcal{W}_i$ as follows:

$$
\begin{aligned}
\mathcal{W}_{c_l} &= \{x \mid c_l \leq \phi(x) \leq c_l + h\} \\
\mathcal{W}_{c_l+1} &= \{x \mid c_l + 1 \leq \phi(x) \leq c_l + h + 1\} \\
&\vdots \\
\mathcal{W}_{c_u-1} &= \{x \mid c_u - 1 \leq \phi(x) \leq c_u - 1 + h\} \\
\mathcal{W}_{c_u} &= \{x \mid c_u \leq \phi(x) \leq c_u + h\}
\end{aligned}
$$

The detection algorithm works by enumerating a set of possible choices for ϕ and checking for the sliding window property with each choice.

Algorithm 3.1 (Sliding Window Detection)
Input: P^{mg}, the output of the Magic Templates rewriting on a program/query pair $\langle P, q\rangle$.
Output: Either a four-tuple (ϕ, h, c_u, c_l) such that P^{mg} has the sliding window property with windows as defined by (ϕ, h, c_u, c_l), or FAIL if no such function could be found.

```
if (Independent(P^mg, P) and AllBoundGround(P^mg))
then while NextPhi(φ, P^mg) do
        if CheckMonotonic(φ, P^mg) and
           FindHeight(φ, P^mg) and
           FindLimit(φ, P^mg, c_l, c_u)
        then return (φ, h, c_l, c_u);
     endWhile;
     return FAIL;
endif;
return FAIL;
```

We consider each of the procedures used in Algorithm 3.1 in turn.

Procedure 3.1 [MagicAndProgramIndependent]
Input: A program P^{mg}.
Output: True, if in P^{mg}, there are no pairs of IDB predicates p, q such that $p \stackrel{*}{\rightarrow} m\_q$ as defined in Definition 3.1; false otherwise.
Method: Construct the Rule/Goal graph for P^{mg}. For each IDB predicate p, check that there is no path from the node for p in this graph to the node for a magic predicate. □

The running time of this procedure is $O(n^2)$, where n is the number of predicates in P^{mg}.

Procedure 3.2 [Groundness]
Input: A program P^{mg}.
Output: True, if the procedure can prove that in the bottom-up evaluation of P^{mg}, every bound argument is a ground term; False otherwise.
Method: It is sufficient to check that the query and all EDB predicates contain only ground terms, and that the magic rules in P^{mg} are range restricted. □

The running time of this procedure is linear in the size of P^{mg}. Procedures 3.1 and 3.2 are independent of the particular class of candidate ϕ functions considered by Algorithm 3.1. By contrast, the remainder of the procedures in that algorithm are highly dependent on the chosen class of candidate ϕ's. We have presented the algorithm in generic form, to accommodate new classes of ϕ's as they are developed. In the remainder of this section, we give specific instances of the algorithms for an especially useful class of ϕ's, as discussed below.

The class of candidate ϕ functions considered in this paper are all based upon the sum of the sizes of a subset of the bound arguments of predicates, or the additive inverse of this sum of sizes. To define the *size* of an argument, we divide the bound arguments into two types: those that contain integers, and those that contain structured terms.

We will use the convention that all terms are uninterpreted, with the exception of arithmetic expressions (as is standard in logic programming). With this convention, the "size" of an argument a, denoted $s(a)$, is defined as follows:

1. For an integer expression e: if e appears in an argument position of a term (that is uninterpreted), then $s(e) = 1$, else $s(e) = e$. (For example, $s(5+6) = 1$ in the term $f(5+6)$, where f is an uninterpreted function symbol.)
2. The size of a variable X is unknown, and is represented by $s(X)$.
3. The size of an atom in a structured argument is 1.
4. The size of a term $f(t_1, t_2, \ldots, t_p)$ is defined by

$$s(f(t_1, t_2, \ldots, t_p)) = 1 + \max(s(t_1), s(t_2), \ldots, s(t_p))$$

For example, suppose that ϕ is the sum of arguments one and three of p, and that argument one has been determined to be a structured argument while argument three is an integer argument. Then

$$\begin{aligned}
&\phi(p(f(g(X), 6+10), X, 4+2)) \\
&= s(f(g(h(X)), 6+10)) + s(4+2) \\
&= 1 + \max(s(g((h(X))), s(6+10)) + 4 + 2 \\
&= 1 + \max(1 + s(h(X)), 2) + 6 \\
&= 1 + \max(2 + s(X), 2) + 6 \\
&= 8 + s(X)
\end{aligned}$$

Note that in this case the expression $6 + 10$ appears in a structured argument, hence is uninterpreted, and has size 2. On the other hand, the expression $4 + 2$ appears in an integer argument, hence it evaluates to 6. Also, this example indicates that the result of applying ϕ to a term may involve a variable. The function ϕ computes the sum of the *sizes* of a subset of the bound arguments of its argument goal or predicate. For this class of ϕ functions, we can enumerate all such choices for ϕ by enumerating the subsets of the bound arguments.

Procedure 3.3 [NextPhi]
Input: A program P^{mg}.
Output: True, if there are alternatives for ϕ that have not been returned by any previous call to NextPhi. In this case the variable ϕ is set to the next such alternative. The procedure returns False otherwise.
Method: NextPhi considers only functions ϕ that are defined to be the sum of the sizes of a subset of bound arguments, or the negative of the sum of the sizes of a subset of bound arguments. NextPhi retains in static storage the subset of bound arguments it returned in the previous call, and whether it was returned with positive or negative sign; on each call it updates this static storage and returns either the additive inverse of the current subset or the next subset in some order of enumeration. If in previous calls all subsets have been returned with both positive and negative sign, NextPhi returns false. □

Each call to NextPhi can be processed in time linear in the number of bound arguments. The total number of calls to NextPhi by Algorithm 3.1 is exponential in the number of bound arguments in a given predicate. However, since we expect the number of bound arguments to be small, an exhaustive algorithm is sufficient. (In the examples considered in this paper, at most two arguments are bound; this means that the number of cases to be considered is six, three with each sign.)

Next we turn to the procedures that check each candidate ϕ function to see if it satisfies the sliding window property.

Procedure 3.4 [CheckMonotonic]
Input: A program P^{mg} and a candidate windowing function ϕ.
Output: True, if for every possible instantiation of a rule in the bottom-up evaluation of P^{mg}, if p is the predicate instance in the head of the rule, and $p_1, \ldots, p_k$ are the recursive predicate instances in the body of the rule,

$$\phi(p) \geq \max(\phi(p_1), \ldots, \phi(p_k))$$

must hold.
Method: For each modified original rule r in P^{mg},

1. Apply ϕ to each literal in the head and body of r, and simplify using the definition of size of an argument and standard arithmetic.
2. Determine the literal p_i in the body of r such that $\phi(p_i)$ is maximal over all predicates in the body.
3. Attempt to produce an arithmetic tautology by simplifying $\phi(p) \geq \max(\phi(p_1), \ldots, \phi(p_k))$ using standard arithmetic.

If the procedure succeeds on step 3 for all rules in P, return true, else return false. □

Example 3.2 Consider again the Fibonacci program of Example 2.1. Since in the Fibonacci program there is only one bound argument, the choices for ϕ are just $\phi(fib(N, x)) = N$ or $\phi(fib(N, x)) = -N$. The following shows that Procedure CheckMonotonic succeeds with the first alternative.

To test for monotonicity, Procedure CheckMonotonic must compare the first instance of fib in the body of the rule and the instance in the head. That is, it must test if $\phi(fib(N, X)) \geq \phi(fib(N-1, X1))$ for all possible instantiations of the two predicates. We have that

$$\phi(fib(N, X)) \geq \phi(fib(N-1, X1))$$

holds if and only if $s(N) \geq s(N-1)$. Since this reduces to $0 \geq -1$, which holds unconditionally, the monotonicity constraint is satisfied. The monotonicity constraint for the second instance of fib and the instance of m_fib are similar.

For an example where CheckMonotonic returns with failure, consider the transitive closure

$t(X,Y) :- e(X,Y).$
$t(X,Y) :- e(X,W), t(W,Y).$

and the query $t(1,Y)$? Here we will need to test monotonicity between $t(X,Y)$ and $t(W,Y)$. Again, since there is only one bound argument, there are only two choices for ϕ: either $\phi(t(X,Y)) = s(X)$ or $\phi(t(X,Y)) = -s(X)$. Considering the first option, we have

$$\phi(t(X,Y)) \geq \phi(t(W,Y)) \iff s(X) > s(W)$$

Since no more simplifications are possible, and in general $s(X) \geq s(W)$ does not hold, the condition fails. The case for $\phi(t(X,Y)) = -s(X)$ is similar. □

Next consider finding the height of the windows.

Procedure 3.5 [FindHeight]
Input: A program P^{mg} and a candidate windowing function ϕ.
Output: True, if there is a constant h such that for every possible instantiation of a rule r in the bottom-up evaluation of P^{mg}, if p_1 through p_k are the recursive predicates in the body of r, the difference

$$\max(\phi(p_1), \phi(p_2), \ldots, \phi(p_k)) - \min(\phi(p_1), \phi(p_2), \ldots, \phi(p_k))$$

is at most h. In this case also return h; otherwise return false.
Method: For each rule r in P,

1. Apply ϕ to each predicate instance in the head and body of r, and simplify using the definition of size of an argument and standard arithmetic.
2. Determine $\max(\phi(p_1), \phi(p_2), \ldots, \phi(p_k))$ and $\min(\phi(p_1), \phi(p_2), \ldots, \phi(p_k))$ and their difference.
3. Attempt to reduce this difference to a constant h by using standard arithmetic.

If the procedure succeeds on step 3 for all rules in P^{mg}, return true and the maximal constant h used in step 3 over all rules; else return false. □

Example 3.3 Returning again to the Fibonacci example, Procedure FindHeight must find an h that bounds the difference of

$$\max(\phi(m\_fib(N)), \phi(fib(N-1, X1)), \phi(fib(N-2, X1)))$$

and

$$\min(\phi(m\_fib(N)), \phi(fib(N-1, X1)), \phi(fib(N-2, X1)))$$

Since

$$\max(\phi(\phi(m\_fib(N)), fib(N-1, X1)), \phi(fib(N-2, X1)))$$

reduces to $\max(s(N), s(N-1), s(N-2))$, which is just N, and

$$\min(\phi(m\_fib(N)), \phi(fib(N-1, X1)), \phi(fib(N-2, X1)))$$

reduces to $\min(s(N-1), s(N-2))$, which is $N-2$, the difference $N-(N-2) = 2$. It is simple to verify that $h = 2$ works for the other rules of P^{mg} and the inverted magic rules as well. For an example where FindHeight returns false, consider the binary transitive closure:

$t(X,Y) :- t(X,W), t(W,Y).$
$t(X,Y) :- e(X,Y).$

and the query $t(1,Y)$? Here, again, there are only two choices for ϕ, the size of the first argument of t, or the additive inverse of the size of the first argument. With the first choice, we will have to verify that there exists an h such that the difference

$$\max(\phi(t(X,W)), \phi(t(W,Y))) - \min(\phi(t(X,W)), \phi(t(W,Y)))$$

is less than w. Here we have

$$\max(\phi(t(X,W)), \phi(t(W,Y))) - \min(\phi(t(X,W)), \phi(t(W,Y)))$$

reduces to $\max(s(X), s(W)) - \min(s(X), s(W))$. No more simplifications can be made, since nothing is known about X and W, so the test fails. The case for the other choice of ϕ is similar. □

Finally, we turn to finding limits on the windows involved.

Procedure 3.6 [Find Limit]
Input: A program P^{mg} and a candidate windowing function ϕ.
Output: True, if there are constants c_u and c_l such that any answer a to q must have $\phi(a) \leq c_u$ and any magic fact m produced in the bottom-up evaluation of P^{mg} must have $\phi(m) \geq c_l$. In this case return c_u and c_l. Otherwise return false.
Method: We consider the cases for c_u and c_l separately. There are two cases to consider, depending on whether $\phi(X) \geq 0$ for all facts X, or $\phi(X) \leq 0$ for all facts X. One or the other must hold, since sizes are never negative and ϕ is either a sum of sizes or the inverse of a sum of sizes.
First, consider the case where $\phi \geq 0$.

1. Since by definition every answer agrees with q on the bound arguments, we may always set c_u to $\phi(q) - h$.
2. The constant c_l is more complex. Suppose that ϕ considers the arguments $a_1, \ldots, a_m$. Then if argument a_i contains structured arguments, define $l_i = 0$. Otherwise, a_i must contain an integer argument. If the variable in a_i is $X - k$, and X appears in the body of the rule in a predicate $X \geq y$, then define $l_i = y - k$. Define $c_l = \sum_1^m l_i$. Verify by expansion that c_l works for all predicates in P.

For the case where $\phi \leq 0$, the roles of c_u and c_l are reversed. If Step 2 was successful, return c_l and c_u, otherwise return failure. □

Example 3.4 Returning once more to the Fibonacci example, consider the magic rule

$m\_fib(N-2) :- N > 1, m\_fib(N).$

and assume that we are considering $\phi(fib(N,X)) = \phi(m\_fib(N)) = N$. Then FindLimit must find some constant c_l such that

$$\phi(m\_fib(N-2)) \geq c_l$$

In the body of the rule, we have that $N > 1$, which implies that $N - 2 \geq 0$, so this condition is satisfied for $c_l = 0$.

Furthermore, if the query is $fib(n, X)$?, since we have already determined that $h = 2$, we set $c_u = n - 2$.

Combining everything in Examples 4.2 through 4.4, we get that for the query $fib(n, X)$? on the fibonacci example, the relevant windows are

$$\begin{aligned}
\mathcal{W}_0 &= \{m_fib(N) \text{and} fib(N, X) \mid 0 \leq N \leq 2\} \\
\mathcal{W}_1 &= \{m_fib(N) \text{and} fib(N, X) \mid 1 \leq N \leq 3\} \\
&\vdots \\
\mathcal{W}_{n-3} &= \{m_fib(N) \text{and} fib(N, X) \mid n-3 \leq N \leq n-1\} \\
\mathcal{W}_{n-2} &= \{m_fib(N) \text{and} fib(N, X) \mid n-2 \leq N \leq n\}
\end{aligned}$$

□

Theorem 3.1 *If Algorithm 3.1 returns successfully when invoked on $\langle P, q\rangle$, then $\langle P, q\rangle$ has the sliding window property.*

4 Sliding Window Tabulation

In this section we consider Sliding Window Tabulation in more detail.

4.1 A "Naive" Description

The focus in Sliding Window Tabulation is on how we can discard facts as early as possible. An orthogonal concern is how to avoid repeating the same inferences. The Seminaive bottom-up evaluation algorithm can be adapted to ensure that Sliding Window Tabulation does not repeat any inferences. We consider this adaptation in the next subsection. (We consider the adaptation, or some equivalent technique for avoiding repeated inferences, to be an integral part of Sliding Window Tabulation. We have presented the ideas separately for ease of exposition.)

Sliding Window Tabulation of a (rewritten) program P^{mg} proceeds in two phases. In phase one, the "down" phase, only the magic rules are applied. Initially, the only magic fact is the query, which is in the highest window. For each window, processing consists of repeatedly applying the magic rules until no new facts can be derived. The important constraint is that in applying a magic rule, only facts in the current window can be used to instantiate the body. The Monotonicity condition in the definition of the sliding window property ensures that generated facts belong to either the current window or to some lower window; those in lower windows are saved for processing later.

After a window is processed, we discard all facts in this window that are not also in subsequent windows, except for "fringe" facts. A *fringe* fact is a magic fact m such that

1. m appears in the current (processed) window, but does not appear in any lower window, and
2. m was never used to instantiate a magic rule in the down phase.

Intuitively, fringe magic facts correspond to leaf or basis nodes in a derivation. All fringe facts are saved. (Also, recall that EDB facts are never discarded.) The "down" phase terminates when we have processed the lowest window.

In phase two, the "up" phase, the fringe facts, which are the only facts saved from the down phase, are used to initialize derived program facts in the lowest window; fringe facts in other windows are retained for initializing these windows when they are processed. For each window, processing consists of repeatedly applying the modified program rules in P^{mg} and the inverted magic rules until no new facts can be derived. As in the "down" phase, in applying a rule, only facts in the current window can be used to instantiate the body. Also as in the "down" phase, in the "up" phase the Monotonicity condition in the definition of the sliding window property ensures that generated facts belong to either the current window or to some higher window; those in higher windows are saved for processing later. After a window is processed, we discard all facts in this window that do not belong in subsequent windows also. The "up" phase terminates when we have processed the highest window; all answers to the query are contained in the facts that belong to this window.

4.2 A "Seminaive" Formulation

In this subsection we describe how Seminaive evaluation [Ban85] can be adapted to Sliding Window Tabulation.

4.2.1 Seminaive Evaluation

We present a brief overview of Seminaive evaluation. Seminaive evaluation works by identifying "differentials," which are new predicates that contain tuples produced in the last iteration. Consider a program that is to be evaluated using Seminaive evaluation. The program is first rewritten in order to define the new "differential" predicates. Suppose the program contains the following rule:

p :- $p_1, p_2, \ldots, p_k, q_1, q_2, \ldots, q_m$.

Let the p's be derived predicates and the q's be EDB predicates. A set of rewritten rules is generated from this rule, each of the form δp^{new} :- term, $q_1, \ldots, q_m$. There is one such rewritten rule for each term in the expansion of $(p_1^{old} + \delta p_1^{old}) \ldots (p_n^{old} + \delta p_n^{old}) - (p_1^{old} \ldots p_n^{old})$. In evaluating the program, in each iteration each of the seminaive rules is applied, followed by updating the relations as follows:

$p_i^{old} := p_i^{old} + \delta p_i^{old}$;
$\delta p_i^{old} := \delta p_i^{new} - p_i^{old}$;
$\delta p_i^{new} := \emptyset$;

The iteration continues until all the relations δp_i^{old} are empty.

4.3 Sliding Window

For convenience, we introduce the function Φ_w, where for a set of facts S, we define $\Phi_w(S)$ to be the subset of S that is contained in window w. We also define *unused*(S) to be all facts in S that were never used in any instantiation of a rule.

Recall that during the "down" phase of Sliding Window Tabulation, only the magic rules are applied. In the "up" phase, on the other hand, only the modified program rules and the inverted magic rules are applied. Sliding Window Tabulation differs from Seminaive in the updating phase following each iteration, both in the down and up phases.

Also, for each magic predicate m_p_i, we introduce the predicates $m\_p_i^{save}$ and $m\_p_i^{fringe}$ (in addition to the predicates introduced by the standard seminaive rewriting.) Intuitively, $m\_p_i^{save}$ stores magic facts that belong to a window other than the window currently being processed; m_p^{fringe} stores the fringe magic facts encountered during the "down" phase. As is discussed below, there is no need for p_i^{save} or p_i^{fringe} for program predicates p_i.

Consider first the down phase. The initialization is simple — the current window is set to the highest window, $\mathcal{W}_{c_u}$ and all relations are empty, with the exception that $\delta m\_q^{old} = seed$, where q is the query predicate and $seed$ is the magic fact corresponding to the query.

As with the standard Seminaive evaluation, all magic rules are applied in every iteration. However, instead of performing the Seminaive updates at the end of an iteration in the processing of window w, perform the following assignments for each magic predicate m_p_i:

1. $m\_p_i^{save} := m\_p_i^{save} + \delta m\_p_i^{new} - \Phi_w(\delta m\_p_i^{new})$;
2. $m\_p_i^{old} := m\_p_i^{old} + \delta m\_p_i^{old}$;
3. $\delta m\_p_i^{old} := \Phi_w(\delta m\_p_i^{new}) - m\_p_i^{old}$;
4. $\delta m\_p_i^{new} := \emptyset$;

Step 1) just saves facts in lower windows for later processing. Steps 2) – 4) are the usual seminaive updates, except that Step 3) only retains facts in the current window from $\delta m\_p_i^{new}$ (recall that the remaining facts are saved for later processing, in Step 1)).

The processing of a window continues until all relations $\delta m\_p_i^{old}$ are empty. Next, between the processing of windows w and $w-1$, unless of course w is the lowest window, the following updates must be performed:

5) $\delta m\_p_i^{old} := \Phi_{w-1}(m\_p_i^{save})$;
6) $m\_p_i^{old} := \Phi_{w-1}(m\_p_i^{old})$;
7) $m\_p_i^{fringe} := m\_p_i^{fringe} + unused(\Phi_w(m\_p_i^{old}) - \Phi_{w-1}(m\_p_i^{old}))$;

Steps 5) – 6) initialize the processing of the next window $(w-1)$. Note that Step 6) just discards some facts from window w that have been processed and are no longer needed. Step 7) saves "fringe" facts, to be used later in the up phase.

The down phase terminates after the processing of the lowest window has been completed. At this point, we perform the update

8) $m\_p_i^{fringe} := unused(m\_p_i^{old})$;

for each magic predicate.

To initialize the "up" phase, we begin with the updates

9) $m\_p_i^{save} := m\_p_i^{fringe}$;
10) $\delta m\_p_i^{old} := \Phi_{c_l}(m\_p_i^{save})$;

In the up phase, we fire the program rules and the inverted magic rules. The updates to the magic predicates after each iteration are identical to those of the down phase. However, the updates to the program predicates are simplified, for the following reason: for any program predicate p_i, the predicate p_i^{save} is uniformly empty. This follows because every original program rules with head p_i is "guarded" by the magic predicate $m\_p_i^{old}$, which at all times contains only facts in the current window. Hence the updates for program predicates are just the usual seminaive updates, repeated here for convenience:

11) $p_i^{old} := p_i^{old} + \delta p_i^{old}$;
12) $\delta p_i^{old} := \delta p_i^{new} - p_i^{old}$;
13) $\delta p_i^{new} := \emptyset$;

In between the processing of windows w and $w+1$ on the up phase, where w is not the top window, the following updates must be performed:

14) $\delta m\_p_i^{old} := \Phi_{w+1}(m\_p_i^{save})$;
15) $m\_p_i^{save} := m\_p_i^{save} - \Phi_{w+1}(m\_p_i^{save})$;
16) $m\_p_i^{old} := \Phi_{w+1}(m\_p_i^{old})$;
17) $p_i^{old} := \emptyset$;

Step 14) initializes the processing of the next window $(w+1)$. Step 15) removes facts that have been selected for processing (in Step 14)) from $m\_p_i^{save}$. Steps 16) – 17) discard facts from window w that have been processed and are no longer needed. We illustrate this evaluation algorithm with an example.

Example 4.1 Consider again the Fibonacci program of Example 2.1 with the query $fib(n, X)$?. The result of applying Magic Templates to this program appears in Example 2.1. We now turn to evaluating this program using Sliding Window Tabulation.

First we consider the down phase. Recall that the windowing function here is $\phi(m\_fib(N)) = N$, the window size is 2, and that if the original query was $fib(n, X)$?, the bounds are $c_u = n-2$ and $c_l = 0$. Table 1 gives the values for the relations in question at the end of each iteration in the "down" phase for the query $fib(5, X)$?. The starting window for the "down" phase is $\mathcal{W}_3$. Table 2 gives the values for the relations in question at the end of each iteration in the "up" phase. □

4.4 Properties

First, we verify that Sliding Window Tabulation correctly evaluates programs that have the sliding window property.

Theorem 4.1 *Let $\langle P, q\rangle$ have the sliding window property. Then Sliding Window Tabulation computes all answers to q and terminates.*

Next, we turn to the efficiency of the tabulation. One key advantage of seminaive as compared to naive is that it never repeats a derivation. This property is known as the "seminaive property." In the next theorem we show that this is true of Sliding Window Tabulation also has the seminaive property.

Theorem 4.2 *Let $\langle P, q\rangle$ have the sliding window property. Then Sliding Window Tabulation has the seminaive property.*

Window	Iteration	δm_fib^{old}	m_fib^{old}	m_fib^{save}	m_fib^{fringe}
3	init	{(5)}	$\emptyset$	$\emptyset$	$\emptyset$
3	1	{(4),(3)}	{(5)}	$\emptyset$	$\emptyset$
3	2	$\emptyset$	{(5),(4),(3)}	{(2),(1)}	$\emptyset$
2	init	{(2)}	{(4), (3)}	{(1)}	$\emptyset$
2	1	$\emptyset$	{(4),(3),(2)}	{(1),(0)}	$\emptyset$
1	init	{(1)}	{(3),(2)}	{(0)}	$\emptyset$
1	1	$\emptyset$	{(3),(2),(1)}	{(0)}	$\emptyset$
0	init	{(0)}	{(2),(1)}	$\emptyset$	$\emptyset$
0	1	$\emptyset$	{(2),(1),(0)}	$\emptyset$	$\emptyset$
end		$\emptyset$	$\emptyset$	$\emptyset$	{(0),(1)}

Table 1: "Down" phase of evaluation of $fib(5, X)$?.

The overall goal of Sliding Window Tabulation is to limit the storage required by the program evaluation. The following set of definitions, culminating in Theorem 4.3, give bounds on the space efficiency of Sliding Window Tabulation.

Definition 4.1 Let $\langle P, q\rangle$ have the sliding window property, with ϕ the ordering function on the facts and goals of P^{mg}. Also, let $\mathcal{P}$ be the set of goals and facts produced in the Sliding Window Tabulation of P^{mg}. Then the *goal width* of $\langle P, q\rangle$ is the maximum, over all constants c, of the number of goals $g \in \mathcal{P}$ such that $\phi(g) = c$. Similarly, the *fact width* of $\langle P, q\rangle$ is the maximum, over all constants c, of the number of facts $f \in \mathcal{P}$ such that $\phi(f) = c$. The *width* of $\langle P, q\rangle$ is the larger of the goal or fact widths for $\langle P, q\rangle$. □

Definition 4.2 Let $\langle P, q\rangle$ have the sliding window property, with ϕ the ordering function on the goals and facts of P. Then the *goal span* of $\langle P, q\rangle$ is the maximal value s such that for some rule firing in the Sliding Window Tabulation of P^{mg}, goal m_p_1 appears in the head, p_2 appears in the body, and $\phi(m\_p_1) - \phi(p_2) = s$. Similarly, the *fact span* of $\langle P, q\rangle$ is the maximal value s such that for some rule firing in the Sliding Window Tabulation of P^{mg}, fact p_1 appears in the head, p_2 in the appears in the body, and $\phi(p_1) - \phi(p_2) = s$. The *span* of $\langle P, q\rangle$ is the larger of the goal span or fact span of $\langle P, q\rangle$. □

Definition 4.3 Let $\langle P, q\rangle$ have the sliding window property. Then the *basis width* b of $\langle P, q\rangle$ is the number of relevant basis facts determined by the down phase of the sliding window evaluation of $\langle P, q\rangle$. □

Theorem 4.3 *Suppose that $\langle P, q\rangle$ has width w, span s, height h, and basis width b. Then Sliding Window Tabulation stores at most $w(s + h) + b$ goals or facts at any given time.*

Corollary 4.1 *Suppose $\langle P, q\rangle$ has constant width, span, height, and basis width, and furthermore that any goal or fact of $\langle P, q\rangle$ can be stored in constant space. Then Sliding Window Tabulation runs in constant space.*

Example 4.2 Returning to the Fibonacci example, we have that $s = 2$, $w = 1$, $h = 2$, and $b = 2$. Since each fact is an integer and each goal is a pair of integers, if we assume that an integer can be stored in constant space, then Sliding Window Tabulation is constant space on Fibonacci. An example of a program on which Sliding Window Tabulation runs in linear space is given in Section 5. □

The time efficiency of Sliding Window Tabulation is more difficult to analyze than the space efficiency. The simplest way to calibrate the performance of Sliding Window Tabulation on $\langle P, q\rangle$ appears to be a comparison with the seminaive evaluation of P^{mg}. Even this comparison is not straightforward, for the following reasons:

1. In some cases, sliding window "overtabulates". That is, it may compute facts that are not computed by seminaive evaluation of P^{mg}; these facts are not relevant to the query.

   To understand why, consider the "up" phase. This phase is initialized using the set of relevant basis facts. Subsequently, the modified original rules in P and the inverted magic rules are fired repeatedly. This eliminates the need to store all magic facts from the "down" phase, but it raises the possibility of computing irrelevant program facts. Intuitively, this happens when some magic fact m can be generated from two distinct magic facts, say m_1 and m_2, where only one of the magic facts was produced on the way down. On the way back up, there is no way to know which of the two magic facts produced m on the way down, so both are generated on the way up.
2. There are overheads in Sliding Window Tabulation that are not present in seminaive evaluation of P^{mg}. For example, when a new fact f is produced, we must evaluate $\phi(f)$ before deciding where the fact should be saved. As another example, when "sliding" the window, any facts in the saved relations that belong in the new current window must be found, and moved from the save relation to the corresponding "*new*" relation.
3. On the other hand, often the number of facts stored by sliding window tabulation is much less than that stored

Window	Iteration	δfib^{old}	fib^{old}	δm_fib^{old}	m_fib^{old}	m_fib^{save}
0	init	$\emptyset$	$\emptyset$	{(0),(1)}	$\emptyset$	$\emptyset$
0	1	{(0,1),(1,1)}	$\emptyset$	{(2)}	{(0),(1)}	{(3)}
0	2	{(2,2)}	{(0,1),(1,1)}	$\emptyset$	{(0),(1),(2)}	{(3),(4)}
0	3	$\emptyset$	{(0,1),(1,1),(2,2)}	$\emptyset$	{(0),(1),(2)}	{(3),(4)}
1	init	$\emptyset$	{(1,1),(2,2)}	{(3)}	{(1),(2)}	{(4)}
1	1	{(3,3)}	{(1,1),(2,2)}	$\emptyset$	{(1),(2),(3)}	{(4),(5)}
1	2	$\emptyset$	{(1,1),(2,2),(3,3)}	$\emptyset$	{(1),(2),(3)}	{(4),(5)}
2	init	$\emptyset$	{(2,2),(3,3)}	{(4)}	{(2),(3)}	{(5)}
2	1	{(4,5)}	{(2,2),(3,3)}	$\emptyset$	{(2),(3),(4)}	{(5),(6)}
2	2	$\emptyset$	{(2,2),(3,3),(4,5)}	$\emptyset$	{(2),(3),(4)}	{(5),(6)}
3	init	$\emptyset$	{(3,3),(4,5)}	{(5)}	{(3),(4)}	{(5)}
3	1	{(5,8)}	{(3,3),(4,5)}	$\emptyset$	{(3),(4),(5)}	{(6),(7)}
3	2	$\emptyset$	{(3,3),(4,5),(5,8)}	$\emptyset$	{(3),(4),(5)}	{(6),(7)}

Table 2: "Up" phase of evaluation of $fib(5, X)$?.

by seminaive. The large number of facts to be stored can slow their retrieval, in the worst case requiring a great deal of I/O that is not required by sliding window tabulation.

We can, however, prove the following two theorems. The first, Theorem 4.4, gives a worst case upperbound on the number of inferences; the second, Theorem 4.5, shows that much better performance can be guaranteed if the magic rules in P^{mg} are "invertible".

Theorem 4.4 *Sliding Window Tabulation of $\langle P, q\rangle$ never infers more program facts than the seminaive evaluation of P.*

Note that the above theorem only addresses the set of program facts that are inferred; it can be extended by noting that Sliding Window never infers more facts than seminaive evaluation of P plus the magic facts inferred in seminaive evaluation of P^{mg}.

Definition 4.4 [Invertibility] A magic rule r in P^{mg} is said to be invertible if the following holds. Suppose that a r is instantiated so that the head is $m1$ and the (only) magic fact in the body is $m2$. Given $m1$ and r, we should be able to determine $m2$. □

Theorem 4.5 *Suppose that all the magic rules in P^{mg} are invertible, and that the Seminaive evaluation of P^{mg} makes M magic fact inferences and P program fact inferences. Then Sliding Window Tabulation of $\langle P, q\rangle$ makes at most $2M$ magic fact inferences and P program fact inferences.*

5 An Example

Deductive database technology has been proposed as a useful tool in DNA sequence analysis. The Longest Common Subsequence (LCS) is representative of some of the low-level problems that are involved in this type of analysis. We are given two strings, say $A = a_0a_1 \ldots a_{m-1}$, and $B = b_0b_1 \ldots b_{n-1}$, where the a_i and b_j are drawn from some common alphabet. The desired answer is the maximal x such that there is a string $C = c_0c_1 \ldots c_{x-1}$, where C is a subsequence of both A and B. Note that "subsequence" differs from "substring" in that the members of a subsequence C of A and B need not appear contiguously in either A or B; all that is required is that the elements of C appear in the same order in A and B.

Hirchberg [Hir75] gives the following program to compute the LCS of two strings. (This is also discussed by Bird [Bir80] in the context of tabulation.) To express the problem in logic programming notation, we represent the string $A = a_0a_1 \ldots a_{m-1}$ by the facts $a(0, a_0)$, $a(1, a_1)$, ..., $a(m-1, a_{m-1})$. Similarly, the string $B = b_0b_1 \ldots b_{n-1}$ is represented as $b(0, b_0)$, $b(1, b_1)$, ..., $b(n-1, b_{n-1})$. Then the following program defines the relation $lcs(M, N, X)$, with the intended meaning that the longest common subsequence of A beginning at a_M and B beginning at B_N is of length X.

$$
\begin{aligned}
&lcs(m, N, 0).\\
&lcs(M, n, 0).\\
&lcs(M, N, X) \text{ :- } M < m, N < n, a(M, C), b(N, C),\\
&\qquad lcs(M+1, N+1, X-1).\\
&lcs(M, N, X) \text{ :- } M < m, N < n, a(M, C), b(N, D), C <> D,\\
&\qquad lcs(M+1, N, X1), lcs(M, N+1, X2),\\
&\qquad X = \max(X1, X2).
\end{aligned}
$$

The longest common subsequence of the two strings is given by the query $lcs(0, 0, X)$?.

First, if we use Prolog to evaluate this query, in the worst case the running time is $\Omega(\binom{m+n}{m})$. As noted in the introduction, this is impractically large for all but the smallest m and n. Another approach to evaluating the query is to use Magic Templates to rewrite the program, then to evaluate the result bottom-up. The resulting Magic rules are:

$$
\begin{aligned}
&m\_lcs(1, 1).\\
&m\_lcs(M+1, N+1) \text{ :- } M < m, N < n, a(M, C), b(N, C),\\
&\qquad m\_lcs(M, N).\\
&m\_lcs(M+1, N) \text{ :- } \quad M < m, N < n, a(M, C), b(N, D),
\end{aligned}
$$

$$
\begin{aligned}
& && C <> D, m\_lcs(M,N).\\
& m\_lcs(M,N+1) \text{ :- } && M < m, N < n, a(M,C), b(N,D),\\
& && C <> D, m\_lcs(M,N).
\end{aligned}
$$

while the modified original rules are

$$
\begin{aligned}
& lcs(m,N,0) \text{ :- } && m\_lcs(m,N).\\
& lcs(M,n,0) \text{ :- } && m\_lcs(M,n).\\
& lcs(M,N,X) \text{ :- } && m\_lcs(M,N), M < m, N < n,\\
& && a(M,C), b(N,C), lcs(M+1,N+1,X-1).\\
& lcs(M,N,X) \text{ :- } && m\_lcs(M,N), M < m, N < n,\\
& && a(M,C), b(N,D), C <> D,\\
& && lcs(M+1,N,X1), lcs(M,N+1,X2),\\
& && X = \max(X1,X2).
\end{aligned}
$$

One may verify that when evaluated bottom-up using Seminaive, and assuming constant time access to memoed facts, this program is $O(mn)$ time in the worst case. However, the program is also $O(mn)$ in space. For large m and n, this will also be impractical. Sliding Window Tabulation can be used to reduce the space requirement to $O(m+n)$ (with time complexity remaining $O(mn)$).

In order to apply Sliding Window Tabulation, we first need to verify that the LCS program does indeed have the sliding window property by running Algorithm 3.1. First, Algorithm 3.1 attempts to find a windowing function ϕ. Since there are two bound arguments in the magic program, there are six choices for ϕ. The Sliding Window Detection algorithm will choose $\phi(lcs(N,M,X)) = \phi(m\_lcs(N,M)) = -(N+M)$. This means that the relevant windows for the query $lcs(m,n,X)$? are

$$
\begin{aligned}
\mathcal{W}_{-(m+n)} &= \{\text{all } lcs(N,M,X) \text{ and } m\_lcs(N,M) \mid \\
&\quad -(m+n) \leq M+N \leq -(m+n)+1\}\\
\mathcal{W}_{-(m+n)+1} &= \{\text{all } lcs(N,M,X) \text{ and } m\_lcs(N,M) \mid \\
&\quad -(m+n)+1 \leq M+N \leq -(m+n)+2\}\\
&\ \vdots\\
\mathcal{W}_{-2} &= \{\text{all } lcs(N,M,X) \text{ and } m\_lcs(N,M) \mid \\
&\quad -2 \leq M+N \leq -1\}\\
\mathcal{W}_{-1} &= \{\text{all } lcs(N,M,X) \text{ and } m\_lcs(N,M) \mid \\
&\quad -1 \leq M+N \leq 0\}
\end{aligned}
$$

Since $0 \leq M \leq m$ and $0 \leq N \leq n$, each window can contain at most $n+m$ facts or facts.

Now consider the specific instance of the problem $A = acbc$ and $B = cabb$. The query we wish to ask is $lcs(0,0,X)$?, and the correct answer is $lcs(0,0,2)$. (There are two subsequences of length 2: *ab* and *cb*.) To save space we do not show the value of every relation on every iteration of the evaluation. Instead, in Table 3 we show tuples of m_lcs and lcs computed by the up and down phases of Sliding Window Tabulation, partitioned by ϕ value.

Note that unlike the case with the Fibonacci example, not all of the basis program facts are relevant. Specifically, none of $m\_lcs(0,4)$, $m\_lcs(2,4)$, $m\_lcs(4,4)$, or $m\_lcs(4,0)$ are generated. Also, notice that the basis facts do not all appear in the same window.

Also, this is an example of overtabulation. Specifically, the magic facts $m\_lcs(1,1)$, $m\_lcs(2,0)$, or $m\_lcs(3,0)$ are not generated on the way down. However, in the "up" phase, the facts $m\_lcs(1,1)$, $m\_lcs(2,0)$, and $m\_lcs(3,0)$, and the corresponding lcs facts, are computed. If the magic facts were retained for the "up" phase, instead of being recomputed by the inverted magic rules, these lcs facts would not have been generated.

6 Conclusion

We have presented a broad framework for compile-time garbage collection in bottom-up evaluation of logic programs. Since the space requirements for bottom-up methods are typically much greater than for top-down methods, this is an important area for optimization. Our results can be extended in many ways; in particular, we are considering the following problems.

- Refining sliding window techniques.

  The techniques presented here can be refined in many ways, including devising stronger tests for applicability, and developing techniques for "sliding" windows in bigger increments, thereby minimizing the processing of windows that contain few or no facts.

- Multiple recursive cliques.

  The Independence condition for the applicability of Sliding Window Tabulation disallows the dependence of magic predicates on any derived program predicates. If a program P contains more than one recursive clique, the magic predicates of one clique may depend upon program predicates from other cliques. It is desirable to extend Sliding Window Tabulation to deal with such programs.

- Dynamic methods.

  Sliding Window Tabulation is a static method in that it tries to determine window functions ϕ at compile time. Sometimes, it may be possible to design suitable windows only at run-time. A good example is a program that traverses an acyclic graph, say a part-subpart hierarchy, and (possibly) does some additional computation. In such cases, an interesting problem is to devise *dynamic* strategies that, possibly through some auxiliary run-time computation and/or additional stored facts, identify a set of windows that result in significant space savings overall.

- Integrating with general bottom-up evaluation.

  Finally, a number of issues must be addressed in order to incorporate the tabulation techniques investigated here into a system based upon rewriting and seminaive evaluation. For example, it is likely that while tabulation is not applicable to the entire program, it is applicable to a subprogram. To deal effectively with this situation, techniques must be developed to integrate the optimizations that are possible for the subprogram into the evaluation of the entire program.

	down phase	up phase	
ϕ	*m_lcs*	*m_lcs*	*lcs*
0	{(0,0)}	{(0,0)}	{(0,0,2)}
−1	{(1,0),(0,1)}	{(1,0),(0,1)}	{(1,0,2),(0,1,2)}
−2	{}	{((1,1),(2,0)}	{((1,1,1),(2,0,1)}
−3	{(2,1),(1,2)}	{(2,1),(1,2),(3,0)}	{(2,1,1),(1,2,1),(3,0,1)}
−4	{(3,1),(2,2),(1,3)}	{(3,1),(2,2),(1,3)}	{(3,1,0),(2,2,1),(1,3,1)}
−5	{(4,1),(3,2),(2,3),(1,4)}	{(4,1),(3,2),(2,3),(1,4)}	{(4,1,0),(3,2,0),(2,3,1),(1,4,0)}
−6	{(3,3),(4,2)}	{(3,3),(4,2)}	{(3,3,0),(4,2,0)}
−7	{(3,4),(4,3)}	{(3,4),(4,3)}	{(3,4,0),(4,3,0)}
−8	{}	{}	{}

Table 3: Values for *m_lcs* and *lcs* in up and down phases in evaluation of $lcs(0, 0, X)$?.

The tradeoff between recomputation and storage has received little attention in the domain of deductive database programs, and to our knowledge has not been addressed at all in the context of bottom-up evaluation strategies. This paper demonstrates the potential gains from considering this problem by presenting bottom-up evaluation schemes that avoid recomputation without saving every intermediate result for the duration of the computation.

Acknowledgements

Divesh Srivastava and S. Sudarshan made several valuable comments on an earlier draft of this paper.

References

[Ban85] Francois Bancilhon. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Database and AI Systems.* Springer-Verlag, 1985.

[Bir80] R. S. Bird. Tabulation techniques for recursive programs. *Computing Surveys*, 12(4):403–417, December 1980.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, Cambridge, Massachusetts, March 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[Hil76] J. Hilden. Elimination of recursive calls using a small table of "randomly" selected function values. *Nordisk Tidsckrift For Informationsbehandling (BIT)*, 16(1):60–73, 1976.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.

[MR90] Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1990. To appear.

[NRSU89] Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.

[Ram88] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[RBK88] Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.