Incremental Evaluation of Tabled Prolog: Beyond Pure Logic Programs

Diptikalyan Saha and C. R. Ramakrishnan

Department of Computer Science,
State University of New York at Stony Brook
Stony Brook, New York, 11794-4400, U.S.A.
E-mail: {dsaha, cram}@cs.sunysb.edu

Abstract. Tabling, or memoization, enables incremental evaluation of logic programs. When the rules or facts of a program change, we need to recompute only those results that are affected by the changes. The current algorithms for incrementally maintaining memo tables treat insertion of facts/rules differently from their deletion. Hence these techniques cannot be directly applied for incremental evaluation of arbitrary tabled programs, especially those involving Prolog built-ins such as `findall`, other aggregation operations, or non-stratified negation. In this paper, we explore a simpler incremental evaluation algorithm that, based on the dynamic call graph, invalidates and re-evaluates entire calls. The algorithm is agnostic to whether a dependency adds or removes answers from tables, and hence can be applied uniformly to programs with negation, even when the negation is implicit (as is the case with certain aggregation operations). We find that the call-based algorithm is very effective in examples where the call dependencies are largely acyclic (e.g. dynamic programming examples) and is moderately effective when the dependencies contain independent cyclic components (e.g. data flow analysis problems). This is the first practical algorithm to handle all legal tabled logic programs for which incremental evaluation is meaningful.

1 Introduction

Tabled resolution for logic programs [6, 26] alleviates some of the well-known problems of Prolog, including susceptibility to looping, repeated subcomputations, and unsatisfactory semantics for negation. Tabled resolution-based systems evaluate programs by memoizing subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a set of tables. When resolving a subgoal, if it is present in the call table, then it is resolved against the answers recorded in the corresponding answer table; otherwise the subgoal is entered in the call table, and its answers, computed by resolving the subgoal against program clauses, are also entered in the answer table. Implementations of tabling [9, 20, 27, 28, e.g.] have become stable and efficient and practical applications can be developed by encoding them as high-level logic programs [8, 18].

Tabling enables *incremental* evaluation: when some facts or rules in a program change, we can recompute only the results affected by the changes, instead of re-evaluating the program from scratch. The crucial questions for incremental evaluation are how to detect which table entries need to change, and how to compute the changes.

Based on earlier works on view maintenance in databases [10, e.g.], we have developed time- and space-efficient techniques for incremental evaluation of tabled logic programs [22, 23, 25]. These techniques, based on maintaining dependencies between answers, use separate algorithms for handling additions and deletions incrementally. These techniques have been highly effective for incremental evaluation of large definite logic programs (e.g. points-to analysis for C programs), and have been integrated into experimental versions[1] of the XSB logic programming system [27].

However, these techniques cannot be readily applied to arbitrary tabled logic programs, especially those that use aggregation and other Prolog built-ins, or have non-stratified negation. In the presence of non-monotonic operators, it is often difficult to determine whether the addition of an answer to a table results in addition or deletion of an answer to another table.

In this paper, we present an incremental evaluation algorithm that is based on *call* dependencies instead of answer dependencies, and process insertions as well as deletions using a single method. At a high level, the technique works as follows. When facts or rules of a program change, we first mark all calls in tables whose answers may be affected by this change. In the next step we re-evaluate the marked calls. Naive re-evaluation is often inefficient since the call dependencies are too coarse. Our algorithm chooses calls to be re-evaluated optimally, and sequences the re-evaluations judiciously to minimize the number of wasteful computations (see Section 3).

The salient advantages of this technique are:

- The technique can be used on any tabled program, regardless of the use of intermediate non-tabled predicates and Prolog built-ins.
- The technique is agnostic to the sign of a dependency— i.e. whether a call depends negatively or positively on another— and hence can be used without change on general logic programs: *even those with non-stratified negation*.
- The re-evaluation phase issues calls in an optimal order, re-evaluating calls only when needed, and resulting in good performance in practice.
- Call graphs are generally small, and hence the technique scales to large examples.

We also present an extensive experimental evaluation of this new technique (see Section 4). We present the results for evaluating a wide variety of programs: dynamic programming examples, points-to analysis for C programs, data flow analysis of C programs, and validation of XML documents with respect to DTDs. We survey the closely related prior work in Section 5 and conclude with a discussion on the extensions to the new incremental evaluation techniques (Section 6).

2 Preliminaries

We first review certain concepts from SLG resolution that help formalize our incremental algorithm. We assume familiarity with the standard logic programming definitions of terms, formulas, predicates, Horn clauses, rules, facts, and unification [14].

Our technical development is based on the SLG resolution [6]; however the definitions as well as the results of this paper can be ported to other tabled evaluation schemes

[1] See `http://www.lmc.cs.sunysb.edu/~dsaha/symspt/`

as well [9, 28, e.g.]. Given a program P and an initial query q, the set of call tables constructed by SLG resolution is denoted by *calls*(q, P). The set of answers computed for a subgoal q over program P is denoted by *ans*(q, P). The set of all answer tables constructed during evaluation of a query q, denoted by *answer_tables*(q, P) is given by the collection $\{ans(q', P) \mid q' \in \textit{calls}(q, P)\}$.

In SLG resolution derivations are captured as a proof *forest*, with each tree in the forest corresponding to an answer table. The model we present here abstracts away operational details that are irrelevant to the results of this paper. A more fine-grained abstract operational model of tabled resolution can be found in [5]. Moreover, for simplicity, the following definition is based on definite Horn clause programs (*i.e.* no negative literals in clause bodies); nevertheless, the definitions can be extended to cover general logic programs (see [21]).

SLG Resolution: SLG resolution [6] associates an answer table with each tree in the proof forest. Given a program P and a query q, tabled resolution proceeds by building the proof forest using a sequence of the following four operations, starting with a Program Clause Resolution operation for q.

Program Clause Resolution: A proof tree in the forest is extended by one step using OLD-resolution [26].

New Subgoal: This operation is applicable whenever a tabled subgoal g is the selected literal at a node that currently appears as a leaf, and there is no tree with g as the root. This operation creates a new proof tree with g as the root for computing the answers for g using program clause resolution.

New Answer: Applicable whenever a new answer a has been computed for a tabled subgoal g (*i.e.*, whenever a success leaf is derived in the tree for g), this operation places a in the answer table for g.

Answer Clause Resolution: A proof tree in the forest is extended by one step by resolving a tabled subgoal g with one of the answers in g's answer table.

The construction of the proof forest terminates when none of the above operations can be applied. The above description of SLG resolution follows the development of operational semantics of SLG in [21]. Note that *Completion* operation of SLG resolution does not lead to a growth in SLG forest and hence is hence treated separately from the above four operations.

Definition 1 (Subgoal Dependency Graph [21]) *The subgoal dependency graph due to evaluating query q over a program P is a directed graph (V, E) such that (i) V is the set of all tabled subgoals that occur as roots of trees in the SLG forest (i.e. the entries in the call table); and (ii) $(c_1, c_2) \in E$, i.e. there is an edge from c_1 to c_2 if c_2 occurs as a selected literal in a tree rooted at c_1.*

Subgoals are also known as *calls* and an edge (c_1, c_2) in the subgoal dependency graph means that c_1 calls c_2. The subgoal dependency graph obtained when resolving the query `r(1,X)` over the program in Figure 1(a) is given in Figure 1(b).

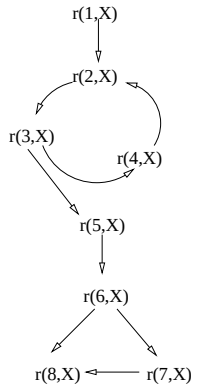
3 Incremental Evaluation Based On Call Dependencies

We consider incremental evaluation of tabled programs, where facts or rules may be added or deleted after query evaluation is completed. Each complete query evaluation

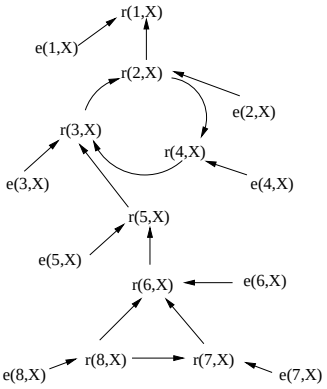
```
:- table r/2.
r(X,Y) :- e(X,Y).
r(X,Y) :- e(X,Z),
          r(Z,Y).

e(1,2).
e(2,3).
e(3,4).
e(3,5).
e(4,2).
e(5,6).
e(6,7).
e(6,8).
e(7,8).
```

(a)



(b)



(c)

Fig. 1. Example program (a); and the subgoal dependency graph (b) and called-by graph (c) for evaluating `r(1,X)`

is called a *run*. Between each run, a set of rules in the program may change. We denote this set by C and partition C into two sets C^+ and C^- that contain the added and deleted rules respectively. Given a program P, the changed program P' obtained by applying the changes in C is given by $P' = P \cup C^+ - C^-$. Note that our technical development is general and considers changes to a program's *rules*. Facts, which are rules with empty bodies, naturally become a special case.

Our algorithm is based on tracking dependencies between calls during query evaluation. In the absence of any other information (e.g. dependencies between answers), least work that a call-dependency-based algorithm can do to incrementally maintain tables after a program change is related to the number of calls whose answers have to be modified due to the change. Hence the set of changed calls, defined formally below, gives a upper bound on the performance of our incremental algorithms.

Definition 2 (Changed Calls) *Let P be a program, $C = C^+ \cup C^-$ be the set of rules that are changed, and $P' = P \cup C^+ - C^-$ be the changed program. Let Q be the set of calls due to evaluation of some query over P. The set of changed calls, denoted by changed(P, C) is the set of all calls in Q such that ans$(q, P) \not\equiv$ ans(q, P').*

In terms of our implementation in the XSB system, predicates whose definition may change are marked explicitly by the user as `volatile`. It should be noted that the XSB system supports the declaration of facts and rules that are dynamically loaded (i.e. not compiled) as `dynamic`. Not all dynamically loaded fact/rule bases change from run to run, and hence we use the `volatile` declaration to specifically indicate which facts/rules may be subject to change. For instance, in the program in Figure 1(a), the set of edge facts may change, and this is denoted by the declaration `volatile e/2`.

Our call-dependency-based incremental evaluation technique is based on an extension of the transpose of the subgoal dependency graph, known as the *called-by* graph.

Definition 3 (Called-By Graph) *The called-by graph due to the evaluation of query q over program P is a directed graph (V, E) such that (i) $V = V_t \cup V_f$ where V_t is the set of tabled subgoals that occur as roots of trees in the SLG forest, and V_f is the set of*

selected literals in the SLG forest that unify with the head of some volatile rule; and (ii) $(c_1, c_2) \in E$ *if and only if* c_1 *is a selected subgoal in a tree with* c_2 *as the root (i.e.* c_1 *is called by* c_2*).*

The called-by graph after evaluation of query `r(1,X)` over the program in Figure 1(a) is given in Figure 1(c). The graph captures the dependencies between tabled calls and calls to volatile predicate. It is first generated in the initial (non-incremental) evaluation, and maintained over subsequent incremental runs.

The incremental algorithm has two phases. The first is the *invalidation* phase, where calls that may be affected by the change are marked as *affected*.

Definition 4 (Initially Changed Calls) *Given a called-by graph* $G = (V, E)$ *and a non-empty set* $C = C^+ \cup C^-$ *of rules that were changed (inserted or deleted) since the last run, the set of initially changed calls, denoted by init*(G, C) *are those* $v \in V$ *such that* v *unifies with the head of some rule in* C*.*

Definition 5 (Affected Calls) *Given a called-by graph* $G = (V, E)$ *and a non-empty set* $C = C^+ \cup C^-$ *of rules that were changed (inserted or deleted) since the last run, the set of affected calls, denoted by affected(G, C), is the smallest set such that* $v \in$ *affected*(C, G) *if*

- $v \in$ *init*(G, C)*, or*
- $\exists v' \in$ *affected*(G, C) *such that* $(v', v) \in E$*.*

The set of affected calls (based on the above definition) can be found by simply traversing the called-by graph starting from the vertices that unify with changed rule heads (case (i) above). Note that the direction of edges in the called-by graph is reversed from those in the subgoal dependency graph. It is this choice of direction that enables the traversal of the called-by graph to compute the set of affected calls.

The idea behind the invalidation phase is calls that are not deemed affected are unchanged by the modification, as formally stated below:

Theorem 1 *Let* P *be an initial program,* $C = C^+ \cup C^-$ *be the set of changed rules, and* $P' = P \cup C^+ - C^-$ *be the changed program. Let* $G = (V, E)$ *be the called-by graph for some query over* P*. Then, every changed call is affected; i.e. changed*$(P, C) \subseteq$ *affected*(G, C)*.*

Naive Re-Evaluation: Theorem 1 means that when some program rules change, it is sufficient to re-evaluate the set of affected calls. Our first "naive" strategy is to remove all table entries corresponding to the affected calls (i.e. their entries in the call table, as well as their answer tables) and re-issue all affected calls, thereby computing them using SLG resolution. This phase of incremental computation is called the *re-evaluation* phase. Note that *all* affected calls are deleted to ensure that any answer derived for an affected call is based only on valid information: either rederived answers of another affected call, or existing answers of an unaffected call. While deleting the table entries for an affected call, we also remove the corresponding vertex and the edges incident on it from the called-by graph. Note that the re-evaluation may generate new vertices and edges in the called-by graph. Thus the called-by graph itself is (incrementally) modified when processing incremental changes.

For example, consider the deletion of the fact `e(3,5)` from the program in Figure 1(a). The invalidation phase identifies the calls `e(3,X)`, `r(3,X)`, `r(2,X)`, `r(4,X)` and `r(1,X)` as affected. Since these calls will be re-evaluated, the edges incident on these vertices, i.e. $e(3,X) \rightarrow r(3,X)$, $r(5,X) \rightarrow r(3,X)$, $e(4,X) \rightarrow r(4,X)$, $r(4,X) \rightarrow r(3,X)$, $e(2,X) \rightarrow r(2,X)$, $r(2,X) \rightarrow r(4,X)$, $e(1,X) \rightarrow r(1,X)$, and $r(2,X) \rightarrow r(1,X)$, are deleted from the called-by graph. In the re-evaluation phase, the call `r(1,X)` gives rise to calls `r(2,X)`, `r(3,X)`, and `r(4,X)`, and their answers are subsequently computed. These calls and the corresponding edges are added (back) to the called-by graph. Note that, answers to unaffected calls can be found directly from the tables. For example, the call `r(3,X)` uses already existing answers for `e(3,X)` and `r(5,X)`; calls such as `r(5,X)` are unaffected by the deletion and are not re-evaluated, thereby saving expensive program clause resolution steps.

Optimal Re-evaluation: The set of affected calls over-approximates the set of changed calls. In many cases, the approximation may be severe and the naive re-evaluation strategy wastefully re-evaluates unchanged calls. Consider the deletion of fact `e(7,8)` from the program Figure 1(a). The invalidation phase identifies the calls `e(7,X)`, `r(7,X)`, `r(6,X)`, `r(5,X)`, `r(3,X)`, `r(2,X)`, `r(4,X)`, and `r(1,X)` as affected. However, the set of changed calls is only `e(7,X)` and `r(7,X)`, but the naive strategy also re-evaluates all other affected calls.

We can define a better approximation to the *changed* set by considering which calls need to be recomputed (even to determine that their answers have not changed). This set, called the *recomputed* set is defined below.

Definition 6 (Recomputed Set) *Let P be a program, $C = C^+ \cup C^-$ be the set of changed rules, and $P' = P \cup C^+ - C^-$ be the changed program. Let $G = (V, E)$ be the called-by graph for some query q over P. Then, the set of recomputed calls, denoted by recomputed(G, C), is the smallest set such that $c \in$ recomputed(G, C) if*

1. *$c \in$ init(G, C), or*
2. *there is some c' such that $(c', c) \in E$ and $c' \in$ changed(P, C), or*
3. *there is some c' such that c and c' are in the same strongly connected component of G, and $c' \in$ recomputed(G, C).*

The recomputed set represents the smallest set of calls that need to be re-evaluated. The intuition behind this definition follows from the following observations:

1. Every changed call needs to be re-evaluated.
2. Every call that immediately depends on a changed call needs to be re-evaluated (even if it itself is not changed). Note that the called-by graph contains no more qualitative information on *how* the change of a call affects another. Only the program has this information embedded in it, and hence the only way to determine whether or not such a call changes is to re-evaluate it.
3. If a re-evaluated call is in a SCC, then all calls in that SCC need to be re-evaluated. For instance, when `e(3,5)` is deleted from the program in Figure 1(a), `e(3,X)` is changed, and hence `r(3,X)` is recomputed. Note that we cannot simply delete `r(3,X)`'s tables are re-evaluate it: since `r(4,X)` currently contains the answer `X=5`, and `e(3,4)` holds, we will then (incorrectly) conclude that `r(3,5)` still holds. Hence, we have to re-evaluate all mutually dependent calls simultaneously (`r(3,X)`, `r(4,X)` and `r(2,X)`, in this case).

It follows from the definition that every changed call is also in the recomputed set. It can also be readily shown that every call in the recomputed set is affected. Formally,

Proposition 2 *Let P be a program, $C = C^+ \cup C^-$ be the set of changed rules, and $P' = P \cup C^+ - C^-$ be the changed program. Let $G = (V, E)$ be the called-by graph for some query q over P. Then changed$(P, C) \subseteq$ recomputed$(G, C) \subseteq$ affected(G, C).*

The key to incremental re-evaluation based on call dependency information is to re-evaluate only the calls in the *recomputed* set. We need two basic mechanisms to accomplish this: (a) one to determine whether a re-evaluated call is changed or not, and (b) another to determine SCCs in the called-by graph.

a. Change Marking: First of all, instead of deleting all the affected tables in the invalidation phase, we simply mark their answers as (currently) invalid. Invalid answers are ignored when doing answer clause resolution. With each affected call, we also keep the number of invalid answers (in a counter called *invalid_count*), initialized to the total number of answers at the beginning of the re-evaluation phase. Finally, we keep a flag with each affected call (called *addl_answer*) to indicate whether a new answer was added to this call's answer table in the re-evaluation phase. During re-evaluation, whenever an answer is added to a table (New Answer operation in SLG), if the answer already exists but is invalid, we remove the invalid mark and decrement *invalid_count* for the table. If the answer did not exist before, we add the answer and set *addl_answer* of the call to true. When a call is completely re-evaluated (at the Completion operation of SLG), we can determine that the call is *changed* iff *addl_answer* is true or *invalid_count* is non-zero.

b. Evaluating SCCs: Finding SCCs in the called-by graph is fundamental to evaluating the *recomputed* set. Apart from the explicit use of SCC information in its definition, note that we determine whether or not a call is *changed* only after completion. This means that we need to evaluate the calls "bottom-up" through the called-by graph, and triggering re-evaluations at higher levels only after confirming that the lower-level calls have changed. This strategy, when applied to acyclic graphs has been shown to be optimal [19] (see Section 5 for a detailed discussion).

Algorithms for finding SCCs typically need an additional pass over the graph. We now describe a technique to find SCCs without making this additional pass, by slightly modifying the traversal used in the invalidation phase. This technique is based on Kosaraju and Sharir's SCC computation algorithm [7, pages 488–493], which works as follows. To find SCCs in a graph G, we first traverse G and give post-order numbers to the vertices in G. We then traverse G^T, the transpose of G, starting from the vertex with the highest post-order number; this traversal builds a spanning tree for one SCC of G. Whenever the traversal ends, we begin a new traversal from the unvisited vertex with the highest post-order number, thereby building a spanning tree for another SCC. This process continues until all vertices have been visited, enumerating all SCCs of G. The order in which SCCs are found by the Kosaraju-Sharir algorithm is a topological order in the SCC-reduced graph of G: if (v_1, v_2) is an edge in E, then the SCC containing v_1 is found at least as early as the one containing v_2.

The Re-Evaluation Algorithm: We now describe a re-evaluation algorithm that implicitly finds SCCs. In the invalidation phase, we traverse the called-by graph and assign a post-order number to each affected call.

In the re-evaluation phase, shown in Figure 2, we maintain a sequence of calls to be re-evaluated in a global sequence known as the *working sequence* (variable *ws* in the algorithm). This sequence is maintained using a heap data structure, keeping the calls in the descending order of their post-order numbers. During re-evaluation, we pick the call with the highest post-order number from this and invoke the call. Re-evaluation continues until the working sequence becomes empty. When the re-evaluation of a call c is complete, if c has changed, we add all its immediate successors in the called-by graph to the working sequence.

```
   re_eval(G, C)
1.   ws := init(G, C);
2.   while (ws is not empty)
3.      remove c, the call with the
                highest PO number from ws;
4.      call(c);

   In SLG's Completion Op. for call c:
1.   if (c.addl_answer) or
                        (c.invalid_count > 0)
2.      foreach c' such that (c, c') ∈ E
3.        if not c'.processed
4.           add c' to ws
5.           c'.processed := true
```

Fig. 2. Optimal Re-Evaluation Algorithm

Note that, during re-evaluation, if call c_2 needs answers from c_1's table, then (c_1, c_2) is an edge in the called-by graph. *Thus re-evaluation implicitly traverses the transpose of the called-by graph*. If c_1's table is either unaffected or has been recomputed completely, then c_2 can use the answers from that table. Otherwise, c_1 will also be re-evaluated. This ensures that all calls in an SCC of the called-by graph will be evaluated simultaneously.

The correctness of the algorithm, stated in the following theorem, can be established following the properties of the Kosaraju-Sharir algorithm and the definition of *recomputed* set.

Theorem 3 *The set of calls picked by the re-evaluation algorithm (line 3 of re_eval in Figure 2) is the same as the recomputed set.*

In the example, when `e(7,8)` is deleted, the reverse postorder of affected calls is given by the sequence `e(7,X)`, `r(7,X)`, `r(6,X)`, `r(5,X)`, `r(3,X)`, `r(4,X)`, `r(2,X)`, `r(1,X)`. The set of initially changed calls is {`e(7,X)`}. When `e(7,X)` is re-evaluated, its answer `e(7,8)` is removed, and hence we deem the call to have changed. This causes `r(7,X)` to be added to the working sequence. When this call is re-evaluated, it too is deemed to have changed (answer `r(7,8)` is no longer derivable). Hence we add `r(6,X)` to the working sequence. Re-evaluating `r(6,X)`, we find that it has not changed. The working sequence is now empty and the re-evaluation is complete. Thus, among the 8 affected calls, we re-evaluated only 3. It should be noted that answers to all affected calls have been marked invalid, and only a few of the affected calls are re-evaluated. Hence the re-evaluation phase ends by cleaning up: i.e. that removes the invalid mark from answers of tables that are affected but not re-evaluated. This step is straightforward and not shown in Figure 2.

4 Experiments

We evaluated the performance of the naive and optimized algorithms on various classes of table logic programs. Below we present the results of our experiments. The algo-

rithms are implemented by extending XSB logic programming system [27] (ver 2.7.1). All measurements are taken on a PC with 3GHz Pentium 4 processor with 2GB of physical memory running Linux (RedHat) version 2.6.9. Our implementation, benchmarks, additional experimental results on simple reachability analysis and push down model checking are available at [24].

Dynamic Programming: We now present the performance of incremental evaluation on a set of familiar dynamic programming problems, which are canonical examples of the advantages of memoization in both functional programming and logic programming worlds. Support graph based incremental techniques [25] cannot be directly used to capture the answer dependencies in these problems due to the use of aggregation operations (min, max etc.). Figure 3 summarizes the relative time performance of incremental evaluation (w.r.t. from-scratch evaluation time) averaged over several possible changes for different dynamic programming problems: longest common subsequence (LCS), minimum edit distance (EDD), and matrix chain multiplication (MM). The figure presents the average performance

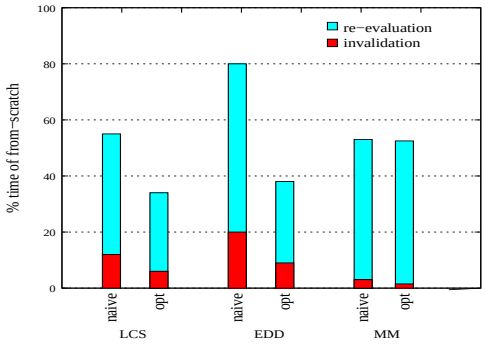


Fig. 3. Performance on Dynamic Prog. problems

LCS: We evaluated the performance of incremental evaluation on LCS by changing the character at some position in one of the strings. On average, 50% calls are affected, and 11% of are changed and 15% are recomputed. Although only 15% of the calls are re-evaluated by our optimized incremental algorithm, the time taken for re-evaluation is close 30%. This is due to the overhead of answer clause resolution that our current implementation performs (from the top-level) even for calls that are not recomputed. Incremental evaluation of LCS is sensitive to positions of characters in the string that were changed. This can be readily seen from Figure 4.

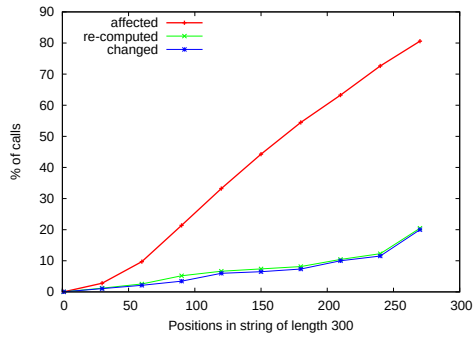


Fig. 4. The effect of the changed position on the performance of incremental evaluation of LCS.

EDD: The solution to EDD is very similar to that of LCS. The two problems differ in the number of dependent calls for each call. Every call in EDD evaluation is connected to 3 calls in the call-by graph whereas in LCS each call is connected to at most 2 calls. Hence the number of affected calls in higher in EDD, resulting in higher invalidation time.

MM: For matrix chain multiplication, we deleted one matrix from the chain and measured the incremental and from-scratch time to do the evaluation. Each affected call is also re-

computed. Hence the optimal algorithm does not show better performance over the naive one.

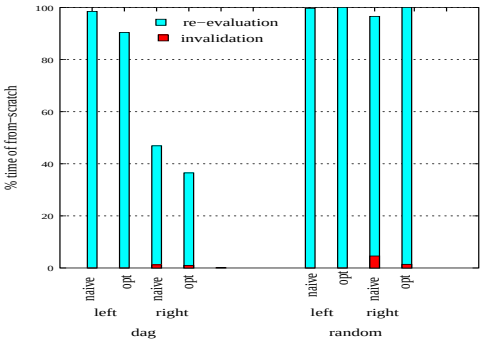


Fig. 5. Performance of Incremental Algorithms on All-Pair Shortest Path

All-Pair Shortest Path: We experimented with encodings of the all-pair shortest path problem on a directed acyclic graph having 50K nodes and randomly generated graph having 50K edges and 250 nodes (close to complete graph). We performed separate experiments with two different logic program encodings (with left and right recursion, resp.). For the almost-complete graph, incremental evaluation algorithms are not effective since almost all calls are recomputed. For DAGs, the left-recursive version shows poor incremental performance due to lack of call dependency information.

Data Flow Analysis Reaching definition analysis for imperative programs is a well-known data flow analysis which determines, for each program point, the set of variable definitions (assignments) that may reach that point [2]. We extended the intra-procedural analysis to an inter-procedural setting using the classical approach of replacing procedure calls with jumps: from the call site to the entry point of the callee, and from the exit point of the callee to the statement following the call site. The experiments were performed on various large C programs and for each benchmark 100 random statements (one per incremental run) were chosen for replacement with a skip statement. The logic programming formulation of data flow analysis uses stratified negation, and the techniques based on answer dependency [25] cannot be readily used in this case.

Benchmark	Non	Non-opt. Incremental			Opt. Incremental			% of calls	% of aff. calls	
	Incr.	Invalid	Re-eval	%	Invalid	Re-eval	%	affected	recomputed	changed
assembler	5.9477	0.0013	3.6001	60.6	0.0011	3.6393	61.2	23.5	85	1
diff	4.5451	0.0009	2.2256	49.0	0.0008	2.2358	49.2	30.9	97	1
dixie	1.7306	0.0006	0.9609	55.6	0.0005	0.9405	54.4	26.8	95	7
gnugo	4.4097	0.0008	2.3761	53.9	0.0007	2.4185	54.8	30.6	99	1
learn	1.2925	0.0005	0.5250	40.7	0.0004	0.5354	41.4	26.6	93	9
smail	5.5063	0.0010	2.8868	52.4	0.0008	2.8455	51.7	25.4	98	2

Table 1. Reaching Definitions; One statement replaced with skip

Table 1 shows that incremental algorithms takes on average 50% of from-scratch time although number of affected calls is close to 30%. Closer inspection reveal that for these examples 90% of the call nodes belong to a few non-trivial SCCs in the called-by graph. The formation of such large SCCs is due the inter-procedural jumps which introduce cycles even when the original program had no recursion. Due to the large SCCs, most affected calls are also recomputed. For example in benchmark `learn` 93% of the affected calls are recomputed but only 9% of the affected calls are changed. This suggests that the flow analysis program itself is readily incrementalized. It remains

to be seen whether the program can be reformulated to enable incremental evaluation (analogous to converting right-recursion to left to obtain effi cient tabled programs).

Pointer Analysis We used the call-graph based techniques for the incremental evaluation of Anderson's Points-to analysis [3] encoded as a tabled logic program [23]. We measured the performance of the analyzer on programs taken from C benchmarks available with PAF [16] compiler suite and SPEC95 benchmarks. The C source code is preprocessed using CIL [15] into Prolog facts representing the primitive assignment statements. Each library function was replaced by a stub representing the data flow between its formal parameters and return value and preprocessed in the same manner. The lines of code for twmc, nethack and vortex are 24959, 33993, and 67110 respectively.

Benchmark	Non	Naive Incremental			Opt. Incremental			% of calls	% of aff. calls	
	Incr.	Invalid	Re-eval	%	Invalid	Re-eval	%	affected	recomputed	changed
m88ksim	0.3911	0.0019	0.0375	10.1	0.0014	0.0252	6.8	1.1	56.4	25.2
vpr	0.6481	0.0123	0.1875	30.8	0.0080	0.1725	27.8	4.0	57.9	6.1
smail	1.6520	0.0141	1.1793	72.2	0.0061	1.1884	72.3	6.0	90.3	25.8
twmc	2.2172	0.0077	0.9345	42.5	0.0030	0.9221	41.7	2.9	85.7	6.0
nethack	0.9778	0.0053	0.8046	82.8	0.0026	0.8020	82.2	5.6	67.2	12.8
vortex	12.44	0.0408	12.1018	97.5	0.0169	11.3504	91.3	5.5	68.3	6.6

Table 2. Performance of naive and optimized algorithms on pointer analysis

Table 2 shows the relative performance of naive and optimized incremental algorithms after removal of one (source-level) statement from the benchmark programs compared to the from-scratch time. Deleting one source level assignment statement may delete multiple primitive assignments statements and hence multiple facts. The experiments results are averaged over 100 randomly chosen deletion of source statements.

Observe that the incremental times for large benchmarks are close to the non-incremental times. We investigated the vortex program to explain its behavior. Pointer analysis of vortex makes 68K calls in total of which on average 4K calls are affected. Close inspection of affected calls revealed the existence of large SCC (consisting 2.7K nodes) in the call graph. Also about 90% of the time taken by pointer analysis is attributed to the calls in the large SCC. Since the nodes in the SCC are part of the affected set, re-evaluation takes almost same time as from-scratch analysis. The calls in the SCC are also in the recomputed set and hence we do not observe any appreciable difference in the performance of the optimized algorithm relative to its naive counterpart.

XML Validation We investigated incremental validation of XML documents with respect to Document Type Defi nitions (DTD) [4]. DTD is an extended context-free grammar which defi nes a regular expression for each element type of XML document. An XML document forms a tree, and the string corresponding to an element of an XML document is the concatenation of the labels of its children. An element E is said to be *valid* with respect to a DTD D if all its children are valid, and the string corresponding to E belongs to the regular language defi ned in DTD corresponding to the type of E. An XML document is said to be *valid* with respect to a DTD D if the root element is valid. Given a document X valid with respect to an a DTD D and an update to the document X, incremental validation determines whether the updated document is still valid with respect to D.

Table 3 shown the result of applying the naive algorithm for incremental validation of XML documents for different number of elements (first column). The example XML documents and DTD describe a library catalog which contains zero or more number of books. Each book contains zero of more number of authors followed by title. Each author has a name, zero or more emails and an address. The elements types name, address, emails and title are defined as PCDATA. We generated XML documents having 1K–50K books, with up to 3 authors per book and up to 3 email addresses per author. Each update consists of deletion of one book element from the chain of book elements of the library. The number of affected calls is less than 0.01% of total number of calls.

No. of Elements	Non-Incr	Non-Opt. Incremental		
		Invalid	Re-eval	%
12K	0.1848	0.000	0.0023	1.25
120K	1.8855	0.000	0.0293	1.55
240K	3.7926	0.000	0.0604	1.59
360K	5.6746	0.000	0.0933	1.64
480K	7.6301	0.000	0.1241	1.62
600K	9.6027	0.000	0.1581	1.64

Table 3. XML Validation; deletion of one element

Note that the basic problem here is to check whether a string belongs to a regular language or not. We encoded this checker as a left-recursive DCG, and hence do not expect to see any benefits due to the optimized algorithm. In the example above, the savings due to incremental evaluation arise from reusing the prior validation of each book element. Since the number of books is large, it results in considerable savings due to incremental evaluation.

Space Overhead We measured the space overhead for keeping the called-by graph for various applications. Note that although the number of nodes in the called-by graph is bounded by the number of tabled calls, the number of edges can large. Observe from Table 4 w that space needed for the called-by graph is about 30% of the table space for most of the applications. For matrix chain multiplication with chain length n, the number of calls is $O(n^2)$ but the number of called-by graph edges is $O(n^3)$. This contributes to the large size of the called-by graph compared to its table space. For such applications, it may be better to not materialize the called-by graph, as described in 6.

Application	Table Space	Called-by Graph Size
Pointer Analysis (vortex)	51.0	13.6
Pointer Analysis (twmc)	18.3	3.5
Matrix Multiplication (chain 200)	4.0	75.0
Longest Common Subsequence (strlen 1000)	168.7	50.1
Minimum Edit Distance (strlen 600)	63.3	21.6
Reaching Definition (diff)	211.0	39.3
XML validation (60K elements)	107.0	13.0

Table 4. Space overhead (in MB) for called-by graph

5 Related Work

The idea of recording the evaluation process as a graph and using a topological order to guide incremental change propagation has been used in various fields of program analysis, model checking, functional programming, and logic programming.

Incremental attribute grammar evaluation [19] generates an acyclic dependency graph to record the functional dependencies among attribute values in the non-circular attribute grammar. The dependency graph considered there is static and acyclic; the topological order in the graph was found in a pre-processing phase and the change propagation was performed in that order. The paper showed that the change propagation was optimal. The augmented dependency graph (ADG) [1], records the dependencies between input and output values in the execution of pure functional programs having branches. The dependencies are dynamic in this setting, and an incremental topological order maintenance algorithm was used for efficiency in change propagation. However, ADGs can represent only acyclic dependencies. The dependency graph considered in our work is potentially cyclic and dynamic. Thus our change propagation algorithm applies to a more general setting than adaptive functional programming.

In [12, 13] Hermenegildo et. al. presented incremental algorithms for re-analysis of logic programs and constraint logic programs respectively. They use call dependency to propagate changes due to insertion and deletion of rules. They also presented a bottom-up deletion algorithm which uses SCC-reduced predicate dependency graph to propagate the changes from a topologically lower predicate SCC to a higher one only after the lower SCC is completely evaluated. In [17] this algorithm was improved by propagating changes through dynamic call graph instead of predicate dependency graph. The paper also points out that the more accurate dependency graph will result in more localized change propagation.

6 Discussion

In this section we discuss possible extensions to the algorithms presented in Section 3.

Lazy re-evaluation. The algorithms presented in Section 3 refreshes all answer tables such that after each incremental phase the set of answers is sound and complete with respect to the changed program. Certain applications, such as ontology management, access tables through a graphical user interface, and access some or all of the answers only when required. In such cases, it will be better to re-evaluate a call only on demand. This can be done by keeping the subgoal dependency graph to propagate demand top-down, while keeping the called-by graph to perform re-evaluations bottom-up.

Insertion for Definite Logic Programs. The algorithm presented in the paper determines which calls need to be re-evaluated, but does not prescribe what technique should be used to re-evaluate them. When the direction of the change (i.e. whether it is an addition, deletion or both) is known, it is possible to derive better techniques for re-evaluating calls. If the change made is an addition and the program has no negation, we can derive a new program that computes these changes efficiently. The rules of the new program are called "delta rules" and are derived by finite-differencing the original definite program [11, 22]. This has a potential to significantly improve incremental

evaluation times. For example, a single statement insertion using delta rules takes on average 8% of from-scratch time for pointer analysis in vortex benchmark whereas it takes 90% of from-scratch time when the affected calls are completely re-evaluated. While it is relatively straightforward to use the "delta rules" program for incrementally processing additions for predicates without negation, light-weight re-evaluation techniques for other kinds of changes and for general logic programs remains an open problem.

Mixed Strategy. In [25] we described a space efficient technique for storing answer dependencies in the form of symbolic support graph for incremental evaluation of an important class of tabled logic programs. Since answer dependency is more fine-grained than call dependency, symbolic support graph based deletion algorithm are extremely efficient in practice— taking less than 5% of from-scratch time in all the applications we have tested. We can combine the two techniques, keeping call dependencies in general but keeping symbolic support graphs to efficiently process deletions whenever possible.

Non-materialized called-by graph. Although the call dependencies are typically smaller than answer dependencies, and the number of calls is bounded by table space, we saw an example (matrix chain multiplication) where the called-by graph takes much more space than the tables themselves. It is hence worth exploring whether we can avoid storing the edges of the called-by graph, and instead compute them on the fly. It is relatively easy to derive the called-by relation a given definite logic programs. For instance, from every rule of the form $p :- q_1, q_2, \ldots, q_n$ we can derive "called-by" rules such as $\texttt{called\_by}(q_i, p) :- q_1, q_2, \ldots, q_{i-1}.\}$. Although it is not clear whether such rules can be derived for arbitrary logic programs (especially those employing impure constructs such as cuts), the computed called-by relation, wherever possible, offers a space-efficient alternative to storing large called-by graphs.

Summary. We presented an incremental evaluation algorithm based on call dependencies that can handle tabled logic programs with negation, aggregation and Prolog builtins. Experiments show that the general algorithm is useful although not as effective as the (more restricted) answer-dependency-based techniques. The algorithm identifies a small set of calls to be re-evaluated and invokes them in a particular order to ensure optimality. The actual re-evaluation itself is performed rather naively, by (effectively) removing all answers from a table to be re-evaluated and using program clause resolution to restore the answer table. More sophisticated techniques that optimize the re-evaluation itself are of significant interest. Our experience with this algorithm shows that programs written for efficient tabled evaluation may not be most suited for efficient incremental evaluation too. Developing a methodology to write efficient incremental programs (analogous to recursion transformations and supplementary tabling for tabled programs) is an important avenue of future research.

References

1. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *POPL*, pages 247–259. ACM Press, 2002.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, pages 585–718. Addison-Wesley, 1986.
3. L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Unversity of Copenhagen, 1994.

4. A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of xml documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.
5. W. Chen, T. Swift, and D. S. Warren. Efficient implementation of general logical queries. *JLP*, 1995.
6. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1):20–74, 1996.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction To Algorithms*. MIT Press, 2nd edition, 1998.
8. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems —a case study. In *ACM PLDI*, pages 117–126, 1996.
9. H. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *ICLP*, pages 181–196. Springer, 2001.
10. A. Gupta and I. Mumick. Maintenance of materialized views: Problems, techniques, and appfications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
11. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.
12. M. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental analysis of constraint logic programs. *ACM Trans. Program. Lang. Syst.*, 22(2):187–223, 2000.
13. M. V. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental evaluation of tabled logic programs. In *ICLP*, MIT Press, pages 797–811, 1995.
14. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.
15. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer-Verlag, 2002.
16. PAF. Prolangs analysis framework. Available at `http://www.prolangs.rutgers.edu/public.html`.
17. G. Puebla and M. V. Hermenegildo. Optimized algorithms for incremental analysis of logic programs. In *SAS*, pages 270–284, 1996.
18. C. R. Ramakrishnan et al. XMC: A logic-programming-based verification toolset. In *CAV*, number 1855 in LNCS, pages 576–580, 2000.
19. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *TOPLAS*, 5(3):449–477, 1983.
20. R. Rocha, F. Silva, and V. S. Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Workshop on Tabling in Parsing and Deduction*, 2000.
21. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified programs. *TOPLAS*, 8(1):1–49, 1999.
22. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *ICLP*, volume 2916 of *LNCS*, pages 389–406, 2003.
23. D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming*. ACM Press, 2005.
24. D. Saha and C. R. Ramakrishnan. A practical framework for incremental evaluation, 2005. Available at `http://www.lmc.cs.sunysb.edu/~dsaha/callg`.
25. D. Saha and C. R. Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evaluation. In *International Conference on Logic Programming (ICLP)*, 2005. See `http://www.lmc.cs.sunysb.edu/~dsaha/symspt/`.
26. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, pages 84–98, 1986.
27. XSB. The XSB logic programming system. Available at `http://xsb.sourceforge.net`.
28. N. Zhou, Y. Shen, L. Yuan, and J. You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.