Symbolic Support Graph: A Space Efficient Data Structure For Incremental Tabled Evaluation

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. In an earlier paper, we described a data structure, called support graph, for efficient incremental evaluation of tabled logic programs. The support graph records the dependencies between answers in the tables, and is crucial for efficiently propagating the changes to the tables when facts are deleted. Incremental computation with support graphs are hundreds of times faster than from-scratch evaluation for small changes in the program. However, the graph typically grows faster than the tables themselves, making it impractical to maintain the full support graph for large applications. Storing only a partial support graph reduces the space overhead, but significantly affects the incremental evaluation time.
In this paper we present a data structure, called symbolic support graph, which represents support information compactly. For a variety of useful tabled logic programs, the size of the symbolic support graph grows no faster than the table size. We demonstrate its effectiveness using a large application: a logic-programming-based points-to analyzer for C programs. The incremental analyzer shows very good scalability in terms of space usage, and is hundreds of times faster than from-scratch analysis for small changes to the program.

1 Introduction

Tabled resolution [26, 5, 7] has become an important evaluation technique in logic programming. Many implementations of tabling have now emerged [18, 29, 10, 27]. Tabling has enabled us to construct many practical applications— program analysis and verification systems [8, 16], in particular— by encoding them as high-level logic programs.

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. During resolution, if a subgoal 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. For instance, the call and answer tables created when evaluating the query `r(6,X)` over the program in Figure 1(a) is given in Figure 1(b). (The answers in the figure are labeled $a1$, $a2$ etc.)

Tabling opens up the possibility of *incremental* evaluation: when some of a program's facts or rules change, we can recompute only the results affected by the changes, instead of re-evaluating the program from scratch.

Background: Incremental evaluation of tabled programs is closely related to the well-investigated problem of materialized view maintenance in databases [12, e.g.]. Most of these works handle two kinds of changes to a program, namely, insertion and deletion of facts; update is treated as deletion followed by insertion. Incremental processing of

```
:- table r/2.
%rule 1
r(X,Y) :- b(X,Y).
%rule 2
r(X,Y) :- c(X,Z), r(Z,Y).

b(1,2). %f1
b(6,2). %f2
b(6,4). %f3

c(1,6). %f4
c(3,6). %f5
c(3,1). %f6
c(6,3). %f7
```

(a)

Calls	Answers	Supports
r(6,X)		
	[a1] **r(6,2)**	[s1] {**b(6,2)**}, [s10] {**c(6,3),r(3,2)**}
	[a2] r(6,4)	[s2] {b(6,4)}, [s11] {c(6,3),r(3,4)}
r(3,X)		
	[a3] **r(3,2)**	[s3] {**c(3,6),r(6,2)**}, [s8] {c(3,1),r(1,2)}
	[a4] r(3,4)	[s4] {c(3,6),r(6,4)}, [s9] {c(3,1),r(1,4)}
r(1,X)		
	[a5] r(1,2)	[s5] {b(1,2)}, [s6] {**c(1,6),r(6,2)**}
	[a6] r(1,4)	[s7] {c(1,6),r(6,4)}

(b) (c)

Fig. 1. Example program (a); calls and answers generated when evaluating query `r(6,X)` (b); and supports for the query evaluation (c).

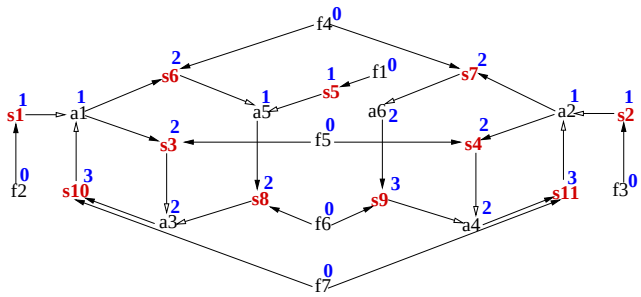

Fig. 2. Support graph for answers to query `r(6,X)` over example program in Figure 1(a).

deletion is more challenging than that of addition, especially for maintaining recursively defi ned views. This paper focusses solely on incremental processing of deletion.

The *DRed* algorithm [11], which subsumes the other recursive view maintenance algorithms, fi rst deletes all answers that may be affected by the deleted facts (the deletion phase). The second (rederivation) phase attempts to rederive the deleted answers without using the deleted facts. For instance, consider the deletion of fact `b(6,2)` from the program in Figure 1(a). Since there is one derivation of `r(6,2)` that contains `b(6,2)`, the DRed algorithm deletes `r(6,2)`. Similarly, `r(3,2)` and `r(1,2)` are also deleted. In the second phase, `r(1,2)` is rederived due to `b(1,2)`. Consequently, `r(3,2)` and `r(6,2)` are also rederived, thereby rederiving all the three originally deleted answers. The deletion-rederivation strategy appears to be universal for handling incremental deletion: it also appears in independently-developed incremental algorithms for program analysis [15, 28] and model checking [23].

Two factors make the DRed algorithm impractical. First, the deletion phase uses program clauses to propagate the deletions. Second, many of the deleted answers are rederived in the second phase, again by applying program clauses. In [19], we proposed a solution to these two problems, as described below.

Support Graph: An instance of a rule that can be used to derive an answer is known as a *support* for that answer. For instance, the supports for the answers used to evaluate the query `r(6,X)` over the program in Figure 1(a), are given in Figure 1(c). The supports in the fi gure are labeled $s1$, $s2$, etc. A support graph has answers, facts and

supports as vertices; the support graph corresponding to Figure 1(c) is shown in Figure 2. An "answer" edge connects a support with the answer it supports (shown with white arrowheads in the figure). An answer/fact contained in a support is connected to the support with a "uses-of" edge (filled arrowheads in the figure).

Our incremental algorithm [19] is based on DRed [11] and has two phases. In the first phase, when a fact is deleted, the supports containing the fact are marked. When a support is marked, the answers it supports are marked; the marks are then further propagated, alternately marking supports and answers. Note that marking is done without using the program rules. The propagation of marks is optimized by (i) annotating the first support used to derive an answer as its *primary* support, and (ii) marking an answer only when its primary support is marked. In Figure 1(c), the primary support is listed first. In our running example, the deletion of `b(6,2)` will mark `r(6,2)`, and consequently `r(3,2)`. The marked supports and answers due to this deletion are shown in boldface in Figure 1(c). Note that `r(1,2)` is not even marked.

More importantly, support graphs significantly simplify rederivation. Observe that after the marks have been completely propagated, if a marked answer has unmarked supports, then it is derivable using any of those supports. Continuing with our running example, we can thus remove the mark on `r(3,2)`. Consequently, the marks on support $\{$`c(6,3),r(3,2)`$\}$, and hence `r(6,2)` are removed. Note that rederivation can be done without using the program clauses.

The Problem: Our incremental algorithm in [19] based on support graphs shows very good time performance: incremental evaluation times for small changes are typically *0.1%* of the from-scratch evaluation time for most programs. However, support graphs are usually very large, and the memory overhead makes it impossible to maintain the full support graph for large applications.

In [20] we proposed an algorithm that kept only a limited number of supports for each answer, making its space requirements linear in the number of answers. The space savings and scalability however come at the price of increased rederivation time. Since all supports are not stored, an answer may have a derivation even when all of its stored supports are marked. Hence, we need to re-evaluate the program clauses to check if the answer can be rederived. The time penalty can be high: incremental evaluation time (for small changes) may be as much as 15% of that of from-scratch evaluation.

This raises an interesting question: *Can we store the entire support graph, which eases rederivation and significantly improves incremental evaluation time, without incurring a prohibitive space overhead*? We address this problem in this paper.

Our Solution: The key to storing the entire support graph is to make use of explicit sharing inherent in the supports. Consider the two answers to call `r(3,X)`, `r(3,2)` and `r(3,4)`, and their supports $s3$ and $s4$ respectively. Observe that the two supports share `c(3,6)`. Also notice that the literals which make the two supports different, i.e. `r(6,2)` and `r(6,4)`, are answers to the call `r(6,X)`. Thus two supports for answers to `r(3,X)` can be represented in intensional form as: `c(3,6), r(6,X)`. This intensional form is represented in a *symbolic support*, which consists of three parts, namely, the set of answers supported (e.g. `r(3,X)`, the common part of all the supports (e.g. `c(3,6)` , and the call whose answers distinguish the supports (e.g. `r(6,X)`). Now, when an answer to `r(6,X)`, say `r(6,2)` is deleted, we can compute, using the symbolic support, that `r(3,2)` may be affected. A symbolic support captures dependencies between certain calls while our earlier notion of supports captured dependencies

Call	Symbolic Supports
r(6,X)	$\langle$r(6,X), {}, b(6,X)$\rangle$, $\langle$r(6,X), {c(6,3)}, r(3,X)$\rangle$
r(3,X)	$\langle$r(3,X), {c(3,6)}, r(6,X)$\rangle$, $\langle$r(3,X), {c(3,1)}, r(1,X)$\rangle$
r(1,X)	$\langle$r(1,X), {}, b(1,X)$\rangle$, $\langle$r(1,X), {c(1,6)}, r(6,X)$\rangle$

Fig. 3. Symbolic supports for query evaluation over the example program in Figure 1(a).

between answers. By lifting this to the level of calls, a symbolic support compactly represents multiple supports.

The symbolic supports for the evaluation of query `r(6,X)` over the program in Figure 1(a) appears in Figure 3. Marking can be readily done using the symbolic supports. Given a marked answer (e.g. `r(6,2)`), we first compute the substitution for the variables in a support corresponding to it (e.g. `r(6,X), X = 2`), and use this substitution to find the supported answer (e.g. `r(3,2)`). When the intensional form does not contain any join operations, we can compute the answer dependencies from the symbolic support in time proportional to the answer size.

Contributions: We propose Symbolic Support Graph (SSG), a data structure for space-efficient and time-efficient incremental evaluation of tabled logic programs (Section 3). We give efficient algorithms for incremental evaluation with SSGs (Section 4). SSGs grow no faster than the tables for an important class of tabled programs and queries (Section 5). In practice, SSGs take much less space than full support graphs, and yet show time performance comparable to the latter (Section 6). We demonstrate the scalability of incremental evaluation using a points-to analyzer for C programs as an example. Our incremental analyzer scales to programs with over 60K lines of code. In many cases, the size of SSG is even smaller than that of partial support graphs used in [20]. Thus, SSGs enable incremental evaluation of large, realistic applications.

The relationship between this paper and the previous work is explored in Section 7. In this paper, we describe SSGs for incremental evaluation when facts are deleted from a definite tabled logic program. The use of SSGs is orthogonal to other issues in incremental evaluation, such as the handling of insertion of facts/rules, deletion of rules, updates, and stratified negation. A brief discussion of these issues appear in Section 8.

2 Preliminaries

We now formally define the notions of supports and support graph. We consider definite logic programs, and partition the predicates into *intensional* and *extensional* predicates. Extensional predicates are defined solely by facts. For simplicity of notation, we assume that only the definitions of extensional predicates may be deleted. The techniques in this paper can be generalized to handle the deletion of rules along the lines described in [19].

Definition 1 (Support) *Let P be a definite logic program, and let T be a set of answer tables obtained when evaluating a query γ over P. A set $\{b_1, b_2, \ldots, b_n\}$ is called a support of an answer a of γ if there exists a clause in P of the form $\alpha :- \beta_1, \beta_2, \ldots, \beta_n$ and a substitution θ, such that $\alpha\theta = a$, and, for all $i \in [1, n]$ $\beta_i\theta = b_i$ and b_i is an instance of an answer in T or a fact in P.*

A support graph maintains the relationships between the answers and supports generated during query evaluation.

Definition 2 (Support graph) *Let P be a definite logic program, and let T be a set of answer tables obtained when evaluating a query γ over P.*

The support graph for the evaluation of γ is a directed graph (V, E) where V contains the facts in P, answers in T and their supports. The set of edges E is such that

- *$(b_i, s) \in E$ for all supports $s \in V$ such that $s = \{b_1, b_2, \ldots b_n\}$, and for all $i \in [1, n]$. We say that $s \in b_i$.uses_of and $b_i \in s$.part_of.*
- *$(s, a) \in E$ for all $a \in T$ and $s \in V$ such that s is a support of a. We say that s.answer $= a$ and $s \in a$.support.*

The primary support of an answer is the first support used to derive the answer in some least fixed point computation procedure. It follows from the property of least fixed point computations that the primary support will be independent of the answer itself. We generalized this to *acyclic supports* in [20], defined using the notion of *derivation length* described below:

$$v.dl = \begin{cases} 0 & \text{if } v \text{ is a fact} \\ 1 + \max\{a.dl \mid v \in a.uses\_of\} & \text{if } v \text{ is a support} \\ s.dl \mid s \text{ is the primary support of } v & \text{if } v \text{ is an answer} \end{cases}$$

The derivation length represents the height of a proof tree for an answer. Note that if the derivation length of a support s is no greater than its supported answer a, then s has a derivation independent of a. A support s is *acyclic* if $s.dl \leq s.answer.dl$. Thus an answer need not be marked in the first phase until all of its acyclic supports are marked. In our running example in Figure 2, we have annotated the vertices with their derivation lengths. Based on the derivation lengths, we determine $s1$, $s2$, $s3$, $s4$, $s5$, $s7$ and $s8$ as acyclic supports. Thus deletion of $f2$ causes only $a1$ to be marked. Note, however, an acyclic support may not remain acyclic after rederivation. For instance, $a1$ is rederived due to $s10$ which now becomes its acyclic support by updating $a1.dl$ to 3. Consequently, the derivation lengths $s3$ and $s6$ are changed to 4 making $s3$ non-acyclic.

3 Symbolic Support Graphs

We now formally define the notion symbolic supports, and describe the data structure to represent symbolic support graphs.

Definition 3 (Symbolic Support) *Let P be a definite logic program with a set of facts F, and let C and A be a set of call and answer tables respectively, obtained when evaluating a query ξ over P. The triple $S = \langle h, s, d \rangle$ is a symbolic support for a call $\gamma \in C$ if there is a clause in P of the form $\alpha :- \beta_1, \beta_2, \ldots, \beta_{n-1}, \beta_n$ and a substitution θ such that*

1. *h, called the* head *of S, is such that $\alpha\theta = h$;*
2. *s, called the* static part *of S, is such that $s = \{b_1, b_2, \ldots b_{n-1}\}$, and $\forall i \in [1, n-1]$, $b_i = \beta_i\theta$ and $b_i \in A \cup F$;*
3. *d, called the* dynamic part *of S, is such that $\beta_n\theta = d$.*

Note that a symbolic support is shared between a non-empty set of answers of a call. The set of non-symbolic supports represented by a symbolic support S are called embedded supports of S, defined below.

Definition 4 (Embedded Supports) *Let P be a definite logic program with set of facts F, and let A be answers in the tables obtained when evaluating a query γ over P. A non-symbolic support s is embedded in a symbolic support $S = \langle h, s', d\rangle$ if there is a substitution σ such that $s.answer = h\sigma \in A$, $s = s' \cup \{d\sigma\}$, and $d\sigma \in A \cup F$.*

Given a symbolic support $S = \langle h, s, d\rangle$, then answer a' is said to be a *supported answer* for an answer/fact a w.r.t. S if there is a substitution σ such that $a = d\sigma$ and $a' = h\sigma$. In that case, we also say that a is a *supporting answer* w.r.t. S.

When the mark on an answer is propagated, we need to find an embedded support that contains this answer. For instance, consider our running example and its symbolic supports in Figures 1(a) and 3 respectively. If `r(6,2)` is marked, since it is an instance of `r(6,X)`, we need to mark supports embedded in $\langle$`r(1,X)`$, \{$`c(1,6)`$\},$ `r(6,X)`$\rangle$ and $\langle$`r(3,X)`$, \{$`c(3,6)`$\},$ `r(6,X)`$\rangle$. This lookup can be efficiently done if the dynamic part of a symbolic support is a tabled call. Moreover, we can maintain, for each tabled call, the set of symbolic supports that contain it. If the dynamic part of the symbolic support is not a tabled call, then we need to maintain additional indexing structures to find the embedded supports. *Hence we do not use a symbolic support when its dynamic part is not a tabled call, and use non-symbolic supports instead.*

Symbolic support graphs (SSG) are an extension of the support graphs that has calls, answers, symbolic as well as non-symbolic supports as vertices and the relationships between them as edges. The edges in an SSG are described below.

- *uses_of*, *part_of*, *support*, *answer*: as in Definition 2.
- *set_uses_of*: If a fact or an answer a is in the static part of a symbolic support *SS* then there is a *set_uses_of* edge from a to *SS*.
- *set_uses_of_call* and *dynamic_call*: If a call C is the dynamic part of a symbolic support *SS* then there is a *set_uses_of_call* edge from C to *SS*. There is also a *dynamic_call* edge from *SS* to C.
- *supported_call* and *symsupport*: If a symbolic support *SS* supports a nonempty set of answers of a call C then there is a *supported_call* edge from *SS* to C, and a *symsupport* edge from C to *SS*.
- *answers* and *subgoal*: If A is the set of answers for call C, then there is a *answers* edge from C to elements of A and a *subgoal* edge from each element of A to C.

The SSG corresponding to the supports in Figure 2 is shown in Figure 4. Note that any tabling engine will give unique identities to each tabled call (e.g. the *subgoal frame* in the SLG-WAM [7]) and tabled answers. We use these identifiers in our implementation of the SSG to denote calls and answers (we use terms to represent calls in examples, for clarity). The information about the variables in the head and the dynamic part, needed to compute the embedded supports, is also kept in a symbolic support. This implementation detail is not shown in the examples.

The *set_uses_of*, *set_uses_of_call*, and *supported_call* edges in an SSG are required for propagation of marks and rederivation. They are analogues of *uses_of* and *answer* in a support graph. The *symsupport* edges are used to adjust the derivation length of an answer after rederivation. Finally, *dynamic_call* is used to compute the embedded supports of a symbolic support.

In addition, we maintain the following attributes with the answers in the support graph. For each answer we maintain the total number of unmarked supports in *total_support_count* and the number of unmarked acyclic supports in *acyclic_support_count*. These attributes

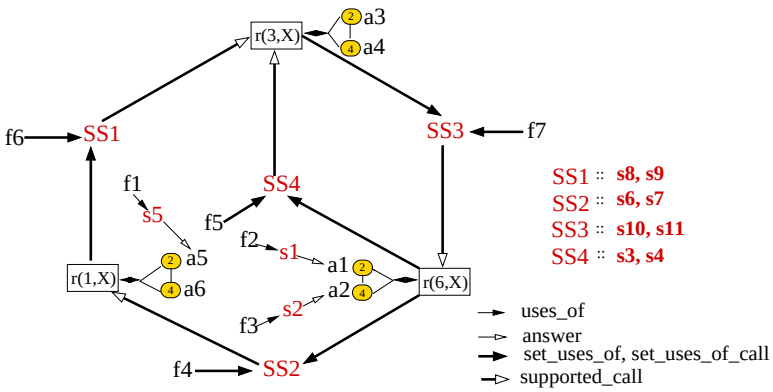

Fig. 4. Symbolic support graph for answers to query `r(6,X)`.

count the number of embedded supports represented by a symbolic support. In Figure 4, *total_support_count* and *acyclic_support_count* of $a1$ are 2 and 1 respectively.

4 The Incremental Table Maintenance Algorithm

We now describe the incremental algorithm for maintaining the tables using symbolic supports. The algorithm extends the one in [19] and handles graphs with a mixture of symbolic and non-symbolic supports. We have already seen in the previous section how to compute the embedded (non-symbolic) supports for each symbolic support. Note that information such as derivation length and marking are specific to the non-symbolic embedded supports; computing this information based on symbolic supports is the key issue in the algorithm. Note also that the static part of a symbolic support is common to all its embedded supports. Hence we associate the information due to the static part in the symbolic support. For each symbolic support node we maintain an attribute *static_maxdl* that stores the maximum of derivation lengths of the answers and facts in its static part. We use this information to compute the derivation length of each embedded support. Similarly, with each non-symbolic support we maintain an attribute *falsecount* which counts the number of marked answers/facts in the support. With each symbolic support, we maintain *static_falsecount* which counts the number of marked answers and facts in its static part.

The algorithm has two phases analogous to the two phases of DRed and other incremental recursive-view maintenance algorithms.

Marking Phase. The algorithm for the marking phase is shown in the Figure 5. The *falsecount* attributes of symbolic and non-symbolic supports are initialized to zero before the marking phase. An answer is marked by setting its *marked* flag to true; this attribute is initialized to false. The answers to be marked are placed in a queue, and the marking phase ends when the queue is empty. The marked answers are placed in a set *marked_set* for processing in the rederivation phase.

The functions *mark_answer* and *mark_fact* propagates the effect of marking an answer/fact to the supports containing it. The function *mark_fact* propagates the effect of deleting a fact to the supports containing it. In addition *mark_answer* places a mark on

```
mark()
  mark_queue = empty
  ∀ deleted facts f
    mark_fact(f)
  while (mark_queue != empty)
    a = dequeue(mark_queue)
    mark_answer(a)

mark_fact(f)
  ∀ Support s ∈ f.uses_of
    mark_support(s)
  ∀ SymbolicSupport S ∈ f.set_uses_of
    mark_static(S)

mark_answer(a)
  a.marked = true
  ∀ s ∈ a.uses_of
    mark_support(s)
  ∀ S ∈ a.set_uses_of
    mark_static(S)
  subg = a.subgoal
  ∀ S ∈ subg.set_uses_of_call
    mark_dynamic(S, a)

mark_support(s)
  s.falsecount++
  if (s.falsecount == 1)
    tans = s.answer
    propagate_mark(tans, s.dl)
```

```
mark_dynamic(S, sans)
  (* Propagate via dynamic part *)
  if (S.static_falsecount == 0)
    tans = supported answer of sans w.r.t. S
    support_dl =
       1 + max(S.static_maxdl, sans.dl)
    propagate_mark(tans, support_dl)

mark_static(S)
  (* Propagate via static part *)
  S.static_falsecount++
  if (S.static_falsecount == 1)
    ∀ sans ∈ answers(S.dynamic_call)
      if (! sans.marked)
        tans = supported answer of sans w.r.t. S
        support_dl =
           1 + max(S.static_maxdl, sans.dl)
        propagate_mark(tans, support_dl)

propagate_mark(tans, support_dl)
  tans.total_support_count−−
  if (tans.dl ≥ support_dl)
    tans.acyclic_support_count−−
    if (tans.acyclic_support_count == 0)
      enqueue(mark_queue, tans)
      marked_set = marked_set ∪ { tans }
```

Fig. 5. Algorithm for Marking Phase

the answer. Function *mark_support* marks a support and propagates this mark to the answer supported by it; functions *mark_static* and *mark_dynamic* mark a symbolic support and if needed propagate this mark to the answer(s) supported by it. Note that a (symbolic) support is marked if its (*static_*) *falsecount* is nonzero.

We illustrate the working of the marking phase using the deletion of $f2$ and $f4$ from Figure 4 as an example. A call to mark_fact($f2$) will call mark_support($s1$), and subsequently propagate_mark($a1$, 1). This will decrement $a1$'s total and acyclic support counts (to 1 and 0, resp.), and place $a1$ in the queue. We will call mark_fact($f4$) next. Since $f4$ is in the static part of symbolic support $SS2$, we call mark_static($SS2$). This sets *static_falsecount* of $SS2$ to 1, iterates over the answers of the dynamic part of $SS2$, i.e. `r(6,X)`. The supported answers of $a1$ and $a2$ w.r.t $SS2$ are $a5$ and $a6$, resp., and $a6$ is added to the queue as $a5$ has an unmarked acyclic support $s5$. Note this is equivalent to propagation of marking through $s6$ and $s7$ in support graph based algorithm. Continuing further, we pick up $a1$ for processing from the queue. Since $a1$ appears in the dynamic parts of $SS2$ and $SS4$ *mark_dynamic* is called for both the symbolic supports. However, *mark_dynamic*($SS2$,$a1$) has no effect as its *static_falsecount* is already 1; *mark_dynamic*($SS4$,$a1$) will call *propagate_mark*($a3$, 2) which reduces the total and

```
rederive()
 ∀ ans ∈ marked_set
   if (ans.total_support_count > 0)
     ans.acyclic_support_count
       = ans.total_support_count
     recalculate_dl(ans)
     enqueue(rq, ans)
 ∀ Answer ans ∈ rq
   rederive_ans(ans)

recalculate_dl(tans)
 spt_max=max{s.dl | s = tans.support ∧
              s.falsecount == 0}
 espt_max=max{max(S.static_maxdl,ans.dl)+1
   | S ∈ tans.subgoal.symsupport
   ∧ S.static_falsecount=0
   ∧ ans is a supporting answer of tans w.r.t S
   ∧ !ans.marked
 tans.dl=max(spt_max,espt_max)

rederive_answer(ans)
 ans.marked = false
 ∀ s ∈ ans.uses_of
   rederive_support(s, ans.dl)
 ∀ S ∈ ans.set_uses_of
   rederive_static(S, ans.dl)
 subg=get_subgoal(ans)
 ∀ S ∈ subg.set_uses_of_call
   rederive_dynamic(S, ans)
```

```
rederive_support(s, dlen)
 s.dl = max(s.dl, dlen+1)
 s.falsecount−−
 if (s.falsecount == 0)
   propagate_rederive(s.answer_of, s.dl)

rederive_dynamic(S, sans)
 if (S.static_falsecount == 0)
   tans = supported answer of sans w.r.t. S
   dlen = max(S.static_maxdl, sans.dl)+1
   propagate_rederive(tans, dlen)

rederive_static(S, dlen)
 S.static_maxdl = max(S.static_maxdl, dlen)
 S.static_falsecount−−
 if (static_falsecount(S) == 0)
   ∀ sans ∈ S.supported_call.answers
     if (!sans.marked)
       tans = supported answer of sans w.r.t. S
       dlen' = max(S.static_maxdl, sans.dl)+1
       propagate_rederive(tans, dlen')

propagate_rederive(ans, dlen)
 ans.total_support_count++
 if (ans.acyclic_support_count==0)
   ans.acyclic_support_count = 1
   ans.dl = dlen
   enqueue(rq, ans)
 else
   if (ans.dl ≥ dlen)
     ans.acyclic_support_count++
```

Fig. 6. Algorithm for Rederivation

acyclic support counts of $a3$ to 1 (due to acyclic embedded support $s8$ in $SS1$). Similarly, processing $a6$ from the queue does not mark $a4$ as it has an acyclic embedded support in $SS4$. Thus at the end of marking phase $a1$ and $a6$ are marked.

Rederivation Phase. Each marked answer that has some unmarked support at the end of the marking phase is known to have a proof not involving its previously known acyclic supports. In addition to resetting its mark, we need to compute its new derivation length (due to the new proofs). In our running example we compute the new derivation length of $a1$ by computing derivation length of its unmarked support ($s10$ in SG) embedded in $SS3$. This is done by finding the supporting answer for $a1$ w.r.t. $SS3$, i.e. answer $a3$ ($dl = 3$), and computing the *dl* of the embedded support. When some of the marked answers are rederived, we propagate rederivation using the function *rederive_answer*. Figure 6 gives the rederivation algorithm, which is very similar to the marking algorithm.

5 Space Complexity of Symbolic Support Graphs

In this section we compare the asymptotic size of SSGs with respect to table size and the size of non-symbolic support graphs for a number of useful tabled programs. For purposes of this comparison, we assume that all supports in the SSG are symbolic. The selected programs and the complexity measures are shown in Figure 7. The apparently simple transitive closure programs (`lreach/2` and `rreach/2`) lie at the heart of a remarkable number of applications of tabled logic programming. For instance, verification of safety properties of systems and implementation of inheritance in object-oriented logics reduce to reachability problem. Context-free language reachability, which is the basis for the verification of push-down systems, has rules that resemble the definition of the simpler same-generation (`sg/2`) predicate. A class of useful tabled logic programs not in the figure are those involving negation and aggregation (e.g. dynamic programming problems). In principle, symbolic supports can be used in these cases also, but other aspects of our implementation (e.g. handling of insertions/updates) need extension (see Section 8). Hence we do not include this class in the comparison.

For the graph traversal examples, we assume that the `edge/2` relation defines a graph with v vertices and e edges. We first consider left-recursive transitive closure (`lreach/2` in Figure 7), with a bound-free query, say `lreach(`a`, X)`. Tabled evaluation of this query will result in only one tabled call, the query itself, and all vertices reachable from a will be answers to this call. Thus, the table size for this query is $O(v)$. Answer `lreach(`a, b`)` has supports of the form $\{$`lreach(`a, Y`), edge(`Y, b`)`$\}$. The number of supports of this form are bounded by the in-degree of b. Hence the total number of supports is $O(e)$. The symbolic supports associated with call `lreach(`a`,X)` are $\{$`edge(`a`,X)`$\}$ and those of the form $\{$`lreach(`a, Y`), edge(`Y`,X)`$\}$. Thus the number of symbolic supports is $O(v)$, i.e. linear in the number of answers.

Now consider a bound-free query to right-recursive transitive closure, say `rreach(`a`,X)`. Tabled evaluation makes $O(v)$ distinct tabled calls to answer this query. Each of these call tables can have $O(v)$ answers, and hence the table size is $O(v^2)$. Each answer `rreach(`b, c`)` has supports of the form $\{$`edge(`b, Y`), rreach(`Y, b`)`$\}$ where Y ranges over neighbors of b. The number of supports for this answer is bounded by the out-degree of b. Since there are $O(v^2)$ answers, the total number of supports is $O(v*e)$. The symbolic supports associated with call `rreach(`a`,X)` are $\{$`edge(`a`,X)`$\}$ and those of the form $\{$`edge(`a, Y`), rreach(`Y`,X)`$\}$. Thus there are two symbolic supports for each edge and hence the number of symbolic supports is $O(e)$. Note that SSG grows slower than the tables for this example.

The asymptotic space complexity for the other examples and queries in Figure 7 are computed along the same lines. The figure shows two versions of the same generation predicate: the naive *sg/2*, and an optimized version *sg_opt/2* obtained by supplementary tabling (i.e. tabling an intermediate join). The latter has better time complexity; observe from the figure that the size of SSG for this program is proportional to table size. For such programs, the space needed for SSG is less than three times the table space in the worst case. In practice the constant factor is close to 1.5 (see next section).

6 Experimental Results

The aim of symbolic support graphs is to make incremental evaluation scale to large applications. To determine the effectiveness of our new data structure and algorithm,

Example programs	Query Modes	Space Complexity		
		Table	SG	SSG
lreach(X,Y):- edge(X,Y).	bb, bf	$O(v)$	$O(e)$	$O(v)$
lreach(X,Y):- lreach(X,Z), edge(Z,Y).	fb, ff	$O(v^2)$	$O(v*e)$	$O(v^2)$
rreach(X,Y):- edge(X,Y).	bf, ff	$O(v^2)$	$O(v*e)$	$O(e)$
rreach(X,Y):- edge(X,Z), rreach(Z,Y).	bb, fb	$O(v)$	$O(e)$	$O(e)$
sg(X,X). *sg(X,Y):- edge(X,Y1),sg(Y1,Y2),edge(Y2,Y).*	all	$O(v^2)$	$O(e^2)$	$O(v*e)$
sg_opt(X,X). *sg_opt(X,Y) :- aux(X,Z),edge(Y,Z).* *aux(X,Y):- edge(X,Z),sg_opt(Z,Y).*	all	$O(v^2)$	$O(v*e)$	$O(v^2)$
Context-Free Language Reachability (see Appendix): N=|nonterms|,G=grammar size	all	$O(N*v^2)$	$O(G*v^3)$	$O(G*v^2)$

Fig. 7. Space complexity of symbolic support graphs

Programs	LOC	Rep. Factor	From Scratch All Points-to			
			Avg. Size	Time(s)	Space(MB)	
					Table	Total
smail	3850	15	24.5	1.45	13	22
parser	11391	15	5.8	1.20	17	32
vpr	17729	15	1.8	0.37	15	33
m88ksim	19093	15	6.0	0.25	10	22
twmc	24951	1	16.7	0.87	9	14
nethack	33993	1	35.0	2.55	6	21
vortex	67110	1	69.8	12.90	31	57

Table 1. Benchmark Characteristics

we measured their performance on a points-to analyzer for C programs. The analyzer itself is a tabled logic program which encodes Anderson's points-to analysis [2, 20]. We measured the performance of the analyzer on programs taken from C benchmarks available with PUF compiler suite and SPEC95 benchmarks. The symbolic support graph based incremental evaluation algorithm was implemented by extending the XSB logic programming system [27] (v2.6). Our incremental points-to analysis system, the benchmarks, and detailed experimental results are available at [22].

We preprocessed the C source code using CIL [14] 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. Performance measurements were taken on a PC with 1.4Ghz Pentium M processor with 512MB of physical memory running Linux (Debian) 2.6.7.

We performed All Points-to Analysis (APA), which computes the points-to relation for all program variables. The characteristics of the benchmarks as well as the results of performing the analysis *without any support for incremental evaluation* are given in Table 1. In the table, "LOC" refers to the number of lines of source code in the benchmark; "Avg. size" shows the average number of of the points-to tuples per variable. The first four benchmarks in the table are relatively small; to remove noise from the results, we replicated the programs, generating new variable names as appropriate. The remaining three benchmarks are large enough to permit stable measurements without replication.

	Support Graph		Partial Support Gr.		Symbolic Support Gr.			mem%
Benchmark	supports	memory	supports	memory	support	symspt	memory	sym/com
smail	3,159.4K	92.2	560K	22.8	42.2K	163.0K	15.5	16.8
parser	1,355.7K	44.2	518K	21.8	130.0K	159.2K	17.9	40.5
vpr	213.1K	9.7	172K	8.7	56.2K	51.9K	8.5	86.9
m88ksim	303.5K	11.8	206K	9.2	34.3K	47.9K	7.1	60.5
twmc	5,727.8K	158.5	396K	16.2	90.5K	105.0K	12.6	8.0
nethack	2,074.8K	59.4	269K	11.2	34.9K	60.4K	8.1	13.6
vortex	33,334.5K	912.0	1,714K	65.2	215.3K	361.4K	46.1	5.1

Table 2. Comparison of support graph sizes for pointer analysis

Benchmark	Table	SG	SSG
smail	0.8	0.9	0.8
allroots	0.5	0.5	0.4
assembler	47.7	67.6	48.6
compiler	51.1	154.0	53.7
compress	7.0	9.2	7.0
loader	5.9	6.9	6.0

(a)

Programs	Graphs								
	chain			complete			tree		
	2000 nodes			50 nodes			10000 nodes		
	Table	SG	SSG	Table	SG	SSG	Table	SG	SSG
lreach	0.3	0.2	0.2	0.1	0.2	0.1	1.6	0.8	0.9
rreach	47.0	96.0	40.2	0.1	3.6	0.3	5.4	5.7	3.2
sg_opt	1.1	0.4	0.4	0.3	7.2	0.5	5.7	1.8	1.9

(b)

Table 3. Support graph sizes (in MB) for: push-down model checking (a); and synthetic benchmarks from Table 7

Space. Table 2 shows the number of supports, and space (in MB) taken by, support graphs [19], partial support graphs with maximum of 2 supports per answer [20], and symbolic support graphs for each benchmark. Observe from the table that the symbolic support graph takes the least space among the three. Note that the symbolic support graph may contain non-symbolic supports; while it is possible to make all supports symbolic, we find that it usually increases space requirements by 20%. Finally, the table shows that the symbolic support graph can be considerably smaller than the (non-symbolic) support graph of [19]. Since symbolic supports keep dependencies between calls instead of answers, the reduction in space is proportional to the number of answers per call (the average points-to size).

Table 3(a) shows the sizes of non-symbolic (SG) and symbolic (SSG) support graphs for performing automata-based dead variable analysis of C programs using the push

benchmark	from scratch (a)	Incr- Support Graph complete (b)	partial (c)	symbolic (d)	% b/a	% c/a	% d/a
smail	1.45	0.0178	0.1073	0.0433	1.22	7.4	2.98
parser	1.19	0.0025	0.0916	0.0108	0.21	7.7	0.90
vpr	0.37	0.0001	0.0048	0.0005	0.03	1.3	0.14
m88ksim	0.25	0.0005	0.0028	0.0015	0.20	1.1	0.60
twmc	2.49	0.0039	0.2092	0.0125	0.16	8.4	0.50
nethack	0.87	0.0005	0.0487	0.0028	0.06	5.6	0.32
vortex	12.80	0.0040	1.9200	0.0200	0.03	15.0	0.16

Table 4. Comparison of running times

down model checker of [4]. The model checker has few answers per call, consequently we see a reduction in space due to SSGs, but not as much as in the points-to analysis.

Recall from Section 5 the size of the symbolic support graph grows at or near the same rate as the table size for bound-free queries to left-recursive and right-recursive transitive closure and same generation programs (from Table 7). Table 3(b) shows that not only the growth rates, but the total space requirements of symbolic supports are also close to those of the tables themselves.

Time. The effectiveness of the incremental techniques were evaluated by removing one (source-level) statement from the benchmark programs, and measuring the time and space taken to redo the analysis from scratch and to maintain the points-to relation incrementally. Deleting one source level assignment statement may delete multiple primitive assignment statements and hence multiple facts.

The results are shown in Table 4. The incremental analysis timings were obtained by repeating the incremental evaluation a number of times to obtain measurable running times. The table shows the incremental evaluation times using the non-symbolic support graph [19] (complete), the partial support graph with at most 2 supports per answer [20] (partial), and the symbolic support graph (symbolic). Observe that the SSG-based algorithm is on average 5 times slower than the complete support graph based one, but is still two orders of magnitude faster than the from-scratch analysis for small changes.

7 Related Work

The idea of recording the evaluation process as a graph to guide incremental change propagation has been used in various fi elds viz. AI, view maintenance, program analysis, model checking, functional programming, and logic programming.

Structures similar to support graph was seen in truth maintenance system [9] (TMS), and later in belief revision systems [3]. With each belief node in TMS a justifi cation set, which represents the reasons for the belief, is kept. This is analogous to the support graph with beliefs as answers and justifi cations as their supports.

Among the works on materialized view maintenance, we reviewed the relationships between our work and the DRed algorithm [11] in the introduction. The Straight Delete (StDel) [13] algorithm keeps the entire proof associated with every answer, thereby eliminating the rederivation phase. While this approach may be feasible in constraint databases, its space complexity is worse than the support graph-based algorithm.

The product graph generated during the model checking process is used by incremental model checking algorithm (MCI) of [23]. The complete graph is kept explicitly and the space issues are not addressed. As mentioned in [19] we have adopted MCI's use of counts to effi ciently compute truth values of nodes during incremental evaluation.

Incremental attribute grammar evaluation [17] generates a dependency graph to record the functional dependencies among attributes in the grammar. The dependency graph is acyclic since only non-circular attribute grammars are considered. Another instance of an acyclic dependency graph is the *augmented dependency graph* [1] which records dependencies between input and output values in the execution of pure functional programs. These graphs also record the ordering among their edges which is used for effi cient change propagation.

None of the above works address the scalability issues related to storing the dependencies between computations. The SSG proposed in this paper grows at or near the rate at which tables grow for many tabled logic programs and queries; in the worst

case, however, SSG's size is not bounded by table size. In [20] we proposed an approach to keep a bounded number of supports with each answer, thereby making the support graph size proportional to the table space for arbitrary tabled logic programs.

In [25, 24] Binary Decision Diagrams (BDDs) [6] are used to represent transition relation and reachable states of a state transition graph in the context of logic synthesis and formal verification of digital circuits. To incrementally maintain reachable states in response to changes in the transition relation, a spanning graph is generated during reachability analysis as the evidence for all the reachable states. This can be considered as an acyclic support graph for the reachable states. BDDs are used to represent the edge relation of the spanning graph, thereby making it space efficient. We attempted to keep the support graph using BDDs. However, an inordinate amount of time was taken to build the support graph BDDs: there appears to be no way to use the efficient set-at-a-time operations of the BDDs to construct the support graph when the query evaluation is done tuple at a time.

8 Conclusion

We presented a space-efficient data structure and incremental algorithms for maintaining tables in the presence of deletion of facts. The techniques can be readily extended to handle deletion of rules by keeping rule information in supports as done in [19]. The symbolic support graph maintains dependencies between certain calls, but in such a way that the dependencies between answers can be readily computed whenever needed. The ease of computation is ensured by keeping the symbolic supports in a "join-free" form, keeping only the last literal of a support as a call. This can be easily extended to keeping as calls the right-most literal in a clause that is followed by simple computations (such as comparison operations). This extension of the notion of symbolic supports will permit us to represent programs with aggregation operations using symbolic supports, thereby enabling incremental evaluation of dynamic programming problems.

In this paper, we considered only definite logic programs, where all predicates are either tabled or defined by facts. We can extend this to programs containing a mixture of tabled and non-tabled predicates along the same lines as in [19]: accumulating support information from non-tabled predicates and storing them with answers. In [21] we give a support-graph-based bottom-up algorithm that interleaves insertion and deletion that permits efficient handling of incremental updates, and programs with stratified negation. We can make the algorithm in [21] space-efficient by using symbolic support graph as its support data structure.

References

1. U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *POPL*, volume 37, pages 247–259, 2002.
2. L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Unversity of Copenhagen, 1994.
3. K. Apt and J. M. Pugin. Maintenance of stratified databases viewed as a belief revision system. In *Principles of Database Systems*, pages 136–145. ACM Press, 1987.
4. S. Basu, K. N. Kumar, L. R. Pokorny, and C. R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *TACAS*, volume 2280 of *LNCS*, pages 236–250, 2002.
5. R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *ILPS*, 1993.

6. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
7. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1):20–74, 1996.
8. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems —a case study. In *PLDI*, pages 117–126. ACM Press, 1996.
9. J. Doyle. A truth maintenance system. *Artifi cial Intelligence*, 12:231–272, 1979.
10. 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.
11. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.
12. A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and appfications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.
13. J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD*, pages 340–351, 1995.
14. 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.
15. L. L. Pollock and M. L. Soffa. An incremental version of iterative data fbw analysis. *IEEE Trans. Softw. Eng.*, 15(12):1537–1549, 1989.
16. C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *CAV*, 2000.
17. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *TOPLAS*, 5(3):449–477, 1983.
18. R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *TAPD*, pages 77–87, Vigo, Spain, September 2000.
19. D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *ICLP*, volume 2916 of *LNCS*, pages 389–406, 2003.
20. D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming (PPDP)*, 2005. To Appear. Available at `http://www.lmc.cs.sunysb.edu/~dsaha/`.
21. D. Saha and C. R. Ramakrishnan. A local algorithm for efficient incremental evaluation of tabled logic programs, 2005. Available at `http://www.lmc.cs.sunysb.edu/~dsaha/local`.
22. D. Saha and C. R. Ramakrishnan. Symbolic support graph: A space efficient data structure for incremental tabled evaluation, 2005. Downloads are available at `http://www.lmc.cs.sunysb.edu/~dsaha/symspt`.
23. O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, volume 818 of *LNCS*, pages 351–363, 1994.
24. G. Swamy. *Incremental Methods for Formal Verifi cation and Logic Synthesis*. PhD thesis, University of California at Berkeley, 1996.
25. G. Swamy, R. K. Brayton, and V. Singhal. Incremental methods for FSM traversal. In *Intl. Conference on Computer Design (ICCD)*, page 590. IEEE Computer Society, 1995.
26. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, pages 84–98, 1986.
27. XSB. The XSB logic programming system. Available from `http://xsb.sourceforge.net`.
28. J. Yur, B. G. Ryder, and W. Landi. An incremental fbw- and context-sensitive pointer aliasing analysis. In *ICSE*, pages 442–451, 1999.
29. 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.

9 Appendix

We provide the tabled logic program for context free language (CFL) reachability problem. The CFL reachability problem determines whether the language generated by a finite state automaton has a non-empty intersection with a given context free language.

The CFL is specified using a context free grammar in Chomsky Normal Form (CNF), as a set of `grammarrule/2` facts. A fact of the form `grammarrule(`A`,` B`)` specifies a grammar rule with A as the left hand side non-terminal symbol, and B as a list of grammar symbols on the right hand side. Note that, since the grammar is in CNF, the list will be in one of three forms: `[]`, signifying an epsilon rule; `[`T`]` where T is a single terminal symbol; or `[`S_1`,`S_2`]` where S_1 and S_2 are two non-terminal symbols.

The automaton is given in form of `trans/3` facts where `trans(`s`,`l`,`d`)` represents a transition from source state s to destination state d labeled with l.

The CFL reachability problem is encoded by the predicate `cfreach/3` defined below. The two given languages has a non-empty intersection iff the query `cfreach(`s_g`,` s_a`,` `F)`, where s_g is the start symbol of the grammar and s_a is the start state of the automaton, succeeds with `F` bound to one of the final states of the automaton.

```
:- table cfreach/3.
cfreach(S,A1,A2):-
        grammarrule(S,[]),
        A1=A2.

cfreach(S,A1,A2):-
        grammarrule(S,[X]),
        trans(A1,X,A2).

cfreach(S,A1,A2):-
        grammarrule(S,[S1,S2]),
        cfreach(S1,A1,A3),
        cfreach(S2,A3,A2).
```