A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs*

Ernie Johnson, C. R. Ramakrishnan, I. V. Ramakrishnan, and Prasad Rao**

Department of Computer Science, State University of New York at Stony Brook
Stony Brook, New York 11794 - 4400
{ejohnson,cram,ram}@cs.sunysb.edu

Abstract. Tabled resolution improves efficiency as well as termination properties of logic programs by sharing answer computations across "similar" subgoals. Similarity based on subsumption of subgoals rather than variance (i.e., identity modulo variable renaming) promotes more aggressive sharing, but requires mechanisms to *index* answers from dynamically growing sets. Earlier we proposed Dynamic Threaded Sequential Automata (DTSA) as the data structure for organizing answer tables in subsumption-based tabling. Using a DTSA, we can retrieve answers one at a time from the table, strictly in the order of their insertion. Although DTSA performed very well, its space usage was high. Here we present an alternative data structure called *Time-Stamped Trie* (TST) that relaxes the retrieval order, and yet ensures that all answers will be eventually retrieved. We show that TST has superior space performance to DTSA in theory as well as practice, without sacrificing time performance.

1 Introduction

Tabled resolution methods in logic programming (LP), beginning with OLDT resolution pioneered by Tamaki and Sato [10], address the well-known shortcomings of the SLD evaluation mechanism of Prolog, namely, susceptibility to infinite looping, redundant subcomputations, and inadequate semantics for negation. Using tabled resolution we can finitely compute the minimal model for datalog programs. More recent methods [1, 2] compute well-founded semantics [11] for normal logic programs. Due to this added power, tabled evaluation enables us to combine LP, deductive databases, and nonmonotonic reasoning, and develop complex applications requiring efficient fixed point computations (e.g., see [7, 6]).

The power of tabled resolution stems from one simple notion: avoid redundant computation by permitting the use of proven instances, or answers, from past computation for satisfying new subgoals. This is achieved by maintaining a *table* of the called subgoals paired with the set of answers derived for each such subgoal

* This work was supported in part by NSF grants C-CR 9711386, C-CR 9876242, and EIA 9705998.

** Currently at Telcordia Technologies (prasadr@research.telcordia.com); work done while at Stony Brook.

(known as answer tables). When a subgoal is selected for resolution, it is first checked against the entries maintained in the call table. If there exists a "similar-enough subgoal", then its associated answers are used for resolving this subgoal (*answer resolution*). Otherwise the subgoal is entered in the call table, and its answers, computed by resolving the subgoal against program clauses (*program resolution*) are entered in the corresponding answer table.

There are two approaches to locate a "similar-enough" goal. One approach, used by the XSB system [8], is to look for an entry that is a variant of the current goal, i.e., identical modulo variable renaming. Although variant-based tabling has been highly successful, this approach permits only limited sharing of answer computations.

The second alternative, called *subsumption-based* tabling, permits greater reuse of computed results. Notice that, given a goal G, any entry in the table, say G', which *subsumes* G will contain in its final answer set all the instances to satisfy G.[1] Using the answers of G' to resolve G avoids computationally-expensive program resolution, and thereby can lead to superior time performance. Space performance may also improve as fewer calls and their associated answer sets need be stored in the tables. However, the mechanisms for efficiently representing and accessing the call and answer tables are more complex. In particular, answer resolution now involves indexing, since not all answers in a selected answer table (say, that of G') may be applicable to the given goal (say, G). This process is especially challenging since all answers to G' may not yet be present in the table when the call to G is made.

In an earlier work [9], we proposed a data structure called Dynamic Threaded Sequential Automaton (DTSA) for representing and retrieving terms from dynamic answer sets. Answer resolution is performed one tuple at a time by backtracking through a DTSA. To ensure that every relevant answer is visited, answers are retrieved strictly in the order of their insertion into the table. Although an implementation based on this strategy shows improvement in time performance on queries where the subsumption of calls is possible, it performs poorly in space when compared to the variant-based tabling engine: potentially quadratic in the size of corresponding tables constructed in a variant-based tabling engine. Moreover, the incremental traversal of the DTSA for resolving each answer forces us to maintain complex state information, thereby increasing choice point space.

In this paper we describe an alternative approach for answer resolution. We tag each answer with a time stamp and store them in *Time-Stamped Tries* (*TST*) which provides indexing based on the symbols in terms as well as the time stamp. Answers relevant to a call are periodically collected from the subsuming call's answer set and cached locally (by the subsumed call) for later consumption. The local cache is updated whenever all answers currently held in the cache have already been resolved against the goal. Each collection phase completely searches the set for answers which have been entered since the previous phase, selecting only those answers which unify with the subsumed goal. Since a complete search

[1] A term t_1 *subsumes* a term t_2 if t_2 is an instance of t_1. Further, t_1 *properly subsumes* t_2 if t_1 subsumes t_2 and t_2 is not a variant of t_1.

is done in each phase, only minimal state information— the time stamp of the last update— is needed between collection phases. The tables stored as TSTs are at most twice as large as the tables in the variant engine, and at most as large as the tables in our earlier DTSA-based engine. Moreover, the space efficiency comes with little or no time penalty.

The rest of the paper is organized as follows. In Section 2, we present an operational overview of tabling operations and answer resolution in subsumption-based tabling. The design and implementation of TSTs appears in Section 3. Comparisons between DTSA, TST and variant-based tabling engines appear in Section 4. In Section 5 we provide performance results of a subsumption-based tabling engine implemented using TSTs.

2 Answer Clause Resolution via Time Stamps

Below we give an overview of the time-stamp-based subsumptive tabling engine. We begin with an abstract description of the operations of a tabling system.

2.1 An Overview of Tabling Operations

We can view top-down tabled evaluation of a program in terms of four abstract table operations: *call-check-insert*, *answer-check-insert*, *retrieve-answer* and *pending-answers*. Below, we describe each of these operations in the context of subsumptive tabling.

Call-check-insert. Given a call c, the *call-check-insert* operation finds a call c' in the call table that subsumes c. If there is no such c', then c is inserted into the call table. Note that *call-check-insert* is independent of the data structures used to represent answer tables.

A subgoal that is resolved using program creates answers to be inserted into the corresponding answer table, say T, and is known as the *producer* of T. A subgoal that is resolved using answer resolution with respect to an answer table T is known as a *consumer* of T.

Answer-check-insert. This operation is used to add the answers computed for a calll into its corresponding answer table. The operation ensures that the answer tables contain only unique answers. Note that, while the data structures used to represent answer tables may be different between subsumptive and variant tabling, the requirements on *answer-check-insert* remain the same.

Retrieve-answer. Answer clause resolution of a call c against a set of terms $T = \{t_1, t_2, \ldots, t_n\}$ in an answer table produces a set R such that $r \in R$ iff $r = t_i\theta_i$ for some t_i, where $\theta_i = mgu(c, t_i)$. This resolution is performed using *retrieve-answer* operations. In a tuple-at-a-time resolution engine, a *consumer choice point* is placed so that the answers can be retrieved one by one using backtracking. To ensure that an answer is returned at most once, we maintain an

answer continuation in the choice point, which represents the set of all answers remaining to be retrieved. Hence the arguments supplied to a *retrieve-answer* operation is split naturally into:

- *first_answer:* Given an answer table T and a call c, return an answer from T that unifies with c, and an answer continuation.
- *next_answer:* Given an answer table T, a call c, and an answer continuation γ, return the next answer a from T that unifies with c as specified by γ, and a modified answer continuation γ' that represents the remaining answers.

Pending-answers. Answer continuation $\perp$ denotes that there are no more remaining answers. When a *retrieve-answer* operation for a call c on an incomplete table T returns $\perp$, the call c will be suspended. The suspended call is later resumed when new answers have been added to T, or when T is known to be complete. Suspension and resumption of calls are performed by the answer scheduler which invokes *pending-answers* to determine whether a suspended call needs to be resumed. Given an answer continuation, *pending-answers* succeeds iff the continuation represents a non-empty set of answers.

2.2 Answer Retrieval in Subsumptive Tabling

The DTSA, proposed as a data structure for answer tables in [9], directly supports the *first-answer* and *next-answer* operations. In this paper, we describe an alternate, two-tier mechanism to realize these operations. At the fundamental level, we decouple the operation of *identifying* the answers relevant to a given goal in an answer set from the operation of *unifying* one of these answers with the goal. This separation frees the identification process from tuple-at-a-time execution. We hence propose an efficient mechanism to identify *all* relevant answers that have been added since the last time the table was inspected. We associate a time stamp with each answer and maintain, as part of the answer continuation, the maximum time stamp of any answer in the set. These answers are stored in a TST and accessed using *identify_relevant_answers* which, given a table T, time stamp τ and goal G, identifies the set of all answers in T with time stamps *greater than* τ that unify with G. It returns this set as a list (with some internal order) as well as the maximum time stamp of any answer in T.

Recall that answers are consumed from a table by a *first_answer* operation followed by a sequence of *next_answer* operations. We can implement these tuple-at-a-time operations based on time stamps as follows. We assume that time stamps are positive (non zero) integers. To compute *first_answer*, we use *identify_relevant_answers* with a time stamp of zero (thereby qualifying all available answers), select one answer from the returned set as the current answer, and store those remaining in an internal data structure called an *answer list*. The answer list, together with the maximum time stamp returned by *identify_relevant_answers*, form the answer continuation. On subsequent invocations of *next_answer*, we simply manipulate the answer list component of the continuation, as long as the answer list is not empty. Should the answer list be empty, we

"refresh" the continuation by another call to *identify_relevant_answers*, using the time stamp component of the continuation to restrict identification to only those answers that have been inserted since the last call to *identify_relevant_answers*.

Let τ be the time stamp returned by an invocation of the access function *identify_relevant_answers*. Since this operation identifies *all* relevant answers, $\langle [\,], \tau \rangle$ represents the empty continuation, $\perp$. Note that the continuation, $\langle [\,], 0 \rangle$, that represents all answers in a table. Thus, *first_answer*(T, c) can be realized simply as *next_answer*$(T, c, \langle [\,], 0 \rangle)$.

The method described above can be formalized readily; see [5] for details. We now state the requirement on *identify_relevant_answers* that is needed to show the correctness of *first_answer* and *next_answer*:

Requirement 1 *Given an answer table T representing a set of answers S, a goal G and a time stamp τ,* identify_relevant_answers(T, G, τ) *returns a permutation of the set $\{a \in S \mid a$ unifies with G and timestamp$(a) > \tau\}$.*

The correctness of operation *retrieve-answer* is then ensured by the following proposition:

Proposition 1. *Given a goal G and answer table T representing a set of answers S, let $\langle a_1, \gamma_1 \rangle, \ldots, \langle a_n, \gamma_n \rangle$ be a sequence of values such that $\langle a_1, \gamma_1 \rangle =$* first_answer$(T, G)$ *and $\langle a_{i+1}, \gamma_{i+1} \rangle =$* next_answer$(T, G, \gamma_i)$*, where $1 \leq i < n$ and $\gamma_n = \perp = \langle [\,], \tau \rangle$. Further, let B be the set $\{b \in S \mid b$ unifies with G and timestamp$(b) > \tau\}$. Then, provided* identify_relevant_answers *satisfies Requirement 1, the sequence $\langle a_1, \ldots, a_n \rangle$ is a permutation of the set $\{b\theta \mid b \in B$, where $\theta = mgu(b, G)\}$.*

3 Time-Stamped Tries

In this section we describe Time-Stamped Trie, which permits indexing on term symbols as well as time stamps. A TST represents a set of terms T and supports two operations: (1) Insert a term t into set T if not already present, and (2) Given a term t, determine the set of all terms $t' \in T$ that unify with t. Below we describe these operations formally. We begin by defining notational conventions and terminology and review answer tries described in [8].

A *position* in a term is either the empty string Λ that reaches the root of the term, or $\pi.i$, where π is a position and i is an integer, that reaches the i^{th} child of the term reached by π. By $t|_\pi$ we denote the symbol at position π in t. For example, $p(a, f(X))|_{2.1} = X$. Terms are built from a finite set of function symbols $\mathcal{F}$ and a countable set of variables $\mathcal{V} \cup \hat{\mathcal{V}}$, where $\mathcal{V}$ is a set of *normal* variables and $\hat{\mathcal{V}}$ is a set of *position* variables. The variables in the set $\hat{\mathcal{V}}$ are of the form X_π, where π is a position, and are used to mark certain positions of interest in a term. We denote the elements of $\mathcal{F} \cup \mathcal{V}$ by α.

A *trie* is a tree-structured automaton used for representing a set of terms $T = \{t_1, t_2, \ldots, t_n\}$. A trie for T has a unique leaf state s_i for every t_i, $1 \leq i \leq n$, such that the sequence of symbols on the transitions from the root to s_i

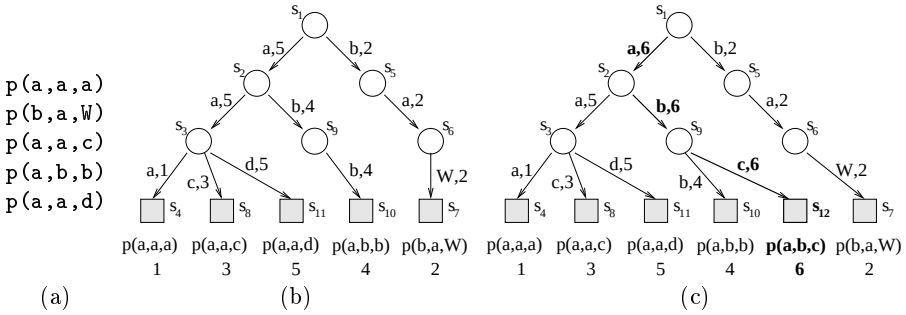

Fig. 1. Set of terms (a), its corresponding TST representation (b), and effect of inserting `p(a,b,c)` to the TST (c)

correspond to the sequence of symbols encountered in a preorder traversal of t_i. Moreover, for every state in a trie, all outgoing transitions have unique labels. With each state s is associated a skeleton, denoted by $Skel_s$, that represents the set of unification operations that must be performed to reach that state. The fringe of $Skel_s$ represents positions where further unification operations need to be performed before unifying a goal term with a term in T. The position in the goal term inspected at s is represented by the first position variable encountered in a preorder traversal of the skeleton. We denote this position by $\pi(s)$. Each outgoing transition from s represents a unification operation involving position $\pi(s)$ in the goal and the symbol labeling this transition. We label transitions by α, where α is a variable or a function symbol. Let the position examined at the current state be π. Then the skeleton of the destination state reached by a transition labeled by α is $Skel_s[t/X_\pi]$, where $t = f(X_{\pi.1}, \ldots, X_{\pi.n})$ if α is a n-ary function symbol f and $t = \alpha$ otherwise. Note that for a leaf state s in the trie, $fringe(Skel_s)$ is empty and that $Skel_s \in T$ is the term represented by s.

A Time-Stamped Trie (see Figure 1) is a trie augmented with information about the relative time its terms were inserted. The time of insertion of each term is called its *time stamp*, and is represented by a positive (non zero) integer. Each transition in a TST is additionally labeled with the maximum time stamp of any leaf reachable using that transition. Hence, all transitions in a TST will be of the form $s \xrightarrow{\langle \alpha, \tau \rangle} d$, where s and d are states in the TST, α is the symbol, and τ is the time stamp. We refer to these attributes of a transition δ as *symbol(δ)* and *timestamp(δ)*, respectively. Since *identify_relevant_answers* looks only for answers with time stamps greater than a given value, the *maximum* time stamp information on transitions is necessary to restrict the search on portions of the TST that correspond to answers with such time stamp values.

3.1 Term Insertion into a TST

Terms are inserted into a TST in a manner analogous to the single-pass check-insert for the trie representation described in [8]. The TST is traversed recursively

algorithm *insert*(s, t, τ)
(* returns **true** iff t was successfully inserted *)
  if (*fringe*$(Skel_s) = \{\}$) **then**
    return (**false**) (* term already exists, hence insert fails *)
  endif
  let $G|_{\pi(s)} = f/n = \alpha$
  if ($\exists \delta : s \xrightarrow{\langle \alpha, \tau' \rangle} d$) **then**
    if (*insert*(d, t, τ)) **then**
      timestamp$(\delta) = \tau$ (* insert successful, so update time stamp *)
      return (**true**)
    else
      return (**false**) (* insert failed; propagate failure up *)
    endif
  else (* match fails at s, so add a new path *)
    create new state d and add transition $\delta : s \xrightarrow{\langle \alpha, \tau \rangle} d$
    $Skel_d = Skel_s\ [f(X_{\pi(s).1}, \ldots, X_{\pi(s).n})/X_{\pi(s)}]$
    insert(d, t, τ)
    return (**true**)
  endif

Fig. 2. Term Insertion into a Time-Stamped Trie

starting at the root. A transition from states s to d, $s \xrightarrow{\langle \alpha, \tau \rangle} d$, is taken if the symbol α matches the symbol in the goal term at the position specified by s. If a leaf state is reached, then the term already exists in the set, and hence the TST is left unchanged. On the other hand, if a match operation fails at a state s, then a new path is added to the TST corresponding to the given term. All time stamps on the transitions along the root-to-leaf path corresponding to this term are updated with a value greater than any other time stamp in the TST. This recursive procedure is specified in Figure 2. The TST obtained from the one in Figure 1(b) by adding the term `p(a,b,c)` is given in Figure 1(c). The new transitions and states, as well as the transitions whose time stamps were modified by the addition, appear in bold face in the figure.

3.2 Identifying Relevant Answers using a TST

We now describe how TST supports *identify_relevant_answers*. Given a goal G and a time stamp τ, answers in a TST, T, are identified by recursively traversing T from the root, at each state exploring all transitions that meet the term indexing as well as the time stamp constraints. The set of transitions to be explored can be formally specified as follows.

Definition 1 (Set of Applicable Destinations). *Given a Time-Stamped Trie T, the set of all destination states that are applicable upon reaching a state s in T with an initial goal G and time stamp τ, denoted* dest(s, G, τ)*, is such that*

$d \in \text{dest}(s, G, \tau)$ *iff (i)* $Skel_d$ *is unifiable with* G*, and (ii)* $s \xrightarrow{\langle \alpha, \tau' \rangle} d$ *is a transition in* T *with* $\tau' > \tau$*.*

In the above definition, condition (i) corresponds to indexing on *terms* while (ii) corresponds to indexing on *time stamps*.

Given a state s in a TST, a goal G, and time stamp τ, the set of all terms represented by the leaves reachable from s with time stamps greater than τ and unifiable with G is given by:

$$relevant(s, G, \tau) = \begin{cases} \{Skel_s\} & \text{if } s \text{ is a leaf} \\ \bigcup_{d \in \text{dest}(s, G, \tau)} relevant(d, G, \tau) & \text{otherwise} \end{cases}$$

Finally, let T be a TST. Then,

$$identify\_relevant\_answers(T, G, \tau) = relevant(root(T), G, \tau) \ .$$

We can establish that the above definition of *identify_relevant_answers* meets Requirement 1.

Proposition 2 (Correctness). *Given a Time-Stamped Trie* T *representing a set of terms* S*, a goal* G *and a time stamp* τ*,* identify_relevant_answers(T, G, τ) *computes the set* $\{a \in S \mid a \text{ unifies with } G \text{ and } timestamp(a) > \tau\}$*.*

Although one can readily derive a computation based on the above definition of *identify_relevant_answers*, its effectiveness depends on the efficient implementation of *dest*. It should be noted that the condition in *dest* for indexing on terms is identical to the one with which transitions to be traversed are selected in a (non time-stamped) trie [8]. The indexing on time stamps, however, is unique to TSTs. We can show that *identify_relevant_answers* can be efficiently computed given an efficient technique to index on time stamps *at each state* in a TST, as formally stated below.

Requirement 2 *Given a Time-Stamped Trie* T*,* $\Delta = \text{dest}(s, G, \tau)$ *is computed in time proportional to* $|\Delta|$*.*

Proposition 3 (Efficiency). *Let* G *be an open goal – i.e., a term whose immediate subterms are all distinct variables –* T *be a Time-Stamped Trie, and* identify_relevant_answers(T, G, τ) *be a non-empty set of terms* S *for some given value of* τ*. Then, if* dest *satisfies Requirement 2, the set* S *can be computed in time proportional to the sum of the sizes of the terms in* S*.*

The structure of TSTs do provide at each state, in addition to the normal index on symbols present in tries, an index on time stamps. Each time index can be maintained as a doubly-linked list of outgoing transitions *in reverse time-stamp order*. This organization allows us to select transitions based on time stamps alone, at constant time per selected transition, thereby satisfying Requirement 2. Moreover, by cross-linking the time and term indices, the time index can be updated in constant time as new transitions are created. Finally, note that TSTs support answer retrieval from *incomplete* answer sets the time index can be deleted when the answer table is complete.

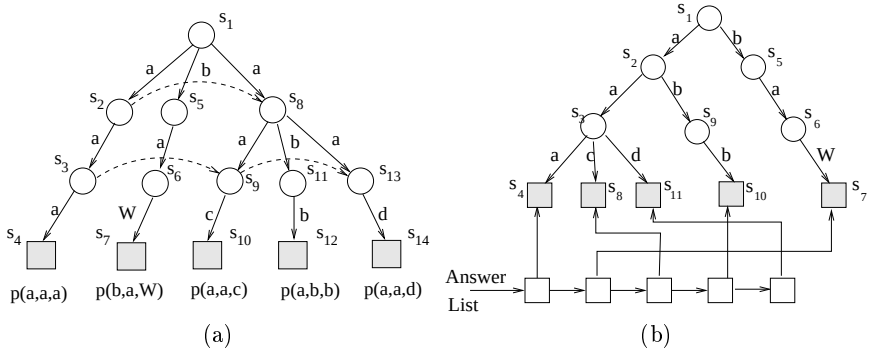

Fig. 3. DTSA (a) and Trie (b) representation of terms in Figure 1(a)

4 Comparison of Tabling Approaches

We now compare variant- , TST-, and DTSA-based tabling engines. To provide the context for comparisons, we now briefly review the data structure called *Dynamic Threaded Sequential Automaton* (DTSA) that was used in our earlier implementation of a subsumption based tabling engine.

The DTSA Structure. DTSA is a trie-like automaton for representing a set of ordered terms $\{t_1, t_2, \ldots, t_n\}$. A DTSA is constructed such that, in a preorder traversal, the leaf representing term t_i is visited before the leaf representing term t_{i+1} (Figure 3(a)). Since transitions from a state s preserve the order of terms represented by the final states reachable from s, there may be multiple transitions with the same label. The loss of indexing due to this duplication is offset by using *threads* which link states that share prefixes. For instance, using the DTSA in Figure 3(a) to find terms unifiable with a given goal term `p(a,a,V)`, after finding the first term (at state s_4), the next term, `p(a,a,c)`, can be retrieved by backtracking to s_3, following the thread to s_9, and making the transition to s_{10}. Observe that both s_3 and s_9 have `p(a,a,`X_3`)` as their skeleton.

4.1 Shared Features

The variant, DTSA- and TST-based subsumption engines share many common features. For instance all engines distinguish between *complete* and *incomplete* table entries. Completed answer tables are organized as *compiled trie code* in all of them (see [8] for details). Although incomplete tables are organized differently they all use substitution factoring whereby only the substitutions for the variables of the call are stored in the answer tries [8]. Once a table entry has completed, all structures created to support answer resolution from the incomplete table are reclaimed. These include the leaf node pointers, the auxiliary structures for indexing on time in the TST-based subsumption engine, and the entire DTSA itself in the DTSA-based subsumption engine.

In the following, we focus on the differences between the three engines.

4.2 Variant- vs. TST-Based Engine

Subsumption engines have a more complex call-check-insert operation than the variant engine. However, this adds very little (constant factor) overhead to the evaluation time. Due to sharing of answer computations, subsumption engines can show arbitrary gains in time performance. The interesting result is that the table space used by a TST-based engine is always within a constant factor of that used by the variant engine. In fact,

Proposition 4 (Space Complexity of TST-Based Engine). *The maximum table space used during computation in a TST-based engine is at most twice that of the variant engine.*

This bound follows from the observation that a TST is structured like a trie with respect to the representation of the answers as sequences of symbols – i.e., a trie and a TST representing the same set of answers have the same number of nodes (states) – and that the size of a TST node is twice that of a trie node (including space for the time index).

4.3 DTSA- vs. TST-Based Engine

The subsumptive engines can be distinguished by their approaches to performing answer retrieval. Subsumption using Time-Stamped Tries divides this operation into two processes: (i) identifying answers relevant to a particular goal, and (ii) unifying a selected relevant answer with that goal. Identification of relevant answers is achieved by a complete search of the TST, yielding a *set* of answers, as discussed in Section 3.2. In contrast, DTSA directly supports the primitive operations of answer retrieval, providing for simultaneous identification and consumption of a *single* answer.

Consequently, DTSAs have more complex continuations, requiring enough state information to resume the traversal; a continuation consists of a *set* of choice points, one for each level in the DTSA. On the other hand, since complete traversals of a TST are performed during each identification phase, only the maximum time stamp of all answers contained in the TST is required for subsequent processing.

DTSA also consumes more table space than TST. In contrast to Proposition 4, the maximum size of a table in the DTSA-based engine is *at least* double that of the representation in the variant engine, as an answer trie (Figure 3(b)) is created in addition to the DTSA. Moreover, the number of nodes in a DTSA may be quadratic in the number of nodes in a corresponding TST. For example, consider the program depicted in Figure 4(a). Answers to the query `a(X,Y)` are discovered in such an order that no sharing of nodes is possible in the DTSA (Figure 4(b)). However, since answers are simply marked with the insertion time in a TST, rather than stored in derivation order, the resulting sharing makes the corresponding TST more compact (Figure 4(c)). It can be shown that the number of nodes required to represent the set of answers to the query `a(X,Y)` in the TST is $n(n-1)/2 + 2k$, whereas in the DTSA, the number of nodes required

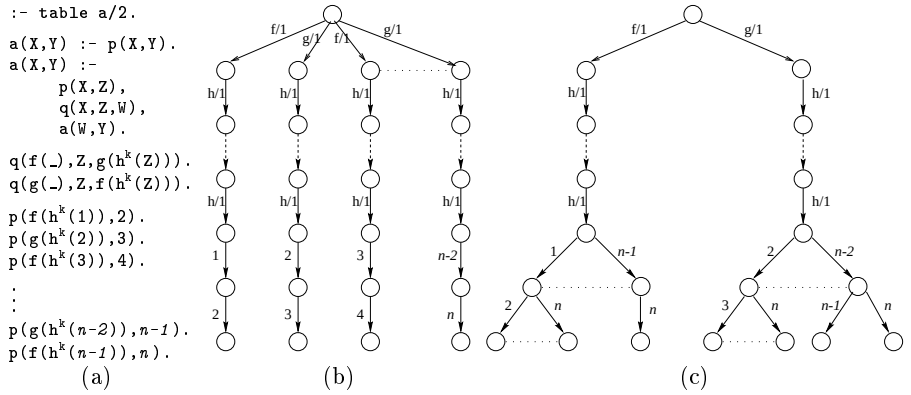

Fig. 4. Program (a) and two organizations of the answer set for the query `a(X,Y)`: That which would be produced for a DTSA (b) and a (Time-Stamped) Trie (c)

is $n(n-1)(2+k)$. As can be seen from both the diagram and these expressions, the size of k adds but a constant factor to the size of the TST, whereas in the DTSA, its effect is multiplicative.

5 Experimental Results

We now present experimental results to compare the performance of the TST engine with that of the variant and DTSA engines. All measurements were taken on a SparcStation 20 with 64MB of main memory running Solaris 5.6. We present the results using the benchmarks presented in [8], derived from the programs: left-, right-, and double-recursive versions of transitive closure (`lrtc/2`, `rrtc/2`, `drtc/2`), and the same generation program (`sg/2`). We note that both the TST and DTSA engines were constructed from different base versions of XSB – the TST engine from version 1.7.2, and the DTSA engine from version 1.4.3. The measurements were made so as to minimize the impact of this difference during each experiment, as discussed below. Because the changes between to two versions did not grossly affect the method of evaluation— in particular, XSB's scheduler— we feel that the following results accurately reflect the relative performance of the systems.

Time Efficiency. The TST engine shows little overhead when the evaluation does not lead to subsumed calls. The overheads result only from the subsumption-checking *call-check-insert* operation. As time-stamp indexes are lazily created, no penalty is incurred due to their maintenance. In addition, updating of time stamps along the path of insertion is avoided by assigning a time stamp value of 1 to all answers entered before the first subsumed call is made. These optimizations enable the TST engine to perform performing within 2% of the speed of the variant engine (see [5] for details).

The performances of variant and TST engines differ when subsuming valls are made. Table 1 shows the execution times of the TST and DTSA engines relative

Table 1. Speedups of DTSA and TST engines in evaluations involving subsumption

Query	Graph	Size	XSB 1.4.3			XSB 1.7.2		
			Variant	DTSA	Speedup	Variant	TST	Speedup
`rrtc(X,Y)`	Chain	512	3.36	3.44	0.97	3.05	2.16	1.41
		1024	16.8	13.8	1.21	14.6	8.74	1.68
	Tree	2048	0.48	0.77	0.62	0.45	0.37	1.21
		4096	1.06	2.08	0.51	1.00	0.83	1.20
`drtc(X,Y)`	Chain	128	6.54	4.60	1.43	5.88	3.09	1.90
		256	52.0	38.7	1.35	51.5	24.1	2.14
	Tree	2048	1.87	1.91	0.98	1.72	1.25	1.38
		4096	4.57	4.86	0.94	4.24	3.00	1.41
`sg(X,Y)`	Chain	512	0.68	0.06	11.3	0.64	0.04	15.8
		1024	2.73	0.12	22.5	2.58	0.09	27.5
	Tree	128	0.12	0.14	0.89	0.11	0.10	1.10
		256	0.49	0.50	0.97	0.43	0.43	1.01
`lrtc(1,X),` `lrtc(2,X)`	Chain	2048	29.4	0.13	219	27.5	0.10	277
		4096	118	0.30	392	110	0.20	574
	Tree	2048	17.6	0.10	176	17.3	0.09	190
		4096	71.1	0.20	351	69.7	0.18	388

to their base variant engines on examples with this behavior. We compare the performance of each engine using the speedups achieved over their base variant engines, thereby removing the noise in the results caused by differences in their base implementations. As each subsumptive engine merely extends the tabling subsystem to support subsumption, this method measures the performance gain achievable by each tabling approach. Notice that in all cases, the TST engine performs at least as well as the DTSA engine.

Table Space Usage. In Table 2 we report on the memory utilized in representing the tables for some of the benchmarks described earlier. This *table space* consists of the call and answer tables, where the latter consists of answer tries, TSTs, DTSA, and answer lists as present in a particular engine. We report the maximum space consumed *during* evaluation, as well as the space consumed by the *completed* tables. Table 2 is divided into three sections: the upper portion shows results for benches that do not make use of subsumption; the middle section shows results for benches for which properly subsumed calls consume answers only from incomplete tables; and the bottom portion shows results for a bench whose subsumed calls consume answers from completed tables only. The space used by the DTSA engine for completed tables is identical to that of TST, and therefore is not repeated.

As mentioned earlier, the subsumptive engines exhibit a 20% overhead in the representation of answer tries as compared to the variant engine. For the benches which do not exhibit subsumption, observe that the actual increase appears lower due to the presence of the call table and answer lists in this measure, which are

Table 2. Space usage for tables in variant and subsumptive engines

Query	Graph	Size	Maximum Table Space			Completed Table Space	
			Variant	DTSA	TST	Variant	TST
`lrtc(1,Y)`	Chain	4096	128 KB	144 KB	144 KB	96.1 KB	112 KB
		8192	256 KB	288 KB	288 KB	192 KB	224 KB
`rrtc(1,Y)`	Chain	512	4.23 MB	4.75 MB	4.75 MB	3.23 MB	3.75 MB
		1024	16.8 MB	18.8 MB	18.8 MB	12.8 MB	14.8 MB
`rrtc(X,Y)`	Chain	512	7.41 MB	9.89 MB	7.73 MB	6.41 MB	3.73 MB
		1024	29.5 MB	39.1 MB	30.8 MB	25.5 MB	14.8 MB
	Tree	2048	1.23 MB	1.64 MB	1.23 MB	1.09 MB	702 KB
		4096	2.68 MB	3.58 MB	2.69 MB	2.36 MB	1.48 MB
`drtc(X,Y)`	Chain	128	492 KB	826 KB	508 KB	429 KB	252 KB
		256	1.87 MB	3.19 MB	1.95 MB	1.62 MB	971 KB
	Tree	2048	1.23 MB	1.71 MB	1.23 MB	1.09 MB	702 KB
		4096	2.68 MB	3.73 MB	2.69 MB	2.36 MB	1.48 MB
`sg(X,Y)`	Chain	512	88.1 KB	110 KB	84.1 KB	84.1 KB	60.1 KB
		1024	176 KB	220 KB	168 KB	168 KB	120 KB
	Tree	128	231 KB	430 KB	309 KB	188 KB	168 KB
		256	855 KB	1602 KB	1190 KB	685 KB	629 KB
`lrtc(1,X)`, `lrtc(2,X)`	Chain	2048	320 KB	128 KB	128 KB	304 KB	112 KB
		4096	640 KB	256 KB	256 KB	608 KB	224 KB
	Tree	2048	276 KB	100 KB	100 KB	260 KB	84.4 KB
		4096	522 KB	200 KB	200 KB	520 KB	168 KB

present in all engines. When answer lists are reclaimed upon completion, the space overhead approaches 20%.

The query `lrtc(1,X),lrtc(2,X)` exhibits behavior similar to nonsubsumptive evaluations during construction of the tables as subsumed calls do not occur until the tables complete. This allows both subsumptive engines to avoid constructing their respective subsumption-supporting data structures. Since answer tables are shared, the subsumption makes the subsumption engines consume less maximum and final spaces than the variant engine.

For those queries which utilize subsumption from incomplete tables, only a single table is constructed under subsumptive evaluation – that of the original query. This table expands throughout the computation until it completes, terminating the evaluation. Under variant evaluation, however, several tables are constructed in addition to the one for the query itself, but are completed incrementally during the computation. Therefore, memory usage is somewhat amortized as space is periodically freed by the tables as they complete. The relative performance in maximum space usage between the two tabling paradigms, then, depends not only on the amount of answer sharing that is possible – and so the extent to which duplicity can be avoided – but also on the pattern of table completion. For these queries, only `sg(X,Y)` on chains exhibits conditions during the evaluation which are conducive to savings under subsumption, and that

Table 3. Maximum choice point space usage for various tabling engines

Query	Graph	Size	Variant	DTSA	TST
`lrtc(1,Y)`	Chain	4096	0.61 KB	0.61 KB	0.61 KB
		8192	0.66 KB	0.66 KB	0.66 KB
`rrtc(1,Y)`	Chain	512	36.5 KB	36.5 KB	36.5 KB
		1024	72.5 KB	72.5 KB	72.5 KB
`rrtc(X,Y)`	Chain	512	36.5 KB	100 KB	30.5 KB
		1024	72.6 KB	208 KB	60.5 KB
	Tree	2048	1.64 KB	288 KB	120 KB
		4096	1.81 KB	592 KB	240 KB
`drtc(X,Y)`	Chain	128	16.4 KB	494 KB	477 KB
		256	32.5 KB	1950 KB	1913 KB
	Tree	2048	1.76 KB	1.22 MB	1.06 MB
		4096	1.94 KB	2.69 MB	2.34 MB
`sg(X,Y)`	Chain	512	0.71 KB	100 KB	30.5 KB
		1024	0.77 KB	208 KB	60.6 KB
	Tree	128	0.76 KB	16.5 KB	7.93 KB
		256	0.82 KB	33.7 KB	15.5 KB
`lrtc(1,X)`, `lrtc(2,X)`	Chain	2048	0.74 KB	0.74 KB	0.74 KB
		4096	0.80 KB	0.80 KB	0.80 KB
	Tree	2048	0.78 KB	0.78 KB	0.78 KB
		4096	0.84 KB	0.84 KB	0.84 KB

by using TSTs only. However, as Table 2 also shows, in *all* subsumptive cases (middle and lower portions of the table) the TST engine yields a more compact representation of the *completed tables*, even though each TST consumes more space than the corresponding answer trie in the variant engine.

Finally, for all queries where subsumed calls are resolved only with incomplete tables, the TST engine outperforms the DTSA engine in maximum space required as predicted (Sect 4.3). The results show that the amount of savings can be significant, in absolute (see `rrtc(X,Y)`) or relative terms (see `drtc(X,Y)`).

Choice Point Creation. In Table 3 we present maximum choice point stack usage which, together with table space usage, accounts for most of the total memory used during query evaluation with subsumption. As the sizes of producer and consumer choice point frames differ between the versions of XSB, we have added padding to these structures to normalize the values and enable a direct comparison. For this reason, only results from one version of a variant engine is shown in this table.

As compared to a variant based evaluation, choice point space is likely to increase as subsumed calls are dependent upon the more general goal, and therefore cannot complete before the general goal itself is completed. Hence, the corresponding consumer choice points must remain active on the choice point stack. In contrast, under variant evaluation, the more specific goals are resolved by program resolution, independent of the more general call, and hence have

the opportunity to complete earlier and free stack space for reuse. Note that the former condition is a characteristic of subsumption-based query evaluation rather than any particular implementation of the tables. In particular, for the query `drtc(X,Y)` executed on trees of depth k, it can be shown that the maximum number of concurrently active choice points is equal to the number of calls to `drtc/2`, which is proportional to $k2^k$, whereas under variant evaluation, the maximum number of active choice points is only $2k$. However, there are cases, such as occurs in the evaluation of `rrtc(X,Y)`, where the initial call pattern is identical under either evaluation strategy. Here, the use of subsumption actually saves space as the representation of a consuming call on the choice point stack is more compact than that of a producer. Note that, even in the worst case, the choice point stack expansion is tied to the number of interdependent calls, and hence proportional to the table space.

As discussed in Section 4.3, the DTSA engine uses more stack space than the TST due to the addition of specialized choice points for performing answer resolution from the DTSA. As the data in Table 3 shows, the TST engine outperforms the DTSA engine in terms of choice point space usage in all examples. Moreover, the overhead is significant, sometimes resulting in usage that is more than triple that of the TST engine. Finally, recall that the table space of the TST engine is proportional to that of the variant engine, and that its choice point space usage is proportional to its table space usage. Therefore, the TST engine's total usage is proportional to that of the variant engine.

6 Discussion

We presented a new organization of tables based on time-stamped tries for subsumption based tabling. We showed from a theoretical as well as practical perspective that it is superior to a tabling engine based on DTSA. Further we have shown that the space performance of such an implementation is no worse than a constant factor away from a variant implementation.

Existence of an efficient subsumption based tabling engine opens up interesting new opportunities for expanding the applicability of top-down evaluation strategies. For instance, programs exist for which top-down, goal-directed query evaluations impose a high factor of overhead during the computation when compared to semi-naive bottom-up evaluation. Preliminary evidence suggests that the performance of our TST-based tabling engine augmented with *call abstraction* is competitive with semi-naive bottom-up evaluation methods (see [5] for details). In general, abstraction of a call c is performed by first making a more general call, c', and allowing c to consume answers from c'. Observe that doing call abstraction within a subsumptive engine on a call c amounts to losing goal directedness as far as the evaluation of c is concerned. Thus by selectively abstracting calls we can vary the degree of goal directness employed during an evaluation without changing the core evaluation strategy.

References

[1] R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *Proc. of the Symp. on Logic Programming*, 1993.

[2] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.

[3] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and T. Swift. Optimizing clause resolution: Beyond unification factoring. In *ICLP*, 1995.

[4] Y. Dong, et al. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS '99)*. Springer Verlag, 1999.

[5] E. Johnson, C. R. Ramakrishnan, I. V. Ramakrishnan, and P. Rao. A space-efficient engine for subsumption-based tabled evaluation of logic programs. Technical report. Available from `http://www.cs.sunysb.edu/~ejohnson`.

[6] Y. S. Ramakrishna, et al. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.

[7] C. R. Ramakrishnan, S. Dawson, and D. S. Warren. Practical program analysis using general purpose logic programming systems - a case study. In *ACM Symposium on Programming Language Design and Implementation*, 1996.

[8] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *JLP*, January 1999.

[9] P. Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A thread in time saves tabling time. In *JICSLP*. MIT Press, 1996.

[10] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *ICLP*, pages 84–98. MIT Press, 1986.

[11] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3), July 1991.

[12] The XSB Group. The XSB programmer's manual, Version 1.8, 1998. Available from `http://www.cs.sunysb.edu/~sbprolog`.