Optimizing Clause Resolution in Tabled Logic Programs

(Extended Abstract)

Steven Dawson
C.R. Ramakrishnan
I.V. Ramakrishnan
Terrance Swift

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

Abstract

The incorporation of tabulation into resolution methods ([3, 5]) has proven effective for computing the well-founded semantics in a goal-oriented manner. It has also given rise to an extremely efficient evaluation method for in-memory deductive databases [12]. Perhaps because these results are quite recent, the problem of optimizing *tabled clause resolution* has remained open, despite the fact that optimizing term indexing has been extensively studied in Functional Programming, and optimizing program clause unification in Logic Programming.

We address the problem of statically optimizing clauses by using a family of *Clause Resolution Automata* (CRA), to characterize the general nature of resolution in tabled programs. The members of the family exhibit different behavior depending on whether a predicate is tabled, on the information known about modes, and on whether clause order is to be preserved. We first derive complexity results for the members of this family. For classes of programs where optimal CRA's can be constructed in polynomial time, dynamic programming algorithms are specified for their construction. For those classes of programs where construction of optimal CRA's is not known to be polynomial, we develop heuristics for constructing CRA's and establish their properties. Finally, we present experimental results that show the impact of our approach on the performance (time *and* space usage) of a set of representative programs.

Contact author: Steven Dawson
E-mail: sdawson@cs.sunysb.edu
Tel: (516) 632-8470
Fax: (516) 632-8334

1 Introduction

SLG resolution [4, 6] is emerging as a powerful technique for combining Deductive Databases, Non-monotonic Reasoning and Logic Programming. SLG is a generalization of SLD resolution with tabling. A form of SLD with tabling has been shown in [13] to be asymptotically equivalent to a variant of magic templates [11]. Among properties relevant to deductive databases, SLG evaluates programs according to the well-founded model [16], terminates for programs with bounded term size, and has polynomial data complexity for Datalog programs with negation. SLG is amenable to efficient implementation, as shown by the performance of the XSB system [12]. In particular, the XSB system has been shown to compute in-memory deductive database queries about an order of magnitude faster than current semi-naive methods, and to compute Prolog queries with little loss of efficiency when compared to well known Prolog systems [14].

Like other tabling methods, SLG evaluates programs by recording subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a table. In particular predicates are marked *a priori* as either *tabled* or *nontabled*, and clause resolution, which is the basic mechanism for program evaluation, proceeds as follows. For nontabled predicates the subgoal is resolved against program clauses. For tabled predicates, if the subgoal is already present in the table, then it is resolved against the answers recorded in the table; otherwise the subgoal is entered in the table, and its answers, computed by resolving the subgoal against program clauses, are also entered in the table. For both tabled and nontabled predicates, program clause resolution is carried out using SLD.

Each clause resolution step results in several elementary operations. Specifically, entering a call or an answer into the table, referred to as *call tabling* and *answer tabling* respectively, requires elementary lookup (to check whether the call or answer already exists in the table) and insert operations. Program clause and answer clause resolution results in elementary unification operations (match operations and variable bindings). For example, resolving the goal $p([+,a],W)$[1] for the first time against the tabled predicate in Figure 1a results in the following operations. To enter it in the table requires 6 table insert operations, one per symbol.[2] The program clause resolution step that follows call tabling results in 4 unification operations (2 match operations and 2 bindings) to unify the call with the first clause head, an additional 4 unification operations to unify with the second clause head, and 2 more before failing to unify with the third clause head. Tabling the answer $p([+,a],[\,])$ requires 6 table insert operations. Backtracking through this answer clause on a subsequent call involves 6 unification operations.

Improving the performance of clause resolution steps by reducing the number of elementary operations is the topic of this paper. For a simple exposition of potential optimizations, consider the above example. Observe that to unify the goal $p([+,a],W)$ with the first two program clause heads, we need only perform 2 match operations (involving the outermost *cons* and $+$) and 4 bindings (for the four head variables), *i.e.,* we can share some of the unification operations. Traditionally, in tabling systems the call is always tabled *prior* to performing any unification operation [4, 15, 17]. But now suppose that in the above example we perform tabling operations *after* unifying the goal with the outermost *cons* and $+$. Since all the answers have the form $cons(+,\ldots)$, and the corresponding unifications have already been performed, we need not enter these two symbols in the answer table. This saves both table space and associated table insert operations. Furthermore, since the outermost *cons* and $+$ have been matched they need not even be entered in the call table.

We present a technique called *clause resolution factoring* that improves the efficiency of clause resolution by sharing some unification operations and avoiding some table lookup/insert operations by suitably interleaving the unification and tabling operations, as illustrated in the above example. Clause resolution factoring is a compile time technique and is presented as a source transformation. For instance, the program in Figure 1a can be transformed to the program in Figure 1b by clause resolution factoring. Resolving the goal $p([+,a],W)$ for the first time in the transformed program results in 7 unification operations and 8 tabling operations to insert both the goal $(p2([a],W))$ and the answer $(p2([a],[\,]))$ into the tables. In general there are different ways to share and interleave operations in a program, yielding transformed programs with differing performance. The design of optimal clause resolution factoring, *i.e.,* factoring that yields a program that has the best performance, is addressed in this paper.

[1] We assume Prolog syntax for lists. *e.g.* the list $[+,a]$ denotes the list $cons(+,cons(a,nil))$.

[2] We assume that each tabled predicate is associated with its own table. Hence we need not store p in the table.

```
:- table p/2.
p([+|X],Y) :- p(X,Z), p(Z,Y).
p([+|X],Y) :- p(X,Y).
p([a|X],Y) :- X = Y.
```

(a)

```
:- table p2/2.
p([X|Y],Z) :- p1(X,Y,Z).
p1(+,X,Y) :- p2(X,Y).
p1(a,X,Y) :- X = Y.
p2(X,Y) :- p(X,Z), p(Z,Y) .
p2(X,Y) :- p(X,Y).
```

(b)

Figure 1: Sample program before (a) and after (b) clause resolution factoring

We model the resolution process in SLG by a *clause resolution automaton* (CRA) whose states and transitions capture elementary unification and tabling operations. By associating appropriate costs with these states and transitions, a least cost CRA would correspond to optimal clause resolution. We present theoretical results related to the construction of optimal CRA's and provide strong experimental evidence of the effectiveness of CRA's in practice. This paper significantly generalizes very recent work in [7] where we developed a technique for sharing unification operations in Prolog-style SLD clause resolution. But SLG's tabling mechanisms add a new and difficult dimension to factoring clause resolution. Moreover, we had left open the problem of efficiently sharing unification operations in SLD resolution when the modes of arguments in the call are known. This problem is also addressed here. The paper is organized as follows. Section 2 formalizes CRA, and Section 3 describes appropriate cost measures for it. Complexity results concerning construction of optimal CRA's appear in Section 4. The table below is a summary of these results.

	Without mode information *Computing optimal CRA*		*With mode information* *Computing CRA cost (Seq. or Nonseq.)*	
	Sequential	*Nonsequential*	*For given CRA*	*For optimal CRA*
Nontabled Datalog	P	**NP-complete**†	**NP-complete***	**NP-hard***
Nontabled general	**P**†	NP-complete	NP-complete	NP-hard
Tabled Datalog	**P***	**NP-complete***	**NP-complete***	**NP-hard***

The column labeled *Sequential* refers to the Prolog-style textual-order clause selection strategy used for program clause resolution, while *Nonsequential* refers to arbitrary clause selection. Results independently proved in this paper are marked with "$*$", while results marked with "†" were proved in [7]. Each of the remaining results is a direct consequence of one of the other results in the table. In Section 5 we develop heuristics for those cases where optimal CRA construction is computationally difficult and provide experimental evidence about their effectiveness. Discussion appears in Section 6.

2 Clause Resolution Automata

A clause resolution automaton provides a formal framework for clause resolution factoring. It is structured as a tree, with the root as the start state, and the edges, denoting transitions, representing elementary unification operations. Every state is labeled either *Tabled* or *Nontabled*, and call and answer tabling operations are done only in the tabled states. Every leaf state represents a program clause, and states encountered and transitions made on the path from the root to a leaf are the tabling and elementary unification operations needed to unify the head of that clause with a goal. CRA's elegantly capture sharing of unification and tabling operations in program and answer clause resolution[3]. Specifically, shared edges represent common unification operations that are needed to unify the goal with two or more clauses, and shared tabled states represent tabling operations common to the clauses. Note that since the transitions represent unification operations, all possible transitions out of a state are attempted; *i.e.*, the automaton is non-deterministic. Below, we formalize the CRA, following (and generalizing) our formalism in [7].

[3] The degree of sharing in answer clause resolution is limited to portions common to program and answer clauses.

Preliminaries We assume the standard definitions of term and the notions of substitution and subsumption of terms. 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, $p(a, f(X))|_{2.1} = X$. We denote the set of all positions by Π. 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 *program* 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 simply as a notational convenience to mark certain positions of interest in a term. The symbol t (possibly subscripted) denotes terms; $\alpha, \alpha', \ldots$ denote elements of the set $\mathcal{F} \cup \mathcal{V}$; $\gamma, \gamma', \ldots$ denote elements of the set $\mathcal{F} \cup \mathcal{V} \cup \Pi$; and f, g, h denote function symbols. The arity of a symbol α is denoted by $arity(\alpha)$; note that the arity of variable symbols is 0. Simultaneous substitution of a term t' at a set of positions P in term t is denoted $t[P \leftarrow t']$. For example, $p(X_1, f(X_{2.1}), X_3)[\{2.1, 3\} \leftarrow b] = p(X_1, f(b), b)$.

Operationally, a CRA is the computational vehicle for performing clause resolution. At any given state in a CRA's computation, we need to capture the operations that have been performed, as well as those that remain to be done. The partial computation of unification is captured by a *skeleton*, which is a term over $\mathcal{F} \cup \mathcal{V} \cup \hat{\mathcal{V}}$. Elements of $\mathcal{F} \cup \mathcal{V}$ in a skeleton represent unification operations that have been performed. Position variables denote portions of the goal where the remaining operations will be performed. Given a skeleton, its *fringe* defines the positions to be explored for unification to progress. Formally,

Definition 2.1 (Skeleton and Fringe) A *skeleton* is a term over $\mathcal{F} \cup \mathcal{V} \cup \hat{\mathcal{V}}$. The *fringe* of a skeleton S, denoted $fringe(S)$, is the set of all positions π such that $S|_\pi = X_\pi$ and $X_\pi \in \hat{\mathcal{V}}$.

For example, for the goal $q(f(U), V, W)$ the skeleton $q(X_1, Y, g(X_{3.1}, X_{3.3}, X_{3.3}))$ captures the fact that the substitution for W has been partially computed (to be $g(X_{3.1}, X_{3.3}, X_{3.3})$), the head variable Y has been unified with the second argument of the goal (V), and that the first argument of the goal has not yet been explored. The fringe of this skeleton is $\{1, 3.1, 3.3\}$.

With each state, either tabled or nontabled, we associate a skeleton that represents the partial computation of unification, and a subset of clauses, called the *compatible set* of the state. The clauses in the compatible set share the unification and tabling operations on the path from the root to that state. Recall that the fringe of a skeleton represents the positions in the partially unified goal that remain to be explored. A state then specifies one such position, and each outgoing transition represents a unification operation involving that position. We label the transition by γ, where γ is either a function symbol or variable in the clause head, or another position in the (partially unified) goal. For example, in Figure 2 the label on the transition from s_2 to s_3 specifies unifying position 1.1 of the goal with $+$. The compatible set for state s_3 consists of clauses 1 and 2.

Let the skeleton of the current state be S and the position examined be π. The skeleton of the destination state reached by a transition labeled γ is obtained by extending the skeleton S using the operation $extend(S, \pi, \gamma)$ defined below. Intuitively, S is extended by replacing all occurrences of X_π in S by the term corresponding to γ. If γ is a function symbol, this term has γ as root and position variables representing new fringe positions as its children. If γ is a position, this term is the position variable X_γ. Otherwise, this term is the variable γ itself.

Definition 2.2 (Skeleton extension) The *extension* of a skeleton S at fringe position π by γ, denoted $extend(S, \pi, \gamma)$, is the skeleton S' such that

$$S' = S[P \leftarrow t] \quad where \quad P = \{\pi' \mid S|_{\pi'} = X_\pi\}$$
$$and \quad t = \begin{cases} \gamma(X_{\pi.1}, \ldots, X_{\pi.k}) & (\gamma \in \mathcal{F} \text{ and } k = arity(\gamma)) \\ \gamma & (\gamma \in \mathcal{V}) \\ X_\gamma & (\gamma \in \Pi) \end{cases}$$

For example, in the skeleton $q(X_1, g(X_{2.1}, X_{2.3}, X_{2.3}))$, the operation of unifying position 1 with $f/1$ results in extending the skeleton to $q(f(X_{1.1}), g(X_{2.1}, X_{2.3}, X_{2.3}))$. The operation of unifying position 1.1 with position 1.2 further extends the skeleton to $q(f(X_{2.1}), g(X_{2.1}, X_{2.3}, X_{2.3}))$.

Definition 2.3 (Clause Resolution Automaton) A *clause resolution automaton* for a set of clauses C and a skeleton S is an ordered tree whose edges are labeled with elements of the set $\mathcal{F} \cup \mathcal{V} \cup \Pi$, and with each node s (a state) is associated a label L_s, a skeleton S_s, a position $\pi_s \in fringe(S_s)$ (if s is not a leaf), and a non-empty *compatible* set $C_s \subseteq C$ of clauses such that:

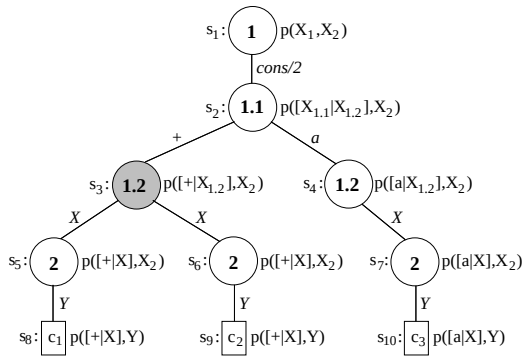


Figure 2: Clause resolution automaton for program in Figure 1a

1. $L_s \in \{\underline{Tabled}, \underline{Nontabled}\}$,
2. Every clause head in C_s is subsumed by S_s,
3. the root state has S as the skeleton and C as the compatible set,
4. for each edge (s, d) with label γ, $S_d = extend(S_s, \pi_s, \gamma)$, and
5. the collection of sets $\{C_d \mid (s, d)$ is an edge$\}$ is a partition of C_s.

The partitioning of the compatible set C_s at a fringe position π_s in the above definition ensures that transitions specify unification operations involving π_s in the goal, and either: 1) a function symbol or variable appearing at π_s *in at least one* of the clause heads in C_s; or 2) another (fringe) position in the goal. Each set in the partition is a compatible set of one of the next states of S_s, and all the clause heads in it share the unification operation specified by the corresponding transition. Note that, for a nontabled predicate, all states in the CRA will be labeled as *Nontabled*. We call such a CRA as a nontabled CRA.

Construction Using the program in Figure 1 (p. 2) and the corresponding automaton in Figure 2 for illustration, we informally describe the construction of a clause resolution automaton. For a predicate p/n an automaton is built incrementally starting with the skeleton $p(X_1, \ldots, X_n)$ for the root state s_1 and the set of clauses defining p/n as its compatible set. If p/n is a tabled predicate, we can choose the label of the start state to be *Tabled*, in which case no other state in the automaton need be tabled. If we label the start state as *Nontabled*, then some descendent states in each of the root-to-leaf path in the automaton need to be labeled as *Tabled*. From a given state s we "expand" the automaton as follows. We first choose a position π_s from the fringe of its skeleton S_s. We then partition the compatible set C_s into sets $C_{d_1}, \ldots, C_{d_k}$ such that in each C_{d_i}, all clause heads specify the same operation of unifying γ_i at π_s[4].

For example, at state s_2 in Figure 2, the compatible set, containing all three clauses, is partitioned into a set containing clauses 1 and 2, which share the operation of unifying position 1.1 in the goal with $+$, and a set consisting of clause 3, which has the operation unifying position 1.1 with a. We create new states $d_1, \ldots, d_k$ such that for each state d_i, C_{d_i} is its compatible set, $S_{d_i} = extend(S_s, \pi_s, \gamma_i)$ is its skeleton, and the edge (s, d_i) is the transition into d_i, labeled with γ_i. The process of expanding the automaton is repeated on all states that have non-empty fringes. The CRA in Figure 2 is an example of applying this construction process to the program in Figure 1. Tabled states in the CRA are shaded (*e.g.*, s_3). Observe that our formalism allows us to model selective tabling of clauses in a predicate. For example, only the first two clauses of predicate p are tabled in Figure 2. It is quite straightforward to transform a CRA into a tabled logic program (see Fig. 1b). The details are omitted.

Operation Unification of a goal with the clause heads of program clauses begins at the root of the automaton. From each state, a transition is made to the next state by performing the specified unification operation on the partially unified goal. If more than one transition is possible, the first such transition is made, and

[4] Variable symbols in distinct clauses are assumed to be distinct

the remaining transitions are marked as pending. When a leaf is reached, the body of the corresponding program clause is invoked.

With each tabled state we associate a *call table* that records all goals encountered by this state. With each goal in the call table, we associate an *answer table* that holds all the answer substitutions computed for the goal. When a tabled state is reached, the partially unified goal is checked against the call table associated with the state. If a variant of the goal[5] is not found in the table (*i.e.*, the goal had not been encountered before), an entry for this goal is made in the table. As the answer substitutions for the goal are computed, they are entered into the answer table associated with the goal. If a variant of the goal is found in the call table, the answer substitutions are computed by backtracking through the entries in the return table. Note that the constant portions of the goal that have been seen during unification need not be entered in the call table. Similarly, they need not be entered in the goal's answer table, as all the answers share this common portion. Whenever failure occurs, either within the automaton (due to failure of a unification operation), or during execution of a clause body, the automaton backtracks to the nearest state with pending transitions. The process then continues with the next pending transition. The automaton fails on backtracking when there are no states with pending transitions. This informal description of the automaton's operation can be formalized, and its soundness and completeness can be readily established; these are routine and omitted.

Recall that while constructing a CRA, the compatible set of a state is partitioned into a collection of sets, each of which in turn is the compatible set for one of the destination states. The implication of such set partitioning is that clauses selected for resolution need not follow Prolog's textual top-to-bottom order. To preserve this sequential order, each state in the automaton must consider its compatible clauses as a *sequence* and partition this sequence into a collection of subsequences. For this we introduce the following concept of *sequential clause resolution automaton* (SCRA).

Definition 2.4 (Sequential Clause Resolution Automaton) A *sequential clause resolution automaton* (SCRA) for a sequence $\langle c_1, c_2, \ldots, c_n \rangle$ of clauses is a CRA A such that, in a left-to-right preorder traversal of A, leaf i is visited before leaf $i+1$, for $1 \leq i < n$.

An SCRA faithfully models Prolog-style selection of program clauses for resolution. (See appendix for an example.)

3 Optimization Criteria

Recall that the elementary operations (and thus, the costs) associated with clause resolution fall into three categories:

- **Unification**: Unifying a goal with the heads of program clauses and answers;
- **Call tabling**: Checking for the presence of a goal in the call table and, if necessary, inserting it;
- **Answer tabling**: Inserting the answers for a goal into its answer table.

Each elementary operation is assumed to have unit cost, and the cost of resolving a goal is the total number of elementary operations performed in the three categories. Note that the unification operations in answer clause resolution modeled by a CRA are limited to those common to program clauses and answers. For simplicity, we account only for unification operations in program clause resolution. However, we can use more refined cost measures that account for unification in answer clause resolution and variations in the costs of different elementary operations, without affecting the results presented in this paper.

For a given goal and CRA, the cost of resolving the goal via the CRA is accounted as follows. At any state, if any outgoing transitions are possible, unification cost is proportional to the number of such transitions. Otherwise, the cost is 1 (for failure). For example, at the root state of the CRA in Figure 3a, two unification operations are performed for the goal $p(b, c, d, b)$. At a tabled state, call tabling cost is proportional to the number of symbols that must be entered into the table. Any subterms in the goal that were unified with constants along the path leading to this state need not be entered into the call table. Thus, in Datalog, for

[5] A term is a variant of another if they are identical up to variable renaming.

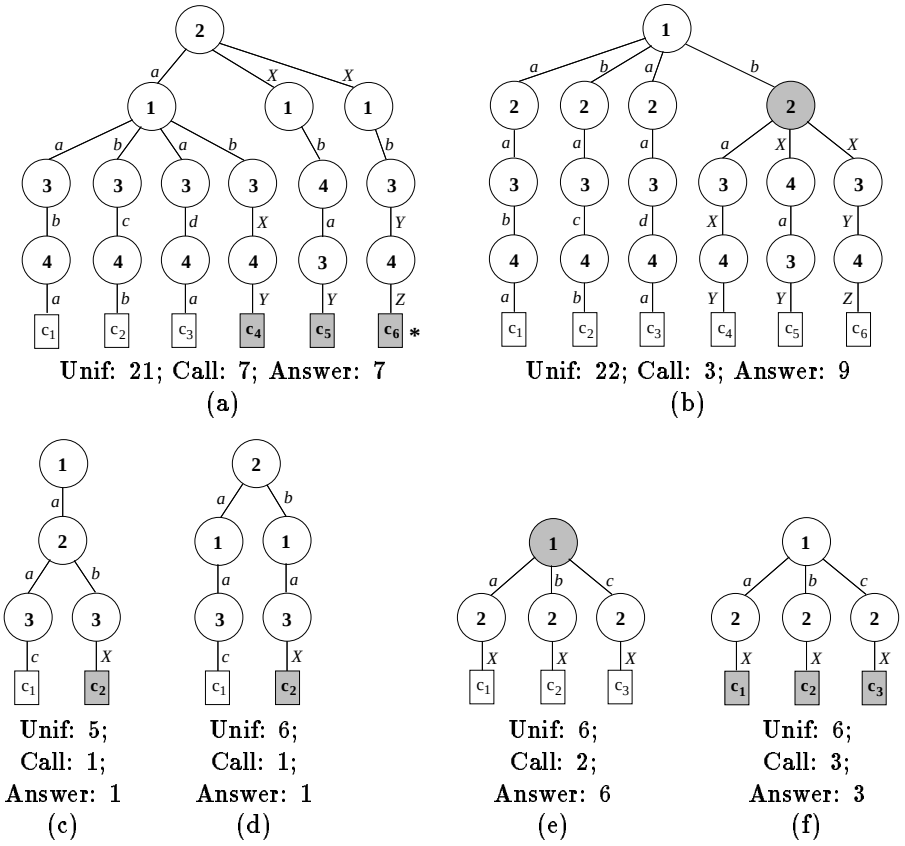


Figure 3: CRA's demonstrating dependence of performance on different operations

example, the number of call tabling operations at a tabled state can be counted as the arity of the predicate, less the number of constants along the path leading to that state. For example, the number of call tabling operations performed at the state marked $*$ in Figure 3a is 3 for goal $p(b, a, c, b)$. Answer tabling costs are counted in the same way, but observe that the operations must be performed for every answer derived.

The number of operations performed depends on both the goal and the CRA with which the operations are performed. For example, the resolution of the goal $p(b, a, c, b)$ via the CRA in Figure 3a requires 14 unification operations, 5 call tabling operations, and 5 answer tabling operations, while its resolution via the CRA in Figure 3b requires 13 unification, 3 call, and 6 answer operations. On the other hand, goal $p(b, c, d, b)$ requires 7 unification, 3 call, and 3 answer operations for the first CRA, and 8 unification, 3 call, and 3 answer operations for the second. Thus the relative costs of two CRA's vary with the goal. While we may have certain information about goals (such as modes) at compile time, the goals themselves are unknown in general. Hence, we choose the *worst case* performance of a CRA as the measure of its cost. An optimal CRA, then, is one that minimizes the worst-case cost.

The overall worst-case cost of a CRA is determined by both its structure (the arrangement of states and transitions) and the choice of tabled states. Unification cost is determined by structure alone, while call and answer tabling costs are determined by both the structure and tabled states, implying that construction of an optimal CRA must take into account simultaneously all three costs. A CRA constructed to minimize one or two of the costs does not always minimize the total cost, as illustrated by the CRA's in Figure 3.

The CRA's in the figure were constructed for Datalog predicates in the absence of mode information. The worst-case cost of such CRA's corresponds to *open* goals, *i.e.,* goals whose arguments are all variables, since (1) an open goal traverses the entire CRA, and (2) for Datalog, tabling an open goal is as expensive as tabling any other goal. The CRA in Figure 3a was constructed to minimize the number of unification operations (21), and then tabled states were selected to minimize the number of call (7) and answer (7) operations for that automaton. However, the CRA in Figure 3b has a lower total number of operations (34) in the worst

case, despite having a greater number of unification operations (22). Hence, first minimizing unification cost does not guarantee an optimal CRA. As a consequence, minimizing only unification and answer costs does not guarantee an optimal CRA, since answer costs are always minimal for any CRA with leaves as tabled states. Similarly, Figures 3c and 3d show that minimizing call costs or call and answer costs alone does not guarantee optimality. Finally, Figures 3e and 3f demonstrate that minimizing unification and call costs together is insufficient. Thus, we can conclude

Theorem 3.1 *Any method for constructing optimal CRA's must simultaneously account for unification, call tabling, and answer tabling costs.*

4 Complexity of Optimizations

The complexity of compile-time optimizations for clause resolution depends on a number of factors: (1) whether a predicate is tabled or nontabled, (2) whether a predicate is Datalog or general, (3) the presence of mode information, and (4) the need to preserve clause order (sequentiality). The special case of nontabled predicates in the absence of mode information was addressed in [7]. Of the remaining cases, Section 4.1 deals with tabled Datalog programs in the absence of mode information. Section 4.2 covers both nontabled and tabled Datalog programs in the presence of mode information. Finally, tabled general programs are addressed in Section 4.3.

4.1 Tabled Datalog without mode information

Recall that, in the absence of mode information, open goals result in the worst-case cost of CRA's for Datalog predicates. When textual clause order must be preserved, we exploit this property to construct an optimal SCRA in polynomial time. On the other hand, when textual order need not be preserved, we show that constructing an optimal CRA becomes NP-complete.

Sequential In an SCRA the compatible set associated with any state is a *sequence* of clauses, and the transitions from a state partition its compatible set into subsequences. Consequently, any subautomaton of an optimal SCRA must be optimal for the subsequence of clauses in the compatible set associated with the root of the subautomaton. Furthermore, for an optimal SCRA it can easily be shown that no two adjacent transitions from a state can have the same label. These properties, along with the fact that the worst-case unification, call, and answer costs result from open goals, enable us to combine the worst-case costs of subautomata to compute the overall worst-case cost. This leads to a recurrence for the worst-case total cost of an optimal SCRA that can be solved in polynomial time using dynamic programming.

To construct an optimal SCRA for a sequence C of clauses and initial skeleton S, we consider, for each position in the fringe of S, the least cost SCRA that can be constructed by first inspecting that position. The transitions out of a state and their order are uniquely determined by the position chosen. Recall that the transitions partition of the compatible set into subsequences. We construct an optimal subautomaton for each subsequence and the corresponding extended skeletons. If the root state is tabled, then we need only consider subautomata that have no tabled states. Otherwise, we need to consider for each subautomaton, one that is tabled at its root, and one that is tabled below the root, retaining the one with lower cost. These optimal subautomata can then be combined into an optimal automaton rooted at the given position.

Given n clauses with at most m symbols in each clause head, an optimal SCRA can be constructed in $O(n^2m(n+m))$ time. Therefore,

Theorem 4.1 *An optimal SCRA can be constructed in polynomial time.*

The details of the dynamic programming solution and the analysis of its complexity are given in the appendix.

Note that computing the worst-case cost of an SCRA depends on knowing the number of answers for each clause. If the number of answers can be estimated at compile time, the dynamic programming algorithm can be used to build an SCRA that is optimal for that measure of answers. When answer costs are unknown

and cannot be estimated reliably at compile time, we need heuristics to build a more optimal SCRA. Below, we propose two heuristics and show bounds on their optimality.

Heuristic 1 *Ignore call tabling costs.* If call costs are ignored, the optimal automaton is the SCRA optimized for unification cost alone, with leaves as tabled states. Note that this automaton, denoted $\mathcal{A}_1$, can be constructed independent of the number of answers per clause, although its cost, denoted (u_1, c_1, a_1), may not be known.

Heuristic 2 *Ignore answer tabling costs.* Clearly, an automaton that minimizes the sum of unification and call costs can be built in polynomial time. Let the cost of this automaton, $\mathcal{A}_2$, be (u_2, c_2, a_2).

Let the cost of the optimal automaton, $\mathcal{A}_{opt}$ be $(u_{opt}, c_{opt}, a_{opt})$. Note that since $\mathcal{A}_1$ is the optimal automaton for unification cost alone, $u_1 \leq u_2$ and $u_1 \leq u_{opt}$. Furthermore, $a_1 \leq a_2$ and $a_1 \leq a_{opt}$. Also, $u_2 + c_2 \leq u_1 + c_1$ and $u_2 + c_2 \leq u_{opt} + c_{opt}$.

Bound on Heuristic 1 The difference between the costs of $\mathcal{A}_1$ and $\mathcal{A}_{opt}$ is at most $(u_1 + c_1) - (u_2 + c_2)$, since $(u_1+c_1+a_1)-(u_{opt}+c_{opt}+a_{opt}) \leq (u_1+c_1+a_1)-(u_2+c_2+a_{opt}) \leq (u_1+c_1+a_1)-(u_2+c_2+a_1)$.

Bound on Heuristic 2 The difference between the cost of $\mathcal{A}_2$ and the cost of $\mathcal{A}_{opt}$ is at most $a_2 - a_1$, since $(u_2 + c_2 + a_2) - (u_{opt} + c_{opt} + a_{opt}) \leq (u_2 + c_2 + a_2) - (u_2 + c_2 + a_1)$.

Nonsequential In general, a nonsequential CRA does not have the property that transitions from a state partition its compatible set into subsequences. Rather, the transitions may partition the compatible set into *subsets,* and the construction of an optimal nonsequential CRA may require enumerating a large number of such subsets of clauses. In fact, we show

Theorem 4.2 *The problem of optimal nonsequential CRA construction is NP-complete.*

The proof is by reduction from the *set cover* problem and appears in the appendix.

4.2 Datalog programs with mode information

When mode information becomes available, the worst-case cost of a CRA no longer corresponds to an open goal, and in general, there is no goal whose resolution traverses the entire CRA. Furthermore, a goal that yields the worst-case cost on one subautomaton may not be the same goal that yields the worst-case cost on another. Hence, in the presence of mode information, we can no longer combine locally optimal solutions to arrive at an optimal CRA as we did in the absence of mode information. In fact, even computing the worst-case of a *given* CRA may require considering a large number of goals to identify one that yields the highest cost. We show that this problem is NP-complete. We also generalize this result to show that the decision problem corresponding to computing the cost of an *optimal* CRA in the presence of mode information is NP-hard. These results hold for both tabled and nontabled CRA's, sequential or nonsequential. We first establish these results for nontabled automata, addressing a problem left open in [7]. We then generalize the results to tabled CRA's.

Sequential nontabled Recall that, for nontabled automata, the only cost is due to unification. Let $unifcost(\mathcal{A}, g)$ denote the cost of unifying goal g via automaton $\mathcal{A}$. The problem of estimating the worst-case unification cost of a given automaton in the presence of mode information is formulated as the following decision problem:

Worst-case cost of given nontabled automaton (NT-Given): *Given a sequence H of Datalog clause heads, a nontabled CRA $\mathcal{A}_H$ for H, and integer k, is there a ground goal g such that $unifcost(\mathcal{A}_H, g) \geq k$?*

Computing the worst-case cost of an optimal nontabled automaton is formulated as

Worst-case cost of optimal nontabled automaton (NT-Opt): *Given a sequence H of Datalog clause heads and an integer k, for every nontabled CRA $\mathcal{A}_H$ for H is there a ground goal g such that $unifcost(\mathcal{A}_H, g) \geq k$?*

By reduction from 3-satisfiability (3-SAT) we show

Theorem 4.3 *NT-Given is NP-complete.*

Theorem 4.4 *NT-Opt is NP-hard.*

The proofs of these theorems are given in the appendix. Note that it is unlikely that NT-Opt $\in$ NP, since the problem of verifying the cost of an automaton (NT-Given) is itself NP-complete. Thus, while the exact characterization of the difficulty of constructing an optimal nontabled CRA in the presence of mode information remains an open problem, it is unlikely that there is any efficient method for building an optimal CRA that relies on computing worst-case unification costs.

Sequential tabled Let $totalcost(\mathcal{A}, g)$ represent the total cost of unification, call and answer table costs for goal g via automaton $\mathcal{A}$. We formulate the problems of computing the worst-case total cost of a given automaton and that of an optimal automaton as follows:

Worst-case cost of given tabled automaton (T-Given): *Given a sequence H of Datalog clauses, a tabled CRA $\mathcal{A}_H$ for H, and integer k, is there a ground goal g such that $totalcost(\mathcal{A}_H, g) \geq k$?*

Worst-case cost of optimal tabled automaton (T-Opt): *Given a sequence H of Datalog clauses and integer k, for every tabled CRA $\mathcal{A}_H$ for H is there a ground goal g such that $totalcost(\mathcal{A}_H, g) \geq k$?*

We show by reduction from 3-SAT that

Theorem 4.5 *T-Given is NP-complete.*

Theorem 4.6 *T-Opt is NP-hard.*

Note that, since minimizing unification cost alone does not in general minimize total cost (Theorem 3.1), Theorems 4.5 and 4.6 are not direct consequences of Theorems 4.3 and 4.4.

Nonsequential Theorems 4.3 and 4.5 can be immediately generalized to nonsequential automata, since sequentiality is a special case of nonsequentiality. Theorems 4.4 and 4.6 also apply to nonsequential automata, although the proofs do not immediately generalize. Instead, these results follow from Theorem 4.2 (and its counterpart for nontabled automata in [7]), which represents a restricted case of mode information (in which all arguments have unknown modes).

4.3 General (non-Datalog) programs

When programs are not restricted to Datalog, the size of a call – and hence, the cost of tabling a call – is in general unbounded. This makes even a reasonable formulation of CRA optimization difficult. The worst-case cost criterion used for Datalog programs is not applicable for general programs, since the worst-case cost is unbounded in general. A modified problem we can pose is to find an automaton that minimizes the worst-case cost for calls of a *given size*. However, one automaton may have better worst-case performance over one range of call sizes, and worse in another range, compared to another automaton. Thus, comparing asymptotic performance of two automata appears inappropriate.

Note, however, that unification cost for linear clause heads is independent of goal size. Thus, the results of Section 4.2 for nontabled Datalog programs in the presence of mode information apply to *nontabled* general programs as well (by restriction). Results for nontabled programs in the absence of mode information, both sequential (polynomial) and non-sequential (NP-complete), are reported in [7].

5 Heuristics and Experimental Results

We now describe heuristics to build a CRA when the construction of optimal CRA's is difficult. The heuristic is used to build CRA's based on the construction process discussed in section 2 (page 4), In that construction, the automaton is expanded from a given state s by choosing a position π_s from the fringe of its skeleton S_s and partitioning the compatible set C_s into sets $C_{d_1}, C_{d_2}, \ldots, C_{d_k}$ such that in each C_{d_i} all clause heads specify the same unification operation at π_s. We define a partition operation that ensures that all common unification operations that *can* be shared *will be* shared. For example, consider the clause set $C = \langle p(a,b), p(b,c), p(b,X), p(a,c)\rangle$. When clause order need not be preserved, choosing position 1 partitions C into $\{p(a,b), p(a,c)\}$ and $\{p(b,c), p(b,X)\}$. When clause order needs to be preserved, and position 1 is *not* known to be ground in all goals, C is partitioned into $\langle p(a,b)\rangle$, $\langle p(b,c), p(b,X)\rangle$ and $\langle p(a,c)\rangle$. On the other hand, if position 1 *is* known to be ground in all goals, C is partitioned into $\langle p(a,b), p(a,c)\rangle$ and $\langle p(b,c), p(b,X)\rangle$. The main function of the heuristic now is choosing the position π_s at each state. This choice is influenced by several considerations, as described below.

All optimal automata have the following property: in any traversal of the automaton, a state where no goal backtracks is always visited before a state where backtracking is possible. Hence, among all positions in the fringe of S_s, we first choose positions that ensure that no goal backtracks at state s. Observe that backtracking cannot occur at any state that has only one outgoing transition. Furthermore, if the symbol at π_s is nonvariable in all goals, then no backtracking is possible at s whenever (1) no outgoing transition from s is labeled by a variable and (2) no two outgoing transitions have the same label (as can happen in SCRA's).

When all positions in the fringe can lead to backtracking at state s, we choose positions that allow us to defer tabling operations to the descendent states without increasing tabling costs. Note that deferring tabling operations to descendent states often decreases call and answer tabling costs. In fact, it never increases answer costs, although it can increase call costs. For example, consider the set of clauses $\{p(a,b,X), p(a,Y,c), p(b,Z,c)\}$. On choosing position 1 the clauses are partitioned into $\{p(a,b,X), p(a,Y,c)\}$ and $\{p(b,Z,c)\}$. If tabling is deferred to the descendent states, the goal $p(U, f(f(...)), c)$ results in entering the subterm $f(f(...))$ in two call tables, thereby increasing call costs. In the following, we describe a condition under which tabling operations can be deferred without increasing tabling costs.

This condition is based on the following notion of *variable overlaps*. Two terms are said to have variable overlaps at π_s whenever one of them has a variable at π_s and the other has a nonground subterm rooted at that position. For instance, in the above example the only variable overlap among clauses $p(a,b,X)$, $p(a,Y,c)$ and $p(b,Z,c)$ occurs at position 2, between the last two clauses. Let π_s partition the compatible set into sets $C_{d_1}, C_{d_2}, \ldots, C_{d_k}$. It can be readily shown that if all variable overlaps occur only among clauses in exactly one of the sets, say C_{d_i}, then the choice of tabled states can be deferred to the descendent states without increasing call costs. For instance, in the example above choosing position 3 partitions the clauses into $\{p(a,b,X)\}$ and $\{p(a,Y,c), p(b,Z,c)\}$, and all variable overlaps occur in the second set. On continuing the construction process, tabling can be deferred to the leaf state of the subautomaton for the first clause set. On the other hand tabling can be deferred only to the root of the subautomaton. The resulting automaton has lower cost than any whose root is tabled.

For a given predicate, an automaton in which unifications are not shared and tabling is performed at the root represents the operation of traditional tabling systems. Let the worst case cost of such an automaton be $cost_d$ and that of the CRA built by the heuristic for the same predicate be $cost_h$. Then,

Theorem 5.1 (Safety of Optimization) $cost_h \leq cost_d$.

5.1 Performance

The effectiveness of clause resolution factoring is indicated by its performance in the XSB system. Table 1 shows the time and space savings that result from clause resolution factoring on a representative sample of tabled programs[6]. The benchmarks used include a (left-recursive) transitive closure (the *ancestor* relation) on two different EDB's (I and II), a join of two relations, a parser for an LR(1) grammar, and a parser for

[6] The times in the table were averaged over multiple iterations.

Benchmark	Time (milliseconds)		Speedup	Table space (bytes)		Savings
	Original	Factored		Original	Factored	
Transitive Closure (I)	60	49	1.22	55.5 K	36.5 K	34%
Transitive Closure (II)	190	110	1.73	344140	52	99%
Join	181	145	1.25	68	52	24%
LR(1) (a)	4600	3170	1.45	3799 K	3330 K	12%
LR(1) (b)	1400	1200	1.17	5705 K	2897 K	49%
LR(1)(det) (a)	4750	3581	1.33	3799 K	3399 K	11%
LR(1)(det) (b)	921	330	2.79	5691 K	1449 K	75%
Ambiguous	28	9	3.10	138 K	77 K	44%

Table 1: Experimental results for clause resolution factoring in XSB

an ambiguous grammar[7]. The Transitive Closure and LR(1) benchmarks require tabled evaluation to ensure termination, whereas the Join and Ambiguous benchmarks need tabling for efficiency.

Observe that factoring has reduced (substantially in many cases) the amount of information that must be tabled, resulting not only in space savings but also savings in execution time, since the number of table insert operations is reduced. Even in the LR(1) benchmark, where the clause heads contained little factorable information, the selective tabling afforded by our CRA model resulted in time and space savings. In order to increase the amount of factorable information, we enlarged the clause heads of the original LR(1) program, yielding the LR(1)(det) program, by incorporating *success* and *context* conditions, using the determinacy analysis technique of [8]. On query (b) this additional head information alone resulted in a speedup over LR(1), and after factoring, the speedup and space savings were still greater. On query (a) there was a slight slowdown on LR(1)(det) as compared with LR(1). However, LR(1)(det) has the potential for even more optimization (*e.g.,* from specialization), indicating the possible benefits of combining clause resolution factoring with other optimization techniques.

6 Discussion

In order to make optimal use of CRA's, a tabling system must in fact treat unification and backtracking uniformly whether it occurs in program or answer clauses. We note that it is possible for engines with different evaluation strategies to have this property. As an instance, the SLG-WAM, the engine underlying the XSB system, can compile predicates so that they are evaluated using a mostly breadth-first strategy that approximates semi-naive evaluation. This strategy will also be able to use CRA's.

It is apparent that most optimization methods for deductive databases address different problems entirely. As one example, supplementary magic sets [2] optimizes the body of clauses, folding EDB calls into new tabled predicates to minimize redundant subcomputations. As another, NRSU-factoring [10] can project instantiated variables out of a clause head, but uses mode information, and information about query reachability to do so. We should mention that both of these techniques have proven useful for the SLG-WAM.

Techniques used in formulating and optimizing CRA's stem from optimization traditions outside of the deductive database community: those of term indexing and partial evaluation. Given a non-tabled predicate together with information that all arguments of input will be fully ground, CRA's can be seen as a non-deterministic discrimination net [1]. Given a non-tabled predicate and no mode information, CRA's reduce to Unification Factoring Automata [7]. CRA's thus generalize both of these techniques.

Optimizing clause resolution using CRA's opens several avenues for further research. One is the the use termination analysis to determine when a subset of clauses of a predicate may need to be tabled, and to incorporate this information into constructing a CRA. We believe that this technique will be especially useful for modules of a program which require tabling to ensure termination, but which are not particularly data-oriented. Another is the use CRA's as a building block for specialization of literals in the bodies of clauses.

[7] All benchmarks were run on a Sun SparcStation 20, using SunOS 5.3 and XSB version 1.4.1. Benchmark programs and queries appear in the appendix.

References

[1] Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In *Theory and Practice of Software Development*, number 668 in LNCS, pages 61–74. Springer Verlag, April 1993.

[2] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3):255–299, 1991.

[3] R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *International Logic Programming Symposium*, 1993.

[4] W. Chen, T. Swift, and D. S. Warren. Efficient implementation of general logical queries. *Journal of Logic Programming*. To Appear. Also available as a technical report, Dept. of Computer Science, SUNY at Stony Brook.

[5] W. Chen and D. S. Warren. Computation of stable models and its integration with logical query evaluation. *IEEE Transactions on Knowledge and Data Engineering*. To Appear. Also available as a technical report, Dept. of Computer Science, SUNY at Stony Brook.

[6] W. Chen and D. S. Warren. Query evaluation under the well-founded semantics. In *ACM Symposium on Principles of Database Systems*, 1993.

[7] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *ACM Symposium on Principles of Programming Languages*, January 1995. To Appear. Also available as a technical report, Dept. of Computer Science, SUNY at Stony Brook.

[8] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting determinacy in logic programs. In *International Conference on Logic Programming*, pages 424–438. MIT Press, 1993.

[9] M. Garey and D. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completness*. W.H. Freeman and Company, 1979.

[10] J. Naughton, R. Ramakrishnan, Y. Sagiv, and J. Ullman. Argument reduction through factoring. In *International Conference on Very Large Data Bases*, pages 173–182, 1989.

[11] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programming. In *Joint International Conference/Symposium on Logic Programming*, pages 140–159. MIT Press, 1988.

[12] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *ACM SIGMOD International Conference on the Management of Data*, 1994.

[13] H. Seki. On the power of Alexander templates. In *ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.

[14] T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *International Logic Programming Symposium*, 1994. To Appear. Also available as a technical report, Dept. of Computer Science, SUNY at Stony Brook.

[15] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98, 1986.

[16] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3), 1991.

[17] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

A Appendix

This appendix contains only supplementary reference material for the convenience of the referees.

A.1 Non-sequential CRA example

The (nontabled) CRA's in Figure 4 illustrate the difference between non-sequential (b) and sequential (c) automata. While the non-sequential automata is able to share the operation unifying argument 1 with a, it does not preserve clause order; the sequential CRA preserves the clause order.

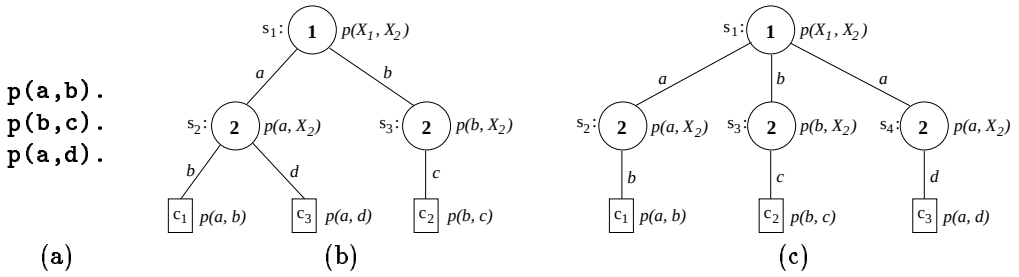


Figure 4: Non-sequential (b) and sequential (c) CRA's for predicate (a)

A.2 Optimal SCRA construction without mode information

Dynamic programming The following three readily established properties of an optimal SCRA are used in its construction.

Property 1 Each subautomaton is optimal for the clauses in the compatible set and skeleton associated with the root of the sub-automaton.

Property 2 Any unification that *can* be shared *will* be shared. That is, in an optimal SCRA, no two adjacent transitions from any given state have the same label.

Property 3 Transitions from a state partition its compatible set of clauses into *subsequences*.

A consequence of the three properties is that worst-case call table and answer costs of an automaton can be estimated in polynomial time. Using the above properties, we formulate a recurrence on the cost (and hence construction) of an optimal SCRA, and solve the recurrence in polynomial time using dynamic programming.

To construct an optimal automaton for a sequence of clauses C and initial skeleton S, we consider, for each position in the fringe of S, the least cost automaton that can be constructed by first inspecting that position. If the root state of this automaton is a tabled state, then all states in the sub-automata are nontabled. On the other hand, if the root of this automaton is not a tabled state, then each of the sub-automata have at least one state that is tabled.

From Properties 2 and 3 it follows that the transitions out of a state and their order are uniquely determined by the position chosen. Since these transitions represent the partition of the compatible set of clauses, to construct an optimal automaton for a given start position, we first compute this partition. We construct two automata for each sequence of clauses in the partition and the corresponding extended skeletons – the optimal automaton with tabled states and the optimal automaton with no tabled states. From Property 1 it follows that these optimal sub-automata can be combined to produce an optimal automaton for the selected start position. The lowest cost automata among the automata constructed for each position is an optimal SCRA for C and S. The recursive construction sketched above lends itself to a dynamic programming solution, and is formalized below:

We use a function *part* to partition the clause sequence C into the minimum number of subsequences that share unification operations at this position.

Definition A.1 (Partition) *Given a sequence $\langle t_1, \ldots, t_n \rangle$ of clause heads corresponding to the sequence of clauses C, a pair of integers (i, i'), $1 \leq i \leq i' \leq n$, and a position π, the* partition of C by π, *denoted $part(i, i', \pi)$, is the set of triples (α, j, j'), $i \leq j \leq j' \leq i'$, such that j and j' are the end points of a maximal subsequence of clause heads in $\langle t_i, \ldots, t_{i'} \rangle$ having symbol α at position π. That is, $(\alpha, j, j') \in part(i, i', \pi)$ iff*

1. *for all $j \leq k \leq j'$, $t_k|_\pi = \alpha$,*
2. *either $j = i$ or $t_{j-1}|_\pi \neq \alpha$, and*
3. *either $j' = i'$ or $t_{j'}|_\pi \neq \alpha$.*

Each triple (α, j, j') computed by *part* represents a transition associated with the unification operation involving π and α. For example, the partition of sequence $\langle p(a,b), p(b,c), p(a,d) \rangle$ at position 1 is $\{\langle p(a,b) \rangle$, $\langle p(b,c) \rangle$, $\langle p(a,d) \rangle\}$. The skeleton of the next state resulting from the transition is $extend(S, \pi, \alpha)$. The compatible clauses of this state form the subsequence $\langle C_j, \ldots, C_{j'} \rangle$.

For each subsequence in a partition there are one or more fringe positions in the skeleton with a symbol common to all clause heads in the subsequence. From Property 2 it follows that the unification operations at these positions will be shared by all clause heads in the subsequence. We use the function *common* to identify such operations and extend the skeleton to record their effect:

Definition A.2 (Common) *Given a skeleton S, a sequence $\langle t_1, \ldots, t_n \rangle$ of clause heads, and a pair of integers (i, i'), $1 \leq i \leq i' \leq n$, $common(S, i, i')$ is a pair (E, S'), where E represents the set of unification operations common to $\{t_i, \ldots, t_{i'}\}$, and S' is the extended skeleton:*

$$common(S, i, i') = \begin{cases} (E \cup \{(\pi, \alpha)\}, S'), & \text{if } \exists \pi \in fringe(S) \text{ such that } t_j|_\pi = \alpha,\ i \leq j \leq i', \\ & \text{where } (E, S') = common(extend(S, \pi, \alpha), i, i') \\ (\{\}, S), \text{ otherwise} & \end{cases}$$

For example, given skeleton $S = p(X_1, X_2, X_3)$ and sequence $\langle p(a,b,c), p(a,b,d), p(a,b,e) \rangle$, $common(S, 1, 3) = (\{(1,a), (2,b)\}, p(a,b,X_3))$.

The cost of a optimal tabled and nontabled automata for the sequence of clauses $[i \ldots j]$ and the skeleton S are denoted by $Cost(\underline{Tabled}, i, j, S)$ and $Cost(\underline{Nontabled}, i, j, S)$ respectively. The costs of optimal tabled automata with start position $\pi \in S$, with the start state tabled and start state not tabled are denoted by $cost(i, j, S, \texttt{at}\ \pi)$, and $cost(i, j, S, \texttt{below}\ \pi)$, respectively. For the sake of uniformity, the cost of an optimal nontabled automaton with start position π is denoted by $cost(i, j, S, \texttt{above}\ \pi)$. The cost of an optimal automaton for a given sequence of clauses and skeleton are given by the following equations.

$$Cost(\underline{Tabled}, i, j, S) = \min_{\pi \in fringe(S)} (\min(cost(i, j, S, \texttt{at}\ \pi), cost(i, j, S, \texttt{below}\ \pi))) \quad (1)$$

$$Cost(\underline{Nontabled}, i, j, S) = \min_{\pi \in fringe(S)} (cost(i, j, S, \texttt{above}\ \pi)) \quad (2)$$

The cost of an automaton, $Cost(i, j, S)$, is a triple $(Cost_{unify}, Cost_{call}, Cost_{answer})$ where $Cost_{unify}$, $Cost_{call}$ and $Cost_{answer}$ denote the unification, call and answer costs of the automaton. The cost of the automata with given start states are denoted by the triple $(cost_{unify}, cost_{call}, cost_{answer})$. We describe the computation of each component of the cost separately, starting with the unification cost.

The worst unification case cost of an SCRA is when all transitions are taken. Note that unification cost is independent of which states of the automaton are tabled. Assuming that all elementary unification operations have unit cost, the unification cost of an optimal SCRA for clause sequence $\langle C_i, \ldots, C_{i'} \rangle$ and skeleton S is expressed by the following recurrence:

$$\begin{aligned} cost_{unify}(i, i', S, \texttt{above}\ \pi) &= \\ cost_{unify}(i, i', S, \texttt{at}\ \pi) &= \\ cost_{unify}(i, i', S, \texttt{below}\ \pi) &= \sum_{\substack{(\alpha, j, j') \in \\ part(i, i', \pi)}} Cost_{unify}(\_, j, j', S') + |E|, \\ & \qquad \text{where } (E, S') = common(S, j, j') \end{aligned} \quad (3)$$

Recall that constants seen on reaching a state are not entered in the call table. We use $var\_pos(S)$ to denote the set of those positions in the skeleton S that are not constants, *i.e.*, $\pi \in var\_pos(S)$ iff $S|_\pi \notin \mathcal{F}$. Assuming that cost of creating a table entry (call or answer table) is the same for a constant and a variable, the table cost is given by

$$cost_{call}(i, i', S, \texttt{above } \pi) = 0 \tag{4}$$

$$cost_{call}(i, i', S, \texttt{at } \pi) = |var\_pos(S)| \tag{5}$$

$$cost_{call}(i, i', S, \texttt{below } \pi) = \sum_{\substack{(\alpha,j,j')\in \\ part(i,i',\pi)}} Cost_{call}(\underline{Tabled}, j, j', S') \quad \text{where } (E, S') = common(S, j, j') \tag{6}$$

The cost of tabling *at* π is clearly the number of positions in the clause heads yet to be seen, and hence is the size of the fringe. When the table points are *below* π, however, we need to table not only the fringe positions at the tabling points, but also the variable positions in the path from π to each of the tabling points.

Using ρ_k to denote the number of answers for the the clause $C_k \in \langle C_i, \ldots, C_{i'} \rangle$, the answer cost is given by:

$$cost_{answer}(i, i', S, \texttt{above } \pi) = 0 \tag{7}$$

$$cost_{answer}(i, i', S, \texttt{at } \pi) = |var\_pos(S)| \cdot \sum_{k=i}^{i'} \rho_k \tag{8}$$

$$cost_{answer}(i, i', S, \texttt{below } \pi) = \sum_{\substack{(\alpha,j,j')\in \\ part(i,i',\pi)}} Cost_{answer}(\underline{Tabled}, j, j', S'), \quad \text{where } (E, S') = common(S, j, j') \tag{9}$$

The cost of answers at a tabling point is the number of entries in the table times the size of the table. The number of entries is simply the sum of the number of answers in each clause that shares this tabling point. The cost of answers at π, when the tabling points are below it, is the sum of answer costs associated with the tabling points.

Note that $|E|$ is the number of common unification operations for a subsequence in a partition. Also note that the recurrence assumes that subsequence $\langle t_i, \ldots, t_{i'} \rangle$ has no common part with respect to skeleton S. Thus, the cost of an optimal automaton for a tabled predicate p/m consisting of n clauses is given by $|E| + Cost(\underline{Tabled}, 1, n, S)$, where $(E, S) = common(p(X_1, \ldots, X_m), 1, n)$.

Using the above recurrence, it is straightforward to construct an optimal SCRA based on dynamic programming, where the optimal tabled and nontabled automata for each subsequence of clauses are recorded in a table. Note that, since $\langle t_i, \ldots, t_{i'} \rangle$ has no common part with respect to the skeleton, a skeleton S is uniquely determined, given i and i'. Hence, the cost recurrence has only two independent parameters, i and i', and the table used in the dynamic programming algorithm will be indexed by i and i'.

Given n clauses with at most m symbols in each clause head, the number of possible subsequences is $O(n^2)$, and hence, the number of table entries is $O(n^2)$. The number of partitioning positions considered for each entry is $O(m)$. For each position, *part* requires $O(n)$ time. Identification of common operations for all subsequences in a partition can be accomplished in $O(m)$ time via a precomputed matrix (that requires $O(mn)$ time and space). Thus, the time required to compute one entry in the table is $O(m(n+m))$, and the overall time needed to compute an optimal automaton is $O(n^2 m(n+m))$.

In the recurrence for the cost of an optimal SCRA, it is assumed that all elementary operations have unit cost, and that recording pending transitions has zero cost. The recurrence can be easily modified to accurately reflect the cost of each elementary operation and the cost of recording pending transitions. Furthermore, by suitably modifying *part* and *common*, as given in [7], the above recurrences can be used even in presence of nonlinear clause heads, to construct an optimal SCRA in polynomial time.

A.3 Nonsequential tabled programs without mode information

The problem of optimal nonsequential CRA construction (in the absence of mode information) is formulated as follows.

NSCRA Given a set of clauses H for a tabled predicate and an integer k, is there a nonsequential CRA $\mathcal{A}_H$ for H, such that, for any open goal g, $totalcost(\mathcal{A}_H, g) \leq k$.

Theorem A.3 *NSCRA is NP-complete.*

Proof: The proof is by reduction from *set cover* and generalizes the proof from [7], which applied only to nontabled automata. The minimum set cover problem can be stated as follows: Given a finite set $U = \{u_1, \ldots, u_n\}$, a collection $C = \{C_1, \ldots, C_m\}$ of subsets of U, and a positive integer $k \leq m$, do there exist k or fewer subsets in C whose union is S?

Let I be an instance of SC. We construct an instance I' of NSCRA consisting of $2n$ clauses of arity $2(n+m)^2 + m + 1$, as follows. Each clause head can be viewed as consisting of four segments: a "*Test*" segment, consisting of m arguments; a "*Blue*" segment, consisting of $(n+m)^2$ arguments; a "*Red*" segment, consisting of $(n+m)^2$ arguments; and the fourth segment consisting of one variable:

$$p(\underbrace{\tau_1, \tau_2, \cdots, \tau_m}_{Test}, \underbrace{\beta_1, \beta_2, \cdots, \beta_{(n+m)^2}}_{Blue}, \underbrace{\rho_1, \rho_2, \cdots, \rho_{(n+m)^2}}_{Red}, X)$$

For each element $u_i \in U$ two clauses are constructed: a *Red* clause and a *Blue* clause. In the *Red* clause, $\tau_j = 0$, for $1 \leq j \leq m$; $\beta_l = i$ and $\rho_l = 0$, for $1 \leq l \leq (n+m)^2$. The i^{th} *Red* clause head thus has the form

$$p(\underbrace{0, \cdots, 0}_{Test}, \underbrace{i, \cdots, i}_{Blue}, \underbrace{0, \cdots, 0}_{Red}, X)$$

In the *Blue* clause, $\tau_j = 1$ if $u_i \in C_j$, otherwise 0, for $1 \leq j \leq m$; $\beta_l = i$ and $\rho_l = 0$, for $1 \leq l \leq (n+m)^2$. The i^{th} *Blue* clause head thus has the form

$$p(\underbrace{\tau_1, \tau_2, \cdots, \tau_m}_{Test}, \underbrace{0, \cdots, 0}_{Blue}, \underbrace{i, \cdots, i}_{Red}, X)$$

Unifying with a symbol in the *Test* segment of a *Blue* clause head can be thought of testing for membership of an element of U in a subset.

Observe that the *Red* and *Test* segments in every *Red* clause are identical, and that each *Blue* segment is distinct. For a set consisting entirely of *Red* clauses, an optimal CRA must examine all positions in the *Red* and *Test* segments before examining any position in the *Blue* segment, since the first test in the *Blue* segment effectively partitions the set into individual clauses (see Figure 5a). The order of testing within the *Red* and *Test* segments is unimportant, as is the order of testing within the *Blue* segment. Tabling costs (both call and answer) are minimized by choosing the leaves as tabled states, since any CRA for *Red* clauses will have some long chains (sequences of states with only one outgoing transition), and selecting a common ancestor of these chains as a tabling point would greatly increase both call and answer costs.

Also observe that the *Blue* segment in every *Blue* clause is identical, and that each *Red* segment is distinct. An optimal CRA for a set of *Blue* clauses must examine all positions in the *Blue* segment before examining any position in the *Red* segment. In general, testing positions in the *Test* segment will incrementally partition the set. Thus, an optimal CRA for *Blue* clauses will typically have the form shown in Figure 5b. Again, tabling costs are minimized by choosing the leaves as tabled states.

The above observations lead to the following bounds on the worst-case total number of elementary operations (unification, call tabling, and answer tabling) in an optimal CRA for a *monochromatic* set of clauses.

Lemma A.3.1 *Given a set S consisting entirely of Red clauses, such that $|S| = n_\rho$, the worst-case unification, call tabling, and answer tabling costs in an optimal CRA for S are $(n_\rho + 1)(n+m)^2 + m + n_\rho$, n_ρ, and n_ρ, respectively.*

Lemma A.3.2 *Given a set S consisting entirely of Blue clauses, such that $|S| = n_\beta$, the worst-case unification cost in an optimal CRA for S is no more than $(n_\beta + 1)(n+m)^2 + (m+1)n_\beta$, and the worst-case call and answer tabling costs are each n_β.*

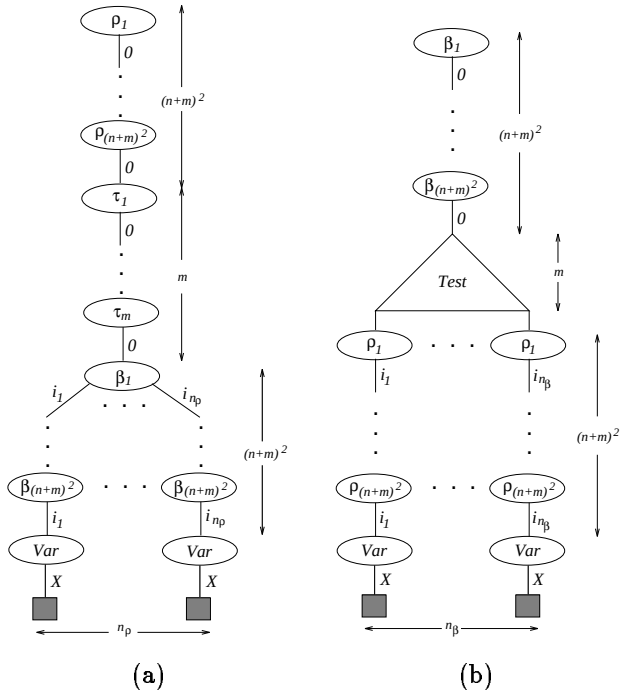


Figure 5: Optimal CRA's for sets of *Red* (a) and *Blue* (b) clauses

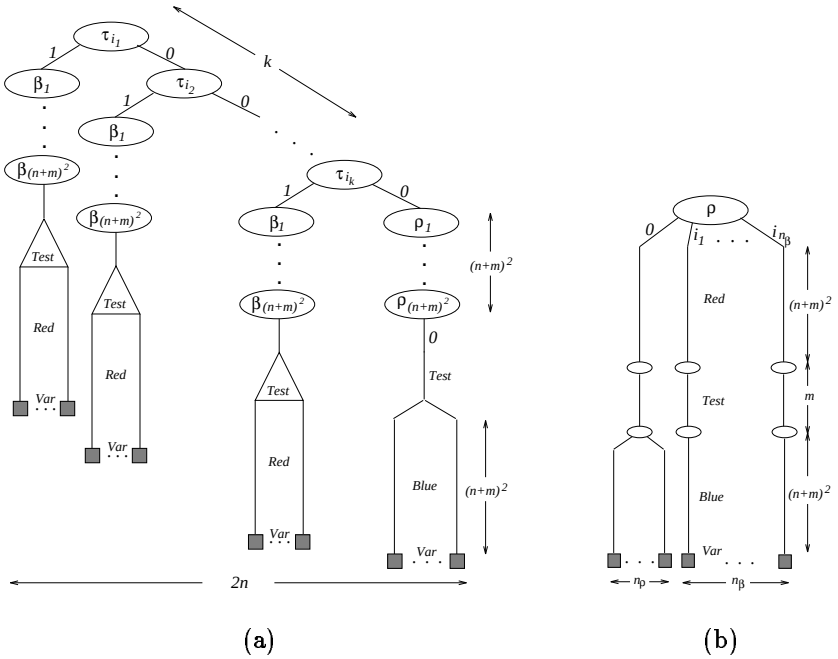


Figure 6: Optimal CRA for $2n$ clauses (a) and suboptimal CRA (b)

The main idea in constructing an optimal CRA for a combination of *Red* and *Blue* clauses is to order the examining of positions such that less partitioning occurs near the root and more occurs toward the leaves, and to select the leaves as tabled states. Therefore, to build an optimal CRA for the $2n$ *Blue* and *Red* clauses constructed from the set cover instance I, positions in the *Test* segment should be examined before positions in the *Blue* and *Red* segments. Observe that examining a position in the *Test* segment partitions a set containing both *Blue* and *Red* clauses into one set containing only *Blue* clauses (on a branch labeled "1") and another containing *Red* and perhaps some *Blue* clauses (on a branch labeled "0"– see Figure 6a). Examining a position in the *Red* (*Blue*) segment, on the other hand, partitions the set into one set containing all of the *Red* (*Blue*) clauses and one set for each of the *Blue* (*Red*) clauses (see Figure 6b). These observations lead to the following proof that U has a cover of size k if and only if there exists a CRA for S having worst-case total cost less than $(2n+k+2)(n+m)^2+6n$

Lemma A.3.3 *If U has a cover of size k, then there exists a CRA for S having worst-case total cost less than $(2n+k+2)(n+m)^2+6n$.*

Proof of lemma: A CRA for S can be constructed as illustrated in Figure 6a. For $1 \leq j \leq k$, let C_{i_j} be a member of the cover of U. Now, for each state τ_{i_j} in the automaton, the subautomaton attached to the "1" edge is an automaton for *Blue* clauses only, since no *Red* clause head contains a 1 in its τ segment. Since $\{C_{i_1}, \ldots, C_{i_k}\}$ is a cover for U, each *Blue* clause is represented in one of the subautomata attached to a "1" edge, and only the n *Red* clauses are represented in the subautomaton attached to the "0" edge of state τ_{i_k}.

By lemma A.3.2, the worst-case total cost of the subautomaton attached to the "1" edge of state τ_{i_j} is no more than $(n_{s_j}+1)(n+m)^2+(m+3-j)n_{s_j}$, where n_{s_j} is the number of clauses represented in the subautomaton. Since $\sum_{j=1}^{k} n_{s_j} = n$, the total cost of all the *Blue* subautomata is no more than $(n+k)(n+m)^2+(m+3)n$. By lemma A.3.1, the worst-case cost of the subautomaton attached to the "0" edge of state τ_{i_k} is no more than $(n+1)(n+m)^2+3n+m-k$. Thus, the worst-case total cost of the CRA is no more than $(2n+k+1)(n+m)^2+(n+1)m+6n+2k$, which is less than $(2n+k+2)(n+m)^2+6n$. □

Lemma A.3.4 *If there exists a CRA for S having worst-case total cost less than $(2n+k+2)(n+m)^2+6n$, then U has a cover of size k.*

Proof of lemma: It suffices to show that any CRA cor S having cost less than $(2n+k+2)(n+m)^2+6n$ must be of the form shown in Figure 6a. Given a set S' containing n_ρ *Red* clauses and n_β *Blue* clauses, the root of an optimal CRA for S' must be a τ state. If, instead, the root were a ρ state, the CRA would have cost at least $(2n_\beta+n_\rho+1)(n+m)^2+3(n_\rho+n_\beta)$ (see Figure 6b). Similarly, a CRA with a β root state would have cost at least $(2n_\rho+n_\beta+1)(n+m)^2+3(n_\rho+n_\beta)$. Replacement of any of the k τ states in the CRA in Figure 6a by a ρ or β state would therefore result add a cost of at least $(n-k)(n+m)^2$. Thus, any CRA having worst-case cost less than $(2n+k+2)(n+m)^2+6n$ must be of the form shown in Figure 6a, which can exist only if U has a cover of size k. □

Lemmas A.3.3 and A.3.4 together complete the reduction from SC to NSCRA. The remaining details showing that the construction is polynomial in the size of I and that NSCRA $\in$ NP are standard and omitted. ■

A.4 Datalog programs with mode information

Recall that Section 4.2 identified the following four optimization problems for sequential automata:

Worst-case cost of given nontabled automaton (NT-Given): *Given a sequence H of Datalog clause heads, a nontabled SCRA $\mathcal{A}_H$ for H, and an integer k, does there exist a ground goal g such that $unifcost(\mathcal{A}_H, g) \geq k$?*

Worst-case cost of optimal nontabled automaton (NT-Opt): *Given a sequence H of Datalog clause heads and an integer k, for every nontabled SCRA $\mathcal{A}_H$ for H does there exist a ground goal g such that $unifcost(\mathcal{A}_H, g) \geq k$?*

Worst-case cost of given tabled automaton (T-Given): *Given a sequence H of Datalog clauses, a tabled SCRA $\mathcal{A}_H$ for H, and integer k, does there exist a ground goal g such that $totalcost(\mathcal{A}_H, g) \geq k$?*

Worst-case cost of optimal tabled automaton (T-Opt): *Given a sequence H of Datalog clauses and integer k, for every tabled SCRA $\mathcal{A}_H$ for H does there exist a ground goal g such that $totalcost(\mathcal{A}_H, g) \geq k$?*

The proofs of the theorems given in Section 4.2 for these problems appear below.

Theorem A.4 *NT-Given is NP-complete.*

Proof: The proof is by reduction from 3-satisfiability (3SAT) [9]. Given an instance I of 3SAT consisting of a set V containing m variables and a set C containing n disjunctions (each having three literals), we construct an instance I' of NT-Given, consisting of a sequence H of $16n$ $(m+1)$-ary Datalog facts and a nontabled SCRA $\mathcal{A}_H$ for H, as follows. For each disjunction $D_i \in C$ of the form $(L_{i_1} \vee L_{i_2} \vee L_{i_3})$, where $L_j \in \{X_j, \overline{X_j}\}$ and $X_j \in V$, a sequence of sixteen Datalog facts is created. Let $a_1 = (t,t,t), a_2 = (t,t,f), \ldots, a_8 = (f,f,f)$ be the eight possible truth assignments for the variables $(X_{i_1}, X_{i_2}, X_{i_3})$ in D_i. Exactly one of these assignments, say a_f, falsifies D_i; all others satisfy D_i. For each a_k an $(m+1)$-ary Datalog fact is created, with argument 1 as "1", argument $i_l + 1$ as $a_k(l)$, $1 \leq l \leq 3$, and each of the remaining $m-3$ arguments as a distinct variable. If $a_k \neq a_f$ the pattern is duplicated, yielding fifteen facts. The sixteenth and final fact has all of its $m+1$ arguments as distinct variables (see Figure 7 for an example of this construction). H is formed by concatenating the n sequences of facts constructed for each $D_i \in C$, yielding a total of $16n$ facts in H.

Now, for some $D_i \in C$, consider an SCRA for the 16 facts created from c that examines, along any root-to-leaf path, first argument 1, then arguments i_1, i_2, and i_3, followed by the remaining $m-3$ arguments in any order. Such an SCRA will have the form shown in Figure 7c. SCRA $\mathcal{A}_H$ is constructed by merging together the n automata corresponding to the disjunctions in C (see Fig. 7d), completing the construction of I'. This construction can clearly be done in time polynomial in n and m.

To complete the reduction, we show that

$$C \text{ is satisfiable} \Leftrightarrow \text{there exists a ground goal } g \text{ such that } unifcost(\mathcal{A}_H, g) \geq n(3m-1),$$

where each elementary unification operation is assumed to have unit cost. Consider again the SCRA in Figure 7c for the 16 facts in Figure 7b. The unification of any ground goal via the SCRA that traverses the three edges corresponding to one of the seven satisfying assignments of $(X_1 \vee \overline{X_2} \vee X_4)$ will traverse a total of $3m-1$ edges (*e.g.,* the $4 + 2(m-3)$ dashed edges and $m+1$ dotted edges). The unification of any other ground goal must traverse fewer edges. In particular, the unification of a goal that traverses the three edges corresponding to the falsifying assignment will traverse a total of $2(m+1)$ edges (the $m+1$ bold edges and $m+1$ dotted edges).

Now, consider the unification of any ground goal g via SCRA $\mathcal{A}_H$. If the arguments of g correspond to a satisfying assignment for C (and argument 1 is "1"), the unification of g will traverse a total of $n(3m-1)$ edges in $\mathcal{A}_H$ ($3m-1$ edges in each of the n smaller automata comprising $\mathcal{A}_H$). The unification of any other goal must traverse fewer edges. In particular, for a goal whose arguments correspond to an falsifying assignment for C, unification of the goal will traverse at most $2(m+1)$ edges in at least one of the n subautomata from which $\mathcal{A}_H$ was formed. Therefore, C is satisfiable iff there exists a ground goal g such that $unifcost(\mathcal{A}_H, g) \geq n(3m-1)$.

In order to show NT-Given $\in$ NP, it suffices to observe that the cost of unifying any given goal via $\mathcal{A}_H$ can be computed in $O(nm)$ time. ■

Theorem A.5 *NT-Opt is NP-hard.*

Proof: The construction used in the proof of Theorem A.4 is used here to prove that

$$C \text{ is satisfiable} \Leftrightarrow \text{for every SCRA } \mathcal{A}_H \text{ there exists a ground goal } g \text{ such that } unifcost(\mathcal{A}_H, g) \geq n(3m-1).$$

The proof of Theorem A.4 establishes the existence of an SCRA that, when C is not satisfiable, always has unification cost less than $n(3m-1)$, leaving to be proved

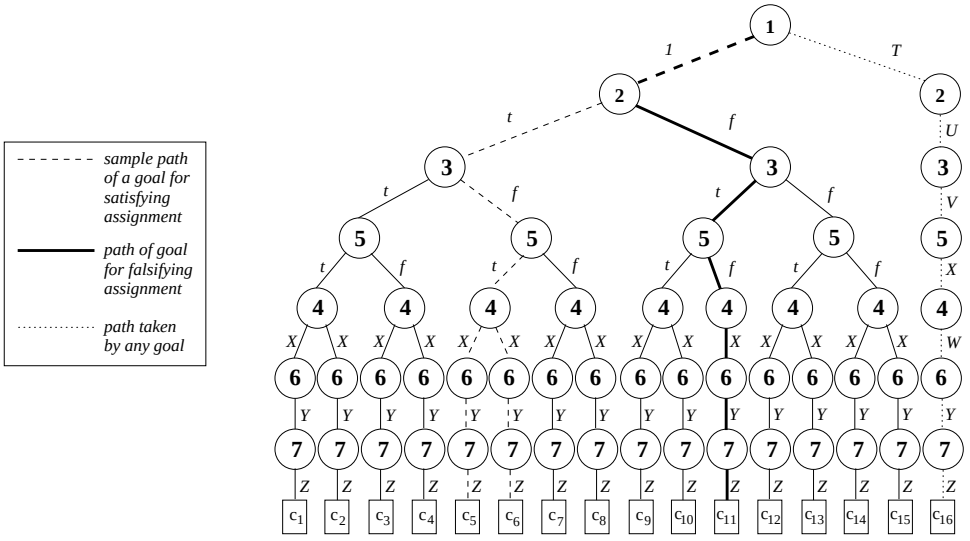
$$C \text{ is satisfiable} \Rightarrow \text{for every SCRA } \mathcal{A}_H \text{ there exists a ground goal } g \text{ such that } unifcost(\mathcal{A}_H, g) \geq n(3m-1).$$

$(X_1 \vee \overline{X_2} \vee X_4)$

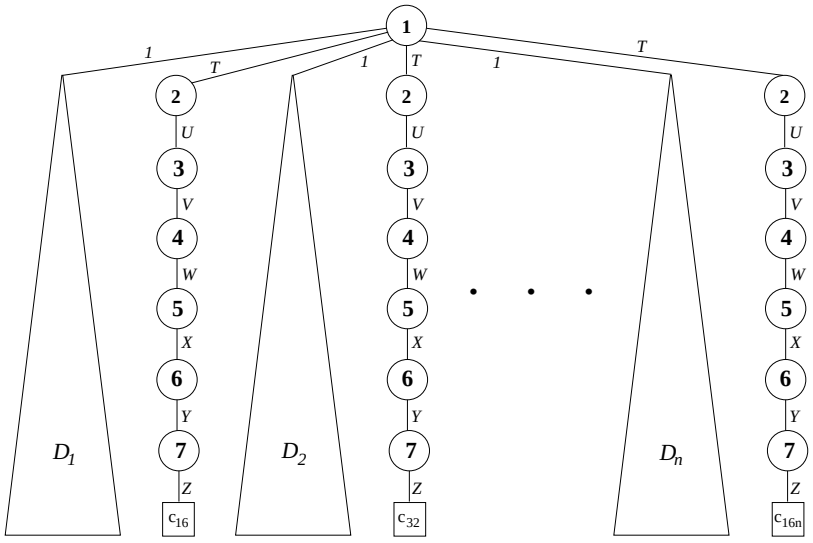
(a)

```
p(1,t,t,X,t,Y,Z).    p(1,f,t,X,t,Y,Z).
p(1,t,t,X,t,Y,Z).    p(1,f,t,X,t,Y,Z).
p(1,t,t,X,f,Y,Z).    p(1,f,t,X,f,Y,Z). (falsifying assignment)
p(1,t,t,X,f,Y,Z).    p(1,f,f,X,t,Y,Z).
p(1,t,f,X,t,Y,Z).    p(1,f,f,X,t,Y,Z).
p(1,t,f,X,t,Y,Z).    p(1,f,f,X,f,Y,Z).
p(1,t,f,X,f,Y,Z).    p(1,f,f,X,f,Y,Z).
p(1,t,f,X,f,Y,Z).    p(T,U,V,W,X,Y,Z).
```

(b)



(c)



(d)

Figure 7: 3SAT to SCRA – (a): CNF clause; (b): facts created from (a); (c): SCRA for (b); (d): full SCRA.

This is proved by showing that, for any possible SCRA for H (the $16n$ Datalog facts derived from C), any ground goal (with argument 1 as "1") corresponding to a satisfying assignment for C will have unification cost at least $n(3m-1)$. The following two observations lead to this result:

Observation 1 Any ground goal (with first argument equal to "1") corresponding to a satisfying assignment for C unifies with exactly $3n$ facts, since in each of the n sequences of 16 facts derived from the disjunctions of C, there are two facts (duplicates) exactly matching four arguments in the goal (unifying with the rest), and one fact that unifies with any goal.

Observation 2 In any SCRA for H, no edges can be shared between subautomata representing facts derived from two or more distinct disjunctions in C. Recall that all facts that unify with a goal must be unified in textual order. The all-variable fact appearing at the end of each of the n sequences of facts prohibits such edge sharing, since it shares no edges with any fact, and it must be unified with a goal after any fact appearing before it, and before any fact appearing after it (see Figure 7d).

Now, in any SCRA for H, the unification of any ground goal corresponding to a satisfying assignment for C with three facts from any of the n sequences must traverse at least $3m-1$ edges in the SFA: $m+1$ edges for the all-variable fact, $2(m-3)$ edges for the (non-shared) variable positions in the two other facts, and at least 4 edges for the remaining positions in those two facts. Since the goal unifies with three facts from each of the n sequences, the total cost of unifying the goal is at least $n(3m-1)$. ■

Note that Theorem A.4 means that it is unlikely that NT-Given $\in$ NP.

To generalize the above results to tabled automata requires taking into account simultaneously the three costs of tabled evaluation: unification, call tabling, and answer tabling (recall Theorem 3.1). In general, determining answer costs for a given CRA is difficult, since the number of answers is not known at compile time. However, even if the number of answers for a given query is known, the problem of determining the overall cost (unification + call + answer) of a given SCRA is NP-complete. As with unification cost alone, this result is extended to show that the decision problem corresponding to SCRA cost minimization is NP-hard. The proofs are similar to those for Theorems A.4 and A.5 and are sketched below.

Theorem A.6 *T-Given is NP-complete.*

Proof: (*Sketch*) The proof is by reduction from 3SAT, and the construction is similar to that used in the proof of Theorem A.4, with the following modifications. First, the clauses in H are no longer simple facts, but have bodies fixed to produce a certain number of answers per clause (as defined below). Second, each clause head has one extra variable argument (for holding answer substitutions). Third, no clause is duplicated. Thus, there are nine clauses per disjunction in C (having the same form as before) and $9n$ clauses total in H. A tabled SCRA $\mathcal{A}_H$ is constructed having the same form as before, except that the paths corresponding to satisfying assignments for individual disjunctions no longer branch. To make $\mathcal{A}_H$ a tabled CRA, the leaves are marked as tabling points. Finally, for each of the $7n$ clauses whose heads correspond to satisfying assignments for disjunctions in C, the number of answers is fixed at $9(m+2)(n+1)$. For each of the $2n$ remaining clauses (whose heads are all variables or correspond to falsifying assignments), the number of answers is fixed at 1. For the remainder of the proof, we assume goals that are ground in all arguments but the last (the newly added argument to hold answers). Now, for a goal corresponding to a satisfying assignment for C, two clauses for each of the n disjunctions will succeed, one resulting in a large number of answers ($9(m+2)(n+1)$), and one resulting in one answer. For a goal corresponding to an falsifying assignment, there will be two clauses for at least one of the n disjunctions with only one answer each, yielding a lower total number of answers. Unification and tabling costs are the same for either type of goal. The following table summarizes these costs for both types of goal:

Cost	Satisfying goal	Falsifying goal
Unification	$2n(m+2)$	$2n(m+2)$
Call	$n((m-2)+(m+2))$	$n((m-2)+(m+2))$
Answer	$n(9(m+2)(n+1)+1)$	$< n(9(m+2)(n+1)+1)$
Total	$n(9(m+2)(n+1)+4m+5)$	$< n(9(m+2)(n+1)+4m+5)$

Thus, we have

$$C \textit{ is satisfiable} \Leftrightarrow \textit{there exists a goal } g \textit{ such that } totalcost(\mathcal{A}_H, g) \geq n(9(m+2)(n+1)+4m+5),$$

As in the proof of Theorem A.4, the total cost of a given goal with respect to $\mathcal{A}_H$ can be verified in $O(mn)$ time, and hence, T-Given $\in$ NP. ■

Theorem A.7 *T-Opt is NP-hard.*

Proof: (*Sketch*) Using the same construction as above, the main idea is to show that, for any tabled SCRA that can be constructed for H, the lowest combined call and answer cost occurs for tabling at the leaves of the automaton. An SCRA with the lowest total cost will then be one that minimizes unification cost (since unification cost is independent of tabled states), with the leaves as tabling points.

To show that tabling at the leaves is optimal for any SCRA for H, we first consider tabling at the root. Here the call tabling cost is just $(m+2)$. If the tabling points are moved to the children of the root, the call tabling cost increases to at most $9n(m+2)$ for goals corresponding to satisfying assignments (consider an SCRA that first examines argument $m+2$, which is a variable for all clauses). Now, the root of the SCRA examines either position $m+2$, or some other position. If it examines position $m+2$, there is a $9n$-way branch from the root, forming $9n$ chains to the leaves. In this case, the tabling points can be moved to the leaves without further increasing call tabling costs, while reducing answer tabling costs by $(m+1)n(9(m+2)(n+1)+1)$, far outweighing the increase in call tabling cost. If the root examines any other position, some of the outgoing edges must be labeled by constants. By moving the tabling points to just below the root, answer costs are reduced by at least $9(n+1)(m+2)$, still outweighing the increase in call tabling cost. Hence, tabling below the root costs less than tabling at the root for any SCRA for H.

With tabling points below the root shown to have lower cost, we can now (by Observation 2 in the proof of Theorem A.5) consider separately the subautomata corresponding to the n individual disjunctions of C. Since any goal will succeed along at most one path in the subautomaton, the lowest combined call and answer cost will occur for an automaton that first examines positions that have constants in the clause heads, with tabling points at the leaves (call cost $(m-2)+(m+2)$, answer cost (satisfying) $9(m+2)(n+1)+1$). Moving tabling points any higher will increase answer cost far more than the savings in call cost. Thus, for any SCRA for H, the least combined call and answer cost occurs when the leaves are the tabled states.

Using an argument similar to that used in the proof of Theorem A.5, it is easily established that the SCRA constructed in the proof of Theorem A.6 is minimal for unification cost. Since it has tabling points at the leaves, it is also minimal for overall cost. Hence,

$$C \textit{ is satisfiable} \Leftrightarrow \textit{for any tabled SCRA } \mathcal{A}_H \textit{ for } H \textit{ there exists a goal } g \textit{ such that } totalcost(\mathcal{A}_H, g) \geq n(9(m+2)(n+1)+4m+5).$$

■

A.5 Benchmark programs

Transitive closure

```
% Original:

a :- anc(_, _), fail; true.

:- table anc/2.
anc(X, Y) :- parent(X, Y).
anc(X, Y) :-
    anc(X, Z),
    parent(Z, Y).


% Factored:

a1 :- anc1(_, _), fail; true.

:- table anc11/2.
anc1(X, Y) :- parent(X, Y).
anc1(X, Y) :- anc11(X, Y).
anc11(X, Y) :-
    anc1(X, Z),
    parent(Z, Y).
```

EDB I

```
/* 3-level tree
with fanout of
10 and a total
of 1110 facts */

parent(1, 10).
parent(1, 11).
parent(1, 12).
parent(1, 13).
parent(1, 14).
parent(1, 15).
parent(1, 16).
parent(1, 17).
parent(1, 18).
parent(1, 19).
parent(10, 100).
.
.
.
parent(11, 110).
.
.
.
parent(100, 1000).
.
.
.
parent(199, 1999).
```

EDB II

```
parent(2,1).
parent(3,1).
parent(4,1).
parent(5,1).
parent(6,1).
parent(7,1).
parent(8,1).
parent(9,1).
parent(10,1).
.
.
.
parent(10238,1).
parent(10239,1).
parent(10240,1).
```

Join

```
% parent/2 and parent2/2 are EDB I above
parent_salt(s(X),s(Y)):- parent(X,Y).
parent_salt(d(X),d(Y)):- parent2(X,Y).

% Original

joinsalt2 :- suppalt1(X), parent_salt(X,W).

:- table(suppalt1/1).
suppalt1(X):- parent_salt(X,_),parent_salt(X,_).


% Factored

joinsalt3 :- suppalt2s_extra(X), parent_salt(X,W).
joinsalt3 :- suppalt2d_extra(X), parent_salt(X,W).

suppalt2s_extra(s(X)):- suppalt2s(X).
suppalt2d_extra(d(X)):- suppalt2d(X).

:- table(suppalt2s/1).
suppalt2s(X):- parent(X,_),parent(X,_).
:- table(suppalt2d/1).
suppalt2d(X):- parent2(X,_),parent2(X,_).
```

Ambiguous

% Query: $(++c-+cc)^{140}$

```
% Original:
:- table s/2.

p(X) :- s(X, []), fail; true.

s([+|X], Y) :- s(X, Z), s(Z, Y).
s([+|X], Y) :- s(X, Y).
s([c|X], X).


% Factored:

:- table s12/2.
p1(X) :- s1(X, []), fail; true.

s1([U|X], Y) :- s11(U, X, Y).

s11(+, X, Y) :- s12(X, Y).
s11(c, X, X).

s12(X, Y) :- s1(X, Z), s1(Z, Y).
s12(X, Y) :- s1(X, Y).
```

LR(1)

```
% Queries:
% (a):  a^100 b^100
% (b):  (ab)^300


% Original:

:- table s/2.
p(X) :- s(X,[]), fail ; true.

s(X, Y) :- s(X, [a|Z]), s(Z, [b|Y]).
s(X, X).


% Factored:

:- table s11/2.
p1(X) :- s1(X,[]), fail; true.

s1(X, Y) :- s11(X, Y).
s1(X, X) :- s12(X).
s11(X, Y) :- s1(X, [a|Z]), s1(Z, [b|Y]).
s12(_).
```

LR(1)(det)

```
% Original:
:- table s0/2.
p0(X) :- s0(X,[]), fail; true.

s0([a|X],[]) :- s0([a|X],[a|Z]), s0(Z,[b]).
s0([a|X],[a|Y]) :-
    s0([a|X],[a|Z]), s0(Z,[b,a|Y]).
s0([a|X],[b|Y]) :-
    s0([a|X],[a|Z]), s0(Z,[b,b|Y]).
s0([],[]).
s0([a|X],[a|X]).
s0([b|X],[b|X]).

% Factored:
:- table s24/1, s25/2, s26/2.
:- index s21/2-2.
:- index s23/3-2.

p2(X) :- s2(X,[]), fail; true.

s2([a|X],Y) :- s21(X,Y).
s2([],[]).
s2([X|Y],[X|Y]) :- s22(X).
s21(X,[]) :- s24(X).
s21(X,[Y|Z]) :- s23(X,Y,Z).
s23(X,a,Y) :- s25(X,Y).
s23(X,b,Y) :- s26(X,Y).
s24(X) :- s2([a|X],[a|Z]), s2(Z,[b]).
s25(X,Y) :- s2([a|X],[a|Z]), s2(Z,[b,a|Y]).
s26(X,Y) :- s2([a|X],[a|Z]), s2(Z,[b,b|Y]).
s22(a).
s22(b).
```