Caching and Lemmaizing in Model Elimination Theorem Provers*

Owen L. Astrachan
Department of Computer Science
Duke University
Durham, NC 27706
ola@cs.duke.edu

Mark E. Stickel
Artificial Intelligence Center
SRI International
Menlo Park, CA. 94025
stickel@ai.sri.com

Abstract

Theorem provers based on model elimination have exhibited extremely high inference rates but have lacked a redundancy control mechanism such as subsumption. In this paper we report on work done to modify a model elimination theorem prover using two techniques, *caching* and *lemmaizing*, that have reduced by more than an order of magnitude the time required to find proofs of several problems and that have enabled the prover to prove theorems previously unobtainable by top-down model elimination theorem provers.

1 Introduction

Model Elimination (*ME*) [20, 22] is a complete inference procedure for the first-order predicate calculus. It is the method underlying the Prolog Technology Theorem Prover (*PTTP*) [33, 34], the *SETHEO* prover [19], and several or-parallel theorem provers [31, 8, 2]. The use of model elimination, an input proof procedure, has enabled *ME*-based provers to draw on techniques developed by the logic programming community (hence the name *PTTP*) that enable the implementations to be very efficient in the use of space, to have a high inference rate, and to be readily parallelized. *METEOR* is a high-performance implementation of *ME* written in C that runs under the UNIX operating system. It compiles clauses into a data structure that is then "interpreted" at run-time by a uniprocessor, a multiprocessor, or a network of uniprocessors. *METEOR* and *PTTP* perform exactly the same number of inferences in solving the problems reported in [34] when *METEOR* is run using inference count as the depth measure (see Section 3).

Unfortunately, *ME* provers are also susceptible to a lack of control in the search mechanism and can have highly redundant search spaces. This paper reports on work done

*This research was supported by the National Science Foundation under Grant CCR-8922330. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the National Science Foundation or the United States government.

to modify the sequential version of *METEOR* [2] in an attempt to address these issues. We report on two mechanisms—caching and lemma use—that have enabled *METEOR* to prove theorems not previously obtainable by top-down *ME* provers and that can reduce by more than an order of magnitude the time required to find proofs of some "difficult" theorems.

In general, the lack of both a redundancy-control mechanism such as subsumption [42] and a best-first search methodology are severe impediments to finding deep proofs. Many theorems obtainable by *OTTER* [25] cannot be proven in our systems because the size of the search space and the lack of redundancy control quickly overwhelm the inference rate. We have investigated several methods of changing the search mechanism used in *METEOR* in an attempt to redress this lack of control. We use the two-dimensional grid in Figure 1 to categorize our approaches.

	replace search	**augment search**
discovery cost	caching	
other cost	heuristic caching	lemmaizing

Figure 1: Changing the search mechanism.

In a broad sense, the cost referred to in Figure 1 is a measure of the computational resources used to find a solution of a goal. More precisely, *METEOR* employs an iterated form of depth-first search called iterative deepening [36, 18] in which the maximum depth of search is bounded during each iteration. This bound limits the computational resources available to solve a goal; the resources actually used to find a solution of a goal constitute the discovery cost of the solution. Details of the search mechanism and the depth bounds used in *METEOR* are given in Section 3.

In our terminology, *caching* refers to a mechanism that optionally replaces the normal search mechanism at a lower computational cost, but yields essentially identical results to search. Cached goals are solved by lookup instead of search. Proofs will be found with the same cost bound as when caching is not used, and no more inferences will be performed with caching than without (in practice, many fewer inferences are required when caching is used). Caching reduces the number of inferences because replacing search for solutions of previously seen (cached) goals by lookup avoids repeating inferences on "failure branches" of the search tree explored during the search for the cached solution; lookup ideally will return each distinct solution only once, whereas search may repeatedly generate the same solution, and fewer more general solutions may be retrieved from the cache instead of many more specific ones. Whether there is a net performance gain depends on how efficiently the cache is implemented, i.e., what are the relative costs of search and cache lookup.

For caching to reproduce essential features of the search and, in particular, to guarantee that use of the cache does not result in more rather than fewer inferences being performed, it is necessary for the cache to return solutions with the same cost bounds as search would have found. Caching charges discovery cost to reproduce essential features of search, e.g., the same solutions with the same cost. Charging discovery cost is not the only option, however. If some solution looks particularly useful, perhaps because of its generality, it might be desirable to charge less than discovery cost for it, to make it easier to use. Or if all solutions look alike (e.g., they have only constant arguments and no function symbols, as in the case with Datalog programs) despite being discovered with different costs, it may be reasonable to charge a uniform minimum cost for them instead of distinguishing among them on the basis of how deep their proofs were. Charging some cost other than discovery cost leads to what we call *heuristic caching*, which is identical to caching in concept and implementation except for the cost charged for looked-up solutions. The guarantee that caching will not result in an increase in the number of inferences is absent for heuristic caching, but in some domains heuristic caching can be extremely successful.

The objective of caching is to make effective use of results discovered in past search. Caching simply stores results of past searches and replaces future searches by cache lookup. Another way to use results discovered in past searches is to record some seemingly useful solutions as *lemmas* and use them in future extension operations in the same way as input clauses are used. Note that lemmas, unlike caching, can introduce substantial additional redundancy in the search space, since theorems can then be proved both either entirely from the input clauses as before or by use of lemmas. Allowing lemmas to be treated as input clauses thus increases the branching factor of the search space, but use of lemmas may still shorten the proof enough to compensate for the increased branching factor. Note that it makes no sense to charge discovery cost for lemmas. This would result in an increased branching factor and no reduction in proof length—goals would be solved by both input clauses and lemmas with the same cost. Lemmas (as stored solutions) can be beneficial only if less is charged for their use than for their solution from the input clauses and, even then, a lemma must actually be used in the proof of the top goal for there to be any reduction in the total size of the search space. We use the word *lemma-izing* or *lemmaizing*[1] to refer to this mechanism that augments the search by introducing lemmas that are treated as input clauses .

This paper is organized as follows: In Section 2 we briefly describe the model elimination proof procedure. In Section 3 we outline the search mechanism used in *METEOR* and our modifications to this mechanism. These modifications are further described in Section 4 and Section 5. In Section 6 we briefly describe the implementation of these modifications, results generated using this implementation are given in Section 7. Related work is outlined in Section 8 and conclusions presented in Section 9.

2 Model Elimination

In this section we give a description of the *ME extension* and *reduction* inference rules and other *ME* terminology sufficient for an understanding of the remaining sections. We assume familiarity with terminology of resolution proof procedures, e.g., *terms*, *atomic formulas (atoms)*, *literals*, *clauses* and *unification*. For a description of these, see [22], which also gives a complete description of the model elimination procedure. We use Prolog notation in which variables are represented by capital letters and functions, constants and predicates are represented by lowercase letters.

[1]Although the juxtaposition of vowels in this word may be inharmonious, recall memo-izing [26] used to mean essentially what we call caching.

The *ME* proof procedure uses a kind of annotated clause called a *chain*. Roughly, the annotations in a chain record previous inferences that have been made in the current sequence of inference steps and identify information that can be used as the proof is expanded. Literals in a chain are either *B-literals* or *A-literals*. An A-literal has been used in an extension operation (and is thus in some sense an ancestor literal) and may participate in the *ME* reduction operation. Initially all literals are *B-literals*.

The *ME* procedure begins with some designated input clause as the initial chain. The leftmost literal in this chain is unified with a complementary literal of an input clause. The leftmost literal in the chain is designated as an A-literal (ancestor literal), the other literals (if any) in the input clause are added to the front of the chain, and the unifying substitution is applied. This is the *ME* extension operation. It is the same as the Prolog inference operation except that it retains the unified literal as an A-literal, which may then be used in subsequent *ME* reduction operations. A-literals appear in brackets in the following descriptions. We assume that all chains and clauses are renamed apart so that they are variable disjoint as necessary. Formally we have

Definition 2.1 *Given chain C_1 of the form l_1C_0 with leftmost B-literal l_1 and input clause C_2 with literal l_2 complementary to l_1 such that the atoms of l_1 and l_2 are unifiable with most general unifier (mgu) θ, the ME* <u>extension</u> *operation of C_1 with C_2 on l_2 yields the chain $\{(C_2 - l_2)[l_1]C_0\}\theta$ where $[l_1]$ is an* A-literal *and the literals in $(C_2 - l_2)$ may be reordered. We use the notation* $\underline{\text{extend}(C_1, l_1, C_2, l_2, \theta)}$ *to denote the result of the extension.*

Example: If $C_1 = q(f(X), Y)[r(Y, Z)]p(X, Z)$ and $C_2 = \neg q(f(a), c)\neg r(c, b)$ (note that C_1 has one A-literal and two B-literals) then C_1 extended with C_2 on $\neg q(f(a), c)$ is $\neg r(c, b)[q(f(a), c)][r(c, Z)]p(a, Z)$.

Definition 2.2 *Given chain C_1 of the form l_1C_0 with leftmost B-literal l_1 and complementary A-literal l_2 such that the atoms of l_1 and l_2 are unifiable with mgu θ, the ME* <u>reduction</u> *operation yields chain $C_0\theta$. We use the notation* $\underline{\text{reduce}(C_1, l_1, l_2, \theta)}$ *to denote the result of the reduction.*

Example: In the resultant chain above, $\neg r(c, b)[q(f(a), c)][r(c, Z)]p(a, Z)$, the (only possible) reduction operation yields $[q(f(a), c)][r(c, b)]p(a, b)$.

Note that both the reduction operation and extension with a unit clause (in which no literals are added to the chain) can make the leftmost literal of the chain an A-literal. As the *ME* inference rules require the leftmost literal to be a B-literal, any leftmost A-literals are removed after the extension and reduction operations are performed. This is the *contraction* operation as defined in [22]. In the chain used in the examples above, the chain $[q(f(a), c)][r(c, b)]p(a, b)$ is contracted to the chain $p(a, b)$. In practice this operation is incorporated into the extension and reduction operations. The A-literals that are removed by contraction represent solved goals or lemmas.

3 Search Mechanism

Although *ME* is a complete proof procedure in that there is always an *ME* derivation of the empty chain from an unsatisfiable set of input clauses, a complete search strategy must be employed to ensure that such a derivation is found. Prolog, for example, uses unbounded depth-first search and may fail to find a proof because of infinite branches in the search tree.

Rather than employ a breadth-first strategy with its exponential storage requirements, *METEOR* and *PTTP* use iterative deepening [36, 18] to ensure completeness of the search strategy. Iterative deepening is asymptotically optimal among brute-force search mechanisms[2] and has minimal storage requirements, being in essence a depth-first strategy. Rather than storing intermediate results as is done in breadth-first search, results are recomputed at each stage of the iterative deepening search.

We impose a cost bound on prospective proofs. Our cost bounds are not bounds on the entire search space (except implicitly), but rather only on each portion of it that forms a single (partial) proof. Thus, for example, a bound on the number of inference steps in a proof is a cost measure, but a bound on the total number of inferences performed in the process of finding the proof, including those on failing branches of the search space, is not. A finite cost bound d makes the search tree finite while allowing all proofs with cost bounded by d to be discovered. If no proof is found, the bound is incremented and the entire search tree is re-explored with the larger bound. When bounded search is used, each goal has an associated cost bound (derived from the current global bound) that must not be exceeded during an attempt to solve the goal. We use the notation $\langle \mathcal{G}, n \rangle$ to refer to a single literal goal $\mathcal{G}$ with associated cost bound n. The search mechanism employed in *METEOR* is described in Figure 2.

```
boolean
Solve(chain $\mathcal{C}$,resource $n$)
[0]  if $\mathcal{C}$ is the empty chain then
         return TRUE
[1]  goal $\mathcal{G} \leftarrow$ leftmost literal in $\mathcal{C}$
[2]  R $\leftarrow$ A-literals of $\mathcal{C}$ potentially unifiable with complement of $\mathcal{G}$ (for reductions)
[3]  E $\leftarrow$ input clauses with literals potentially unifiable with complement of $\mathcal{G}$ (for extensions)
             /* try to solve $\langle \mathcal{G}, n \rangle$ */
[4]  for each $l_R$ in R do
[5]      $n_{\text{new}} \leftarrow$ resources available if reduction made
[6]      if $n_{\text{new}} \geq 0$ and $l_R$ and complement of $\mathcal{G}$ unify with mgu $\theta$ then
[7]          if Solve(reduce($\mathcal{C}$, $\mathcal{G}$, $l_R$,$\theta$),$n_{\text{new}}$) then
[8]              return TRUE
     endfor (reduction)
[9]  for each clause C in E with literal $l_{\text{C}}$ do
[10]     $n_{\text{new}} \leftarrow$ resources available if extension of $\mathcal{C}$ with C is made
[11]     if $n_{\text{new}} \geq 0$ and and $l_{\text{C}}$ and complement of $\mathcal{G}$ unify with mgu $\theta$ then
[12]         if Solve(extend($\mathcal{C}$,$\mathcal{G}$,C,$l_{\text{C}}$,$\theta$), $n_{\text{new}}$) then
[13]             return TRUE
     endfor (extension)
[14] return FALSE
```

Figure 2: The search procedure.

There are several optimizations that can be applied in the search mechanism without affecting its completeness [34]. Many of these optimizations are implicit in the definition of an acceptable chain and the accepting transformation that is applied to chains in the original presentation of *ME* [20, 21, 22]. The most effective of these tends to be the

[2]The optimality result applies only in the absence of redundancy control mechanisms such as subsumption.

identical ancestor pruning rule. Before any reductions or extensions are attempted, the A-literals in the chain to the right of $\mathcal{G}$ are examined to see if any are identical to $\mathcal{G}$. If this is the case, **Solve** returns FALSE; it is not necessary to solve a goal in the context of a previous attempt to solve the same goal. This pruning rule is highly effective but its use is limited when caching is employed, in a manner described later.

In *PTTP*, proofs of minimal length (in number of *ME* inferences) are found since the total number of nodes in a proof tree is bounded during each stage of iterative deepening with successively higher resource limits. We call this depth measure *inference depth* or D_{inf}. Alternatively, bounding the depth of the proof tree yields what we call *A-literal depth* or D_{Alit} since the depth of a proof tree is the number of A-literals present in the chain that represents the current state of the deduction. This metric is the default depth measure used in *SETHEO* and was used in one of the earliest implementations of *ME* [13]. Neither of these measures is clearly superior to the other in that there seems to be no *a priori* method for determining which measure yields a proof more quickly for any particular theorem.

3.1 Modifying the Search Mechanism

The high inference rate and modest storage requirements of *PTTP* and *METEOR* make them attractive inference engines useful for seeking shallow proofs. In some domains, the high inference rate may overcome the lack of redundancy control and permit the discovery of hard theorems with deep proofs. For example, using *METEOR* we are able to find proofs of two problems [2] from a set of real-analysis challenge problems [5] that are difficult if not unobtainable for *OTTER* [25], a prover which employs both subsumption and a notion of best-first search. The proof of the third challenge problem from this set is too deep for *METEOR* to obtain without some modification. Parallel implementations of model elimination theorem provers have resulted in very high performance provers [31, 8, 2], but none of these provers has yet produced a proof of a theorem previously unobtainable by running the prover on a single processor. The high inference rate obtainable with *PTTP*, *METEOR* and similar systems cannot overcome the highly redundant, exponential search space of hard problems with deep proofs. We have modified the search mechanism used in *METEOR* by the addition of caching—which replaces search, and lemmaizing—which augments search. Our goal has been to implement these modifications with minimal degradation of the high inference rate.

4 Caching

By *caching* we mean the use of a device (the *cache*) that on occasion replaces the regular search mechanism and yields substantially identical results to search. This means that solutions should be retrieved from the cache only when it is known that the cache contains complete information, i.e., it contains all solutions that would be generated by search. When the cache is complete in this sense its use can replace the normal search mechanism. To this end, the cache consists of two logical parts: the *cache directory*, which stores information about which goals have solutions stored in the cache, and the *cache store*, which contains the solutions. When a goal and its associated resource bound are submitted to the **Solve** procedure (see Figure 2), the directory is consulted and the cache store used, if possible, before line 4. If the cache store is used, procedure **Solve** is exited (with success or failure) before lines 4–14 are executed. If the cache store is not used, lines 4–14 are executed as in the regular search procedure. The cache is intended to be a more efficient mechanism than the normal search procedure. Its effectiveness in decreasing the time to find a proof depends on the efficiency with which it is implemented, the number of cache "hits" that occur, and on other costs (e.g., increased storage) incurred by its use.

The caching method we describe here is applicable only to cases of model elimination in which the reduction operation is not used; this includes problems expressed in Horn clauses. For such problems all solutions of the pair $g_1 = \langle \mathcal{G}, n \rangle$ are also solutions of the pair $g_2 = \langle \mathcal{G}, m \rangle$ if $n \leq m$ since the sequence of inferences that solve $\mathcal{G}$ in g_1 will also solve $\mathcal{G}$ in g_2 provided the identical-ancestor pruning rule is partially disabled.

If during the search for solutions of some goal $\langle \mathcal{G}, n \rangle$ a branch of the search tree below $\mathcal{G}$ is pruned using identical-ancestor pruning with an ancestor $A_{\mathcal{G}}$ of $\mathcal{G}$, a solution might be missed that would be found in another context in which $A_{\mathcal{G}}$ did not appear as an ancestor. To prevent such inconsistencies, and to avoid the need for storing an environment of ancestor literals, identical-ancestor pruning of subgoals of cacheable goals is disabled. More precisely, a goal cannot be pruned by any ancestor of a cacheable goal. Pruning is permitted if the pruning goal is not being stored in the cache, or its descendants are not.

For non-Horn problems the sequence of deductions that solve $\mathcal{G}$ in g_1 may include reductions with ancestors of $\mathcal{G}$. This same sequence of deductions will solve $\mathcal{G}$ in g_2 only if the same reductions are possible, i.e., only if the necessary A-literals are present in the chain. Thus caching with non-Horn problems would seem to require that a cached solution contain some record of the A-literals present when the solution is generated.

Although a modification of *ME* has been proposed [29] that can decrease the number of A-literals that are stored with a cached solution (at the expense of potentially longer proofs), and a method of generalized A-literals is addressed in [1], in this paper we do not address the issue of caching with non-Horn problems. We do, however, note several successful applications of lemmaizing to non-Horn problems in Section 7.

4.1 The Cache Mechanism

The cache store contains all the cached solutions. A cached solution consists of a substitution instance of the subgoal and the resource bound used in obtaining the solution as defined in Definition 4.1.

Definition 4.1 *A* <u>cached solution</u> *is a pair* $\langle \mathcal{G}', n_{\mathcal{G}'} \rangle$ *where* $\mathcal{G}'$ *is* $\mathcal{G}\theta$ *for some goal* $\mathcal{G}$, θ *is the composition of unifiers used in solving* $\mathcal{G}$, *and* $n_{\mathcal{G}'}$ *is the measure of the resources used in producing* $\mathcal{G}'$.

A cached solution stores only the instantiation $\mathcal{G}'$—nothing to identify the goal $\mathcal{G}$ it was used to solve. Thus, the cache store contains solutions (provable literals) divorced from the goals during whose proof they were found.

The cache directory, which is consulted to determine if the cache store of solutions should be used, consists of cache templates defined in Definition 4.2.

Definition 4.2 *A* cache template *or* template *is a triple $\langle \mathcal{G}, m, m_S \rangle$ that indicates that the cache is m-complete for goal $\mathcal{G}$. If $m_S \leq m$, then m_S is the smallest resource needed to solve $\mathcal{G}$; if $m_S > m$, then $\mathcal{G}$ has no solution with cost $\leq m$.*

The templates in the cache directory are used to determine when the solutions in the cache store include a complete set of solutions for any particular goal.

Definition 4.3 *A cache is* complete *for $\langle \mathcal{G}, n \rangle$ if all solutions of (sub)goal $\mathcal{G}$ that can be obtained with a resource bound of at most n are in the cache store. In this case we say that the cache is* n-complete for $\mathcal{G}$.

Given a goal pair $\langle \mathcal{G}, n \rangle$, the cache directory is searched to see if $\mathcal{G}$ appears as the first component of a template (there may be more than one applicable template if we are using template-subsumption, see Section 4.2). If a template is found and it indicates that the cache is m-complete for $\mathcal{G}$ with $m \geq n$ then the cache can be used in lieu of the regular search mechanism. As an optimized special case, we note that if $m \geq n$ and $m_S > n$ then there are no solutions bounded by n, so further cache lookup to find solutions is unnecessary. This use of the cache directory to indicate failure corresponds to the *failure cache* outlined in [12]; when templates are used in this way we call them *failure templates.*

If the cache is complete for a goal, solutions to the goal can be found by retrieving from the cache store all cached solutions that are instances of the goal. If subsumption of cached solutions is employed, however, unifiable solutions rather than instances must be retrieved from the cache store.

Our cache differs in use from the cache described in [28] in which the cache may be used even when it is not complete. It is similar to an unimplemented modification developed for iterative deepening of the ET^* algorithm for Datalog programs outlined in [11].

When caching is used, the modifications indicated in Figure 3 are made to the search routine of Figure 2.

```
        /* added before line 4 in Figure 2 */
[3.1]   ⟨$\mathcal{G}$,$m$,$m_S$⟩ ← template corresponding to ⟨$\mathcal{G}$,$n$⟩
              /*  more than one template may be applicable
                  if template-subsumption is being used */

[3.2]   if $m \geq n$ then
[3.3]         if $n \geq m_S$ then
[3.4]               return **CacheSolve**($\mathcal{C}$,$n$)
[3.5]         else
[3.6]               return FALSE
        /* if we reach here then use regular search mechanism */
```

Figure 3: Determining if the cache should be used.

The procedure **CacheSolve** called in Figure 3 is shown in Figure 4.

In its implementation in *METEOR*, the code in Figure 3 is guarded by a statement that enables cache use only when n, the resource available, is above some user-specified threshold value. In the current implementation, the same threshold is used to guard both

```
boolean
CacheSolve($\mathcal{C}$,$n$)
     {$\mathcal{G}$ is the leftmost literal in $\mathcal{C}$}
[1]  L $\leftarrow$ all solution pairs $\langle \mathcal{G}', n_{\mathcal{G}'} \rangle$ such that $\mathcal{G}'$ is potentially unifiable with $\mathcal{G}$ and
          such that $n_{\mathcal{G}'} \leq n$

[2]  for each $\langle \mathcal{G}', n_{\mathcal{G}'} \rangle$ in L do
[3]       if $\mathcal{G}$ and $\mathcal{G}'$ unify with mgu $\theta$ then
[4]            $n_{\text{new}} \leftarrow n - n_{\mathcal{G}'}$
[5]            if Solve(extend($\mathcal{C}$,$\mathcal{G}$,$\mathcal{G}'$, $\mathcal{G}'$, $\theta$), $n_{\text{new}}$) then
[6]                 return TRUE
     end for
[7]  return FALSE
```

Figure 4: Using an m-complete cache.

solution storage and template retrieval. Cache use is limited by a threshold for several reasons:

- The cost of retrieving cache templates and solutions may exceed the cost of the regular search mechanism for small n.
- The identical ancestor pruning rule, whose use often results in large decreases in search space size, must be at least partially disabled when caching is used (see Section 4).
- The efficiency of the cache tends to decrease as the number of entries in it increases.

Because the identical-ancestor pruning rule is so effective and because the normal inference rate in *METEOR* is high enough that using the cache for shallow searches is counterproductive cache use is prevented when the global cost bound is below the threshold. We present data in Section 7 showing how different threshold values affect the performance of the prover; in general, low thresholds severely degrade cache performance.

4.2 Storing Templates and Solutions

To conserve cache storage and to minimize the effective branching factor of the search space, a solution subsumed by an entry already in the cache store is not entered in the store. Subsumption of solution pairs is defined in Definition 4.4.

Definition 4.4 *If $S_1 = \langle \mathcal{H}, n_{\mathcal{H}} \rangle$ and $S_2 = \langle \mathcal{G}, n_{\mathcal{G}} \rangle$ are solution pairs then the pair S_1* subsumes *the pair S_2 if and only if $\mathcal{H}$ subsumes $\mathcal{G}$ (there exists a substitution σ with $\mathcal{H}\sigma = \mathcal{G}$) and $n_{\mathcal{H}} \leq n_{\mathcal{G}}$.*

The resource bound of a potentially subsumed solution pair must be compared with the resource bound of the subsuming pair to ensure that the subsuming pair is at least as general, e.g., that it will be retrieved from the cache in every context that the subsumed solution pair would be retrieved. If the resource bound $n_{\mathcal{H}}$ of a pair $\langle \mathcal{H}, n_{\mathcal{H}} \rangle$ is greater than the resource bound $n_{\mathcal{G}}$ of a pair $\langle \mathcal{G}, n_{\mathcal{G}} \rangle$ then both pairs must be stored in the cache even if $\mathcal{H}$ subsumes $\mathcal{G}$ since the solution $\mathcal{G}$ might be usable when $\mathcal{H}$ is not due to its lower resource requirements.

Goal templates must be provided for each goal seen in a deduction and updated when new information is obtained concerning a goal's completeness level or when a new (potentially minimal) solution is found. When a goal template is retrieved (line 3.1 in Figure 3) for a goal $\mathcal{G}$ that has not previously been seen, a template $\langle\mathcal{G}$,-1,$\infty\rangle$ is created and stored in the cache directory. Previously unseen goals must be solved by search. Each time a solution to the goal is found, the template is updated if the new solution requires fewer resources than the minimum currently registered in the template. The initial value ∞ ensures that the resources of the first solution found will be used correctly to update the template. When the **Solve** routine returns FALSE for a pair $\langle\mathcal{G},n\rangle$ (line 14 in Figure 2), a call is made to a cache directory updating procedure which registers that the template corresponding to $\mathcal{G}$ is now n-complete. The initial value -1 ensures that the cache store will not be used (line 3.2 in Figure 3).

Since the cache replaces search with a (hopefully) more efficient mechanism, and since previously unseen goals cannot use the cache, it is worth investigating methods that allow the search for solutions of an unseen goal to be replaced with a cache lookup. This is the motivation for the concept of *template-subsumption* (defined in Definition 4.5): to allow the cache to be used when a specific goal is encountered for the first time with a given resource bound.

When a goal pair $\langle\mathcal{G},n\rangle$ is encountered for the first time, it is possible that the cache is m-complete for a more general goal with $m \geq n$. In this case the cache may be used instead of the regular search mechanism; we say that the pair $\langle\mathcal{G},n\rangle$ is *template-subsumed* as defined in Definition 4.5.

Definition 4.5 *If cache template* $T = \langle\mathcal{H},m,m_S\rangle$ *and goal pair* $G = \langle\mathcal{G},n\rangle$ *then* T template-subsumes G *if and only if* $\mathcal{H}$ *subsumes* $\mathcal{G}$ *and* $m \geq n$.

If such a goal pair $\langle\mathcal{G},n\rangle$ is template-subsumed by a cache template $\langle\mathcal{H},m,m_S\rangle$, then since the cache is m-complete for the more general goal $\mathcal{H}$ all solutions of $\mathcal{H}$ are stored in the cache. These solutions are a superset of the solutions of $\mathcal{G}$ given that $\mathcal{H}$ subsumes $\mathcal{G}$ and that the resource bounds m and n satisfy the constraints of Definition 4.5. Thus the cache is m-complete for the goal pair as well and the cache replaces search.

In practice there may be more than one subsuming template for a given goal pair $\langle\mathcal{G},n\rangle$. In *METEOR* (potentially) all such subsuming templates are examined and the template with the largest m_S (minimal solution depth) is returned as the applicable template on line 3.1 of Figure 3. This ensures that if a subsuming template can serve as a failure template such a template is used. Since failure templates enable a branch of the search tree to be pruned without making any inferences, they allow a potentially greater saving than results from using the cache in lieu of the normal search mechanism.

The use of template-subsumption in the cache is a parameter that the user of the system can set. Results in Section 7 indicate that the more frequent cache access enabled by template subsumption more than compensates for the increased lookup time of a subsuming template.

4.3 Heuristic Caching

As indicated in Figure 1, we have investigated an alternative to caching in which a cost other than discovery cost is incurred when a cached solution is used. We call the method *heuristic caching* and as shown in Section 7 we have found a domain in which its use yields substantial performance gains over the normal caching mechanism.

When a solution is retrieved from the cache, a cost is incurred as shown on line 4 of Figure 4. When caching recreates the search space this cost reflects the resources used in creating the solution. It is possible, however, to reduce the available resource by less than the creation cost for certain solutions. Using such a solution in effect permits a search beyond that constrained by the current resource bound. Thus deep proofs may be found at shallow depths. Of course charging less than the discovery cost can also permit many deep but fruitless paths to be searched as well. In general, it is difficult to identify those solutions that should be stored with a resource less than the discovery cost. In certain domains, however, it may be possible to treat all solutions uniformly and realize a substantial performance gain over caching.

We report on such a domain—function-free Datalog—in Section 7. Great care must be taken, however, when there is no bound on the number of solutions as there is in this domain since the increasing number of solutions generated by heuristic caching can overwhelm cache storage and increase the branching factor to the point that a proof may not be found.

5 Lemmaizing

The difficulties of caching for non-Horn problems and the potential for large caches for deep Horn problems have led us to investigate alternatives to caching that can decrease the storage requirements inherent in caching all solutions and that can allow deep proofs to be discovered at shallow depths. In this section we investigate one such approach characterized by the lower right corner of Figure 1, an approach we have called *lemmaizing.*

Lemmaizing differs from heuristic caching in that not all solutions are stored for a given goal, but only some (it is hoped relevant) solutions are stored thus augmenting rather than replacing search. Since lemmas are not needed for completeness, we may impose syntactic and semantic criteria in deciding which lemmas to retain. The idea is to store lemmas that are used to eliminate repeated subdeductions. In this sense the use of lemmas allows us to combine an aspect of bottom-up reasoning with the top-down reasoning in *METEOR*. By imposing strict criteria on lemmas we retain a complete inference theorem that will hopefully allow us to prove theorems otherwise unobtainable. In one of the first implementations of *ME* [13], this kind of lemma use was explored in an ad-hoc manner. No notion of lemma cost was explored and lemmas were not found to be useful in general since the potential for a shorter proof was not realized due to the increased branching factor induced by allowing lemmas as alternatives for extension.

5.1 Lemma Storage and Retrieval

If (repeated) stored solutions used in the proof of a goal incur a smaller cost than other solutions, the proof might be found more quickly. The intuition here is that these repeated solutions function as *lemmas*, they reflect useful information to be used without requiring that the information be rederived each time it is used and whose use can make a proof easier to understand as well as shorter. Identifying useful lemmas is a non-trivial task and an important one in automated reasoning systems [6]. We are only beginning to explore this use of stored solutions and report on several successful uses of lemmaizing in Section 7.

Consider treating lemmas as input clauses. If only "good" lemmas are placed in the lemma store, a proof may be found quickly. However, if all solutions are treated as input clauses the lemma store may be quickly overwhelmed with irrelevant lemmas whose use may generate still more irrelevant lemmas. We have used several syntactic and semantic criteria in determining what lemmas to store. The primary criterion we have employed is to limit the nesting depth of terms that may appear in lemmas. This is done in an attempt to circumvent the kind of combinatorial explosion in the number of lemmas and their size that the use of arbitrary lemmas permits.

In some domains (e.g., group theory) demodulators may be used to rewrite solutions to a canonical form. This reduces redundancy since subsumption checks permit rewritten solutions to be discarded that might otherwise appear (redundantly) in the lemma store. Demodulation also permits terms that might violate a syntactic criterion such as nesting depth to be rewritten to a form that does not violate the criterion. Although demodulation and resolution may not, in general, result in a complete proof procedure, we can impose any restrictions on lemmas (including demodulating them) and retain completeness in *ME*. We have experimented with demodulating lemmas in group and ring problems by including the complete set of rewrite rules as demodulators as well as using demodulators generated during the search to rewrite all lemmas; we report on several successful applications in this area in Section 7.

In the current system we use lexicographic recursive path ordering [10] based on a user-specified total ordering of symbols to determine if a rewrite rule applies. The lexicographic order is also used to rewrite orientable instances of unorientable rules such as $f(xy) = f(yx)$. This system is similar to the LEX demodulators used in *OTTER* [25] and is a feature of the unfailing Knuth-Bendix procedure [10]. The user may specify if a set of rewrite rules is to be used in addition to or independently of any dynamically generated demodulators.

6 Implementation

Because cache templates and solutions must be added at runtime, it is not feasible to compile cache entries in the same way that input clauses are compiled in *PTTP*. In *METEOR*, however, input clauses are compiled into a data structure that is subsequently interpreted by the theorem-proving engine(s). The cache can be compiled into a similar structure so that once a cached solution is retrieved, making an inference with it is no more expensive than making an inference with an input clause.

Despite this efficiency, some pruning of the solutions retrieved from the cache must be made or the normally high inference rate obtainable in *METEOR* would decrease due to a large number of unsuccessful attempts to unify goals with cached solutions. Caching and lemmaizing both require fast associative retrieval of terms suitably related to goals to achieve this efficiency. This necessitates some type of term indexing as is often employed in Prolog implementations and other theorem proving systems [35]. In our system we employ a modified *trie* [17]. When used to store terms and expressions in this manner, tries are often referred to as discrimination trees or nets; variations of these structures have been employed in many different theorem proving systems [24, 14, 9]. Discrimination trees are especially good for retrieving generalizations [24, 35].

In our system we have employed compression techniques similar to those used in a PATRICIA trie [17]. In addition, many runtime parameters can be used to experiment with different cache structures. In the results reported in this paper the default settings are used except as noted in Section 7. For a complete description of our cache implementation see [1].

7 Results

In this section we include results for a variety of problems we have run using the caching and lemmaizing methods outlined in the previous sections. Rather than give an exhaustive set of results based, for example, on the problems reported in [34]; we include problems that have been historically difficult for *ME* based provers.

We have used heuristic caching in proving SAM's lemma [41] and achieved spectacular results for top-down or *ME* theorem provers. Although Otter solves the same formulation of SAM's lemma in about seven seconds, it has been an intractable problem for provers not employing some form of redundancy control. In the formulation we use, the input clauses for this problem are from the domain of function free Datalog problems. For this and other Datalog-like problems storing all solutions but charging unit retrieval cost seems a promising method.

We have also experimented with lemmas in several group theory problems. By imposing limits on the nesting depth of function symbols that appear in lemma terms (and by use of demodulation) we have been able to prove both the commutator problem and the theorem that if $x^2 = x$ in a ring then the ring is commutative [41] whose proofs have, heretofore, been unobtainable by top-down *ME* theorem provers.

We also note a successful proof of the intermediate value theorem of calculus (as formulated in [40]). This non-Horn problem is proved using lemmaizing and retaining all lemmas with a nesting depth of less than six. To our knowledge, this problem has been beyond the capabilities of linear provers.

All the results in this section are based on running an unoptimized version of *METEOR* (in the sense that the compiler debug rather than optimize flags were set) on a Sun SPARC-station 2 with 64 megabytes of memory.

In Figure 5 the label "fail.temp." means that the cache was used only for pruning using failure templates (with template subsumption used); "cache" is based on the best cache threshold over several runs (see Figure 6) with template subsumption used; "unit lemma" indicates that only solutions with function symbols nested at most 1 were stored and retrieved with unit cost; and "demod" indicates the same run but with lemmas rewritten using the complete set of reductions for free groups as well as with any demodulators meeting the nesting criterion generated during the proof.

Each run indicates the number of seconds needed to find the proof (the top number) and the number of successful inferences made.

problem	$\mathbf{D}_{\text{inf}}$	**fail. temp.**	**cache**	**unit lemma**	**demod.**
wos10	13.06	26.51	3.72	6.40	2.73
	78,669	129,643	10,714	19,562	7,979
wos 1	19.64	35.96	6.39	316	0.48
	139,068	223,455	10,551	1,221,686	1,273
wos21	283.43	840.2	85.51	584	39.73
	2,200,583	5,397,293	368,426	2,134,087	132,307
wos15	13,841		1,356	29.8	4.37
	91,879,275		5,399,388	104,883	15,701
sam	10^{14}‡		280.37	40.83†	
	$5(10^{17})$‡		948,444	155,480	
wos22	11,388		1,565		
	71,143,961		7,217,820		

†heuristic caching
‡projected measure

Figure 5: Results of caching.

In Figure 6 results are given using different cache thresholds, i.e., varying the minimum level at which the cache is consulted rather than relying solely on the normal search mechanism. Statistics are given in seconds and number of inferences. These results show that a low threshold uniformly degrades cache performance; the higher inference rate of the normal search procedure more than compensates for the reduction in inferences. Note that low thresholds also preclude the use of the identical ancestor pruning rule in more cases (see above). When these results are examined in light of the depth of search needed to find a proof (the number of steps in a proof found using D_{inf}) we see that there is a threshold window such that for runs made within the window performance increases as the threshold increases (note, for example, that for wos21 a threshold of 8 results in a 540,000 inference proof found in 93 seconds and a threshold of 9 results in a 1 million inference proof found in 154 seconds). Although the user can set the threshold, the default threshold used in *METEOR* is five.

	Cache Threshold Level						
problem	**1**	**2**	**3**	**4**	**5**	**6**	**7**
wos10	4.87	4.66	4.12	3.72	4.45	6.54	8.83
	8,482	8,482	8,482	10,714	17,563	31,698	46,561
wos 1	11.57	9.43	7.12	6.39	6.68	7.96	13.11
	8,499	8,499	8,499	10,551	14,136	23,059	49,582
wos21	492	483	358	226	133	96	86
	133,536	133,536	139,843	156,766	203,618	292,732	368,426
wos15	31,822	31,043	16,203	5,720	2,141	1,515	1,356
	1,507,114	1,507,114	1,509,839	1,721,907	2,413,466	3,765,384	5,399,388
sam†	42	42	41	41	43	52	
	126,650	126,650	127,451	130,328	156,433	242,347	
wos22	36,078	35,009	17,288	6,075	2,504	1,686	1,565
	1,846,619	1,846,619	1,921,009	2,280,075	3,231,997	4,942,331	7,217,820

†heuristic caching

Figure 6: Using different cache thresholds.

When template subsumption is not employed, the cache stores exactly the same number of solutions, but is accessed less frequently. In addition the number of templates stored greatly increases further degrading performance. Figure 7 gives statistics for the same problems and parameters as given in Figure 6, but without employing template subsumption.

	Cache Threshold Level (no template subsumption)				
problem	**2**	**3**	**4**	**5**	**6**
wos10	25.39	21.21	15.72	13.83	14.85
	39,036	39,032	42,570	51,293	68,088
wos 1	25.77	16.83	11.49	10.44	10.61
	15,953	15,953	18,576	27,330	35,964
wos21	937	727	455	285	184
	275,448	284,512	319,149	392,029	510,506
sam†	70	71.7	72	64	67.7
	239,251	240,044	254,946	268,438	355,197
wos22			51,403	18,823	8,398
			8,452,066	9,966,389	14,083,812‡

†heuristic caching
‡4,754 secs. and 17,191,011 inferences at threshold 7

Figure 7: Different cache thresholds (no template subsumption).

Figure 8 shows results from running several group theory problems using demodulators generated during the proof in addition to using the standard demodulators for free groups. No back demodulation was employed and all terms appearing in lemmas were restricted by limiting the level to which function symbols could be nested.

	Dynamic Demodulation				
problem	**time (secs)**	**inferences**	**# proof steps proof depth**	**# stored lemmas**	**lemmas generated**
wos10	2.73	7,979	7/8	38	7,101
wos 1	0.48	1,273	7/7	10	1,585
wos21	39.73	132,307	9/9	33	92,640
wos15	4.37	15,701	7/9	21	12,872
commutator	430.7	1,281,052	7/9	417	611,148
$x^2 = x$ ring	1,495	4,763,795	5/10	170	2,349,653

Figure 8: Using demodulation with lemma generation.

In Figure 9 results are given for lemmaizing with several non-Horn problems. The results are given as number of seconds and number of inferences needed to solve the problems. The problem labeled ivt is the intermediate value theorem. As noted above, this problem has been beyond the range of top-down, linear provers. It is proved by the STR+VE prover [7] and the HD-PROVER in [40]. With the addition of several non-automatically constructed rewrite rules it is proved by the prover in [30]. The problem labeled nonobv is a problem given in [27] and subsequently cited in [23]. We should note that when D_{Alit} is used as the depth measure *METEOR* solves this problem in under one second. The problem labeled salt is Lewis Carrol's salt and mustard logic puzzle.

METEOR and *PTTP* are attractive inference engines because of minimal memory requirements and because of the high inference rate. One of the potential drawbacks incurred by caching and lemmaizing is the (potentially large) size of the cache and lemma store. Figure 10 shows the memory requirements for the problems reported in this section.

For problems that are solved quickly, the requirements are quite modest which makes

problem	normal search	lemmaizing
ivt D_{Alit}		915
		3,216,208
nonobv D_{inf}	657	1.42
	6,526,914	12,071
salt D_{Alit}	60.3	35.7
	660,774	309,434

Figure 9: Lemmaizing with non-Horn problems.

problem	# solutions	# trie nodes	size (Mbytes)
wos10	473	1,300	0.05
wos1	2,301	7,269	0.38
wos21	8,867	28,428	1.5
sam†	143	407	0.013
wos15	55,076	176,037	10.2
wos22	79,871	243,227	15.38
commutator‡	417	1,094	0.48
$x^2 = x$ ring‡	170	469	0.19
ivt‡	91	289	0.14
nonobv‡	31	80	0.002

†heuristic caching
‡lemmaizing

Figure 10: Memory requirements for several problems.

caching and lemmaizing attractive methods for this class of problem. For hard problems, the memory requirements needed to store all solutions as is done in caching makes lemmaizing an attractive alternative. Of course lemmaizing requires the identification of useful lemmas which is a difficult task itself.

8 Related Work

Much work has been done in the area of query optimization for deductive databases [3]. This work tends to focus on reducing redundant (recursive) derivations by program transformation techniques [4], by introducing a control language [15, 16], and by run-time analysis [38]. In general, these techniques are designed to work with function-free, Horn (Datalog) programs. As our results with SAM's lemma indicate, caching and heuristic caching can work well for this class of problem. The framework of SLD-AL resolution [39] is closely related to our framework, the concept of a lemma in SLD-AL resolution corresponds exactly to (and is antedated by) the use of lemmas in model elimination; the QSQR implementation [38] of SLD-AL resolution also uses iterative deepening. The OLDT resolution procedure [37] is very closely related and involves an iterative deepening search of Datalog programs. As database optimizations, these methods concentrate on reducing redundancy when all solutions to a goal are desired; in a theorem proving context we (usually) only search for one proof.

Extension tables as used in [11] are closely related to the OLDT procedure. Although an outline of an iterative deepening prover is given there, no empirical data is given and it appears that the method has not yet been implemented. Plaisted [29] has implemented a theorem prover in which solved goals are stored although no notion of cache completeness is used. Although he reports some favorable results, caching in his prover could lead to longer proofs, did not work for the same class of problems we report on here, and did not admit proofs for problems previously unprovable in his system. In fairness, his implementation was not optimized and access to the store of solved goals can be particularly slow in the system employed in the prover.

Elkan [12] reports on the idea of caching to reduce redundancy in a resolution based prover for Horn problems, but only reports on the use of caching to solve one problem. His prover has been used in the realm of explanation based learning and the use of the prover with what we call lemmaizing is reported in [32]. The EBL domain is slightly different since the principle aim is to "train" the prover by storing solutions for a class of problems and then use these solutions to solve other problems of the same type. The only kinds of lemmas stored in the system are generalizations of goals. This makes it possible to prune the search with success when a match is found with a stored solution. Our work indicates that this methodology is of limited success for the kinds of problems traditionally addressed in theorem proving domains.

9 Conclusions

We have outlined two modifications to the *ME* search mechanism used in *METEOR*. These modifications, caching and lemmaizing, have enabled *METEOR* to prove theorems previously unobtainable by top-down model elimination theorem provers and have reduced by more than an order of magnitude the time required to prove some typically difficult theorems.

Other work in this area has focused almost exclusively on lemmaizing. We have studied, and shown to be successful, a method (caching) that consciously replaces search rather than augmenting it. In our implementation we have succeeded not only in reducing the number of inferences (which is easy and guaranteed for exhaustive searches), but in reducing the time required to find reduced-inference proofs, which is not so easy. The volume of data caching and lemmaizing uses demands indexing schemes unnecessary for ordinary *ME*; adding and using such schemes in an already fast theorem prover indicates not only the promise of the methods, but the versatility of our prover.

In the future we hope to address optimizations to the caching mechanism that will increase its efficiency both in terms of storage requirements and in its redundancy reducing capabilities thus permitting caching to be applicable to a larger class of problem. We also plan to investigate methods for identifying useful lemmas that will allow us to combine aspects of bottom-up reasoning with the goal-directedness of top-down provers in solving both Horn and non-Horn problems.

References

[1] O.L. Astrachan. *Investigations in Theorem Proving based on Model Elimination.* PhD thesis, Duke University, 1992. (expected).

[2] O.L. Astrachan and D.W. Loveland. METEORs: High Performance Theorem Provers using Model Elimination. In R. Boyer, editor, *A Festschrift for W.W. Bledsoe.* Kluwer Academic Publishers, 1991.

[3] F. Bancilhon and F. Ramakrishnan. Performance Evaluation of Data Intensive Logic Programs. In J. Minker, editor, *Foundations of Deducative Databases and Logic Programming*, chapter 12, pages 439–517. Morgan Kaufmann, 1988.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the 6th Symposium on Principles of Database Systems*, pages 269–283, 1987.

[5] W. W. Bledsoe. Challenge Problems in Elementary Calculus. *Journal of Automated Reasoning*, 6(3):341–359, 1990.

[6] W.W. Bledsoe. Some thoughts on proof discovery. In *IEEE Symposium on Logic Programming*, pages 2–10, 1986.

[7] W.W. Bledsoe and L. Hines. Variable Elimination and Chaining in a Resolution-Based Prover for Inequalities. In *Fifth Conference on Automated Deduction*, pages 281–292. Springer-Verlag, 1980.

[8] S. Bose, E. Clarke, D. E. Long, and S. Michaylov. Parthenon: A parallel theorem prover for non-Horn clauses. In *Symposium on Logic in Computer Science*, 1989.

[9] J. Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 1992. (to appear).

[10] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[11] S. W. Dietrich. Extension tables: Memo Relations in Logic Programming. In *IEEE Symposium on Logic Programming*, pages 264–272, 1987.

[12] C. Elkan. A conspiratorial and caching and/or tree searcher for theorem-proving. In *International Joint Conference on Artifical Intelligence*, 1989.

[13] S. Fleisig, D. Loveland, A. Smiley, and D. Yarmash. An implementation of the model elimination proof procedure. *Journal of the Assocation for Computing Machinery*, 21:124–139, January 1974.

[14] S. Greenbaum. *Input Transformations and Resolution Implementation Techniques for Theorem Proving*. PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[15] A. R. Helm. Detecting and eliminating redundant derivations in logic knowledge bases. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 145–161. Elsevier Science Publishers, 1990.

[16] A. R. Helm. On the Elimination of Redundant Derivations During Execution. In Saumya Debray and Manuel Hermenegildo, editors, *North American Conference on Logic Programming*, pages 551–568, 1990.

[17] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[18] R. E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Articial Intelligence*, 27:97–109, 1985.

[19] R. Letz, S. Bayerl, J. Schumann, and W. Bibel. SETHEO — a High-Performance Theorem Prover. (to appear in Journal of Automated Reasoning).

[20] D. W. Loveland. Mechanical theorem proving by model elimination. *Journal of the Assocation for Computing Machinery*, 15(2):236–251, April 1968.

[21] D. W. Loveland. A simplified format for the model elimination procedure. *Journal of the Assocation for Computing Machinery*, 16(3):349–363, July 1969.

[22] D. W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, 1978.

[23] R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In *Ninth International Conference on Automated Deduction*, pages 415–434. Springer-Verlag, 1988.

[24] W. McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Technical Report MCS-P191-1190, Mathematics and Computer Science Division Argonne National Laboratory, January 1991. (to appear in Journal of Automated Reasoning).

[25] W. W. McCune. *OTTER 2.0 Users Guide*. Argonne National Laboratory, March 1990.

[26] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.

[27] F.J. Pelletier and P. Rudnicki. Non-obviousness. *AAR Newsletter*, (6):4–5, 1986.

[28] D. Plaisted. Non-Horn Clause Logic Programming without Contrapositives. *Journal of Automated Reasoning*, 4(3):287–325, September 1988.

[29] D. Plaisted. A sequent style model elimination strategy and a positive refinement. *Journal of Automated Reasoning*, 6(4), 1990.

[30] D. Plaisted and S.-J. Lee. Inference by clause linking. Technical Report TR90-022, University of North Carolina, Department of Computer Science. Chapel Hill, NC, 1990.

[31] J. Schumann and R. Letz. PARTHEO: A High Performance Parallel Theorem Prover. In *Tenth International Conference on Automated Deduction*, pages 40–56, 1990.

[32] A. Segre and Scharstein D. Practical caching for definite-clause theorem proving. [draft], September 1991.

[33] M.E. Stickel. A Prolog technology theorem prover. *New Generation Computing*, 2(4):371–383, 1984.

[34] M.E. Stickel. A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler. *Journal of Automated Reasoning*, 4:343–380, 1988.

[35] M.E. Stickel. The path-indexing method for indexing terms. Technical Report 473, SRI International, Artificial Intelligence Center, October 1989.

[36] M.E. Stickel and W.M. Tyson. An analysis of consecutively bounded depth-first search with applications in automated deduction. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 1073–1075, August 1985.

[37] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, 1986.

[38] L. Vieille. Recursive axioms in deductive databases: the query/subquery approach. In *Proceedsings 1st International Conference on Expert Database Systems*, pages 179–193, 1986.

[39] L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.

[40] T.C. Wang and W.W. Bledsoe. Hierarchical Deduction. *Journal of Automated Reasoning*, 3:35–77, 1987.

[41] L. Wos. *Automated Reasoning 33 Basic Research Problems.* Prentice Hall, 1988.

[42] L. Wos and R. Overbeek. Subsumption, a sometimes undervalued procedre. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson.* MIT Press, 1991.