Caching and Lemmaizing in Model Elimination Theorem Provers*

Owen L. Astrachan[1] and Mark E. Stickel[2]

[1] Department of Computer Science, Duke University, Durham, NC 27706, ola@cs.duke.edu
[2] 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*) [17, 19] is a complete inference procedure for the first-order predicate calculus. It is the method underlying the Prolog Technology Theorem Prover (*PTTP*) [28, 29], the *SETHEO* prover [16], and several or-parallel theorem provers [26, 9, 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 [29] when *METEOR* is run using inference count as the depth measure (see Section 3).

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

* 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.

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 [7] that are difficult if not unobtainable for *OTTER* [21], 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 [26, 9, 2], though they have not produced a proof of a theorem infeasible to obtain by running the prover on a single processor.

In general, the lack of both a redundancy-control mechanism such as subsumption and a best-first search methodology are severe impediments to finding deep proofs. Many theorems obtainable by *OTTER* cannot be proven in our systems because the size of the search space and the lack of redundancy control quickly overwhelm the high inference rate. In this paper we report on the modification of 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. These modifications have enabled *METEOR* to prove theorems not previously obtainable by top-down *ME* provers and can reduce by more than an order of magnitude the time required to find proofs of some "difficult" theorems. We use the two-dimensional grid in Fig. 1 to categorize our approaches.

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

Fig. 1. changing the search mechanism.

In a broad sense, the cost referred to in Fig. 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 [30, 15] 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.

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. 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 inferences 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. 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*[3] 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*. Our modification to this mechanism are described in Section 4 and Section 5. Results generated using these modifications are given in Section 6. Related work is outlined in Section 7 and conclusions presented in Section 8. A more detailed account of the work presented here can be found in [3].

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 [19], which also gives a complete description of the model elimination procedure. We use Prolog notation in which variables are repre-

[3] Although the juxtaposition of vowels in this word may be inharmonious, recall memoizing [22] used to mean essentially what we call caching.

sented by capital letters and functions, constants and predicates are represented by lowercase letters.

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* <u>extend($C_1, l_1, C_2, l_2, \theta$)</u> *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* <u>reduce(C_1, l_1, l_2, θ)</u> *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 [19]. 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 [30, 15] to ensure completeness of the search 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. 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* [12]. 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.

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 Fig. 2.

Several optimizations can be applied in the search mechanism without affecting its completeness [29]. 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* [17, 18, 19]. The most effective of these tends to be the *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.

```
boolean
Solve(chain 𝒞,resource n)
[0]  if 𝒞 is the empty chain then
          return TRUE
[1]  goal 𝒢 ← leftmost literal in 𝒞
[2]  R ← A-literals of 𝒞 potentially unifiable with complement of 𝒢 (for reductions)
[3]  E ← input clauses with literals potentially unifiable with complement of 𝒢 (for extensions)
               /* try to solve ⟨𝒢,n⟩ */
[4]  for each l_R in R do
[5]       n_new ← resources available if reduction made
[6]       if n_new ≥ 0 and l_R and complement of 𝒢 unify with mgu θ then
[7]            if Solve(reduce(𝒞, 𝒢, l_R,θ),n_new) then
[8]                 return TRUE
     endfor (reduction)
[9]  for each clause C in E with literal l_C do
[10]      n_new ← resources available if extension of 𝒞 with C is made
[11]      if n_new ≥ 0 and and l_C and complement of 𝒢 unify with mgu θ then
[12]           if Solve(extend(𝒞,𝒢,C,l_C,θ), n_new) then
[13]                return TRUE
     endfor (extension)
[14] return FALSE
```

Fig. 2. the search procedure.

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 Fig. 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 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 [24] 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 6.

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* cached solution *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 [11]; 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. When caching is used, the modifications indicated in Fig. 3 are made to the search routine of Fig. 2.

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

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

```
      boolean
      CacheSolve(C,n)
          {G is the leftmost literal in C}
[1]   L ← all solution pairs ⟨G′,n_G′⟩ such that G′
       is potentially unifiable with G and
       such that n_G′ ≤ n

[2]   for each ⟨G′,n_G′⟩ in L do
[3]    if G and G′ unify with mgu θ then
[4]     n_new ← n − n_G′
[5]     if Solve(extend(C,G,G′, G′, θ), n_new) then
[6]      return TRUE
      end for
[7]   return FALSE
```

Fig. 3. determining if the cache should be used.

In its implementation in *METEOR*, the code in Fig. 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 solution storage and template retrieval. Because the identical-ancestor pruning rule is so effective (and must be partially disabled when caching is used—see Section 4) 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 6 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 <u>subsumes</u> *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* $n_{\mathcal{H}} \leq n_{\mathcal{G}}$*.*

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.

Since the cache replaces search with a (it is hoped) 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.

The use of template-subsumption in the cache is a parameter that the user of the system can set. Results 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

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 6, 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 Fig. 3. When caching recreates the search space this cost reflects the cost used in creating the solution. Consider charging some other cost, e.g., a cost less than that used to discover the solution. In such a case a solution may be used whose discovery cost exceeds the current cost bound. Using such a solution in effect permits a search beyond that constrained by the current cost bound. 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 cost 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 in Section 6.

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 high-cost proofs to be discovered with low cost bounds. In this section we investigate such 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* [12], 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 solutions used in the proof of the top goal incurred 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. 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 6. 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.

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 a 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 6. Details of the rewriting techniques employed can be found in [3].

6 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. Detailed accounts of the implementation of these methods can be found in [1, 3] Rather than give an exhaustive set of results based, for example, on the problems reported in [29]; we include problems that have been historically difficult for *ME* based 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.

We have used heuristic caching in proving SAM's lemma [35] 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 [35] 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 [34]). 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.

In both Tables 1 and 2 each cell of the table indicates the number of seconds needed to find the proof (the top number) and the number of successful inferences made.

In Table 1 the columns indicate (respectively) the problem name, statistics using the default (D_{inf}) search strategy, using failure templates only, the best caching results over several runs (see Table 2), using all solutions with function symbols nested at most one as lemmas with unit cost, and rewriting these lemmas using using a complete set of reductions as well as demodulators generated during the proof.

In Table 2 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. Although the user can set the threshold, the default threshold used in *METEOR* is five.

Table 3 shows results from running several group theory problems using demodulators generated during the proof in addition to using the standard de-

Table 1. results of caching.

problem	D_{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

Table 2. using different cache thresholds.

	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

modulators 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.

In Table 4 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 [8] and the *HD-PROVER* in [34]. With the addition of several non-automatically constructed rewrite rules it is proved by the prover in [25]. The problem labeled nonobv is a problem given in [23] and subsequently cited in [20]. 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 po-

Table 3. using demodulation with lemma generation.

	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

tential drawbacks incurred by caching and lemmaizing is the (potentially large) size of the cache and lemma store. Table 4 shows the memory requirements for the problems reported in this section. For problems that are solved quickly, the

Table 4. non-Horn problems

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

memory requirements

problem	# sol's	# 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

requirements are quite modest which makes 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.

7 Related Work

Much work has been done in the area of query optimization for deductive databases [4]. This work tends to focus on reducing redundant (recursive) derivations by program transformation techniques [5], by introducing a control language [13, 14], and by run-time analysis [32]. 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 [33] 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 [32] of SLD-AL resolution also uses iterative deepening. The OLDT resolution procedure [31] 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 [10] 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 [24] 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 [11] 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 [27].

8 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 Model Elimination Based Theorem Proving*. PhD thesis, Duke University, 1992. (also technical report CS-1992-21).
2. O.L. Astrachan and D.W. Loveland. METEORs: High performance theorem provers using model elimination. In R.S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Kluwer Academic Publishers, 1991.
3. O.L Astrachan and M.E. Stickel. Caching and lemmaizing in model elimination theorem provers. Technical Report 513, SRI International, Menlo Park, CA, December 1991.
4. F. Bancilhon and F. Ramakrishnan. Performance evaluation of data intensive logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 12, pages 439–517. Morgan Kaufmann, 1988.
5. 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.
6. W.W. Bledsoe. Some thoughts on proof discovery. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 2–10, 1986.
7. W.W. Bledsoe. Challenge problems in elementary calculus. *Journal of Automated Reasoning*, 6(3):341–359, 1990.
8. W.W. Bledsoe and L. Hines. Variable elimination and chaining in a resolution-based prover for inequalities. In *Proceedings of the Fifth Conference on Automated Deduction*, pages 281–292. Springer-Verlag, 1980.
9. S. Bose, E. Clarke, D.E. Long, and S. Michaylov. Parthenon: A parallel theorem prover for non-Horn clauses. In *Proceedings of the Symposium on Logic in Computer Science*, 1989.
10. S.W. Dietrich. Extension tables: Memo relations in logic programming. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 264–272, 1987.
11. C. Elkan. A conspiratorial and caching and/or tree searcher for theorem-proving. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, 1989.
12. S. Fleisig, D. Loveland, A. Smiley, and D. Yarmash. An implementation of the model elimination proof procedure. *Journal of the Association for Computing Machinery*, 21:124–139, January 1974.
13. 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.
14. A.R. Helm. On the elimination of redundant derivations during execution. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 551–568, 1990.
15. R.E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
16. R. Letz, S. Bayerl, J. Schumann, and W. Bibel. SETHEO—a high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.
17. D.W. Loveland. Mechanical theorem proving by model elimination. *Journal of the Association for Computing Machinery*, 15(2):236–251, April 1968.
18. D.W. Loveland. A simplified format for the model elimination procedure. *Journal of the Association for Computing Machinery*, 16(3):349–363, July 1969.
19. D.W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, 1978.

20. R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in Prolog. In E. Lusk and R. Overbeek, editors, *Proceedings of the Ninth International Conference on Automated Deduction*, pages 415–434. Springer-Verlag, 1988.
21. W. McCune. *OTTER 2.0 Users Guide*. Argonne National Laboratory, March 1990.
22. D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.
23. F.J. Pelletier and P. Rudnicki. Non-obviousness. *AAR Newsletter*, (6):4–5, 1986.
24. D. Plaisted. A sequent style model elimination strategy and a positive refinement. *Journal of Automated Reasoning*, 6(4), 1990.
25. 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.
26. J. Schumann and R. Letz. PARTHEO: A high performance parallel theorem prover. In *Proceedings of the Tenth International Conference on Automated Deduction*, pages 40–56, 1990.
27. A. Segre and D. Scharstein. Practical caching for definite-clause theorem proving. [draft], September 1991.
28. M.E. Stickel. A Prolog technology theorem prover. *New Generation Computing*, 2(4):371–383, 1984.
29. M.E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:343–380, 1988.
30. 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.
31. H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, 1986.
32. L. Vieille. Recursive axioms in deductive databases: the query/subquery approach. In *Proceedings of the 1st International Conference on Expert Database Systems*, pages 179–193, 1986.
33. L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69(1):1–53, 1989.
34. T.C. Wang and W.W. Bledsoe. Hierarchical deduction. *Journal of Automated Reasoning*, 3:35–77, 1987.
35. L. Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice Hall, 1988.