SRI International

Technical Note 513 • December 1991

Caching and Lemmaizing in Model Elimination Theorem Provers

Prepared by:

Owen L. Astrachan
Department of Computer Science
Duke University
Durham, North Carolina 27706

Mark E. Stickel
Principal Scientist
Artificial Intelligence Center
Computing and Engineering Sciences Division

This research was supported by the National Science Foundation under Grant CCR-8922330. The views and conclusions contained herein are those of the author 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.

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (415) 326-6200 • FAX: (415) 326-5512 • Telex: 334486

Abstract

Theorem provers based on the model elimination theorem-proving procedure 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*) [22, 23, 24] is a complete inference procedure for the first-order predicate calculus. It is the method underlying the Prolog Technology Theorem Prover (*PTTP*) [38, 39], the *SETHEO* prover [21], and several or-parallel theorem provers [35, 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* [2] 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 runtime 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 [39] when *METEOR* is run using inference count as the cost 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 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* [27], a prover that 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 [35, 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 [49] and a best-first search methodology are severe impediments to finding deep proofs. Many theorems obtainable by *OTTER* [27] 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 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 [41, 20] 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. Concretely, the number of steps in the solution of a goal (i.e., the length of its proof or the size of its proof tree) or the maximum depth of its proof tree may be bounded. 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 increase 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 .

The potentially large number of goals and solutions in caching or lemmas in lemmaizing poses the problem of large numbers of formulas not unifiable with a goal being found in the cache or among the lemmas if they are simply stored in linear lists. The resulting slowdown would overcome any possible saving from a reduced number of inference operations. Successful implementations of caching and lemmaizing must provide efficient means to reduce the number of doomed attempts to unify goals with solutions in the cache or lemmas. It is easy to use caching or lemmas to reduce the number of inferences required to find a proof. Reducing the total time as well requires that cache or lemma lookup costs be minimized. We have succeeded in obtaining substantial reductions in time as well as inference count. This requires efficient cache/lemma storage and retrieval and restrictions on their use. For example, cache lookup is not used if the cost bound is low, since such shallow searches can be completed quickly without caching, and lemmas may be restricted in size, or be demodulated.

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 characterize our modifications to this search mechanism. Caching is described in Section 4 and lemmaizing in Section 5. In Section 6 we describe the implementation of these modifications; results generated using

[1] Although the juxtaposition of vowels in this word may be inharmonious, recall memo-izing [30] used to mean essentially what we call caching.

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 [24], 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.

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. Intuitively, as the deduction progresses the chain is extended (more literals may be added) or reduced (literals are removed) until the empty chain is generated, signifying that the initial set of clauses is unsatisfiable. The literals in a chain are a set of *goals* to be refuted. We will speak of them being solved, since removal of a goal by a sequence of *ME* inference operations also constitutes a proof (relative to the rest of the chain) of the complement of the goal. Literals in a chain are either *B-literals* or *A-literals*. An A-literal has been used in an extension operation and may participate in the *ME* reduction operation. A-literals represent ancestor goals of all the literals to their left in the chain.

The *ME* procedure begins with some designated input clause as the initial chain. The leftmost literal in this chain is unified with a literal of opposite sign (i.e., a positive literal (atom) if the leftmost literal in the chain is a negative literal (negated atom) and vice versa) in 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 $l_1 C_0$ with leftmost B-literal l_1 and input clause C_2 with literal l_2 of opposite sign to l_1 such that the atoms of l_1 and l_2 are unifiable with most general unifier (mgu) θ, the ME extension 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 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

$\neg r(c,b)[q(f(a),c)][r(c,Z)]p(a,Z)$.

Besides applying the Prolog-like extension operation to the leftmost literal of a chain, the *ME* procedure also allows the leftmost literal, which is always a B-literal after mandatory contraction (see below) is applied, to be removed if it matches the complement of an A-literal. The reduction operation implements a form of reasoning by contradiction: if P is provable from $\neg P$ and Q, then it is provable from Q alone. Extension and reduction together comprise a sound and complete inference system for the first-order predicate calculus, not just the Horn clause subset handled by the extension operation alone as in Prolog.

Definition 2.2 *Given chain C_1 of the form $l_1 C_0$ with leftmost B-literal l_1 and A-literal l_2 of opposite sign to l_1 such that the atoms of l_1 and l_2 are unifiable with mgu θ, the ME* reduction *operation yields chain $C_0\theta$. We use the notation* reduce(C_1, l_1, l_2, θ) *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 $[q(f(a),c)][r(c,b)]p(a,b)$. Many people have been confused about reduction, thinking that if reduction can be applied, no other rule need be used.[2] In general, reduction is not a mandatory operation—the search for extensions or other reductions cannot be pruned after a successful reduction. However, in the special case when the unifying substitution is empty (e.g., in the propositional case), successful reduction does obviate the need to try other reduction or extension alternatives.

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 [24]. 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 whose use is outlined in Sections 4 and 5. Since we use and describe *ME* as a refutation procedure, an A-literal removed by contraction is refuted and its complement is considered proved and treated as a cached solution or lemma.

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.

[2] In [10] we see: "When an ordered resolvent is generated, we always check whether it is reducible. If it is, we always reduce it to the reduced ordered clause."

Rather than employ a breadth-first strategy with its exponential storage requirements, *METEOR* and *PTTP* use iterative deepening [41, 20] to ensure completeness of the search strategy. Iterative deepening is asymptotically optimal among brute-force search strategies[3] 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. For example, when the bound is d the initial chain has a cost bound of d; the resulting cost bound for each derived chain is computed in a manner dependent on the cost measure being used (see below). A chain's cost bound is used as a bound on the attempt to solve its leftmost-literal goal. We use the notation $(\mathcal{G},n)$ 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. In the description in Figure 2, we treat input clauses as unordered with each literal in a clause a potential candidate for extension. In practice, each n-literal clause is represented as n ordered clauses all of which have the same literals but distinct leftmost literals; only the leftmost literals are eligible for extension.

Several optimizations can be applied in the search mechanism without affecting its completeness [39]. Many of these are implicit in the definition of an acceptable chain and the accepting transformation that is applied to chains in the original presentation of *ME* [22, 23, 24]. 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 it must be partially disabled when caching is employed, in a manner described later.

If no proof is found during a stage of iterative deepening, the global cost bound is increased and the process repeated. To ensure that a minimal proof is found, the bound is incremented by 1 each time, unless the previous search demonstrates that a larger increment will be needed to find a proof. There are several possible cost measures; we mention two here. For a more detailed account of different measures, see [1].

[3]The optimality result applies only in the absence of redundancy control mechanisms such as subsumption. For example, breadth-first search would keep duplicate formulas and make further inferences from them instead of recognizing their redundancy and discarding them.

```
boolean
Solve(chain C,cost 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
```

Figure 2: The search procedure.

We envision the search tree as an and-or tree. An and-node represents an inference and has a branching factor equal to the number of literals introduced into the chain by the inference. These are and-nodes since each such literal must be removed in order to derive the empty chain. Extension with an n-literal clause produces an $(n-1)$-branching and-node, one for each literal introduced as a result of the extension. Note that reduction and extension with a unit clause result in a zero-branching and-node. Such a node closes off a path in the search tree since it effectively solves the branch that produced the node. An n-literal top goal is represented by an and-node with n branches since there are n literals that must be removed to derive the empty chain.

Each and-node branch corresponding to a literal $\mathcal{G}$ leads to an or-node whose potential branching factor is the number of clauses that are candidates for extension with $\mathcal{G}$ plus the number of A-literals that are candidates for reduction with $\mathcal{G}$. The actual branching factor is determined by the number of successful extension and reduction operations, each of which produces an and-node as described above. For purposes of calculating cost in the search tree (and thus the resources available to solve a goal), or-nodes are disregarded. Only and-nodes visited during a search affect cost because a (partial) proof is an and-subtree; or-nodes reflect alternative attempts at proofs.

In *PTTP*, cost is measured by the number of *ME* inferences performed; we call this measure

inference depth or D_{inf}. A naïve calculation of D_{inf} decrements the cost bound by 1 with each extension or reduction: $n_{\text{new}} \leftarrow n - 1$ on lines 5 and 10 in Figure 2. Thus, for example, a bound of 10 will permit the discovery of any proof with 10 or fewer extensions and reductions, i.e., any proof of length less than or equal to 10. Note that an extension with an n-literal clause yields an and-node requiring a minimum of $n - 1$ inferences to solve. Exploring below such a node when the cost bound is less than that makes little sense since the deduction cannot succeed. This leads to a *predictive* calculation of D_{inf} in which the bound is decremented as goals are added to the search tree instead of when they are extended or reduced. Line 10 in Figure 2 calculates n_{new} by

$$n_{\text{new}} \leftarrow n - (\text{number of literals in C} - 1)$$

for extension operations, and line 5 calculates n_{new} by

$$n_{\text{new}} \leftarrow n$$

for reduction operations. Note that $n_{\text{new}} = n$ for reduction operations and extension operations with unit clauses when the predictive calculation is used. This is because the minimum possible single inference required to solve a goal is subtracted from the bound when the goal is added to the search tree by an extension operation with a nonunit clause. This predictive method is used in *PTTP* and in calculating D_{inf} in *METEOR*.

As an alternative to D_{inf}, consider using the depth of the search tree as the cost measure (only and-nodes are used in calculating the depth of the tree). This metric is the default cost measure used in *SETHEO* and was used in one of the earliest implementations of *ME* [15]. The depth of the search tree corresponds to the number of A-literals present in the chain that represents the current state of the deduction; we call this *A-literal depth* or D_{Alit}. Note that, although when D_{inf} is the cost measure, each and-branch has fewer resources than the branch to its left (if the tree is traversed in left-to-right order), when D_{Alit} is used, every and-branch successor of a given node has the same resource. When D_{Alit} is used, for extension with a nonunit clause, we have $n_{\text{new}} \leftarrow n - 1$ since the extension produces one new A-literal (and an and-node one level deeper in the search tree). For extension with unit clauses and reduction, the value of n_{new} depends on the chain C since the number of A-literals may decrease (and cannot increase) as a result of these operations. The A-literals removed by the contraction operation in Figure 2 can produce and-nodes with a larger resource.

Using D_{inf} ensures that the length (number of steps) of the proof is minimized; using D_{Alit} ensures that the depth (in A-literals) of the proof tree is minimized. 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. Note that whereas the resources available when D_{inf} is used as the cost measure decrease monotonically with each inference (the resources available to solve a chain are less than or equal to the resources available to solve its parent), inferences can increase the resources available when D_{Alit} is used (extension by a unit clause

or reduction may result in a chain having fewer A-literals than its parent). This nonmonotonicity allows more inference steps and larger terms to be constructed, which can result in a problem of needing to store many large terms in the cache (see Section 5).

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 cost 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. For exhaustive searches, its use will never result in more inferences being made than when the cache is not used. 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.

In the grid of Figure 1, note that caching occupies a cell corresponding to use of "discovery cost". This is the amount by which the cost bound is decreased by the solution of the goal. If Solve(chain $\mathcal{C}$,cost n) leads to Solve(chain $\mathcal{C}'$,cost n') being called after the leftmost literal $\mathcal{G}$ of $\mathcal{C}$ is solved, the discovery cost of that solution of $\mathcal{G}$ is $n - n'$. For caching to reproduce the search space, it is important that cache lookup for $\mathcal{G}$ result in the same reduction in the cost bound n as a search for solutions would, i.e., the discovery cost of the solutions of $\mathcal{G}$ should be charged when solutions are looked up.

Caching is a sound and complete replacement for search. Every cached solution is discoverable by search: if solving the leftmost literal $\mathcal{G}$ of $\mathcal{C}$ by cache lookup in the call Solve(chain $\mathcal{C}$,cost n) results in Solve(chain $\mathcal{C}''$,cost n'') being called with leftmost literal $\mathcal{G}$ solved, then search leads to the same call on Solve (possibly differing only in the names of variables in $\mathcal{C}''$). Every solution discoverable by search is an instance of a cached solution: if searching for a proof causes the call Solve(chain $\mathcal{C}$,cost n) to lead to Solve(chain $\mathcal{C}'$,cost n') being called after the leftmost literal $\mathcal{G}$ of $\mathcal{C}$ is solved, then cache lookup of $\mathcal{G}$ will lead to Solve(chain $\mathcal{C}''$,cost n'') being called, where $\mathcal{C}''$ is equal to or a generalization of $\mathcal{C}'$ and $n'' \geq n'$. Caching ideally returns a minimal set of solutions, omitting duplicates and solutions that are less general or cost more than others. Solutions may be generated in different orders by searching and cache lookup, so a proof could conceivably be found

after more inferences with caching than without due to the potentially different order of solutions in the final iteration of the iterative deepening search. It is guaranteed, however, that no more inferences will be performed with caching than without for exhaustive searches. The cache is not normally used when the cost bound is small because of conflicts with the identical-ancestor pruning rule and because the overhead of a cache lookup may be too high compared to the time needed in *METEOR* to completely search a shallow tree.

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 (ancestor goals) are present in the chain. For further discussion of the problem of caching in non-Horn problems and one possible partial solution, see Section 4.4.

4.1 The Cache Mechanism

For a given pair $\langle \mathcal{G}, n \rangle$, it must be possible to determine if the cache should be used to solve the goal or if the regular search mechanism should be used. This decision is based both on the goal and on the current cost bound, since the set of solutions of a goal depends on the bound.

The cache store contains all the cached solutions. A cached solution consists of a substitution instance of the subgoal and the cost bound used in obtaining the solution.

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 cost 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 [14]; 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. Later we describe eliminating subsumed solutions from the cache store to reduce its size. This will require retrieving cached solutions unifiable with the goal instead of instances of it.

Our cache differs in use from the cache described in [32] 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 [13].

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

```
    /* added before line 4 in Figure 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 */
```

Figure 3: Determining if the cache should be used.

- The efficiency of the cache tends to decrease as the number of entries in it increases.

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

A-literals removed by the *ME* contraction operation represent solved goals and, therefore, potentially cacheable solutions. For Horn problems, contraction occurs when extension is made with a unit clause and when all the subgoals introduced by extension with a nonunit clause are solved. In each of these situations the (possibly instantiated) goal can be entered in the cache store with the cost used to solve the goal as a solution pair $\langle \mathcal{G}\theta, n_{\mathcal{G}\theta} \rangle$. To conserve cache storage and to minimize the effective branching factor of the search space, a solution subsumed by an entry $\langle \mathcal{G}', n_{\mathcal{G}'} \rangle$ in the cache store is not stored. 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 cost bound of a potentially subsumed solution pair must be compared with the cost 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 cost bound $n_{\mathcal{H}}$ of a pair $\langle \mathcal{H}, n_{\mathcal{H}} \rangle$ is greater than the cost 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, because of its lower resource requirements.

Consider the solution pairs S_1 = ⟨p(a),3⟩ and S_2 =⟨p(a),4⟩. There is no reason to store S_2 in the cache if S_1 is stored since, whenever the current cost bound allows S_2 to be retrieved from the cache, S_1 must be retrievable as well. Redundant search results if both solutions are retrieved.

```
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
```

Figure 4: Using an m-complete cache.

Consider the solution S_1 = ⟨p(X),3⟩ and the less general solution S_2 = ⟨p(a),3⟩. Again in every case that S_2 is retrieved from the cache and used successfully in a deduction, S_1 must be retrievable and useful as well, so only the more general solution should be stored to reduce redundancy in the search generated by cache retrievals.

Because of employing this subsumption technique, a cache lookup requires potentially unifiable solution pairs (as opposed to instance pairs) to be retrieved from the cache store (line 1 in Figure 4). Consider cache lookup for solutions of the goal p(a). If the solution pair S_2 = ⟨p(a),3⟩ is not stored in the cache because of the presence of the subsuming solution pair S_1 = ⟨p(X),3⟩, then the latter, which is unifiable with the goal, must be used by **CacheSolve** to solve the goal p(a).

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 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 ⟨$\mathcal{G}$,-1,∞⟩ 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 a lower cost than the minimum currently registered in the template. The initial value ∞ ensures that the cost of the first solution found will be used correctly to update the template. When the **Solve** routine returns FALSE for a pair ⟨$\mathcal{G}$,n⟩ (line 14 in Figure 2), a call is made to a cache directory updating procedure that 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 (it is hoped) a more efficient mechanism, and since previously unseen goals cannot use the cache, it is worth investigating methods that allow the search for solutions of a previously 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 cost bound if the cache contains solutions to a more general goal.

When a new template is constructed for the pair $\langle \mathcal{G},n \rangle$ (i.e., $\mathcal{G}$ has not been seen before), 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 cost 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 subsuming templates are examined and the template with the largest m_S (minimal solution cost) is returned as the applicable template on line 3.1 of Figure 3. The search for a subsuming template is stopped, however, if a template is found with $m_S > n$. 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.

In the current implementation, no new template is constructed when a goal pair is template-subsumed. It is possible that construction of a new template in such cases would be beneficial when the minimal solution cost for the subsumed goal is greater than that stored with the more general subsuming goal. Storing such a template would permit its subsequent retrieval as a failure template in more situations since the minimal cost bound is greater than that recorded with the subsuming template.

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

Caching is used to replace search and cannot result in more inferences than are made when the normal search mechanism is employed. This is true precisely because the cache is consulted only when it is complete for a given goal and because the use of a cached solution incurs a cost equal to the cost required to generate the solution using the normal search mechanism. 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, 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 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. For example, a solution whose discovery cost is ten but for which a cost of one is charged can be used twice in a proof when the the global cost bound is at least two whereas a cost bound of at least twenty is needed if caching is used. This method has the potential for finding proofs that would require a much higher cost bound than if caching or the normal search mechanism were used. 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 7.

In our experiments with heuristic caching, each retrieved cached solution incurs a cost less than or equal to the discovery cost of the solution. It is possible that some mechanism might identify certain cached solutions as extremely unpromising (i.e., their use would be ineffective in leading to a proof) and charge a cost greater than the discovery cost for such solutions. If effective, such a mechanism would prune unpromising paths in the search space. Identifying such solutions seems quite difficult; in the ensuing development heuristic caching refers to the method in which the cost incurred by the use of a cached solution is no greater than the discovery cost of the solution.

Just as caching substitutes cache lookup for search, heuristic caching also is used as a substitute for search rather than to augment search. For every solution returned by a normal cache lookup, a heuristic cache lookup returns the same solution (or a more general one) with the same or less cost. Unlike caching, however, heuristic caching may return extra solutions, i.e., solutions whose discovery cost exceeds the current cost bound. These extra solutions are then used to probe more deeply into the search tree in the sense that the potential exists for finding high-cost proofs with low cost bounds as noted above.

Whereas caching always returns the same set of solutions for a given goal pair, it is possible for heuristic caching to return an increasing number of solutions for the goal pair on successive cache lookups. This is due to the retrieval of a cached solution with high discovery cost stored in the cache as a solution with a low retrieval cost between successive cache lookups for the same goal pair. When the universe of instantiated goals is finite as it is in the function-free Datalog domain, there is a bound on the number of solutions. Great care must be taken, however, when there is no such bound, since the increasing number of solutions can overwhelm cache storage (see Section 5.1

and Figure 5) and increase the branching factor to the point that a proof may not be found.

4.4 Caching with Non-Horn Problems

Recall that the leftmost literal in a chain represents the current goal. In Horn problems, if we do not employ the identical-ancestor pruning rule, this goal occurs independently of the other literals in the chain since they are not used in inference sequences involving the goal. In non-Horn problems A-literals can contribute to the solution of a goal via the reduction operation. Thus for non-Horn problems a goal cannot be considered in isolation, but must be considered in the context of the A-literals in the chain. These A-literals constitute an environment in which attempts to solve a goal are made. Since a goal template is intended to provide information about possible solutions for a given goal, and solutions are based on this environment, a template must somehow convey information regarding the A-literals that constitute the particular environment of a potentially cached goal. We give an abstract definition of such a template and then briefly discuss methods of implementing this abstraction. Note that for Horn problems this definition reduces to Definition 4.2.

Definition 4.6 *A* non-Horn cache template *or* non-Horn template *for a non-Horn goal $\mathcal{G}$ is a quadruple $(\{A_{\mathcal{G}}\},\mathcal{G},m,m_S)$ that indicates that the cache is m-complete for goal $\mathcal{G}$ in the environment $\{A_{\mathcal{G}}\}$ and that m_S is the smallest cost needed to solve $\mathcal{G}$ in this environment.*

The naïve method for constructing a goal template calls for actually storing the A-literals that can be used in reduction operations during the solution of $\mathcal{G}$ in $\{A_{\mathcal{G}}\}$. When a template for a goal $\mathcal{G}$ is retrieved (line 3.1 of Figure 3), an applicable template must have the property that each A-literal in the environment of $\mathcal{G}$ is an instance of an A-literal stored in the template environment.

Examination of "chain dumps" for several non-Horn problems indicates that cache hits would be very rare and this method does not appear viable for non-Horn problems (Plaisted noted this as a potential problem with caching using model elimination in [33]). Consider a snapshot of the search tree for a particular theorem. For Horn problems, nodes in the search tree (unexpanded and-nodes that correspond to subgoals) can be considered for caching independently of the position at which they occur. In the naïve approach we have outlined for non-Horn problems, it is the root-to-node path that is considered for caching (where each node other than the last constitutes an A-literal). It is, perhaps, not surprising that such paths are not often candidates for cache retrieval. There are methods for reducing the size of the set of A-literals cached using the naïve approach. Plaisted has shown [33] that it is possible to limit the set of literals usable in reduction operations to negative A-literals and retain a complete inference system. This positive refinement (so called because only positive subgoals are considered for reduction with negative A-literals) will lead to a smaller environment at the expense of potentially longer proofs. However, our examination of chain dumps indicates that this restriction appears to be no more promising than the naïve approach.

In order to increase the number of potential cache hits, we have been led to consider a generalized environment as an alternative to the naïve method. For a goal $\mathcal{G}$, rather than storing each A-literal

on the root-to-node path in $\{A_{\mathcal{G}}\}$, we consider storing a completely general A-literal, i.e., a distinct variable for each A-literal. A count of the number of the (generalized) A-literals would be stored rather than the literals themselves. When considering a cache-lookup for a goal $\mathcal{G}$ we need only ensure that the number of A-literals that are potential candidates in reduction operations for solving $\mathcal{G}$ is less than or equal to the number stored in the environment of the retrieved cache template. In order for this method of caching to work, all attempts to solve a goal $\mathcal{G}$ must be made with completely general A-literals. When a goal is solved, the instantiated generalized A-literals are compared to the A-literals that actually occur as ancestors of $\mathcal{G}$. (Note that this comparison entails unifying the instantiated generalized A-literals with the actual A-literals.) If each instantiated generalized A-literal occurs as an actual ancestor, then the current deduction is valid, a solution is stored, and the proof search continued. To ensure that the cache is complete, the solution is stored even if the comparison of instantiated generalized A-literals to actual A-literals fails. In either case, the stored solution contains the instantiated generalized A-literals.

This method incurs the overhead of yielding many general solutions that fail to be actual solutions when the instantiated generalized A-literals fail to appear as ancestors of a particular goal. We call this method *generalized non-Horn caching*; we are investigating its implementation. In both the naïve method and the generalized method cached solutions must store the A-literals used in reduction operations to solve a cached goal. Cached solutions for non-Horn problems are defined in Definition 4.7.

Definition 4.7 *A* cached non-Horn solution *is a triple* $(\{A_{\mathcal{G}'}\},\mathcal{G}',n_{\mathcal{G}'})$ *where* $\mathcal{G}'$ *is* $\mathcal{G}\theta$ *for some goal* $\mathcal{G}$, θ *is the composition of the unifiers used in solving* $\mathcal{G}$, $\{A_{\mathcal{G}'}\}$ *is the set of (instantiated) A-literals used in reduction operations to solve* $\mathcal{G}$, *and* $n_{\mathcal{G}'}$ *is the measure of the cost used in solving* $\mathcal{G}$.

These cached solutions correspond to what was termed a lemma in the original presentation of *ME* [22, 23, 24]. There it is shown that the conjunction of the negation of a solved goal (actually a contracted A-literal) with the negation of the A-literals used in reduction operations to solve the goal represents a clause that might be generated using resolution methods. Specifically, such a clause is a logical consequence of the input clauses and can thus be treated as an input clause if desired. In the Horn case, when no reduction operators are used, a solved goal in our context is a unit lemma. In the non-Horn case, unit lemmas are generated only when a goal is solved without any reduction operations, and nonunit lemmas are generated when reduction operations have been used to solve a goal and correspond to our cached non-Horn solutions.

If a template is retrieved that indicates that the cache can be used to solve a goal $\mathcal{G}$, the instantiated A-literals stored in a cached solution are matched against the A-literals that are actually ancestors of $\mathcal{G}$ (these may be generalized A-literals). If there is a match, which involves unifying the cached A-literals against the actual A-literals, the cached solution is used and replaces the search used in constructing the solution.

The explosive increase in the number of these A-literal dependent cached solutions combined with the apparent lack of cache hits using the naïve method leads us to believe that generalized caching, even with its additional overhead, is the method of choice for non-Horn problems. However, guided by our examination of chain dumps, our intuition leads us to believe that caching may still not be viable for any large class of non-Horn problems. As our experiments with its implementation progress, we hope to report more fully on caching for non-Horn problems. It may prove, however, that lemmaizing is more productive than caching for non-Horn problems (see Section 7).

5 Lemmaizing

The use of the cache described in Section 4 replaces some search with a cache lookup. The cache functions as a more efficient search engine; it recreates the same search space that would be explored without its use and hopes to do so more efficiently. We have outlined the difficulties of caching for non-Horn problems and note that even for deep Horn problems the number of potential cache entries may preclude caching as an alternative because of both memory considerations and the concomitant increase in cache lookup time. This has 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 another 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. We reserve the word caching for use with the mechanism in which cache lookups replace search. We have used heuristic caching to refer to a method that replaces search but for which certain cached solutions may be used although resource requirements are not met. We use lemmaizing to refer to a method in which only certain solutions are stored (rather than all solutions); these solutions are used to augment the normal search mechanism. The term *lemma store* has the same meaning as its cache counterpart but refers to the storage of solutions used in lemmaizing. In lemmaizing, the same kind of solution pair $\langle G, n_G \rangle$ used in caching is stored as a lemma in the lemma store.

Heuristic caching and lemmaizing allow certain stored solutions to be used although cost bounds would not be sufficient to allow them to be used in caching. By identifying a relevant solution whose derivation may require many inference steps, but not deducting the cost of the derivation from the available resources when the stored solution is used, it is possible to discover high-cost proofs with low cost bounds.

Since lemmas are not needed for completeness, we may impose arbitrary syntactic and semantic criteria when 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 system that will hopefully allow us to prove theorems

otherwise unobtainable. In one of the first implementations of *ME* [15], 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 smaller search space was not realized because of the increased branching factor induced by allowing lemmas as alternatives for extension.

5.1 Lemma Costs

When some, but not all, solutions are stored as lemmas and used to augment the search, the cost of using such lemmas (as deducted from the current cost bound) must be less than the discovery cost of the lemma if use of the lemma is to be productive.

For example, if solutions used in the proof of the top goal incurred a smaller cost than other solutions, the proof could be found more quickly. The intuition here is that these 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. The information may be used "for free" (or at a cost less than that required to originally prove the lemma) much as a mathematician makes use of lemmas to make a proof simpler. Identifying useful lemmas is a nontrivial 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 when predictive D_{inf} is used as the cost measure. 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. Consider the snapshot in Figure 5 of part of the search process for a proof in a group theory problem (r is the equality predicate, p is the predicate for the group's binary operation, e is the identity element, and $p(X, Y, f(X, Y))$ is the closure axiom).

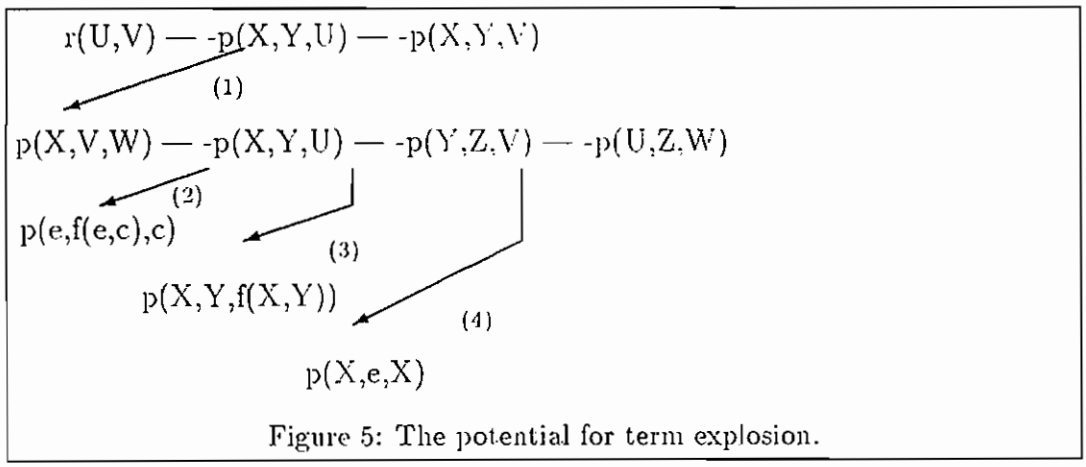

Figure 5: The potential for term explosion.

In Figure 5, the extension labeled (2) is an extension with a lemma. As a result of the sequence of extensions shown, the top goal $\neg p(X, Y, U)$ is instantiated to $\neg p(c, f(f(e, c), e), c)$. If this new

solution is stored as an input clause (i.e., a zero-cost lemma) and substituted for $p(e, f(e, c), c)$ in Figure 5, the same sequence of inference steps will result in the solution $\neg p(e, f(f(f(e, c), e), e), c)$. Repeating this process results in the storage of solutions with ever-increasing term depth and size. Retrieving such solutions, checking them for subsumption, and using them as potential unifiers will quickly degrade cache performance. As a sidenote, consider the same example but using caching rather than lemmaizing with D_{Alit} as the cost measure. Note that in this case the sequence of deductions outlined in Figure 5 requires resource two since the depth of the proof tree is two (a maximum of two A-literals appear in all proof tree branches). If the solution $\neg p(e, f(f(e, c), e), c)$ is stored and retrieved in a cache lookup, the same sequence of steps (using a cost bound increased by one) will result in the storage of the same solutions outlined above when storing lemmas as input clauses. It can be shown that using D_{Alit} can result in cached terms that are exponentially larger than the terms stored for the same problem when D_{inf} is used.

5.2 Storing Lemmas

Since lemmaizing augments the search space and has the potential to increase the branching factor in the search tree to the point that a proof cannot be found, some care must be taken in determining what lemmas to store and what to charge for the use of each lemma.

We have used several syntactic and semantic criteria in determining what lemmas to store. In our experiments to date we have treated all lemmas as input clauses in charging for their use. We report on the successful use of lemmaizing for several Horn and non-Horn problems in Section 7; we give a brief summary here of the general methods we have used in determining what solutions to store as lemmas.

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 as illustrated in Figure 5. In our experiments we generally limit the nesting of function symbols to one or two except as described below when demodulation is used. We have also experimented with limiting the size of the terms that may appear in lemmas, both independently of and in conjunction with limiting the nesting depth of terms. Although limiting term size is important, all of our favorable results depend to a large degree on limiting the nesting depth of terms that 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 the group and ring problems by including the complete set of rewrite rules for free groups or rings as demodulators as well as using demodulators generated during the search to rewrite all lemmas. This leads to a substantial reduction in the number of lemmas stored, thus decreasing the effective branching factor of the lemma search space. In addition, rewriting lemmas allows the early discovery of lemmas that would be discovered deeper in the search tree if demodulation is not employed. Both of these factors can greatly decrease the time to find a proof for problems in which demodulation is possible as shown in Section 7. We have not yet implemented back demodulation, which promises to further reduce the branching factor of the lemma search space.

In the current system we use lexicographic recursive path ordering [12] based on a user-specified total ordering of symbols to determine if a rewrite rule applies. The ordering is also used to rewrite orientable instances of unorientable rules such as $f(X,Y) = f(Y,X)$. This usage is similar to that of LEX demodulators in *OTTER* [27] and is a feature of the unfailing Knuth-Bendix procedure [3]. The user may specify if a set of rewrite rules is to be used in addition to or independently of any dynamically generated demodulators. For further details on the results obtained using demodulators in conjunction with lemmaizing, see Section 7.

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 because of a large number of unsuccessful attempts to unify goals with cached solutions. We must implement the operations *template lookup* (line 3.1 in Figure 3) and *solution lookup* (line 1 in Figure 4) as efficiently as possible while returning only templates and solutions that are "close" matches (i.e., we want to maximize the ratio of successful to attempted unifications).

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 [40].

Cache and lemma lookup operations for a goal $\mathcal{G}$ require the retrieval of terms that are exact matches, generalizations, or possible unifiers for $\mathcal{G}$. If stored solutions can cause the removal of less general solutions already in the cache (back subsumption), then instances of $\mathcal{G}$ need to be retrieved from the cache as well. In order to implement these operations efficiently, we have developed a modification of the *trie* [19] data structure. Tries provide efficient mechanisms for storage and

retrieval of strings. Whereas in a (binary) search tree the search key is compared to the key stored at each node of the tree to determine which branch to follow, a trie uses the structure of the key itself to determine which branch to follow and performs a comparison of keys only at leaf nodes. Consider the expressions $p(X, Y, f(X, Y))$, $p(X, e, X)$, and $p(b, g(f(b, e)), e)$ stored in the trie of Figure 6. As indicated in the figure, we map all variables to an anonymous variable V to decrease the branching factor in the trie and to minimize the computation done in flattening a term (see below). This method gives perfect retrieval for linear terms (i.e., terms with no repeated variables), but can give false matches for nonlinear terms due to conflicts in variable bindings.

When used to store terms and expressions in this manner, tries are often referred to as discrimination trees or nets; variations of tries have been employed in many different theorem-proving systems [11, 16, 28]. Discrimination trees are especially good for retrieving generalizations [28, 40].

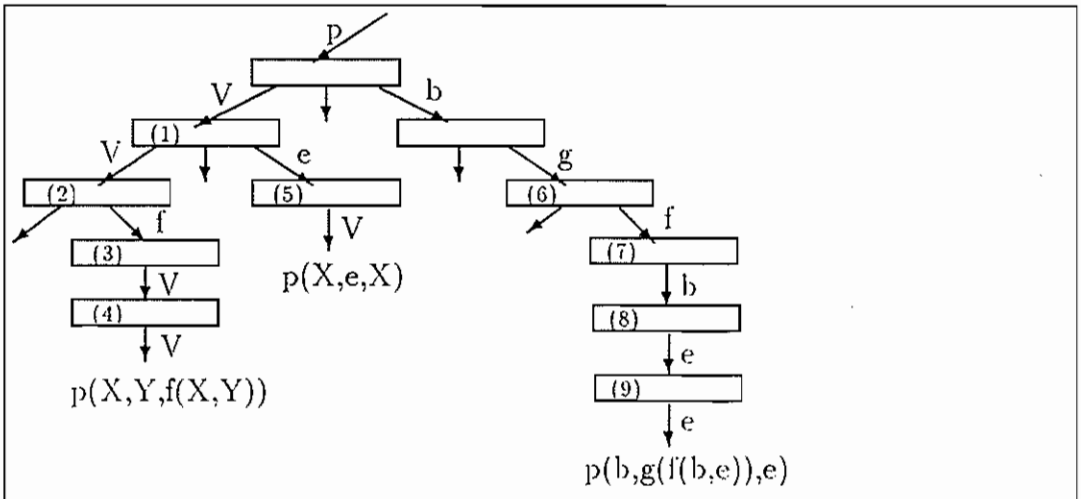

Figure 6: The cache data structure.

For example, every leaf node in the trie shown in Figure 6 will be retrieved as a term unifiable with $p(X, Y, Z)$, only the leaf nodes pointed to by the nodes labeled 4 and 5 will be retrieved as possible generalizations of $p(X, e, f(X, Y))$, and the leaf nodes pointed to by the nodes labeled 5 and 9 will be retrieved as possible instances of $p(X, X, e)$.

An analysis of tries shows that they support sublinear (actually $O(\log n)$) cache operations for (exact) string matching. Retrieval time may be greater and multiple values may need to be returned for more general pattern matching (e.g., all leaf nodes must be returned when retrieving instances of the variable term X). In many of the problems we have run, the number of nodes in the trie quickly becomes quite large. This leads to an increase in lookup time (for both templates and solutions) and can increase the paging activity of a run, further slowing down the prover. This has led us to investigate various trie compression alternatives. Many of these alternatives eliminate those nodes with a single branch that do not discriminate between different keys (see below). Although compression of the trie as described eliminates nodes with a single branch, in keeping with standard

terminology we refer to the elimination of *one-way branches* rather than single-branching nodes.

Consider the nodes labeled 3, 4, 5, 7, 8 and 9 in Figure 6, which are redundant from the point of view of string search operations. For example, the term $p(X, Y, f(X, Y))$ can be referenced by the pointer labeled f in node 2 since nodes 3 and 4 do not differentiate between the term and other terms stored in the trie. Similarly, the term $p(b, g(f(b, e)), e)$ can be referenced by the pointer labeled f in node 6, eliminating nodes 7, 8, and 9 that do not differentiate that term from any other term stored in the trie. By eliminating nodes with a single branch, we decrease the storage required for the trie and decrease the number of nodes visited during a trie search.

Compressing a trie by eliminating one-way branches results in a PATRICIA trie [19, 43]. Formal analysis shows that such tries result in decreases in both time and storage compared to a standard trie. The analysis, however, is based on string-searching applications (where we view any sequence of bits as a string). In our application, we use the trie for pattern matching rather than for exact matches. Consider the elimination of the one-way branches in Figure 6. In searching the uncompressed trie for the expression $p(b, g(f(e, e)), e)$ the search is cut off before a leaf node is reached. In the compressed trie this term must be compared with the term stored in the leaf node since elimination of the one-way branches also eliminates the early cutoff. Since we use the trie to retrieve instances, generalizations, and candidates for unification for a given literal, the comparison at the leaf node may be an expensive operation (e.g., unification) as opposed to a simple test for equality.

6.1 Some Implementation Details

In our example, compression resulted in the "shrinking" of one-way branches that lead to a leaf node (external one-way branches). In a PATRICIA trie all one-way branches are eliminated. As a result of the elimination of internal one-way branches, each node of the trie must store information that indicates how many bits in the search pattern should be skipped when moving to the next node. Although the code for implementing such tries is not overly complex for exact matches [36], the loss of information caused by compression (resulting in more attempted unifications, for example) has led us to consider a compressed trie in which the only one-way branches eliminated correspond to nodes all of whose children are one-way branches (a similar device was used in [16]). For example, if the only branch from the node referenced by p in Figure 6 was the anonymous variable branch (referencing the node labeled (1)), that branch would not be removed since the node labeled (1) has more than one child (in addition to having a child with more than one child—node (2)). By eliminating only such external chains of one-way branches, no auxiliary information is stored in internal nodes and the addition and removal of leaf nodes is relatively straightforward.

We also consider limiting the maximum depth of the trie so that after m levels, for example, a leaf node is replaced by a linked list of leaf nodes. Both this modification and the compressed trie reduce the size of the trie considerably as well as decreasing the time used in searching the trie.

This leads to substantial performance gains for many problems, although both mechanisms yield less perfect retrieval than is possible in a full trie, e.g., false matches occur even for linear terms. Statistics regarding various combinations of these mechanisms are given in Section 7.

Each internal node in our trie holds an array of pointers that represent the possible branches from the node and a counter of how many pointers are non-nil in this array. As each function and constant symbol in the input clauses is parsed, it is given a unique number that serves as an index into the array of pointers. This technique is outlined in [11]. The count of the number of branches is used to recursively remove internal nodes that are removed when cached solutions are removed during back-subsumption. In practice this operation rarely removes nodes other than leaf nodes, especially when external one-way branches are compressed.

6.2 Possible Optimizations

Although the trie is an efficient data structure supporting the retrieval of expressions for unification and subsumption, expressions are often retrieved that fail to unify with (or subsume) the subgoal that invoked the retrieval. This is most often due to conflicts that occur in variable bindings, sometimes as a result of mapping all variables to an anonymous variable. In our implementation, no bindings are made during trie traversal; bindings occur only during the unification or subsumption routines that are called after a potential match is found. For problems in which caching and lemmaizing are successful, the ratio of successful unifications to attempted unifications using solutions retrieved from the cache store ranges from 50% to 80%. In contrast, caching is detrimental for the condensed detachment problems described in Section 7.5 for which this ratio is often less than 5%. For these problems, early binding of terms to variables may result in early rejection of alternate solutions and increase this ratio, thus making caching more tractable [28].

In our implementation the solution store supports finding generalizations of terms more efficiently than finding instances of terms (cf. [28]). This is especially true if the tree form of a term is used. To find generalizations, an anonymous variable branch is followed in the trie and the corresponding nonvariable term skipped in the target expression. When the tree form of terms is employed, the arguments of a functor or predicate of arity n occupy the next n places after the functor or predicate symbol so that skipping one of these arguments is quite simple. In contrast, if the stringized form of terms is used, skipping a term involves calculating where the next term begins. When instances are retrieved from the solution store, these calculations must be made since the branches in the trie constitute a flattened representation of terms. Since finding potentially unifiable solutions also involves these term calculations, long terms tend to degrade performance. It is possible to alleviate some of these problems by incorporating an extra pointer in internal trie nodes referencing the node at which the next term begins (one such pointer is necessary for each possible branch). Following these "jump" pointers avoids the overhead associated with calculating the beginning of the next term. These pointers do add to the storage cost, and when external

nodes are compressed, updating these pointers can be an expensive operation (e.g., during the "uncompressing" of a compressed sequence when a new term is added).

Both these ideas are incorporated to a degree in the *OTTER* theorem prover [27]. Jump pointers are employed explicitly in the *flatterm* representation of terms in [11] (but not in the trie) and are used in the discrimination tree employed in the *SNARK* theorem prover [42]. We are investigating these potential optimizations, but they are not employed in the results given here.

We have investigated one method that has the potential to improve the ratio of successful to attempted unifications. Our unification routine attempts to unify two terms by traversing the terms from left to right, performing substitutions of terms for variables as necessary. Since the trie is normally constructed by traversing a term from left to right as well, attempts at unification that fail late in the left-to-right unification traversal may not be pruned during trie traversal because of compression.

It is possible that using a right-to-left term traversal in searching the trie might lead to better performance since the right-to-left trie traversal and left-to-right unification traversal might result in less redundancy in the matching of literals than if left-to-right traversals were used in both the retrieval and the unification routine. This could lead to earlier unification failures although the ratio of unification successes to failures would be the same.

Since there is no performance penalty in traversing terms from right-to-left as compared to left-to-right during retrievals, this bidirectional traversal scheme will *prima facie* yield a performance gain. However, the effect may be small; experiments show little observable gain over the normal method of left-to-right traversals in both retrieval and unification.

6.3 Physical and Logical Solution Stores

In our caching and lemmaizing mechanisms the allocation and initialization of trie nodes is handled by a single module. In this sense, there is a single physical store that is divided into two logical stores: the template directory and the solution store. Each of these two logical stores is further divided into substores with one substore for each signed predicate. This requires two pointers for each signed predicate symbol—one for the first entry in each of the directory and the store. The store pointer is labeled p in Figure 6. In order to decrease the number of entries in the solution store, it is seeded with unit clauses. These clauses are not retrieved as solutions, but serve as subsuming solutions preventing more specific solutions from being entered in the solution store.

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 [39], we include problems that have been historically difficult for *ME*-based provers.

We show that caching can reduce the time needed to solve several difficult problems by a significant amount, sometimes by more than an order of magnitude. For these results we provide figures indicating that template subsumption is useful when caching is employed and show that the value used for the cost threshold to restrict caching can have a significant impact on cache performance.

We provide a successful application of both caching and heuristic caching in proving SAM's lemma [48] in which we have 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 or ring 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 [48], whose proofs have, heretofore, been unobtainable by top-down *ME* theorem provers. These results indicate the potential for lemmaizing since we succeeded by using syntactic criteria for lemma retention, although we note that the use of demodulation is a powerful tool.

We also succeeded in finding a proof of the intermediate value theorem of calculus (as formulated in [47]). This non-Horn problem is proved using lemmaizing and has to our knowledge been beyond the capabilities of linear provers. Success with this and other non-Horn problems indicates the potential for lemmaizing in this domain.

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.

7.1 Caching and Lemmaizing with Horn problems

In Figure 7, we provide comparisons of the performance of caching and lemmaizing with several Horn problems. For all the problems we report on, caching reduces both the number of inferences and the time to find a proof.

The label "fail.temp." means that the cache was used only for pruning using failure templates (with template subsumption used). Although these results seem to indicate that by themselves failure templates are not useful, recall that caching requires the disabling of the identical-ancestor pruning rule. The time and number of inferences required for these problems without this rule (and without caching) indicate that failure templates improve the performance of the prover but do not compensate for the disabling of the pruning rule.

The label "cache" is based on the best cache threshold over several runs (see Figure 10) 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. These results indicate both the potential for lemmaizing and the need for demodulation when it is applicable.

Each run indicates the number of seconds needed to find the proof (the top number) and the number of successful inferences made. Missing figures indicate that the the method failed for the problem (e.g., no proof was found in the allotted time).

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

Figure 7: Results for caching and lemmaizing with Horn problems.

7.1.1 Use of Demodulation

Figure 8 shows results from running several group or ring theory problems using demodulators generated during the proof in addition to using the standard demodulators for free groups or rings. No back demodulation was employed and all terms appearing in lemmas were restricted by limiting the level to which function symbols could be nested. As these results show, demodulation in conjunction with term-size restrictions is very successful in reducing the number of generated lemmas to a manageable size.

7.2 Lemmaizing with non-Horn Problems

We have noted the difficulties that occur when caching is used with non-Horn problems. Although caching requires storage of all solutions, lemmaizing, since it augments search instead of replacing it, can be more selective. An essential difference between Horn and non-Horn problems is the possibility of nonunit lemmas in the latter. Lemmas are formed during the contraction operation

	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.

when leftmost A-literals are removed from a chain. The removed A-literal is disjoined with all A-literals to its right that were used to remove by reduction B-literals formerly to its left. Only if no such reduction operations occurred (as none can in the case of Horn problems) is a unit lemma formed from the A-literal alone. Since lemmas are not required for completeness, we can restrict ourselves to storing only unit lemmas even for non-Horn problems, just as in the case of Horn clauses. This allows lemma handling to be as efficient as in the Horn case. Considering only unit lemmas is often effective, but not uniformly so. There are problems in which unit lemmas are too scarce or discovered too late in the search to be useful.

Results are given in Figure 9 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 [47]. With the addition of several nonautomatically constructed rewrite rules it is proved by the prover in [34]. The problem labeled nonobv is a problem given in [31] and subsequently cited in [26]. We should note that when D_{Alit} is used as the cost measure *METEOR* solves this problem in under one second. The problem labeled salt is Lewis Carrol's salt and mustard logic puzzle (non-propositional version) taken from [25].

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.

7.3 Parameters Affecting Caching and Lemmaizing

Results are given in Figure 10 using different cache thresholds, i.e., varying the minimum level at which the cache is consulted. 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. 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 $D_{\inf}$, see Figure 12) 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 10: 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 11 gives statistics for the same problems and parameters as given in Figure 10, but without employing template subsumption.

In Figure 12, the number of steps in the proof found using different parameters is given. Note that for caching results the search tree is explored to the same depth as when $D_{\inf}$ is used; the number of steps in a proof is shorter when a solved goal is used in place of a sequence of deductions. The two entries under the cache heading correspond to whether template subsumption is employed.

7.4 Memory Requirements

METEOR and *PTTP* are attractive partly because of their minimal memory requirements. One of the potential drawbacks incurred by caching and lemmaizing is the (potentially large) size of the cache and lemma store. Figure 13 shows the memory requirements for the problems reported in this section.

The use of lemmaizing imposes only modest memory requirements when tight restrictions on

	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 11: Different cache thresholds (no template subsumption).

	D_{inf}	cache		unit	demod.
problem		**subsume**	**no subsume**	**lemma**	
wos10	10	4	7	4	7
wos 1	10	7	7	7	7
wos21	12	5	8	9	
wos15	15	10		7	7
sam†	29	7	9		
wos22	14	6	10		

†heuristic caching

Figure 12: Number of steps needed to find proof.

what lemmas to keep are employed. Results using caching for hard problems (e.g., wos15 and wos22) indicate that unconstrained caching may not be viable for very hard problems without some modification of the caching mechanism. 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.

7.5 An Un-cacheable Problem Class

Although caching works well for a class of problems, there are many problems for which the increased storage costs and retrieval time incurred by the cache do not compensate for the reduced number of inferences. We have used *METEOR* to solve several of the problems given in [29], which are based on the inference rule of condensed detachment. While many of the problems are solved quickly in *METEOR*, we fail to find proofs for nearly 70 out of 112 problems. The search for a proof of some of these problems generates very long terms—on the order of 100 symbols appear in the terms for these problems. Because *METEOR* uses the standard technique of copy on use for binding terms to variables, and because these terms are so long, the cache quickly fills with terms whose subsequent

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 13: Memory requirements for several problems.

binding results in a severe degradation in overall performance. Although no copying of terms is done in forward subsumption tests, terms are copied during unification and backward subsumption. While the number of inferences is reduced for these problems when caching is used the inference rate often decreases by more than an order of magnitude. This could be alleviated to some extent by binding terms to variables during trie traversal, so that variable-binding conflicts, the most common reason for unification failure for these problems, will be detected more quickly. Such a mechanism is employed in the *OTTER* system [28]. The use of lemmas may hold more promise for these and other problems, however.

8 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 [17, 18], and by runtime analysis [45]. 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 [46] is closely related to our framework, and 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 [45] of SLD-AL resolution also uses iterative deepening. The OLDT resolution procedure [44] 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) search for only one proof.

Extension tables as used in [13] 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 [32] 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 his prover.

Elkan [14] 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 [37]. The EBL domain is slightly different since the principal aim there is to "train" the prover by storing solutions for a class of problems and then using 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 further improve the caching mechanism in terms of its storage requirements and its redundancy reduction capabilities and thus permit it 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.

Acknowledgments

We would like to thank Donald Loveland and Richard Waldinger for their valuable comments on earlier drafts of this paper.

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.S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe.* Kluwer Academic Publishers, 1991.

[3] L. Bachmair, N. Dershowitz, and D.A. Plaisted. Completion without failure. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, vol. II: Rewriting Techniques.*, pages 1–30. Academic Press, 1989.

[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] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press. 1973.

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

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

[13] S.W. Dietrich. Extension tables: Memo relations in logic programming. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 264–272, 1987.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] R. Letz, S. Bayerl, J. Schumann, and W. Bibel. SETHEO--a high-performance theorem prover. (to appear).

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

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

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

[25] E. Lusk and R. Overbeek. Non-horn problems. *Journal of Automated Reasoning*, 1:103–114, 1985.

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

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

[28] 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).

[29] W. McCune and L. Wos. Experiments in automated deduction with condensed detachment, 1991. (results presented at 1991 Joint Japanese-American Workshop on Automated Theorem Proving).

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

[31] F.J. Pelletier and P. Rudnicki. Non-obviousness. *AAR Newsletter*, (6):1–5, 1986.

[32] D. Plaisted. Non-Horn clause logic programming without contrapositives. *Journal of Automated Reasoning*, 4(3):287–325, September 1988.

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

[34] 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.

[35] 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.

[36] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

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

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

[39] M.E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:343–380, 1988.

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

[41] 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.

[42] M.E. Stickel and R. Waldinger. Proving properties of rule-based systems. In *Proceedings of the Seventh Conference on Artificial Intelligence Techniques*, pages 81–88, 1991.

[43] W. Szpankowski. Patricia tries again revisited. *Journal of the Association for Computing Machinery*, 37(4):691–711, October 1990.

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

[45] 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.

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

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

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

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