Lemma Management Techniques for Automated Theorem Proving

Yuan Zhang and Geoff Sutcliffe

University of Miami, USA
yuan@mail.cs.miami.edu, geoff@cs.miami.edu

Abstract. Lemmas can provide valuable help for constructing a proof, by providing intermediate steps. However, not all the formulae supplied to an ATP system as lemmas are necessarily helpful. It is therefore necessary to develop lemma management techniques that use the right lemmas at the right time, to improve the problem-solving ability of ATP systems. This paper presents three lemma management techniques, reports on their implementation, and illustrates their potential with example problems.

1 Introduction

Automated Theorem Proving (ATP) is concerned with the development and use of systems that automate sound reasoning: the derivation of conclusions that follow inevitably from facts. This capability lies at the heart of many important computational tasks. In this work we are dealing with ATP for 1st order classical logic, which has well known computational properties, and henceforth all discussion is in that context. Current ATP systems are capable of solving non-trivial problems. In practice however, the search complexity of most interesting problems is enormous, and many problems cannot currently be solved within realistic resource limits. Therefore a key concern of ATP research is the development of more powerful techniques and systems, capable of solving more difficult problems within the same resource limits.

In the mathematics world people often use *lemmas* to help construct proofs for hard theorems. They first make some trials by using existing lemmas that are pertinent to the problem. If they cannot solve the problem, they then find or derive more lemmas that might help solve the problem, and continue the same process until the theorem can be proved. For example, the famous mathematician Gauss proved the Gauss lemma as a step along the way to the quadratic reciprocity theorem [1]. There have been several previous efforts to use lemmas in ATP systems. Lemmas have been used in model elimination based systems in the context of an ongoing proof attempt, to avoid repeated subdeductions, e.g., [2, 3]. Lemmas have also been used to augment a problem before starting a model elimination system, with a filter being used to select the lemmas that seem most likely to be useful [4]. A higher level approach to using lemmas, which breaks a hard problem down into manageable chunks, has been used to find proofs of hard problems in logical calculi [5]. The approach taken in this work is to augment the axioms of a problem with lemmas, and prove the theorem from the axioms and lemmas. The lemmas are then proved from the axioms, either directly, or using the same

technique recursively. The lemmas' proofs are combined with the theorem's proof, to form a proof of the theorem from the axioms alone. The final proof may be viewed at a *proof structure* level, showing the dependencies between the axioms, the lemmas, and the theorem, or a fully detailed level that includes the inference steps of each component proof.

Lemmas can provide valuable help for constructing a proof, by providing intermediate steps. However, not all the formulae supplied to an ATP system as lemmas are necessarily helpful. Some may be not provable in the current theory (i.e., they are not really lemmas), some may not be relevant to the conjecture, and some of them may make only small steps in the overall proof at the expense of an increased search space. It is therefore necessary to develop lemma management techniques that use the right lemmas at the right time, to improve the problem-solving ability of ATP systems. This paper presents three lemma management techniques, reports on their implementation, and illustrates their potential with examples. The first technique, *iterative lemma usage*, relies on the lemmas being provided in an order that allows each lemma to be proved from the axioms and the preceding lemmas, even if some of the lemmas are not part of the final proof structure. The second two techniques, *recursive lemma selection* and *recursive lemma minimization*, are robust to the order in which the lemmas are supplied, and can cope with the lemma set containing formulae that cannot be proved in from the axioms or are irrelevant to the conjecture. Additionally, iterative lemma usage is likely to fail is the proof structure is branching, i.e., requires multiple lemmas to be used together in a component proof, while recursive lemma selection and minimization are independent of the proof structure. However, in their current forms, the second two techniques are likely to perform poorly if the lemma set is very large - two possible solutions are proposed in the conclusion.

2 Iterative Lemma Usage

Art Quaife successfully used the ATP system Otter to prove theorems in several fundamental mathematical theories, such as Von Neumann-Bernays-Gödel set theory [6]. Proofs in these theories are often difficult for ATP systems; theorems that are very easy for humans to prove are very hard for ATP systems to prove. To attack those challenging problems, Quaife used a systematic method in which theorems were proved sequentially, from basic simple theorems through to advanced hard theorems. The sequence in which the theorems were proved was determined by Quaife, based on his mathematical knowledge. Once a theorem was proved, it was added to the axiom list as a lemma to help prove the next harder theorem. By such *iterative lemma addition* (also referred to as lemma adjunction [7]), Quaife proved over 400 theorems in set theory. Iterative lemma addition has been implemented in our YiLT system, and is activated in the `ILA` mode of YiLT. A time limit is imposed on each proof.

Although iterative lemma addition is helpful for solving hard problems, it has some weaknesses. Lemmas that have been added to the axioms may be redundant with respect to (in the sense of being easily proved from) subsequently proved lemmas. Humans are good at ignoring redundant lemmas and focusing on only useful ones, but for ATP systems redundant lemmas act as noise, disturbing the search for a proof. An alternative to

iterative lemma addition, which counters this adverse effect, is *iterative lemma replacement*. In iterative lemma replacement each previously proved lemma is replaced by the newly proved lemma, until the conjecture is proved. Iterative lemma replacement has been implemented in YiLT, and is activated in the `ILR` mode. Iterative lemma replacement has the weakness that even if a lemma is not redundant, it is always replaced by the next lemma proven. Iterative lemma replacement thus cannot produce a branching proof structure.

Iterative lemma addition and replacement are at two extremes in terms of retaining or discarding lemmas. A mechanism that retains selected useful lemmas is desirable. One approach is to discard any previously proved lemmas that are easily proved from the newly proved lemma, the axioms, and other previously proved (but not discarded) lemmas. Such easily proved lemmas are redundant with respect to the axioms and the other lemmas. This technique is called *iterative lemma selection*. A lemma is considered to be "easily" proved if the CPU time taken for the proof is below a specified threshold. In [4] a refined version of this technique is presented, and used to filter out redundancy from a set of lemmas before they are used to augment a problem. Iterative lemma selection has been implemented in YiLT, and is activated in the `ILS` mode.

On average, iterative lemma selection performs better than iterative lemma addition and replacement. However, iterative lemma addition and replacement have strengths in some cases. Section 5 shows a case when iterative lemma addition outperforms iterative lemma selection and replacement. All three variants have the key weakness that each lemma in turn has to be provable from the axioms and preceding lemmas, and thus fails if unprovable lemmas are encountered. Additionally, iterative lemma usage is likely to fail if the proof structure is a branching.

3 Recursive Lemma Selection

The formulae provided as lemmas for a problem may be arbitrarily ordered, may not all be provable in the current theory (i.e., not really lemmas), may not all be relevant to the conjecture, and their use may induce a branching proof structure. These situations may prevent iterative lemma usage from finding a proof of the theorem. We have therefore developed a demand-driven approach to lemma usage that can cope with these situations.

Recursive lemma selection starts with the conjecture as the initial *target formula*. *Helper sets* are formed from different combinations of increasing numbers of lemmas, starting with no lemmas. If the target formula can be proved (within a time limit) from the axioms and a helper set, then immediately the members of the helper set are iteratively treated as the target formula, in a recursive fashion. When all the target formulae have been proved at all the levels of recursion, with the target formulae at the deepest levels being proved directly from the axioms (i.e., with empty helper sets), a proof of the theorem has been found. If at any stage a target formula cannot be proved, the next alternative helper set is considered. At all stages no helper set element may be a descendant of the target formula in the proof structure, to prevent circular arguments. A cache is used to recall and reuse previous proofs of target formulae. This technique has been implemented in our YuLM system.

The power of recursive lemma selection lies in its robustness with respect to the lemmas supplied. Recursive lemma selection identifies lemmas necessary for a proof, and uses them to construct the proof. This robustness is achieved through the combinatorial formation of helper sets of increasing size. If the proof has a branching structure, in which multiple lemmas are required to prove the theorem or some lemma, a helper set with all the necessary lemmas is used. As the helper sets are formed in increasing order of size, less branching is preferred at each stage. The formation of all alternative helper sets makes it possible for recursive lemma selection to find multiple proof structures for the theorem, which may then be compared in terms of some quality measure, e.g., proof size. Section 5 shows cases when recursive lemma selection solves problems that cannot be solved by iterative lemma usage.

4 Recursive Lemma Minimization

Recursive lemma selection has no regard for proof quality. This is due to the greedy immediate recursion to prove the members of a successful helper set. We have therefore developed a modified branch-and-bound style approach to lemma usage, which makes it possible to find a proof that is optimized with respect to the number of lemmas used or CPU time taken, while maintaining the robustness of recursive lemma selection.

Recursive lemma minimization starts with an initial *proof candidate*, formed by placing the conjecture of the problem in the *target queue* of the initial proof candidate. This proof candidate is the initial *target proof candidate*. A list of *alternative proof candidates* is initialized to empty. At each iteration, the head of the target queue of the target proof candidate is the *target formula*. *Helper sets* are formed from different combinations of increasing numbers of lemmas, starting with no lemmas. If the target formula can be proved (within a time limit) from the axioms and a helper set, then no larger helper sets are considered. All helper sets of that size, for which a proof of the target formula can be obtained from the axioms and the helper set, are collected. Each collected helper set is used to form a new proof candidate, by appending the helpers to the target queue of the target proof candidate. (This is akin to the extension operation of tableaux based ATP systems.) If the quality of the best new proof candidate is not more than a (user supplied) *tolerance factor* worse than the quality of the best proof candidate on the alternatives list, then the best new proof candidate is the target proof candidate for the next iteration, and the remaining new proof candidates are added to the alternatives list. If the quality of the best new proof candidate is more than the tolerance factor worse than the quality of the best proof candidate on the alternatives list, then the best proof candidate is removed from the alternatives list as the target proof candidate for the next iteration, and all the new proof candidates are added to the alternatives list. The quality of a proof candidate is measured as either the number of lemmas used, or the CPU time taken for all proofs in the candidate (the quality of the initial proof candidate is optimal - no lemmas, no CPU time taken). When a proof candidate with an empty target queue is the target proof candidate, a proof has been found. It's quality is within the tolerance factor of optimal. If the alternatives list becomes empty then no proof can be found (with the time limit). At all stages no helper set element may be a descendant of the target formula in the proof structure, to prevent circular arguments. A cache is

used to recall and reuse previous proofs of target formulae. This technique has been implemented in our YuLM+ system.

Besides possessing all the strengths of recursive lemmas selection, recursive lemma minimization finds a proof that is within the tolerance factor of optimal, with respect to the number of lemmas used or CPU time taken. Recursive lemma minimization is also more stable than recursive lemma selection, and finds the same proof regardless of the order in which the lemmas are supplied. This is due to the policy of using all helper sets of the successful size at each iteration. Finally, the tolerance factor can be used to tune the performance of the approach, with a larger tolerance factor leading to a less optimal proof, but with less swapping between alternative proof structures and therefore less overall CPU time taken. As the tolerance factor goes to infinity YuLM+ converges to YuLM. Section 5 illustrates situations where these advantages are evident.

5 Illustrative Experiments

YiLT, YuLM, and YuLM+ have been implemented as meta-systems on top of the SystemOnTPTP [8] interface to ATP systems. This allows flexible selection and control of the ATP system used for each proof. Final proof structures are output in TPTP format [9], and can optionally include the full details of the component proofs. Output in TPTP format allows use of the YuTV proof tree viewer to examine proof structures.

The potential of the three systems is illustrated here with three example problems: the "graph triangles" problem, to prove that the maximal length of a shortest path between two vertices in a complete directed graph is the number of triangles in the graph plus one; the "short 5 lemma part 2" [10] that proves surjectivity in a given commutative diagram of homological algebra; and the "kitchen sink" problem [11] in a first-order encoding of the event calculus [12]. Lemmas for each problem were extracted from human proofs of the theorems, producing 11 lemmas for the graph triangles problem, 15 lemmas for the short 5 lemma, and 12 lemmas for the kitchen sink problem. The lemmas are all known to be provable from the axioms, but as the results show, not all are necessary for an automated proof. The lemmas were supplied to the systems in the order they were used in the hand-proofs, and additionally for the kitchen sink problem in reversed order and two randomized orders. Using the lemmas, the proof structure of the graph triangles problem is linear, while the proof structures of the short 5 lemma and the kitchen sink problems are branching, i.e., expected to be out of the reach of YiLT.

SPASS 2.1 [13] was used as the ATP system inside YiLT, YuLM, and YuLM+. Neither the graph triangles problem nor the short 5 lemma problem can be solved by SPASS alone with a 6200s time limit. The kitchen sink problem can be solved by SPASS in 400s, so the use of lemmas may be considered unnecessary, but the results usefully illustrate differences between our three systems. For the testing, YuLM was configured to stop when the first proof was found. For YuLM+ the tolerance factor was set to 1, i.e., forcing YuLM+ to find an optimal proof, and the quality measure was to minimize the number of lemmas in the proof structure. The tests were done on a 930MHz Pentium III computer with 512MB memory, running Linux 2.4. A 20s CPU limit was imposed on each SPASS proof. Table 1 summarizes the results. Each result gives the number of

lemmas in the final proof structure in ()s, followed by the total CPU time taken (to the nearest second) to find the proof structure.

Table 1. YiLT, YuLM, and YuLM+ Results

System	Graph triangles	Short 5 lemma	Kitchen sink Ordered	Random 1	Random 2	Reversed
YiLT ILA	(11) 37	Failed	Failed	Failed	Failed	Failed
YiLT ILR	Failed	Failed	Failed	Failed	Failed	Failed
YiLT ILS	(1) 88	Failed	Failed	Failed	Failed	Failed
YuLM	(1) 34	(9) 4195	(5) 1509	(6) 2422	(8) 2333	(11) 2036
YuLM+	(1) 4882	(8) 5042	(5) 5315	(5) 5312	(5) 5320	(5) 5310

These are only illustrative test problems, and extrapolating general conclusions from the results is not possible. The results do however illustrate performance features of the systems. As expected, YiLT fails on the two examples that have a branching proof structure, illustrating the value of the more general lemma management techniques. Note that iterative lemma addition outperforms iterative lemma replacement and selection in the graph triangles problem. The solutions of the graphs triangles and short 5 lemma problems show how the use of lemmas can extend the capabilities of SPASS.

The consistency of the results for YuLM+ across the four lemma orderings of the kitchen sink problem contrasts with the variation of the results for YuLM. The extra CPU time taken by YuLM+'s search for an optimal proof produces the desired result - the same optimal proof regardless of the order in which the lemmas are supplied. With a higher tolerance factor YuLM+ takes less time and produces less optimal proofs. Table 2 illustrates this for the kitchen sink problem with the ordered lemmas.

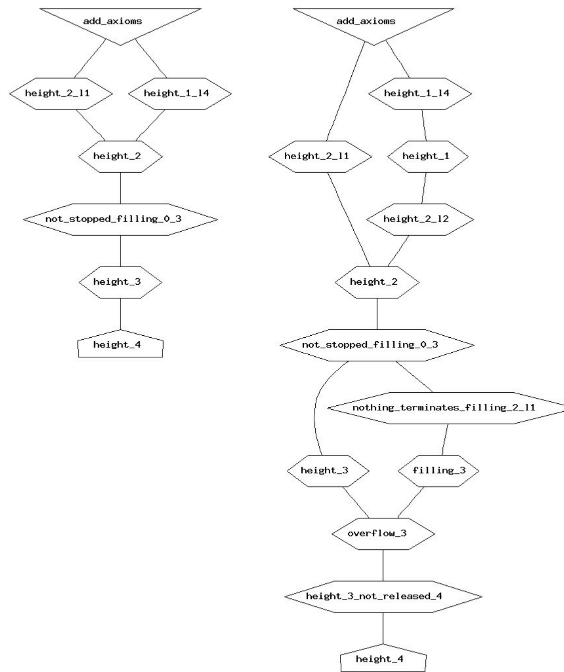
Table 2. YuLM+ Results for different Tolerance Factors

	TF = 1	TF = 2	TF = 3	TF = 4	TF = 5
YuLM+	(5) 5315	(8) 4922	(9) 4530	(9) 3215	(9) 2318

Figure 1 shows the proof structures for YuLM and YuLM+ for the kitchen sink problem. The left structure is from YuLM using the ordered lemmas and YuLM+ for all lemma orders. The right structure is from YuLM using the reversed lemmas. The inverted triangle `add_axioms` represents all the axioms of the problem, the elongated hexagons are lemmas, and the `height_4` house is the final theorem. The lines from the axioms to the lemmas have been drawn for only those lemmas that were proven directly from the axioms, but of course the axioms are used in all proofs. The right structure illustrates how YuLM greedily takes many small steps when the lemmas are

provided in reverse order. At each stage it uses the next lemma(s) in the reverse lemma sequence, hence using all 11 lemmas. This is in contrast to the smaller structure on the left, produced by the other configurations, using only 5 of the lemmas.

Fig. 1. YuLM and YuLM+ Proof Structures



6 Conclusion

This paper presents three lemma management techniques, reports on their implementation, and illustrates their potential with example problems. Appropriate lemma management allows ATP systems to use lemmas to their advantage, and provides robustness against poorly constituted lemma sets.

The principle weakness of the two recursive approaches is their combinatorial formation of helper sets. If a large set of lemmas is supplied, a very large number of helper sets can be formed. This can be overcome by pruning the lemma set before use. Pruning may be achieved using the redundancy elimination technique described in [4], or by using the Prophet tool[1] to select lemmas that seem most relevant to the conjecture.

[1] To be documented in a paper real soon.

The next phase of this project will be to use the AGInT system [14] to generate the lemmas, rather than rely on a human source. This will provide a strong challenge to the lemma management techniques, because the automatic generation of lemmas is more likely to supply lemmas that are irrelevant to the conjecture at hand. The lemma pruning techniques will almost certainly have to be used.

References

1. Nagell, T.: Introduction to Number Theory. Wiley (1951)
2. Astrachan, O., Loveland, D.: The Use of Lemmas in the Model Elimination Procedure. Journal of Automated Reasoning **19** (1997) 117–141
3. Fuchs, M.: Controlled Use of Clausal Lemmas in Connection Tableau Calculi. Journal of Symbolic Computation **29** (2000) 299–341
4. Draeger, J., Schulz, S.: Improving the Performance of Automated Theorem Provers by Redundancy-free Lemmatization. In Russell, I., Kolen, J., eds.: Proceedings of the 14th Florida Artificial Intelligence Research Symposium, AAAI Press (2001) 345–349
5. Veroff, R.: A Shortest 2-Basis for Boolean Algebra in Terms of the Sheffer Stroke. Journal of Automated Reasoning **31** (2003) 1–9
6. Quaife, A.: Automated Development of Fundamental Mathematical Theories. Kluwer Academic Publishers (1992)
7. Wos, L., Pieper, G.: Automated Reasoning and the Discovery of Missing and Elegant Proofs. Rinton Press (2003)
8. Sutcliffe, G.: SystemOnTPTP. In McAllester, D., ed.: Proceedings of the 17th International Conference on Automated Deduction. Number 1831 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2000) 406–410
9. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In Zhang, W., Sorge, V., eds.: Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Number 112 in Frontiers in Artificial Intelligence and Applications. IOS Press (2004) 201–215
10. Weibel, C.: An Introduction to Homological Algebra. Cambridge University Press (1994)
11. Shanahan, M.: Representing Continuous Change in the Event Calculus. In L.C., A., ed.: Proceedings of the 9th European Conference on Artificial Intelligence , Pitman Press (1990) 598–603
12. Mueller, E., Sutcliffe, G.: Reasoning in the Event Calculus using First-Order Automated Theorem Proving. In Russell, I., Markov, Z., eds.: Proceedings of the 18th Florida Artificial Intelligence Research Symposium, AAAI Press (2005)
13. Weidenbach, C., Brahm, U., Hillenbrand, T., Keen, E., Theobald, C., Topic, D.: SPASS Version 2.0. In Voronkov, A., ed.: Proceedings of the 18th International Conference on Automated Deduction. Number 2392 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2002) 275–279
14. Gao, Y.: Automated Generation of Interesting Theorems. Master's thesis, University of Miami, Miami, USA (2004)