Improving Search in a Hypothetical Reasoning System

Richard A. Hagen
Knowledge Representation and Reasoning Unit
School of Information Technology
Griffith University
Queensland, Australia
r.hagen@mailbox.gu.edu.au
(Correspondence author)

Abdul Sattar
Knowledge Representation and Reasoning Unit
School of Information Technology
Griffith University
Queensland, Australia
a.sattar@mailbox.gu.edu.au

Scott Goodwin
School of Computer Science
University of Windsor
Ontario, Canada
sgoodwin@uwindsor.ca

Abstract

Nonmonotonic reasoning has been a field of vigorous study for a quarter of a century with practical implementations emerging during the last 15 years. Improving the efficiency of these systems is an important step in them gaining acceptance beyond the research sphere. The use of *lemmas* – small results that can be reused later in a proof or derivation – might be one way of improving performance. We have extended an implementation of the THEORIST nonmonotonic reasoning system with four sorts of lemmas: goods, nogoods, derived literals and potential crucial literals. In this paper, we report the results of experiments designed to test whether these lemmas provide general performance boosts.

1 Introduction

Hypothetical reasoning is a form of reasoning that takes into account the possibility of uncertain or incomplete knowledge. THEORIST [11] is a practical version of hypothetical reasoning, implementing a form of *nonmonotonic reasoning.* Relationships between THEORIST and McCarthy's circumscription and Reiter's default logic have been noted by Sattar and Goebel [15] and Poole [11].

THEORIST was probably the first nonmonotonic reasoning system to have a practical implementation [10]. Since then, systems implementing versions of Reiter's default logic (e.g., [19]) and Nute's defeasible logic (e.g., [9]) have appeared.

Implementations of THEORIST are usually based on Loveland's MESON reasoning procedure [8] and use the Prolog logic programming language. Modern Prolog systems offer high inference rates as well as the traditional benefits of inbuilt logical variables and unification.

Lemmas – partial results saved during a proof for later reuse – provide opportunities for improving the efficiency of the reasoning systems. The THEORIST implementation presented in this paper implements four sorts of lemmas: goods, nogoods, derived literals and potential crucial literals. In this paper we evaluate whether these lemmas produce improvements in the execution time required for evaluating queries.

This paper presents an overview of the THEORIST system (Section 2) and the possible opportunities for improving its performance by the use of lemmas (Section 3). We also provide a quick look at the translation made by our new THEORIST compiler (Section 4) and describe some test problems that were used to evaluate our implementation and the performance benefits – if any – of the lemmas (Section 5). Finally, we analyse the results of running our system on the test problems (Section 6).

2 Theorist

A THEORIST system attempts to create an *explanation*[1] of an *observation* by using a knowledge-base made up of *facts* and *hypotheses.* Facts are statements that are held to be absolutely true about the world. Hypotheses are statements that are contingently true – conclusions based on hypotheses may be invalidated in the light of changes in the world or as the result of the acquisition of new information.

An explanation is a set of instances of hypotheses which is both consistent with the facts and which also logically implies the observation.

Reasoning with THEORIST enjoys the property of *semimonotonicity.* As a result, the construction and consistency checking of the explanation can be performed incrementally. This means that the construction of an explanation can be broken down into two distinct but interleaved phases within an implementation: an *explanation phase*, where hypothesis instances are selected for inclusion in a possible explanation; and a *consistency check phase*, where the new hypothesis instances are tested for consistency with the current partial explanation.

THEORIST knowledge-bases may also include *constraints.* These specify restrictions on the context in which hypotheses may be used. Constraints are only used in the consistency check phase, which means that they cannot be used to derive or extend theories.

An example THEORIST knowledge-base can be seen in Figure 3.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

[1] The terms explanation and theory are used interchangeably in this paper.

3 Lemmas

Lemmas are previously proven results that can be reused in a proof. For a logic programming system, the results of a successful derivation can be recorded to possibly reduce the work required to reestablish that derivation. The recording and use of lemmas in systems is sometimes also known as *tabling* or *memoization*. Systems like XSB [14] use tabling. There are deep problems with adding tabling to Prolog systems because of the untamed nature of negation as failure. THEORIST systems don't encounter these problems because its default mechanism is not recursive – a consistency check cannot invoke an explanation phase or another consistency check phase.

THEORIST can exploit two sorts of lemmas: those recording information about partially constructed explanations; and those recording information about derivation trees.

Remembering successful partial theories i.e., those that had been shown consistent with the facts, is achieved by remembering *goods*. Inconsistent explanations are recorded as *nogoods*. This terminology comes by analogy with automatic truth maintenance systems and was suggested by Sattar and Goebel [17].

Completed derivations from the explanation phase are tabled as *derived literals* and information derived from the consistency checking phase is stored as *potential crucial literals* [16, 17]. In [17] derived literals and potential crucial literals can only be accepted if their derivations do not involve any model elimination steps. This restriction is unnecessary. For instance, the caching described by Astrachan and Stickel in [1] does not impose this restriction.

Our THEORIST implementation incorporates all these lemma types in a practical system. It also extends them with some generalisations and refinements.

The recording and reuse of goods and nogoods occurs during the *theory extension step* of THEORIST. The theory extension step occurs when a hypothesis is being considered for inclusion in the current theory, and is carried out in the following way. The proposed new theory including the hypothesis is checked to see if is already recorded as a good or a nogood. If it is recorded as a good, this means that the theory has already passed the consistency check, so the extension step may omit the check for this theory and can succeed directly. If the new theory is recorded as a nogood, it has already failed a consistency check, so the extension step may fail immediately. If the proposed new theory is neither a good nor a nogood, the consistency check call is made for the negation of the new hypothesis being added to the theory. If this call succeeds, the new hypothesis must be incompatible with the existing theory, so the proposed new theory is recorded as a nogood and the extension step fails. If the consistency check call fails, the new hypothesis is consistent with the current theory, so the new theory is recorded as a good and the extension step succeeds. See Figure 4 in the lines starting `%% Extend theory`) for the Prolog code for the extension step.

Derived literals are stored as tuples $< l, s, m^+, m^-, c >$ where: l is the derived literal; s is its *support set*, which is the set of hypothesis instances added to the explanation during the derivation of the derived literal – in Figure 1 the support set for l_1 is $\{h_2\}$; m^+ is the set of positive literals involved in model elimination (MESON) steps during the derivation; m^- is the set of negative literals involved in model elimination steps during the derivation; and c is the explanation at the end of the derivation of l. m^+ and m^- are called *model elimination contexts* and c is called the *explanation context*. The model elimination contexts only contain the literals involved in model elimination steps that used ancestor literals of the derived literal.

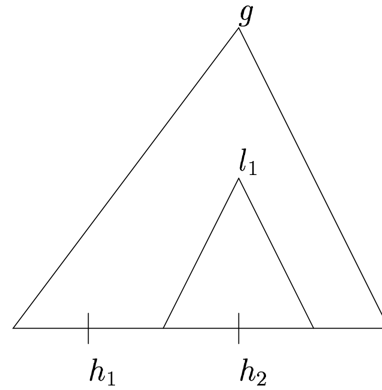

Figure 1: Derived literal l_1

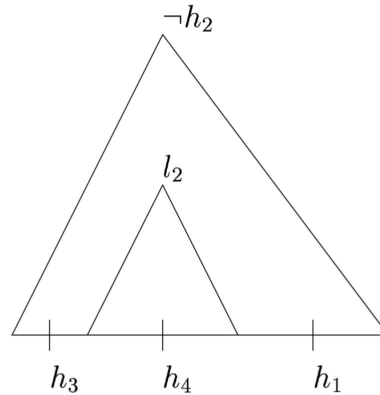

Figure 2: Potential crucial literal l_2

When a derived literal becomes a candidate for reuse, its model elimination contexts are checked against the ancestors of the current goal. If unifying ancestor literals – goals that appear higher up in the derivation tree of the derived literal – are found and the support set is consistent with the current partial explanation then the derived literal may be reused.

When reusing a derived literal, the explanation context can be used to reduce the work of the consistency check. The current theory is checked to see if it is a subset of the recorded explanation context. If it is, then the consistency check for the support set can be omitted completely. Otherwise, the consistency check for the support set can work with the difference of the explanation context and the current theory. Once the derived literal has been vetted and approved, a new derived literal might be recorded, identical to the first but with a new explanation context reflecting the current theory.

Potential crucial literals are stored in a similar manner to derived literals, and their reuse is similar, except that the theory context is not used. Also, potential crucial literals provide more information than is obvious at first glance. If we look at Figure 2, we can see that the support set for l_2 is $\{h_4\}$. We can also see that $\{h_1, h_2, h_3, h_4\}$ is a nogood, since $\neg h_2$ is consistent with $\{h_1, h_3, h_4\}$. From this, it is possible to deduce that the support set for $\neg l_2$ is $\{h_1, h_2, h_3\}$ [17], so we can store potential crucial literal informa-

tion for $\neg l_2$. Note that this inference is only valid if there are no model elimination steps during the derivation of l_2.

4 Implementation

THEORIST is typically implemented by extending Loveland's MESON theorem prover [8]. The first interpreters appeared in the mid to late 1980s and Poole and Goodwin's compiler appeared in 1987 [12]. Though these systems bear similarities to Stickel's Prolog Technology Theorem Prover [20], which is itself derived from the MESON system, they were developed independently.

The system described in this paper is a completely new compiler. It takes a THEORIST knowledge-base as input and produces stand alone Prolog outputs. Inputs can request the inclusion of features like derived literals in the generated code through switches. Attempts have been made to optimise the output code: clauses or calls that always fail are eliminated; code that can never be called doesn't appear in the output. The encoding of the rules in the THEORIST input is split into two sorts of output predicate: *drivers* and *sub-predicates*. Drivers incorporate all the housekeeping of the MESON procedure. Sub-predicates are created from the input rules found in the knowledge-base. This division has the advantage of retaining some of the benefits of argument indexing for sub-predicates.

A driver from the explanation phase generated from a knowledge base for the binary full-adder used in some tests (see Section 5) is shown in Figure 4. This driver includes code for generating and recording nogoods and derived literals. From this code it is easy to see how the basic MESON proof procedure has been extended with the theory extension steps of THEORIST and the lemma recording and reuse we have introduced.

Lemmas are stored by asserting a compiled form into the Prolog database and lemma instances are reused directly through Prolog goals.

The compilation is carried out in the following way:

Derived literals and potential crucial literals The obvious way to remember a literal of the form `p(t1, ..., tn)` with a support set `Support`, positive model elimination context `PosMEC`, negative model elimination context `NegMEC` and theory context `Context` is as `dl(p(t1, ...,tn), Support, PosMEC, NegMEC, Context)` or `pcl(p(t1, ...,tn), Support, PosMEC, NegMEC, Context)`. However, this robs us of the chance to use indexing on the functor name and arity of the literal. Instead, literals are recorded as `dl_p(t1, ..., tn, Support, PosMEC, NegMEC, Context)` or `pcl_p(t1, ..., tn, Support, PosMEC, NegMEC, Context)`. This form can be seen in Figure 4 at the lines after `%% Reuse derived literal`.

Goods and nogoods Goods and nogoods are represented by sorted lists of hypothesis instances, for example `[h1,h3,h4]`. To compile these for assertion in the database, we first take a hash value of the list. For our example, this is 15544339. We then use this value to form the functor for the fact to be asserted, which will be either `good_15544339` or `nogood_15544339` depending on whether we're asserting a good or nogood. Finally, we take the list and use it as the arguments for the asserted fact. Thus, we end up asserting `good_15544339(h1,h3,h4)` or `nogood_15544339(h1,h3,h4)`. We use this method because the most direct method of storing goods and nogoods – e.g. as `good([h1,h3,h4])` – is inefficient, as there is no indexing available on the functor name and arity of the predicate and lists are more expensive to unify than a sequence of predicate arguments.

The compiler implements extended versions of the concepts presented in [17]. These include:

- The inclusion of context arguments for derived literals. This comes from the observation that derived literals are generated in a particular *theory context*. When reuse of a derived literal is considered, the current partial theory can be tested against the context in which the derived literal was derived. If the current theory is a subset of that context, the consistency check of the support set can be simplified or eliminated.
- Allowing the generation of potential crucial literal information wherever a proof tree is rooted at a negated hypothesis. The argument that allows information to be deduced about the negation of a potential crucial literal in the consistency check phase can be readily extended to this more general case.

Excluding support modules – parser, clausal form conversion code, etc.– the compiler consists of about 3000 lines of Prolog.

5 Test Problems

While nonmonotonic reasoning has been a field of study for a quarter of a century, it is surprising to find that very few realistic large problems have been collected[2]. Typically, the problems presented in the literature are small, of the order of complexity of the ubiquitous "birds fly" example [7] and are intended to establish or refute some semantic property of the reasoning system being discussed. Their small size means that they would not stress any practical implementation sufficiently to permit a performance evaluation.

Since the mid 1990s researchers have been using problems derived from graphs generated by Knuth's Stanford GraphBase system [6]. The authors of the DERES [2] and XRAY [19] default reasoning systems used this approach to generate test problems.

The graphs used are *complete graphs* of n vertices. Complete graphs are graphs whose vertices are linked to every vertex by an arc. The problems that we used for our tests involved finding a Hamilton walk over a complete graph. A Hamilton walk is a connected sequence of arcs that reaches each vertex once. It is clear that these problems have a combinatorial aspect – as the number of vertices increases, the number of candidate sequences to be checked rises dramatically. Our tests used complete graphs with 5 to 8 vertices. We used a small PERL script to generate our THEORIST knowledge-bases.

Our system was also tested by running problems based on fault diagnosis using a binary full-adder circuit. The knowledge-base for the full-adder problems uses variables. This gives us the opportunity to test the performance of the system when dealing with nondeterministic goals.

Because of the sensitivity to the ordering of literals in the input (also mentioned in [18]), had reordering applied to the literals of the input rules. The literals

[2] Compare this situation with the field of automated theorem proving which has the TPTP database [21].

were either *ordered*, *reversed* or *randomised* with respect to the standard Prolog ordering of terms. Ten random orderings were produced for each problem.

Ordered literals The literals were ordered according to the Prolog standard ordering on terms. This places uninstantiated variables before integers and atoms. The effect of this ordering is to sort goals lexicographically by functor, and to place goals with variables in a particular argument position before those that have constants in the same argument position.

Reversed literals This is the reverse of the above order. Goals are ordered in reverse lexicographic order on their functor, and goals that have constants in a particular argument position are placed before those that have variables in the same argument position.

Randomised literals The literals were rearranged randomly within the rule bodies. Ten random variations were produced for each problem.

We would expect that the Hamilton walk problems are particularly sensitive to ordering. Problems that perform well with a particular ordering might be expected to perform poorly with that ordering reversed.

Additionally, we kept a version of the input exactly as they were produced by our generator to see if it produced efficient codings for the problems.

These reordered versions were then compiled with switches selecting different combinations of lemmas.

Each executable was run to find only the first solution. This is discussed further in Section 6.

Problem runs were limited to 1000 seconds of CPU time [3]. Three runs of each problem were made. The CPU time recorded was the average of the CPU time for these runs.

6 Results and Analysis

Figures 5 and 6 show the performance results for the problems. Each problem was run under each of the eight possible combinations of lemma types: no lemmas; derived literals only (DL); nogoods only (NG); derived literals and nogoods (DL + NG); potential crucial literals only (PCL); derived literals and potential crucial literals (DL + PCL); nogoods and potential crucial literals (DL + PDL); and, derived literals, nogoods and potential crucial literals (DL + NG + PCL).

The CPU timings are in seconds. The result columns titled “DL”, “G”, “NG” and “PCL” respectively show the number of derived literals, goods, nogoods and potential crucial literals generated for the indicated run.

The results for the Hamilton walk problems show that if the literal ordering is left untouched, then the output of the problem generator without lemmas performs best. Looking at the results for the case where the literals were ordered, we see improvements across the board. In fact, ordered literals seem to help the performance of lemmas. The best overall result is for the case where derived literals and potential crucial literals are combined with ordered literals. This produces an improvement over the case with ordered literals but without lemmas of about 30%. With the exception of derived literals on their own, all combinations of lemmas with ordered literals seem to produce improvement of at least 25% in run time.

For the full-adder problem, it is difficult to extract exact results, given that all the lemma combinations that didn’t involve potential crucial literals produce run times that are below the lower limit of resolution. The only solid conclusion is that it is unwise to use potential crucial literals for this problem. The generation of potential crucial literals requires that each consistency check proof tree be fully generated before information can be extracted from it and failure flagged. However, when potential crucial literals are not being generated, the consistency check proof tree can be abandoned at any time when failure is detected. Because the full-adder uses variables and because potential crucial literals depend on the complete generation of the consistency check proof tree, it might be that the consistency check tree contains a lot of nondeterministic behaviour that is slowing things down. Profiling of the consistency check phase of the system is indicated.

Profiling reveals that as the problems get larger, the overheads of the loop checks and the model elimination checks start to dominate. The current implementation represents the ancestors of a node as a list which has linear complexity for searches. A more efficient representation is obviously required, but it is difficult to see how this might be implemented except by going outside of Prolog and writing some foreign language code to implement some sort of hash table or other more efficient data structure.

An important factor in these results is that they only cover the generation of the first solution. The generation of multiple solutions or all solutions has not been widely canvassed in the literature. Work on theory preference [4] suggest that the ability to efficiently generate many solutions would be a worthwhile contribution.

7 Discussion and Future Work

We have shown that under certain circumstances, lemmas such as goods, nogoods, derived literals and potential crucial literals can produce significant improvement in the execution speed of some problems. It is obvious that other problems need to be investigated to find out what typifies a good situation for the use or non-use of these lemmas.

Compile time measures such as generating code with certain orderings of literals are obviously worthwhile. In our Hamilton walk examples, it was clear that ordering of the literals provided improvements. This might be because the start node for any solution is specified in the database as `fact e(0)`. We conjecture that if this were changed to e.g. `fact e(n)` where `n` is the highest numbered vertex, the reverse ordering of literals would result in a similar improvement.

More sophisticated forms of static analysis such as that used in the HYPER system [13] should also be investigated for efficiency boosts when applied to large problems. The representation used for our Hamilton walk problems deliberately avoided variables, so the techniques of the HYPER system were not applicable. It might be possible to apply them to the full-adder problem.

Using artificial combinatorial problems leaves us uneasy. While it is true that finding Hamilton walks is computationally hard, the problems offer no insight into the real world reasoning. Where are the large, “real life” problems for nonmonotonic reasoning? With the recent release of the OpenCyc portion of Cyc (`http://www.opencyc.org`) this gap might be remedied.

Finally, we believe that with the employment of goods, nogoods, derived literals and potential crucial literals, the run-time lemma types available to systems based on the MESON procedure or SLD reso-

[3] We won’t give exact system details, as we are only concerned with relative performance.

lution have been exhausted. It might be time turn to other techniques derived from other sources e.g., local search as applied to constraint satisfaction problems [22], or linear programming [5].

References

[1] Owen L. Astrachan and Mark E. Stickel. Caching and lemmaizing in model elimination theorem provers. In *Conference on Automated Deduction*, pages 224–238, 1992.

[2] Pawel Cholewinski, Victor W. Marek, and Miroslaw Truszczynski. Default reasoning system DeReS. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kaufmann, San Francisco, California, 1996.

[3] Jürgen Dix et al., editors. *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97*, volume 1265 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, July 1997.

[4] S. Goodwin and A. Sattar. On computing preferred explanations. In *Proceedings of Australian Joint Conference on Artificial Intelligence*, pages 45–52, Melbourne, Victoria, November 1993. World Scientific Publishing Co.

[5] M. Ishizuka and Y. Matuso. SL method for computing a near-optimal solution using linear and non-linear programming in cost-based hypothetical reasoning. *Knowledge-Based Systems*, 15:369–376, 2002.

[6] Donald E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* ACM Press, New York, NY, 1993.

[7] Vladimir Lifschitz. Benchmark problems for formal nonmonotonic reasoning. In M. Reinfrank, J. de Kleer, M. L. Ginsberg, and E. Sandewall, editors, *Proceedings of the Second International Workshop on Non-Monotonic Reasoning*, pages 202–219, Grassau, Germany, June 1988. Springer Verlag. Volume 346, Lecture Notes in Artificial Intelligence.

[8] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North Holland, Amsterdam, The Netherlands, 1978.

[9] Michael J. Maher, Andrew Rock, Grigoris Antoniou, David Billington, and Tristan Miller. Efficient defeasible reasoning systems. *International Journal on Artificial Intelligence Tools*, 10(4):483–501, 2001.

[10] D. Poole. Compiling a default reasoning system into Prolog. Research report, Department of Computer Science, University of Waterloo, 1988.

[11] David Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, August 1988.

[12] D.L. Poole and S.D. Goodwin. A Theorist to Prolog compiler. Available via `http://www.cs.ubc.ca/~poole/theorist.html`, August 1987.

[13] Helmut Prendinger, Mitsuru Ishizuka, and Tetsu Yamamoto. The hyper system: Knowledge reformation for efficient first-order hypothetical reasoning. In *Pacific Rim International Conference on Artificial Intelligence*, pages 93–103, 2000.

[14] P. Rao et al. XSB: A system for efficiently computing WFS. In Dix et al. [3], pages 430–440.

[15] A. Sattar and R. Goebel. Reducing explanation space for circumscriptive theorem provers. In *Proceedings of Australian Joint Conference on Artificial Intelligence*, pages 290–295, Hobart, Tasmania, November 1992. World Scientific Publishing Co.

[16] A. Sattar and R.Goebel. Consistency-motivated reason maintenance in hypothetical reasoning. Technical Report TR-91-12, Department of Computing Science, University of Alberta, 1991.

[17] Abdul Sattar and Randy Goebel. Consistency-motivated reason maintenance in hypothetical reasoning. *New Generation Computing*, 15:163–186, 1997.

[18] T. Schaub. XRay: A Prolog technology theorem prover for default reasoning (DEMO). In Emil Weydert, Gerd Brewka, and Cees Witteveen, editors, *Proceedings of the 3rd Dutch/German Workshop on Nonmonotonic Reasoning Techniques and their Applications*, pages 177–182, Saarbrücken, 1997. Max-Planck-Institut für Informatik.

[19] T. Schaub and P. Nicolas. The X-Ray system: its implementation and evaluation. In Dix et al. [3], pages 441–452.

[20] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction*, volume 230, pages 573–587, Berlin, 1986. Springer-Verlag.

[21] Christian B. Suttner and Geoff Sutcliffe. The TPTP problem library. Technical Report JCU-CS-96/9, James Cook University, Queensland, Australia, 18 1996.

[22] J. Thornton. *Constraint Weighting Local Search for Constraint Satisfaction.* PhD thesis, School of Computing and Information Technology, Griffith University, Australia, January 2000.

A Figures and Tables

```
fact arc(0, 0). fact arc(0, 1). fact arc(0, 2).
fact arc(1, 0). fact arc(1, 1). fact arc(1, 2).
fact arc(2, 0). fact arc(2, 1). fact arc(2, 2).

fact e(0).

hypothesis w(0, 0): e(0); not e(0); not arc(0, 0).
hypothesis w(0, 1): e(1); not e(0); not arc(0, 1).
hypothesis w(0, 2): e(2); not e(0); not arc(0, 2).

hypothesis w(1, 0): e(0); not e(1); not arc(1, 0).
hypothesis w(1, 1): e(1); not e(1); not arc(1, 1).
hypothesis w(1, 2): e(2); not e(1); not arc(1, 2).

hypothesis w(2, 0): e(0); not e(2); not arc(2, 0).
hypothesis w(2, 1): e(1); not e(2); not arc(2, 1).
hypothesis w(2, 2): e(2); not e(2); not arc(2, 2).

hypothesis w(0, 0): ee(0); not e(0); not arc(0, 0).
hypothesis w(1, 0): ee(0); not e(1); not arc(1, 0).
hypothesis w(2, 0): ee(0); not e(2); not arc(2, 0).

constraint not w(0, 0).
constraint not w(0, 1) <- w(0, 2).
constraint not w(0, 2) <- w(0, 1).

constraint not w(1, 0) <- w(1, 2).
constraint not w(1, 1).
constraint not w(1, 2) <- w(1, 0).

constraint not w(2, 0) <- w(2, 1).
constraint not w(2, 1) <- w(2, 0).
constraint not w(2, 2).

fact goal <- e(0), e(1), e(2), ee(0).
```

Figure 3: Hamilton walk database for complete graph with 3 vertices.

```
explain_ok(Arg1, Support0, Support, PosMEC0, PosMEC, NegMEC0, NegMEC, PCL0, PCL,
           Theory0, Theory, Uninst0, Uninst, Depth, PosAncs, NegAncs) :-
        Lit=ok(Arg1), NegLit=neg_ok(Arg1),
        (   %% Loop check (not shown)
        ;   %% Model elimination check (not shown)
        ;   %% Reuse derived literal
            dl_ok(Arg1, XXSupport, XXPosMEC, XXNegMEC, XXContext),
            mec_confirm(XXPosMEC, NegAncs), mec_confirm(XXNegMEC, PosAncs),
            (   subset(XXSupport, Theory0) -> Theory0=Theory
            ;   subset(Theory0, XXContext)
            ->  union(XXSupport, Theory0, Theory1), sort(Theory1, Theory)
            ;   difference(XXSupport, Theory0, X),
                cc_set(X, Theory0), union(X, Theory0, Theory1), sort(Theory1, Theory)
            ),
            union(XXSupport, Support0, Support),
            append(XXPosMEC, PosMEC0, PosMEC), append(XXNegMEC, NegMEC0, NegMEC),
            Uninst0=Uninst, PCL0=PCL
        ;   %% Reuse hypothesis
            ground(Lit), memberchk(Lit, Theory0),
            union([Lit], Support0, Support), PosMEC0=PosMEC, NegMEC0=NegMEC, PCL0=PCL,
            Theory0=Theory, Uninst0=Uninst
        ;   %% Sub call
            Depth1 is Depth-1,
            explain_ok_sub(Arg1, [], SubSupport, [], SubPosMEC, [], SubNegMEC, [], SubPCL,
                           Theory0, Theory, Uninst0, Uninst, Depth1, [Lit|PosAncs], NegAncs),
            mec_filter(SubPosMEC, Depth, SubPosMECF), mec_filter(SubNegMEC, Depth, SubNegMECF),
            derived_add(dl_, pos, ok, 1, [Arg1], SubSupport, SubPosMECF, SubNegMECF, Theory0),
            union(SubSupport, Support0, Support),
            append(SubPosMEC, SubPosMEC0, SubPosMEC), append(SubNegMEC, SubNegMEC0, SubNegMEC),
            append(SubPCL, PCL0, PCL)
        ;   %% Extend theory
            ground(Lit), \+memberchk(Lit, Theory0),
            sort([Lit|Theory0], Theory),
            (   is_nogood(Theory) -> fail
            ;   is_good(Theory)
            ->  union([Lit], Support0, Support), PosMEC0=PosMEC, NegMEC0=NegMEC, PCL0=PCL
            ;   %% Consistency check call
                prove_neg_ok(Arg1, [], CCSupport, [], CCPosMEC, [], CCNegMEC,
                             [], CCPCL, Depth-1, fail, [], [], Theory0)
            ->  union([Lit], CCSupport, CCNogood0),
                sort(CCNogood0, CCNogood),
                nogood_add(CCNogood),
                (   CCPosMEC=[], CCNegMEC=[] -> derived_list_add(pcl_, CCPCL, CCNogood)
                ;   true
                ),
                (   subset(CCSupport, Theory)
                ->  nogood_add(Theory),
                    fail
                ;   good_add(Theory),
                    union([Lit], Support0, Support),
                    PosMEC0=PosMEC, NegMEC0=NegMEC, PCL0=PCL
                )
            ;   good_add(Theory),
                union([Lit], Support0, Support),
                PosMEC0=PosMEC, NegMEC0=NegMEC, PCL0=PCL
            ),
            Uninst0=Uninst
        ;   %% Nonground hypothesis
            \+ground(Lit),
            Support0=Support, PosMEC0=PosMEC, NegMEC0=NegMEC,
            PCL0=PCL, Theory0=Theory, Uninst=[Lit|Uninst0]
        ).
```

Figure 4: Example explanation phase driver.

			No lemmas	DL		NG			DL + NG			
Problem	Order		Time	Time	DL	Time	G	NG	Time	DL	G	NG
Hamilton 5	Untouched		0.01	0.02	21	0.01	5	21	0.01	21	5	21
	Ordered		< 0.01	< 0.01	16	< 0.01	7	21	< 0.01	16	7	21
	Randomised	Min	< 0.01	0.01	24	0.01	25	70	< 0.01	18	11	23
		Max	0.06	0.07	25	0.06	154	175	0.07	25	154	175
	Reverse		0.02	0.03	21	0.02	14	27	0.03	21	14	27
Hamilton 6	Untouched		0.06	0.08	28	0.07	6	39	0.09	28	6	39
	Ordered		0.05	0.05	19	0.04	8	37	0.04	19	8	37
	Randomised	Min	0.06	0.06	26	0.03	29	57	0.04	26	29	57
		Max	1.38	1.48	105	0.42	424	796	0.48	105	424	796
	Reverse		0.18	0.22	28	0.17	20	51	0.23	28	20	51
Hamilton 7	Untouched		0.55	0.66	36	0.57	7	66	0.68	36	7	66
	Ordered		0.42	0.43	22	0.32	9	58	0.32	22	9	58
	Randomised	Min	0.45	0.47	26	0.26	52	184	0.26	38	52	184
		Max	20.60	21.33	77	5.19	993	1874	5.42	77	993	1874
	Reverse		1.78	2.07	36	1.76	27	86	2.05	36	27	86
Hamilton 8	Untouched		5.31	6.23	45	5.47	8	104	6.26	45	8	104
	Ordered		4.04	4.19	25	2.91	10	84	3.00	25	10	84
	Randomised	Min	49.18	50.71	84	7.29	459	681	7.60	49	459	681
		Max	814.62	837.30	304	143.83	8864	13897	149.41	304	8864	13897
	Reverse		19.26	21.94	45	19.11	35	134	21.65	45	35	134
Full-adder	Untouched		< 0.01	< 0.01	25	< 0.01	6	1	< 0.01	25	6	1
	Ordered		< 0.01	< 0.01	26	< 0.01	6	1	< 0.01	26	6	1
	Randomised	Min	< 0.01	< 0.01	24	< 0.01	6	1	< 0.01	28	6	1
		Max	0.01	0.01	24	0.01	5	1	0.01	25	5	1
	Reverse		< 0.01	< 0.01	30	< 0.01	5	1	< 0.01	30	5	1

Figure 5: Performance results

Problem	Order		PCL		DL + PCL			NG + PCL				DL + NG + PCL				
			Time	PCL	Time	DL	PCL	Time	G	NG	PCL	Time	DL	G	NG	PCL
Hamilton 5	Untouched		0.01	0	0.06	24	0	0.01	5	30	0	0.05	24	25	95	0
	Ordered		0.01	0	0.01	7	0	< 0.01	7	32	0	< 0.01	7	6	13	0
	Randomised	Min	0.02	29	0.01	7	12	0.01	11	31	12	0.01	7	10	10	12
		Max	0.11	44	0.36	25	47	0.09	154	187	44	0.15	25	109	117	47
	Reverse		0.04	31	1.38	20	31	0.04	14	42	31	1.14	20	14	54	31
Hamilton 6	Untouched		0.08	0	1.43	34	0	0.08	6	53	0	0.56	34	73	255	0
	Ordered		0.06	0	0.04	8	0	0.04	8	56	0	0.04	8	7	21	0
	Randomised	Min	0.09	28	0.33	32	35	0.06	29	71	28	0.22	32	39	138	35
		Max	2.36	183	95.35	107	183	0.80	309	457	230	17.71	107	424	896	125
	Reverse		0.31	51	69.68	27	51	0.30	20	77	51	56.12	27	20	101	51
Hamilton 7	Untouched		0.66	0	78.86	46	0	0.64	7	86	0	11.71	46	266	733	0
	Ordered		0.46	0	0.31	9	0	0.35	9	87	0	0.31	9	8	31	0
	Randomised	Min	1.44	60	5.05	52	543	0.53	52	214	57	1.26	50	128	496	131
		Max	49.83	987	166.35	51	528	14.55	993	1919	788	111.86	51	570	690	485
	Reverse		2.72	78	> 1000	–	–	2.63	27	126	78	> 1000	–	–	–	–
Hamilton 8	Untouched		6.16	0	> 1000	–	–	5.98	8	131	0	402.60	60	1214	2531	0
	Ordered		4.49	0	2.86	10	0	3.22	10	125	0	2.88	10	9	43	0
	Randomised	Min	139.34	1758	177.03	92	2072	11.91	459	723	254	30.91	119	836	1492	785
		Max	> 1000	–	> 1000	–	–	555.04	6056	16746	4078	> 1000	–	–	–	–
	Reverse		27.82	113	> 1000	–	–	26.71	35	191	113	> 1000	–	–	–	–
Full-adder	Untouched		169.19	60	169.96	11	53	166.69	6	2	60	167.04	11	6	2	53
	Ordered		9.73	48	9.85	11	48	9.63	6	2	48	9.59	11	6	2	48
	Randomised	Min	2.84	72	4.52	21	72	2.81	6	2	72	4.44	21	6	2	72
		Max	116.08	66	126.17	15	76	115.48	6	2	66	124.12	15	6	1	76
	Reverse		31.52	74	62.96	27	74	31.21	5	2	74	62.21	27	5	2	74

Figure 6: Performance results (continued)