

Code Improvements for Model Elimination Based Reasoning Systems

Richard A. Hagen

Knowledge Representation and Reasoning Unit
School of Information Technology
Griffith University
Queensland, Australia
r.hagen@mailbox.gu.edu.au

Scott Goodwin

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

Abdul Sattar

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

Abstract

We have been investigating ways in which the performance of model elimination based systems can be improved and in this paper we present some of our results. Firstly, we have investigated code improvements based on local and global analysis of the internal knowledge base used by the theorem prover. Secondly, we have looked into the use of a n lists to represent ancestor goal information to see if this gives a performance boost over the traditional two list approach. This n list representation might be thought of as a simple hash table. Thirdly, we conducted initial investigations into the effect of rule body literal ordering on performance.

The results for the code improvements show them to be worthwhile, producing gains in some example problems. Using the n list representation gave mixed results: for some examples it improved execution speed, in others it degraded it. A rule body literal ordering that placed instantiated goals (including hypotheses) early in the bodies of rules showed an improvement in execution time.

1 Introduction

Model elimination theorem provers based on the framework described by Loveland (Loveland 1969, Loveland 1978) provide simple, complete reasoning systems. Several systems have been based on the model elimination template, including Stickel's Prolog Technology Theorem Prover (PTTP) (Stickel 1986), THEORIST (Poole 1988b, Poole & Goodwin 1987), DERES (Cholewinski, Marek & Truszcynski 1996) and XRAY (Schaub 1997).

In (Hagen, Sattar & Goodwin 2003), we presented potential improvements to the THEORIST hypothetical reasoning system. These improvements were based on observations of the THEORIST model itself. In the work presented in this paper, we investigate

possible improvements to the underlying model elimination implementation. In related work, de Waal and Gallagher (de Waal 1994, de Waal & Gallagher 1994) discuss the removal of *useless clauses*, clauses that cannot succeed. In our work, we have investigated closer to the underlying coding of the implementation and turned up some possible improvements. Our research used a THEORIST implementation as its test bed.

In (Astrachan & Stickel 1992), Astrachan and Stickel suggest that an alternative representation for ancestor information might provide runtime improvement. We take up this challenge, proposing and testing an ancestor goal information representation that uses n lists of literals. This contrasts with the traditional two list approach.

The performance of model elimination based systems is quite dependent on the literal ordering within the bodies of the underlying rule set. We investigated four new literal orderings: instantiated literals first and last and hypotheses first and last. Hypotheses are a feature of the THEORIST paradigm, and are explained in Section 3.

In Section 2 we outline the way in which model elimination theorem provers work; Section 3 gives a brief overview of the THEORIST paradigm; in Section 4 we describe our improvements for the generated code; in Section 5 we describe the tests used to evaluate our improvements; and in Section 6 we lay out and analyse the performance results. Finally, Section 7 presents our conclusions based on the tests and presents some ideas for future work.

2 Model Elimination Theorem Provers

Existing depth first model elimination based theorem provers work in a similar way to pure Prolog systems. In fact, they are usually implemented in Prolog to take advantage of the similarities. However, there are differences. Model elimination-based systems guarantee completeness, and they do this through the use of the following features.

Logical Negation. Negation is treated purely logically. Clauses may have negated heads (see below). When matching a goal with a clause head, the negation is taken into consideration. This

contrasts with Prolog’s use of a default negation mechanism (*negation as failure*).

Sound unification. Unification in Prolog systems is typically carried out in an *unsound* manner. That is, unifications of the form $X = f(\dots X \dots)$ succeed. Model elimination systems must guarantee that unification is sound so that such unifications fail.

Contrapositives¹. Input rules are translated to internal clauses that include all contrapositives. For instance, an arbitrary input rule with head a and body $b \ \& \ c$, i.e.,

```
a <- b & c.
```

would be translated into the internal clauses

```
a <- b & c.
¬ b <- ¬ a & c.
¬ c <- ¬ a & b.
```

Model elimination. During the execution of a model elimination based theorem prover, new sub-goals are checked to see if they make a complementary match with any ancestor, in which case a successful derivation is recorded. For example, the following derivation for arbitrary literal p succeeds.

```
p
⋮
¬p
```

Matching is done by way of unification. This is called a *model elimination step*.

Complete Search Strategy. Use of a complete search strategy, e.g., iterative deepening, together with the other features, ensures that the model elimination procedure itself is complete.

Additionally, a *loop checking* step can be performed. In this case, derivations where a goal is identical to an ancestor fails immediately. This is not essential for completeness, but can help efficiency. Throughout this paper, we assume the presence of such a loop check.

3 Theorist

A THEORIST system attempts to create an *explanation* 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 represent classes of statements that are contingently true. For instance, an input knowledge base might contain the hypothesis

```
hypothesis ok(G).
```

This means that we have a licence to hypothesise using ground instances of $ok/1^2$ during the course of a THEORIST derivation. Conclusions based on hypothesis instances 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 taken with the facts logically implies the observation.

¹Baumgartner and Furbach (Baumgartner & Furbach 1994a, Baumgartner & Furbach 1994b) present a method of writing model elimination theorem provers without using contrapositives.

²The notation *name/arity* is used in the Prolog community to state a functor. In this case, the name is *ok* and its arity is 1.

Implementations of THEORIST systems extend the model elimination proof procedure to include the accumulation of an explanation during the proof of a goal. A description of this process can be found in Poole (Poole 1988a).

As an extension, THEORIST implementations also provide *constraints*. A constraint is a formula that restricts the use of hypothesis instances. As an example, the constraint

```
constraint ok(G) <- not faulty(G).
```

specifies that instances of hypotheses $ok(\mathbf{X})$ and $faulty(\mathbf{X})$ for the same value of \mathbf{X} are mutually exclusive and cannot appear together in the same explanation.

4 Improvements

4.1 Code Improvements

The code improvements we investigated include clause level, predicate level and program level analysis of the internal program.

In order to motivate some of the improvements, we require the following definition.

Definition: A *childless* predicate is one whose clauses are all facts.

1. Childless predicates can omit the loop check, since a goal corresponding to a childless predicate cannot be the ancestor of any other goal.
2. The complementary predicate of a childless predicate can omit the model elimination check because a goal corresponding to a childless predicate cannot be the ancestor of any other goal. (The complementary predicate for p is $\neg p$, and vice versa.)
3. Clauses for calls that lead to self loop check induced failure can be removed. If a literal within a clause is identical with the head, that clause will always fail, and the clause can be removed.
4. Clauses with calls to non-existent predicates can be removed. The internal clauses include contrapositives generated from the input rules. These contrapositives might contain literals that refer to non-existent predicates. When executed, this will lead to the failure of some clauses. Such clauses can be removed.
5. Duplicate clauses can be removed. Duplicate clauses might arise due to the generation of contrapositives or through a translation from first-order form to clausal form³. These can be removed in order to reduce redundant computation.
6. Duplicate literals within clauses can be removed. Such literals might arise during a translation from first-order form to clausal form. They can be eliminated to reduce redundant computation.

We were tempted to consider another possible “improvement”: literals in a clause that are the complement of the head can be removed. The rationale for this is that since such goals cause an immediate and trivial model elimination they are redundant. However, it is possible that such literals might succeed in their own right during evaluation, so they should be retained. In systems that do not construct contextual information during evaluation, eliminating these literals *when they are ground* might be a valid improvement.

³The translation of inputs from first order to clausal form is a feature of THEORIST implementations, not model elimination theorem provers in general.

4.2 n List Ancestor Information Representation

THEORIST implementations traditionally use two lists for the storage of ancestor information: one for the positive ancestors, and one for the negative ancestors. The possible improvement we looked into is the use of a n lists for the representation of ancestors.

We replace the two ancestor lists with an ancestor structure.

```
ancs(Pred1Anc, ..., PredNAnc)
```

This structure contains lists of ancestor goals corresponding to each predicate in the program. Our compiler assigns a unique number to each predicate so that accessing the lists of this structure can be done in fixed, constant time.

In our implementation, we update the ancestors prior to executing a goal and reset this update after a successful derivation.

```
setarg(i, Ancs, [pred(...)|PredAncs]),
pred(..., Ancs),
setarg(i, Ancs, PredAncs)
```

setarg/3 undoes its actions on backtracking, so the update is reversed on failure. The value of i is the number assigned by the compiler to the predicate for the called goal.

Model elimination checks can be implemented as a lookup followed by a membership check of the complementary predicate's ancestor lists.

```
arg(j, Ancs, CompPredAncs),
member_unifiable(pred(...), CompPredAncs)
```

member_unifiable/2 is defined as

```
member_unifiable([H | T], E) ->
  ( unify_with_occurs_check(H, E)
  ; member_unifiable(T, E)
  ).
```

Note the use of a sound form of unification. If this constraint was lifted, we could use the standard member/2 predicate.

Loop checks are implemented as a lookup of the appropriate ancestor list, followed by failure if an identical copy of the current goal is a member of this predicate's matching ancestors.

```
arg(i, Ancs, PredAncs),
memberchk_identical(pred(...), PredAncs) ->
  fail
```

memberchk_identical/2 is defined as

```
memberchk_identical([H | T], E) ->
  ( H == E -> true
  ; memberchk_identical(T, E)
  ).
```

One way to view this n list structure is as a hash table. Going further towards a full hash table would incur severe runtime overheads, and it is difficult to see how to generate a hash function that would allow all matches to be made.

The n list representation is simple, but we would argue that it is quite efficient. We avoid the overhead of adding n arguments to each predicate – one for each list – by bundling all the lists into a single structure.

Our THEORIST implementation also has the option to produce code that uses only a single list for ancestor information. We implemented this in order to see if the intuitions underlying the introduction of two lists were founded.

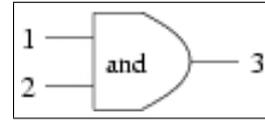


Figure 1: And gate.

```
fact and(0, 0, 0). fact and(0, 1, 0).
fact and(1, 0, 0). fact and(1, 1, 1).

fact opp(1, 0). fact opp(0, 1).

fact bit(B, 0) <- not bit(B, 1).

fact bit(3, C) <- bit(1, A), bit(2, B),
  and(A, B, C), ok(gate).
fact bit(3, C) <- bit(1, A), bit(2, B),
  and(A, B, NC), opp(C, NC), faulty(gate).

hypothesis ok(G).
hypothesis faulty(G).

fact ok(G) <- not faulty(G)

fact bit(1, 0). fact bit(2, 0).

explain bit(3, 0). % -> [ok(gate)].
explain bit(3, 1). % -> [faulty(gate)].
```

Figure 2: Circuit diagnosis knowledge base for *and* gate.

4.3 Literal Ordering

Because many problems are sensitive to literal ordering within clauses, we our THEORIST implementation can produce several different literal orderings: *as-is*, where the literals are left unchanged from the order in which they are presented in the input; *random*, where goal literals are rearranged randomly; *instantiated first*, where instantiated goals with arguments of greater instantiation are placed earlier in the rule; *instantiated last*, where less instantiated goals are placed earlier in the rule; *hypotheses first*, where goals corresponding to hypotheses are placed earlier; and, *hypotheses last*, where hypothesis goals are placed later.

We generate random orderings because performance results for random runs often give information about possible improvements available for some literal orderings.

5 Testing

We used two classes of test problems when evaluating our proposed improvements: circuit diagnosis and Hamilton walk problems.

Figure 1 shows a logical *and* gate. Modelling gates and combinations of gates, we can construct THEORIST knowledge bases that allow us to investigate circuit diagnosis. In this, we set the input bits to known values and ask THEORIST to explain some combination of output bit(s) in terms of *ok* and *faulty* gates. An example knowledge base for the *and* gate is shown in Figure 2.

For one series of tests, we used knowledge bases modelling binary full adders (see Figure 6). This circuit adds three bits (the third bit can be thought of as a carry in) and produces two bits of output, representing the binary value of their sum. This circuit can be extended to an n bit binary adder, by taking the high bit of each adder to feed the carry in of the next higher adder. Our tests involved setting all the input

bits to 0 and asking THEORIST to explain the how the top output bit might be 0 and the next-to-top bit being 1. We performed our tests using n bit adders for values of $n = 4$ to 9. A part of the knowledge base for the adder in Figure 6 is shown in Figure 3.

```

fact bit(N, 1) <- bit(N, 0).

fact ok(G) <- faulty(G).

fact opp(1, 0). fact opp(0, 1).

fact and(0, 0, 0). fact and(1, 0, 0).
fact and(0, 1, 0). fact and(1, 1, 1).

fact or(0, 0, 0). fact or(1, 0, 1).
fact or(0, 1, 1). fact or(1, 1, 1).

fact xor(0, 0, 0). fact xor(1, 0, 1).
fact xor(0, 1, 1). fact xor(1, 1, 0).

hypothesis ok(G). hypothesis faulty(G).

fact bit(4, D) <- bit(1, A), bit(2, B),
xor(A, B, D), ok(x1).
fact bit(4, D) <- bit(1, A), bit(2, B),
xor(A, B, ND), opp(D, ND),
faulty(x1).

...

fact bit(8, H) <- bit(5, E), bit(6, F),
or(F, E, H), ok(o1).
fact bit(8, H) <- bit(5, E), bit(6, F),
or(F, E, NH), opp(H, NH),
faulty(o1).

fact bit(1, 0). fact bit(2, 0). fact bit(3, 0).

explain bit(7, 1), bit(8, 0).

```

Figure 3: Part of a THEORIST knowledge base for 1 bit full adder.

The other tests we ran involved finding Hamilton walks on complete directed n vertex graph. A Hamilton walk is a tour of the graph that visits each vertex only once. A complete n vertex graph is one where every vertex has an arc to every other vertex (see Figure 4 for a complete graph of three vertices). We use complete n vertex graphs because we are guaranteed to find a Hamilton walk. This sort of problem was also used by Schaub (Schaub 1997) in testing XRAY and by Cholewinski, Marek and Truscynski for their DERES system (Cholewinski et al. 1996). While they translated their inputs from Stanford Graph-Base (Knuth 1993) outputs using a system called THEORYBASE (Paawel Cholewiński 1995), we generated ours directly using a small Perl script. We ran our tests using n vertex graphs with $n = 4$ to 9. A THEORIST knowledge base for finding a Hamilton walk on the graph of Figure 4 can be seen in Figure 5.

We note several differences between the problem types: there are several childless predicates in the circuit diagnosis problems (`and/3`, `or/3`, `xor/3`, `opp/2`); circuit diagnosis problems have a more obvious structure than the Hamilton walk problems, which involve more disjunction, which might mean more backtracking is required during execution; circuit diagnosis problems make use of variables, which might exercise the underlying Prolog implementation differently from the Hamilton walk problems, which are wholly ground.

Each input was translated by our THEORIST system to generate unimproved and improved outputs for the three different ancestor list representations (one list, two list, n lists) and all literal orderings (as

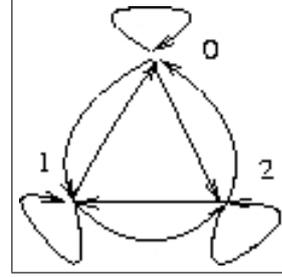


Figure 4: Complete graph with three vertices.

```

fact e(0).

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

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

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

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

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).

explain e(0), e(1), e(2), ee(0).

```

Figure 5: THEORIST knowledge base for 3 vertex Hamilton walk problem.

10 Bit Full Adder		
	Unimproved	Improved
Predicates	19	19
Clauses	800	788
Loop checks	19	7
Model elimination checks	19	6
10 Vertex Hamilton Walk		
	Unimproved	Improved
Predicates	12	10
Clauses	1317	1223
Loop checks	12	8
Model elimination checks	12	5

Table 1: Metrics for output code using two list ancestor representation.

10 Bit Full Adder		
	Unimproved	Improved
Predicates	19	19
Clauses	800	788
Loop checks	19	7
Model elimination checks	8	6
10 Vertex Hamilton Walk		
	Unimproved	Improved
Predicates	12	10
Clauses	1317	1223
Loop checks	12	8
Model elimination checks	10	5

Table 2: Metrics for output code using n list ancestor representation.

is, hypothesis first, hypothesis last, instantiated first, instantiated last, random). These translated outputs were standalone Prolog programs.

Each output program was run 3 times, and the run time was averaged to obtain a final result. Each individual run was limited to 1,000 seconds of CPU time.

6 Results

6.1 Code Improvements

Table 1 contains metrics for the output code when using the traditional two list ancestor representation. Table 2 contains metrics for the output code when using the new n list ancestor representation.

We can see that the improved code has fewer clauses, loop checks and model elimination checks. Our improvements appear to be relevant to the tests. Whether or not this translates into improved run times is revealed in Tables 3 and 4. In these tables, run times are presented in seconds of CPU time. The minimum time taken over the random permutations is displayed in the row “Random (Min.)”. The resolution of the timing mechanism for the Prolog system we used (SWI Prolog) was 0.01 seconds.

It was obvious from our results that our code improvements were effective in almost all cases. There were three results where no improvement is recorded. The run time in these cases was small (0.01, 0.04, 0.07 seconds), so we might be observing the granularity of the timing mechanism rather than a real performance problem. For the full adder problems, the best result was a 70% decrease in run time. The average improvement was 30%. The improvement showed no pattern of correlation to problem size, staying within a range of 25% to 35% improvement.

For the Hamilton walk problems, the best improvement was over 99.9% (702.35 seconds for unoptimised execution vs. 0.03 seconds for optimised execution). The average improvement was nearly 40%. On average, the improvement decreased as problem size increased, closing in on about 20% for the largest

problems.

6.2 n List Ancestor Information Representation

The results for the n list ancestor representation are ambiguous. In the results for the full adder problems, we *almost* always see an improvement in execution speed. However, there were fourteen instances (out of 324 results) where we see a degradation. In one instance, the run time was 43% slower with the n list representation. However, these cases did not correspond to the best case scenario with respect to literal ordering (see below). For the full adder problems we saw a maximum 65% speed up, with an average improvement of about 35%. Looking at timing profiles for the 9 bit runs, we saw that model elimination checks and loop checks dominate. However, the amount of time spent in model elimination checks drops from 2.19 seconds in for two list representation to 1.28 seconds in the n list representation, an improvement of about 40%. The time reported for loop checks degraded slightly, from 1.37 seconds to 1.43 seconds. This is surprising, since there were slightly fewer loop checks, and each check should have been running over a shorter list of ancestor literals on average. Calls to the predicates used to implement the n list representation (`arg/3` and `setarg/3`) accounted for 8.8% or 0.52 seconds of runtime.

Surprisingly, the Hamilton walk problems exhibited degradation as problems grew larger, with 61 instances of degradation (out of 324 results) and 26 instances where no improvement was recorded. Profiling reveals that the Hamilton walk runs were dominated by backtracking. For the 9 vertex runs, model elimination checks took 1.44 seconds for the two list representation and 1.20 seconds for the n list representation. Loop checking took 1.52 seconds and 1.73 seconds respectively – another surprising result for loop check timings. Calls to the predicates used to implement the n list implementation accounted for 8.7% or 1.28 seconds of runtime.

6.3 Literal Ordering

Our literal ordering experiments showed that we could count on improvements in runtime when instantiated literals were placed early in rules. Because the most instantiated literals in our outputs were ground, this corresponds to the constraint programming heuristic of placing a constraint at the earliest point that it can cause failure. The ordering that placed hypotheses first showed similar properties for similar reasons. In these problems, hypotheses are ground at compile time, so placing them at the start of rules can cause early failure.

7 Conclusions and Future Work

In this paper, we have demonstrated that improving code based on observations of the model elimination mechanism produces worthwhile run time gains in some example problems. However, the extension of ancestor goal information from two lists to an n list representation gave mixed results, especially when large amounts of backtracking are encountered. The degradation in runtime in the Hamilton walk problems warrants further investigation.

We have shown that different literal orderings can produce large improvements in run time in the chosen problems. Further work in this area, as well as in dataflow analysis of inputs to determine instantiation patterns would appear to be worthwhile.

Our future work in this area will look at a computational interface to model elimination based implementations. Loveland's work (Loveland 1978) mentions equality and inequality, but incorporating this into a practical reasoning system is a challenge. The ACLP system of Kakas and Michael (Kakas & Michael 1995) might provide a basis for further work in this area.

Stickel, M. E. (1986), A Prolog technology theorem prover: Implementation by an extended Prolog compiler, in J. H. Siekmann, ed., 'Proceedings of the Eighth International Conference on Automated Deduction', Vol. 230, Springer-Verlag, Berlin, pp. 573–587.
URL: citeseer.nj.nec.com/stickel87prolog.html

References

- Astrachan, O. L. & Stickel, M. E. (1992), Caching and lemmaizing in model elimination theorem provers, in D. Kapur, ed., 'Automated Deduction: CADE-11 - Proc. of the 11th International Conference on Automated Deduction', Springer, Berlin, Heidelberg, pp. 224–238.
- Baumgartner, P. & Furbach, U. (1994a), Model elimination without contrapositives, in A. Bundy, ed., 'Proc. 12th Conference on Automated Deduction CADE, Nancy/France', number 918, Springer-Verlag, pp. 87–101.
- Baumgartner, P. & Furbach, U. (1994b), PROTEIN: A PROver with a theory extension INTERface, in 'Conference on Automated Deduction', pp. 769–773.
URL: citeseer.nj.nec.com/baumgartner94protein.html
- Cholewinski, P., Marek, V. W. & Trusczyński, M. (1996), Default reasoning system DeReS, in L. C. Aiello, J. Doyle & S. Shapiro, eds, 'KR'96: Principles of Knowledge Representation and Reasoning', Morgan Kaufmann, San Francisco, California, pp. 518–528.
- de Waal, A. (1994), Analysis and Transformation of Proof Procedures, PhD thesis, Department of Computer Science, University of Bristol.
- de Waal, D. A. & Gallagher, J. P. (1994), The applicability of logic program analysis and transformation to theorem proving, in A. Bundy, ed., 'Automated Deduction-CADE-12', Springer, Berlin, Heidelberg, pp. 207–221.
- Hagen, R. A., Sattar, A. & Goodwin, S. D. (2003), Improving search in a hypothetical reasoning system, in M. J. Oudshoorn, ed., 'Proceedings of the Twenty-sixth Australasian Computer Science Conference (ACSC 2003)', Vol. 16 of *Conferences in Research and Practice in Information Technology*, ACS, Adelaide, Australia, pp. 45–53.
- Kakas, A. C. & Michael, A. (1995), Integrating abductive and constraint logic programming, in L. Sterling, ed., 'Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming', MIT Press, pp. 399–413.
- Knuth, D. E. (1993), *The Stanford GraphBase: A Platform for Combinatorial Computing*, ACM Press, New York, NY.
- Loveland, D. W. (1969), 'A simplified format for the model elimination theorem-proving procedure', *Journal of the ACM (JACM)* **16**(3), 349–363.
- Loveland, D. W. (1978), *Automated Theorem Proving: A Logical Basis*, Vol. 6 of *Fundamental Studies in Computer Science*, North Holland, Amsterdam, The Netherlands.
- Paawel Cholewiński, V. W. M. e. a. (1995), Experimenting with nonmonotonic reasoning, in 'Proceedings of the 12th International Conference on Logic Programming', MIT Press.
- Poole, D. (1988a), Compiling a default reasoning system into Prolog, Research report, Department of Computer Science, University of Waterloo.
- Poole, D. (1988b), 'A logical framework for default reasoning', *Artificial Intelligence* **36**(1), 27–47.
- Poole, D. & Goodwin, S. (1987), A Theorist to Prolog compiler. Available via <http://www.cs.ubc.ca/~poole/theorist.html>.
- Schaub, T. (1997), XRay: A Prolog technology theorem prover for default reasoning (DEMO), in E. Weydert, G. Brewka & C. Witteveen, eds, 'Proceedings of the 3rd Dutch/German Workshop on Nonmonotonic Reasoning Techniques and their Applications', Max-Planck-Institut für Informatik, Saarbrücken, pp. 177–182.

A Figures and Tables

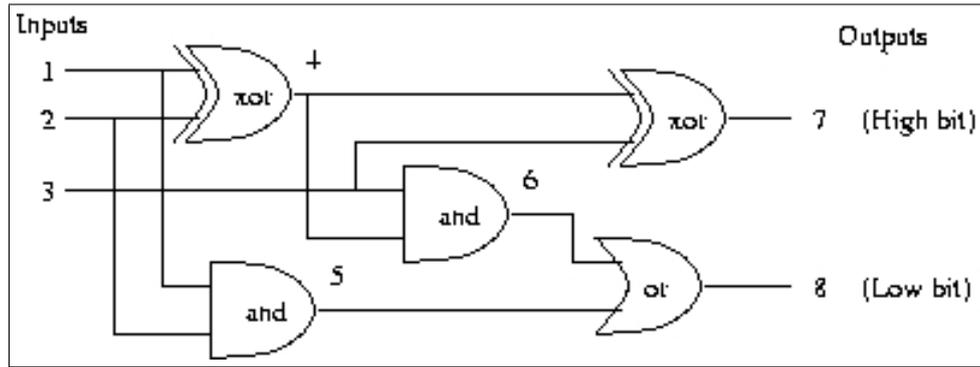


Figure 6: Binary full adder.

Ancestor Representation	Literal Ordering	5 Bit		9 Bit	
		Unimproved	Improved	Unimproved	Improved
One List	As is	6.81	4.72	> 1000	> 1000
	Hyp. First	0.41	0.30	15.22	10.85
	Hyp. Last	479.80	326.56	> 1000	> 1000
	Inst. First	0.09	0.06	0.45	0.29
	Inst. Last	16.99	9.75	> 1000	> 1000
	Random (Min.)	0.64	0.47	42.23	12.57
Two Lists	As is	4.45	2.94	> 1000	> 1000
	Hyp. First	0.29	0.19	8.81	6.32
	Hyp. Last	271.59	191.30	> 1000	> 1000
	Inst. First	0.06	0.04	0.26	0.19
	Inst. Last	9.80	5.44	> 1000	> 1000
	Random (Min.)	0.45	0.28	23.93	7.18
<i>n</i> Lists	As is	2.70	2.46	932.47	870.27
	Hyp. First	0.18	0.17	6.75	5.90
	Hyp. Last	214.08	194.86	> 1000	> 1000
	Inst. First	0.04	0.04	0.17	0.15
	Inst. Last	6.96	5.79	> 1000	> 1000
	Random (Min.)	0.33	0.29	16.73	7.55

Table 3: Run times for full adder problems.

Ancestor Representation	Literal Ordering	5 Vertex		9 Vertex	
		Unimproved	Improved	Unimproved	Improved
One List	As is	0.01	0.01	21.23	12.96
	Hyp. First	0.02	0.01	21.25	14.31
	Hyp. Last	0.12	0.09	> 1000	> 1000
	Inst. First	0.01	0.01	21.41	12.81
	Inst. Last	> 1000	0.03	> 1000	224.44
	Random (Min.)	5.19	0.02	> 1000	> 1000
Two Lists	As is	0.01	< 0.01	16.45	12.64
	Hyp. First	0.01	0.01	17.11	13.26
	Hyp. Last	0.09	0.07	> 1000	935.94
	Inst. First	0.01	0.01	16.87	12.29
	Inst. Last	702.35	0.03	> 1000	200.01
	Random (Min.)	3.30	0.01	> 1000	> 1000
n Lists	As is	0.01	< 0.01	17.04	14.85
	Hyp. First	0.01	0.01	16.46	13.37
	Hyp. Last	0.11	0.09	> 1000	> 1000
	Inst. First	0.01	0.01	16.13	13.38
	Inst. Last	678.01	0.03	> 1000	232.66
	Random (Min.)	2.37	0.02	> 1000	> 1000

Table 4: Run times for Hamilton walk problems.