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

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@windsor.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 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 invalidiated 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.

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 called 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 lemmas 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 l₁ is \{h₂\}; 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.

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 l₂ is \{h₄\}. We can also see that \{h₁,h₂,h₃,h₄\} is a nogood, since \¬h₂ is consistent with \{h₁,h₃,h₄\}. From this, it is possible to deduce that the support set for \¬l₂ is \{h₁,h₂,h₃\} [17], so we can store potential crucial literal informa-
tion for \(-\ell_2\). Note that this inference is only valid if there are no model elimination steps during the derivation of \(\ell_2\).

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 fulladder 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 \(p(t_1, \ldots, t_n)\) with a support set \(S\) is to store the fact \(\text{PosMEC}, \text{NegMEC}\) and \(\text{context}\) for \(p(t_1, \ldots, t_n)\) in the database. Instead, literals are recorded as \(\text{dl}(p(t_1, \ldots, t_n), \text{Support}, \text{PosMEC}, \text{NegMEC, Context})\) or \(\text{pcl}(p(t_1, \ldots, t_n), \text{Support}, \text{PosMEC}, \text{NegMEC, Context})\). 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 hypotheses instances, for example \([h_1, h_3, h_4]\). 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 \(\text{good}_{15544339}\) or \(\text{nogood}_{15544339}\) 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 \(\text{good}_{15544339}(h_1, h_3, h_4)\) or \(\text{nogood}_{15544339}(h_1, h_3, h_4)\). We use this method because the most direct method of storing goods and nogoods — e.g. \(\text{good}(h_1, h_3, h_4)\) — 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. 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

\[\text{Compare this situation with the field of automated theorem proving which has the TPTP database [21].}\]
7 Discussion and Future Work

We have shown that, under certain conditions, potential crucial literals can produce significant improvements in the execution speed of the circuit. It is clear that other problems need to be investigated to find out what types of problems are most effective. However, we believe that the employment of potential crucial literals is a promising way to improve the performance of the circuit.

8 Conclusion

In conclusion, we have shown that potential crucial literals can be effective in improving the performance of the circuit. Further research is needed to determine the best methods for identifying and utilizing potential crucial literals in different types of circuits.
ution 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


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, PasNEO, PasNEG, NegNEO, NegNEG, PCL0, PCL, Theory0, Theory, Uninst0, Uninst, Depth, Pasacs, Negacs) :-
   Lit=NULL, SegLit=NULL, ok(Arg1),
   ( % Loop check (not shown)
     % Manual elimination check (not shown)
   rese Derived literal
   d1 ok(Arg1, TSupport, IMPosNEG, IMNegNEG, IMContext),
   nec confirm(IMPosNEG, Negacs), nec confirm(IMNegNEG, Pasacs),
   ( subset(IMSupport, Theory0) -> Theory0=Theory
     ; subset(TTheory, IMContext)
     X union(IMSupport,Theory0, Theory1), sort(Theory1, Theory)
   ; difference(IMSupport, Theory0, X),
     co_set(X, Theory0), union(X, Theory0, Theory1), sort(Theory1, Theory)
   ),
   union(IMSupport, Support0, Support),
   append(IMPosNEG, PasNEG, PasNEG), append(IMNegNEG, NegNEG, NegNEG),
   Uninst0=Uninst, PCL0=PCL,
   % Reuse hypothesis
   ground(Lit), memberchk(Lit, Theory0),
   union(Lit, Support0, Support),
   PasNEO=PasNEG, PasNEG=PasNEG, PCL0=PCL,
   Theory0=Theory, Uninst0=Uninst,
   % Sub call
   Depth1 is Depth-1,
   explain ok(sub(Arg1, X, SubSupport, X, SubPosNEG, X, SubNegNEG, X, SubPCL,
   Theory0, Uninst0, Uninst, Depth1, LitPosacs, Negacs),
   nec filter(SubPosNEG, Depth, SubPosNEG), nec filter(SubNegNEG, Depth, SubNegNEG),
   derived add(d1, pos ok, i, Arg1, SubSupport, SubPosNEG, SubNegNEG, Theory0),
   union(SubSupport, Support0, Support),
   append(SubPosNEG, SubPosNEG, SubPosNEG),
   append(SubNegNEG, SubNegNEG, SubNegNEG),
   append(SubPCL, PCL0, PCL), Theory0=Theory1)
   % Extend theory
   ground(Lit), memberchk(Lit, Theory0),
   sort(Lit=Theory0, Theory),
   ( is_good(Theory) -> fail
     ; is_good(Theory)
     X union(Lit, Support0, Support),
     PasNEO=PasNEG, NegNEO=NegNEG, PCL0=PCL,
     % Consistency check call
     prove ok(Arg1, X, CSupport, X, COPosNEG, X, CONegNEG, X, COpCL, Depth-1, fail, X, X, Theory0)
     X union(Lit, CSupport, CONeggood),
     sort(CONeggood, CONeggood),
     neggood add(CONeggood),
     ( CONeggood=0, CONeggood=0 -> derived list add(pcl1, OPCL, CONeggood)
     ; true
     )
   ,
   ( subset(CSupport, Theory)
     X neggood add(Theory),
     fail
   ; good add(Theory),
     union(Lit, Support0, Support),
     PasNEO=PasNEG, NegNEO=NegNEG, PCL0=PCL
   )
   ; good add(Theory),
     union(Lit, Support0, Support),
     PasNEO=PasNEG, NegNEO=NegNEG, PCL0=PCL
   )
   Uninst0=Uninst,
   % Nonground hypothesis
   (ground(Lit),
   Support0=Support, PasNEO=PasNEG, NegNEO=NegNEG, PCL0=PCL, Theory0=Theory, Uninst=lit Uninst0)
).

Figure 4: Example explanation phase driver.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Order</th>
<th>No lemmas</th>
<th>DL</th>
<th>NG</th>
<th>DL + NG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamilton 5</td>
<td>Untouched</td>
<td>0.54</td>
<td>0.66</td>
<td>7</td>
<td>66</td>
</tr>
<tr>
<td>Ordered</td>
<td>0.22</td>
<td>0.43</td>
<td>0.32</td>
<td>9</td>
<td>58</td>
</tr>
<tr>
<td>Randomised</td>
<td>Min</td>
<td>0.45</td>
<td>0.47</td>
<td>26</td>
<td>184</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>0.60</td>
<td>0.70</td>
<td>37</td>
<td>184</td>
</tr>
<tr>
<td>Reverse</td>
<td>1.78</td>
<td>2.07</td>
<td>1.76</td>
<td>27</td>
<td>86</td>
</tr>
<tr>
<td>Hamilton 6</td>
<td>Untouched</td>
<td>3.31</td>
<td>6.23</td>
<td>5.4</td>
<td>104</td>
</tr>
<tr>
<td>Ordered</td>
<td>0.04</td>
<td>0.19</td>
<td>0.29</td>
<td>10</td>
<td>84</td>
</tr>
<tr>
<td>Randomised</td>
<td>Min</td>
<td>1.18</td>
<td>2.07</td>
<td>10</td>
<td>681</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>18.62</td>
<td>33.70</td>
<td>304</td>
<td>143.83</td>
</tr>
<tr>
<td>Reverse</td>
<td>19.26</td>
<td>21.94</td>
<td>19.11</td>
<td>35</td>
<td>134</td>
</tr>
<tr>
<td>Hamilton 8</td>
<td>Untouched</td>
<td>&lt;0.01</td>
<td>&lt;0.01</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Ordered</td>
<td>&lt;0.01</td>
<td>&lt;0.01</td>
<td>&lt;0.01</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Randomised</td>
<td>Min</td>
<td>&lt;0.01</td>
<td>&lt;0.01</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Max</td>
<td>&lt;0.01</td>
<td>&lt;0.01</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Reverse</td>
<td>&lt;0.01</td>
<td>&lt;0.01</td>
<td>&lt;0.01</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 5: Performance results
<table>
<thead>
<tr>
<th>Problem</th>
<th>Order</th>
<th>PCL</th>
<th>DL + PCL</th>
<th>NG + PCL</th>
<th>DL + NG + PCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamilton 5</td>
<td>Untouched</td>
<td>0.01 0</td>
<td>0.06 24 0</td>
<td>0.01 5 30 0</td>
<td>0.06 24 25 95 0</td>
</tr>
<tr>
<td></td>
<td>Ordered</td>
<td>0.01 0</td>
<td>0.01 7 0</td>
<td>&lt; 0.01 7 32 0</td>
<td>&lt; 0.01 7 6 13 0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>Min</td>
<td>0.02 20 0</td>
<td>0.01 7 12 0</td>
<td>0.01 11 31 12 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>0.11 44 0</td>
<td>0.30 25 47 0</td>
<td>0.09 154 187 44 0</td>
</tr>
<tr>
<td>Reverse</td>
<td></td>
<td>0.04 31</td>
<td>1.38 20 31 0</td>
<td>0.04 14 42 31 0</td>
<td>1.14 20 14 54 31 0</td>
</tr>
<tr>
<td>Hamilton 6</td>
<td>Untouched</td>
<td>0.09 0</td>
<td>1.34 34 0</td>
<td>0.09 6 33 0</td>
<td>0.09 34 53 253 0</td>
</tr>
<tr>
<td></td>
<td>Ordered</td>
<td>0.09 0</td>
<td>0.04 8 0</td>
<td>0.04 8 50 0</td>
<td>0.04 8 7 21 0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>Min</td>
<td>0.09 26</td>
<td>0.33 32 35 0</td>
<td>0.06 29 71 28 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>2.38 183</td>
<td>95.35 107 183</td>
<td>0.80 309 457 230</td>
</tr>
<tr>
<td>Reverse</td>
<td></td>
<td>0.31 51</td>
<td>69.68 27 51</td>
<td>0.30 20 72 51</td>
<td>56.12 27 20 101 51 0</td>
</tr>
<tr>
<td>Hamilton 7</td>
<td>Untouched</td>
<td>0.66 0</td>
<td>28.84 46 0</td>
<td>0.64 7 86 0</td>
<td>11.71 46 269 735 0</td>
</tr>
<tr>
<td></td>
<td>Ordered</td>
<td>0.46 0</td>
<td>0.31 9 0</td>
<td>0.35 9 87 0</td>
<td>0.31 9 8 31 0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>Min</td>
<td>1.44 60</td>
<td>5.05 52 543</td>
<td>0.53 52 214 57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>49.83 987</td>
<td>166.35 51 528</td>
<td>14.55 993 1919 788</td>
</tr>
<tr>
<td>Reverse</td>
<td></td>
<td>2.72 78</td>
<td>&gt; 1000 - -</td>
<td>2.43 27 126 78</td>
<td>&gt; 1000 - - - -</td>
</tr>
<tr>
<td>Hamilton 8</td>
<td>Untouched</td>
<td>6.16 0</td>
<td>&gt; 1000 - -</td>
<td>5.98 8 131 0</td>
<td>402.6 60 1214 2531 0</td>
</tr>
<tr>
<td></td>
<td>Ordered</td>
<td>4.49 0</td>
<td>2.86 10 0</td>
<td>3.22 10 125 0</td>
<td>2.88 10 9 43 0</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>Min</td>
<td>139.34 1758</td>
<td>177.03 92 2072</td>
<td>11.91 459 723 254</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>&gt; 1000 - -</td>
<td>&gt; 1000 - -</td>
<td>535.0 656 16736 4078</td>
</tr>
<tr>
<td>Reverse</td>
<td></td>
<td>27.82 113</td>
<td>&gt; 1000 - -</td>
<td>26.71 35 191 113</td>
<td>&gt; 1000 - - - -</td>
</tr>
<tr>
<td>Full-adder</td>
<td>Untouched</td>
<td>169.19 60</td>
<td>169.90 11 53</td>
<td>166.84 6 2 60</td>
<td>167.84 11 6 2 53</td>
</tr>
<tr>
<td></td>
<td>Ordered</td>
<td>9.73 0</td>
<td>9.85 11 48</td>
<td>9.63 6 2 48</td>
<td>9.59 11 6 2 48</td>
</tr>
<tr>
<td></td>
<td>Random</td>
<td>Min</td>
<td>2.84 72</td>
<td>4.52 21 72</td>
<td>2.81 6 2 72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>116.68 66</td>
<td>126.17 15 76</td>
<td>115.48 6 2 66</td>
</tr>
<tr>
<td>Reverse</td>
<td></td>
<td>31.52 74</td>
<td>62.96 27 74</td>
<td>31.21 5 2 74</td>
<td>62.21 27 5 2 74 0</td>
</tr>
</tbody>
</table>

Figure 6: Performance results (continued)