The LazyRMS: Avoiding Work in the ATMS

Gerry Kelleher, Linda van der Gaag

RUU-CS-92-20
May 1992

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

The LazyRMS: Avoiding Work in the ATMS

Gerry Kelleher, Linda van der Gaag

Technical Report RUU-CS-92-20
May 1992

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

The LazyRMS: Avoiding Work in the ATMS

Gerry Kelleher
Research Institute for Knowledge-Based Systems
Tongersestraat 6, 6211 LN Maastricht
The Netherlands
e-mail: gerry@riks.nl

Linda van der Gaag
Utrecht University, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht
The Netherlands
e-mail: linda@cs.ruu.nl

Abstract

The basic algorithms involved in reason maintenance in the standard ATMS are known to have a computational complexity that is exponential in the worst case. Yet, also in average-case problem solving, the ATMS often lays claim to a major part of the computational effort spent by a problem solver/ATMS system. In this paper, we argue that within the limits of the worst-case computational complexity, it is possible to improve on the average-case complexity of reason maintenance and query processing by eliminating computation that is of no relevance to the problem solver's performance. To this purpose, we present a set of algorithms designed to control the effort spent by the ATMS on label updating. The basic idea underlying these algorithms is that of lazy evaluation: labels are not automatically maintained on all datums but are computed only when needed (either directly or indirectly) by the problem solver. The algorithms have been implemented in the *LazyRMS* with which we have experimented in the context of model-based diagnosis; our experiments show a substantial saving in the computational effort spent on reason maintenance.

1 Introduction

The assumption-based truth maintenance system (ATMS) of J. de Kleer as presented in [de Kleer, 1986], has been successfully used in a wide range of applications [Smith and Kelleher, 1988], most notably in the areas of automatic diagnosis and planning, see for

example [de Kleer and Williams, 1987], [Morris and Feldman, 1989] and [Mott et al., 1988]. The ATMS, however, suffers from a major drawback in that, unless great care is taken, it can become the major consumer of resources in a reasoning system. In the worst case this problem is insolvable as the algorithms underlying the ATMS have an exponential worst-case computational complexity, [Provan, 1988]. The propensity of the ATMS to lay claim to a major part of the computational effort in average-case problem solving, however, may to some extent be attributed to the ATMS maintaining complete, minimal, sound and consistent labels for all the datums in its justification network at all times; this can lead to expensive computation that is never relevant to anything the problem solver is interested in. There is, for example, the possibility that the problem solver may never ask about (either directly or indirectly) the label of a node; in the standard ATMS such 'unqueried' nodes have their labels updated irrespective of the pointlessness of doing so.

This paper is about improving the average-case performance of the ATMS: it proposes a set of algorithms to control ATMS label updating so that, within the limits of its worst-case computational complexity, the system only does work relevant to the problem solver's queries. The basic intuition underlying the ideas presented here is the same as that underlying the notion of lazy evaluation: label updating is left until the system is actually queried and only labels relevant to the query are computed. The shift in emphasis is away from computation at assertion time to computation at query time. The paper is organised as follows. In Section 2 we briefly review previous work on restricting the computational effort spent by the ATMS within an overall reasoning system and indicate where our works fits in. Section 3 presents the algorithms for a lazy approach to ATMS label updating. The algorithms are illustrated by an example in Section 4. In Section 5 we discuss the relevance of our work in practical applications and present the results we obtained in experiments both on randomly generated justification networks and in the context of model-based diagnosis. Section 6 concludes the paper with some directions for further research.

2 Previous Work

As this paper addresses ATMS label updating, we assume a basic understanding of de Kleer's ATMS, [de Kleer, 1986]. An introduction to RMSs in general can be found in [Kelleher and Smith, 1988]; [Martins, 1990] is an excellent bibliography of work on RMSs.

The propensity of the ATMS to consume a large portion of the computational effort in problem solving has been apparent for some time; however, work on this problem has been relatively restricted when compared to work on extending features of the original ATMS. K.D. Forbus and J. de Kleer describe a problem solver/ATMS architecture designed to help minimise the amount of work done within the overall system by the ATMS, by restricting the information passed on to the ATMS to justifications relevant to the problem solver's performance, [Forbus and de Kleer, 1988]. This is achieved by requiring the problem solver to maintain a *focus environment*; a potential justification whose antecedents are not justified by a focus environment provided by the problem solver is not passed on to the

ATMS. As pointed out by Forbus and de Kleer, although much extra overhead of the ATMS is eliminated, the problem of controlling label updating remains: relevant or not, label updating still occurs globally throughout the ATMS justification network. In response to this difficulty, de Kleer has presented work on a Hybrid RMS (HTMS) which attempts to eliminate overhead involved in label creation [de Kleer, 1991]. This is achieved by providing focus environments which restrict the ATMS to constructing label environments relevant to the focus environment provided. This approach appears to be almost the same as the one proposed by O. Dressler and A. Farquhar in [Dressler and Farquhar, 1991]. Experimental results reported by de Kleer for his HTMS and by Dressler and Farquhar are impressive; the systems show huge savings on the computational effort required from the RMS.

Basically, the problem of controlling label updating consists of two subproblems:

- to update only those environments of a node's label that are relevant to the problem solver's current focus (the *updating* problem), and
- to maintain labels only on nodes which are needed by the problem solver (the *propagation* problem).

The first of these is a problem internally within a node's label. Within a standard ATMS the entire label is created irrespective of its potential for providing useful information to the problem solver. Yet, it may well be that environments are constructed which are of no use in actual problem solving, and in particular this may have been deducible before the environment was created. The provision of focus environments addresses this problem.

There is a degree of overlap between the updating problem and the propagation problem: restricting label updating will, in general, minimise the number of times a label will change and thus the amount of propagation of effects between nodes. More importantly, however, the restricted focus means that within a particular focus only labels of relevance to queries within that focus will be updated. This may seem to suggest that the propagation problem is solved by the provision of focus environments. Unfortunately this is not always true: useless work will still be done if a focus is not minimal with respect to the queries actually generated. This is most obvious for a focus which is established but never queried by the problem solver. If a focus environment provided by the problem solver contains assumptions that are superfluous to the queries actually generated, then these assumptions will be propagated through label updating causing potentially unnecessary computation. The present paper is concerned with the propagation problem.

3 Algorithms

In this section we present a set of algorithms designed to cope with the propagation problem in controlling label updating. We have mentioned before that the general intuition underlying the algorithms is that of lazy evaluation: labels are computed only when necessary. We therefore refer to this set of algorithms as the *LazyRMS*. Before presenting the algorithms in detail we give a general outline.

Consider a reasoning system consisting of a problem solver supported by an RMS. Suppose that new nodes and justifications are passed on to the RMS by the problem solver. The standard ATMS would recompute the labels of all nodes that may be affected by these changes immediately. In contrast, the LazyRMS does not recompute the labels of these nodes but simply marks the nodes as possibly affected. The marking of a node then signifies that its label will have to be recomputed before it can be used in any further label computations itself; an unmarked node is a node whose label is stable given the information we have, that is, the label of an unmarked node needs no further work and may be used as it stands. After marking nodes that may be affected, the LazyRMS is left, potentially, with a partially marked justification network.

Now, suppose the problem solver queries a particular node. The LazyRMS behaves towards the problem solver as if it maintained proper labels on all nodes in the justification network, that is, for each queried node the complete, minimal, sound and consistent label given the justifications in the network is returned. If the queried node is unmarked, then the LazyRMS simply returns the label of the node as it is. However, if the queried node is marked, indicating that its label might not be complete, minimal, sound and consistent, then the LazyRMS has to compute the proper label for the node before it can be returned to the problem solver. There is no need to compute labels on all marked nodes, though. The nodes relevant to the queried node will be restricted to its ancestors in the justification network. It therefore suffices to recursively compute labels only on marked ancestors of the queried node, and to recompute, if necessary, the nogood label, thus allowing the computation of the correct label of the queried node itself.

The two algorithms sketched above constitute the basis of the LazyRMS; they will be discussed in the Sections 3.2 and 3.3. The basic algorithms presented in these sections are extended to handle loops in a justification network in Section 3.4. Section 3.1 provides some additional data structures that are exploited in the algorithms.

3.1 Additional Data Structures

For the algorithms outlined above we introduce three extra components in the standard ATMS's node data structure.

The first addition to a node's data structure is a simple *flag* for marking the node. The second addition is the *antecedent list*, which is a list of all ancestors of the node in the justification network: the antecedent list of a node consists of the node itself, the supporters of its justifications and the members of their respective antecedent lists. Consider for example Figure 1. The antecedent list of node n_9 mentions the nodes $a, b, c, d, n_1, n_2, n_3$ and n_9. Clearly an antecedent list may be quite large, but never larger in length than the total number of nodes in the network. The third addition to a node's data structure is the *mentioned-in list*, which is the set of consequents of the justifications that have the node as a supporter. For example in Figure 1, the mentioned-in list of node n_2 consists of n_3 and n_9.

The antecedent lists and mentioned-in lists will have to be updated when new nodes and justifications are passed on to the LazyRMS. Maintaining the mentioned-in lists for all

nodes in the justification network is computationally trivial: if a new justification is entered into the network, then the consequent of this justification is added to the mentioned-in list of each of the supporters of the justification. Maintaining the antecedent lists for all nodes in the justification network is more complicated. If a new justification is entered into the network, then the supporters of this justification are added to the antecedent list of the consequent of the justification. The change in the antecedent list of this node may affect the antecedent lists of the nodes in the mentioned-in list of this node. The antecedent list of each of these nodes is updated by taking the union of the antecedent lists of the supporters of its justifications and subsequently adding the node itself. This procedure is repeated recursively until all antecedent lists have been updated. Note that in maintaining the antecedent lists for all nodes in the network we can only ever visit any node once.

3.2 Marking Nodes

One of the basic algorithms of the LazyRMS is concerned with marking nodes as having potentially changed labels after a new justification has been added to the justification network. When a new justification is given for a particular node or when a node is otherwise thought affected, the algorithm marks this node and recursively repeats the procedure for the nodes in its mentioned-in list until no new nodes are marked. This algorithm is outlined in the following pseudo-code; the list node-list is initialised with the name of the node which had a new justification.

```
procedure mark-nodes (node-list)

while node-list is not empty do
   if the first element of node-list is marked
   then delete the first element from node-list
   else
      mark the node mentioned as the first element in node-list;
      delete the first element from node-list and insert the mentioned-in list of that node;
      remove duplicates from node-list
   endif
enddo
```

Note that a node is only marked as *potentially* having an altered label; there is no guarantee that its label will actually change. Also note that assumptions and premises are never marked. In addition, we observe that it is possible to check for a node that has no marked nodes in its antecedent list, if its label does in fact change before marking it. This way the number of marked nodes within the justification network and thus the amount of label computation at query time, is reduced. We feel that there is a balance between doing easy label updates to avoid marking and marking to avoid hard label updates; as this balance appears to be application-dependent we have not implemented easy label updating in the algorithms proposed in this paper.

Sofar, we have not taken nogoods into consideration. It will be evident that the LazyRMS will have to take special care of nogoods in order to be able to ensure consistency throughout the justification network. We present the basic algorithms for dealing with nogoods in this and the following section. As soon as a new justification is entered for the nogood or the nogood is otherwise considered affected, its label is recomputed. How the nogood label is recomputed by the LazyRMS will be discussed in Section 3.3; here we first focus on the algorithm for marking after an update of the nogood label. When the nogood label is updated, the algorithm starts with the assumptions mentioned in the nogood label (or better still, with the assumptions mentioned in the updated environments of the nogood label) and then recursively traverses the justification network performing a consistency check for each unmarked node visited. If an unmarked node is encountered having an inconsistent environment in its label, then the mark-node procedure is invoked to recursively mark this node and its consequents. This algorithm is outline below; the list node-list is initialised with the assumptions mentioned in the nogood label.

```
procedure mark-after-nogood-update (node-list)

while node-list is not empty do
   if the first element of node-list is marked
   then delete the first element from node-list
   else
     if some environment in the label of the first element of node-list is inconsistent
     then call mark-nodes for the first element of node-list
     else
       delete the first element from node-list and insert the mentioned-in list of that node;
       remove duplicates from node-list
     endif
   endif
enddo
```

Note that the mark-after-nogood-update procedure is applied only after the nogood label has been updated; the procedure therefore uses a complete and minimal nogood label. This ensures that after the procedure is executed, the labels of the nodes that are still unmarked are consistent.

3.3 Querying a Node

After having marked nodes as having a potentially affected label, the LazyRMS is left with an only partially unmarked justification network. Now, if the problem solver queries the LazyRMS for a node's label, the LazyRMS may have to compute labels for several nodes in the network in order to behave towards the problem solver as if it maintained proper labels on all nodes in the network. Before presenting the algorithm for determining the label of a queried node in detail, we introduce two additional notions.

A *boundary node* is a marked node having an antecedent list consisting of unmarked nodes only; informally speaking, it is on the 'boundary' of the (sub)network of marked nodes. Consider Figure 1 once more; n_5 is an example of a boundary node. A *closely connected component* is a set of two or more nodes mentioning each other in their respective antecedent lists. In Figure 1, for example, the nodes n_1, n_2 and n_3 form a closely connected component. For the sake of brevity, we will in the sequel often refer to a closely connected component as a *loop* in informal discussions.

As the presence of loops in a justification network introduces some complications for the algorithm for determining a queried node's label, loops will be dealt with separately in Section 3.4; here we focus on justification networks not containing any loops. Basically, the algorithm for determining a queried node's label operates by recursively computing labels for marked nodes that are ancestors of the queried node in the justification network. We distinguish between three cases:

- if the queried node is unmarked this means that the node's label has not been affected by changes in the network after its last update and therefore is complete, sound and minimal; furthermore, from the way nogoods are handled it follows that the present label is consistent as well. Therefore, the node's label can simply be returned as it stands.
- if the queried node is a boundary node then the standard ATMS label creation algorithm is used to compute the node's complete, minimal, sound and consistent label.
- otherwise the labels of the supporters of the justifications for the queried node are determined recursively to render it a boundary node.

The algorithm is outlined in pseudo-code below; loops are dealt with in the stabilise-component procedure which will be discussed in the next section.

```
procedure query-node (node)

if node is unmarked
then return node's label
else
    let node-list be the antecedent list of node and remove all unmarked nodes from node-list;
    while node-list is not empty do
        if there are boundary nodes in node-list
        then compute the labels on these nodes and unmark them
        else stabilise-component (node-list)
        endif;
        remove all unmarked nodes from node-list
    enddo;
    query-node (node)
endif
```

In the previous section, we have already mentioned that the LazyRMS has to take special care of the nogood label to guarantee correct behaviour towards the problem solver. We recall that the LazyRMS has to recompute the nogood label as soon as the nogood gets marked to be able to ensure consistency of all unmarked nodes throughout the justification network. The updating of the nogood label is done by the standard label updating algorithm after applying the query-node procedure to all marked supporters of the justifications for the nogood to have their labels updated. After the nogood label is recomputed, the mark-after-nogood-update procedure is called to mark all nodes that may be affected by the new nogood label.

We have implemented a straightforward optimisation of the algorithms for dealing with nogoods by observing that it is possible to postpone computing the nogood label as well. To this purpose we observe that when the nogood label is considered affected, only labels on nodes whose set of antecedent assumptions intersect with the nogood's antecedent assumptions can have become inconsistent. This observation is exploited as follows. When the nogood label is considered affected, the nogood is marked as are those nodes whose set of antecedent assumptions comprise assumptions mentioned in the antecedent list of the nogood. Now, if a queried node's antecedent list specifies assumptions that are antecedents of the nogood, then the label of the nogood is computed before the label for the queried node is considered.

3.4 Handling Loops

In the previous section, we have mentioned that the presence of loops in a justification network complicates the algorithm for querying nodes. We note that a loop can easily be detected within the query-node procedure: if the list of nodes for which a label has to be computed does not specify any boundary nodes, then a loop, or to be more precise, a closely connected component, is encountered. Now observe that this list of nodes may specify more than one closely connected component. As the labels on the members of one such component may be dependent upon the labels of another component's members, it is important to identify a *minimal* closely connected component, in the sense used by J. Goodwin in the context of JTMSs, [Goodwin, 1987]. Exploiting the observation that we are guaranteed by the query-node procedure that every node outside the identified set of closely connected components that is an ancestor of the nodes in these components already has had its label updated, the members of a minimal closely connected component can easily be computed from the node-list that is being examined. We do not elaborate on this issue any further.

The basic idea of handling a minimal closely connected component is to stabilise the labels on the members of the component first before the nodes dependent on the component are considered. Stabilising the labels on the members of a minimal closely connected component involves checking the nodes in the component more than once but never more than the number of members of the component. The algorithm for handling a closely connected component starts with creating a list of nodes to be checked, initially being the set of all nodes inside the component. A particular node is removed from the list and

its label is computed from the labels of its supporters as they are; note that the resulting label may not be complete, sound, minimal and consistent since it may have been computed using labels of marked nodes. If the computed label is the same as the old label nothing happens and a successor element in the component is considered. If there is some change, however, the label is updated accordingly and those members of the component that are in its mentioned-in list are added to the list of nodes to be checked. We repeat this procedure until there are no more nodes left to check.

```
procedure stabilise-component (node-list)

identify minimal component from node-list and initialise to-be-updated with component;
while to-be-updated is not empty do
    update the label of the first element of to-be-updated and remove that element;
    if its label has changed
    then
      add the component members from its mentioned-in list to to-be-updated;
      remove duplicates from to-be-updated
    endif
enddo;
unmark all nodes in component
end
```

Note that the standard ATMS would make no distinction between nodes in a closely connected component and nodes which are dependent on the component. However, there is no reason to update labels on nodes that are dependent on a component until the component itself has been dealt with. Consider Figure 1 once more; node n_9 would be updated (potentially) twice by the standard ATMS but only once by our algorithm. Obviously, our algorithm for stabilising closely connected components before computing labels on nodes that are dependent on such a component could be incorporated into the standard ATMS algorithm and thus save unnecessary computational effort. There is a close relationship between the cycle-cutset algorithm of R. Dechter and J. Pearl for constraint satisfaction problems [Dechter and Pearl, 1987], and the algorithm we propose here. The computational saving occurs because computing labels on nodes in a tree-structured arrangement is much cheaper than computing labels within a closely connected component.

4 An Example

The algorithms described in the previous section are best seen in the context of an example. The following example is intended to make the exposition of the ideas clearer, rather than to illustrate a realistic use of the LazyRMS.

Suppose that the LazyRMS has been given the following history of justifications (the

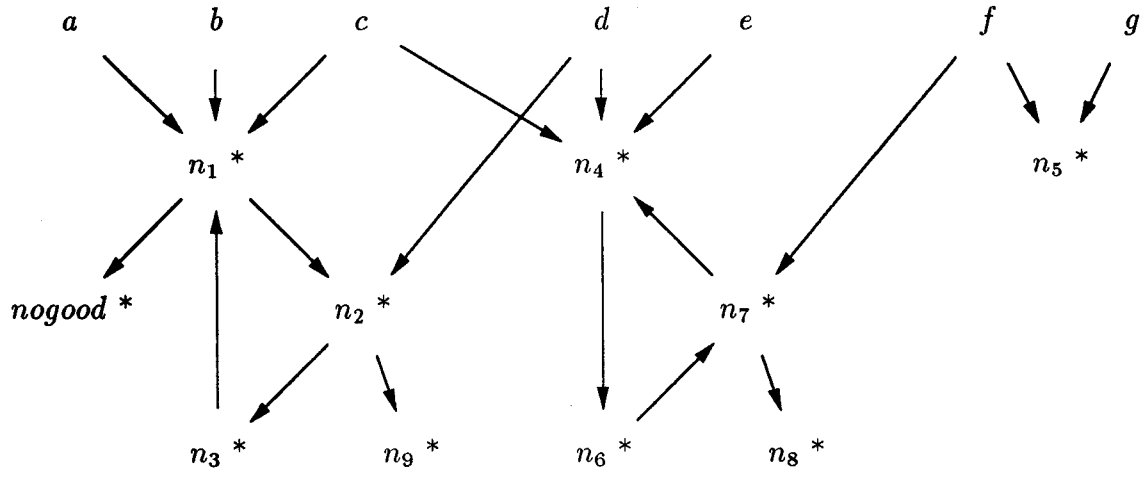

Figure 1: An Example RMS Network.

letters a, b, c, d, e, f, g denote ATMS assumptions):

$(a\ b\ c) \rightarrow n_1$	$(c\ d\ e) \rightarrow n_4$	$(f) \rightarrow n_7$
$(n_1\ d) \rightarrow n_2$	$(n_4) \rightarrow n_6$	$(f\ g) \rightarrow n_5$
$(n_2) \rightarrow n_3$	$(n_6) \rightarrow n_7$	$(n_1) \rightarrow nogood$
$(n_3) \rightarrow n_1$	$(n_7) \rightarrow n_4$	
$(n_2) \rightarrow n_9$	$(n_7) \rightarrow n_8$	

This results in the network shown in Figure 1; an * indicates a node being marked. Initially, all nodes other than assumptions are marked. Within the LazyRMS the only work done so far is the construction of justification lists, antecedent lists and mentioned-in lists. Now suppose that the problem solver asks for the label of node n_5. The LazyRMS applies the query-node procedure to determine the label of this node. Since n_5 is a boundary node and its antecedent assumptions do not intersect with the nogood's antecedent assumptions, the LazyRMS computes n_5's label directly, yielding

$$n_5 : ((f\ g))$$

This is one label computation. Note that in the standard ATMS, all nodes in the network would have had their labels computed. This is a straightforward example of the reduction of work which is possible within our scheme.

A more interesting example follows from a query of node n_9. Since now the antecedent assumptions of this node intersect with those of the nogood, the LazyRMS first updates the nogood label. The only marked supporter for the nogood is n_1. Therefore, the query-node procedure is called for n_1 in order to update its label. The set of antecedents of n_1 consists of a, b, c, d, n_1, n_2 and n_3; removing all unmarked nodes, we have n_1, n_2, n_3. As none of these are boundary nodes we know we have encountered a closely connected component

which is identified to consist of these three nodes. This component is stabilised using the stabilise-component procedure. The resulting labels are:

$n_1 : ((a\ \ b\ \ c))$
$n_2 : ((a\ \ b\ \ c\ \ d))$
$n_3 : ((a\ \ b\ \ c\ \ d))$

All nodes in the closely connected component are now unmarked. The nogood label is computed from the label of node n_1, its only supporter:

$nogood : ((a\ \ b\ \ c))$

The nogood is then unmarked. Walking forward from the assumptions in the nogood label, i.e. a, b and c, node n_1 is found to have an inconsistent environment and is therefore marked; n_2 and n_3 are also marked as consequents of n_1. Subsequently, the problem of finding a label for n_9 is re-addressed. Once more, the stabilise-component procedure is called for the nodes n_1, n_2 and n_3. The new computed node labels are empty labels for all three nodes:

$n_1 : (( ))$
$n_2 : (( ))$
$n_3 : (( ))$

Computing n_9's label from the empty node label of n_1 gives the result

$n_9 : (( ))$

which is returned to the problem solver as n_9's label. Note that in doing this work the labels of n_4, n_6, n_7 or n_8 are never computed: a saving over the standard ATMS.

5 Experiments with the LazyRMS

Clearly the most interesting question to address is in what circumstances the LazyRMS as described in the previous saves computational effort. As the basic algorithms for label updating have an exponential worst-case computational complexity, it is not hard to generate examples which render our approach pointless. This would be a set of queries demanding label updating on every node in the network. Our experience is that such a worst-case situation is rare although not impossible. It is worth noting, however, that our system would not require essentially more computational effort than a standard ATMS on actually computing the labels needed; the only additional work required would be that involved in the computationally trivial 'housekeeping' algorithms. On the other hand, it is equally easy to generate examples of problem solving within which the LazyRMS massively outperforms the standard ATMS. It will be evident that the lazy evaluation scheme will save work when the number of queries by the problem solver is small relative to the number of justifications passed on to the RMS. Also, the more disconnected the justification network

is the better the results rendered by the LazyRMS will be, as the chance of a query needing to compute large numbers of RMS labels is reduced. Furthermore, the LazyRMS helps in circumstances where the justification network must be built whilst the problem solver is operating; if the justification network can be precomputed for problem solver's use as a static data base (as for example in [Kelleher and Bailey, 1989]) our proposals are of no help in improving the performance of a system at runtime.

The above observations are very general. To gain more insight into the performance characteristics of the LazyRMS in situations which were not predesigned, we have conducted two experiments. In Section 5.1 we discuss the results of our first experiment in which we compared the standard ATMS with our LazyRMS on randomly generated justification networks. As in most problem solver/RMS systems the interactions between the two subsystems are heavily optimised, the use of random justification networks may be misleading. In our second experiment, therefore, we have analysed the performance of the LazyRMS in a practical application, namely within the context of model-based diagnosis. Section 5.2 describes the results we obtained.

5.1 The LazyRMS and Random Justification Networks

The first experiment reported here has been designed to compare performance characteristics of the standard ATMS and our LazyRMS on arbitrary justification networks. To conduct this experiment, we have created several sets of randomly generated justification networks. Each of the generated networks consisted of one thousand nodes. Of these, two hundred nodes were assumptions; all other nodes were either assumed nodes or nodes justified by a combination of other nodes. The networks did not comprise any premises. The justifications for the nodes in the network were created using the random number generation facilities of a commercial Common Lisp system, that is, the number of justifications and their content and length were chosen using this facility. The node to be justified by a generated justification was also randomly chosen. The position of the nogood was chosen at random as well; for the nogood, ten justifications were generated.

The justification networks were created to have differing degrees of connectivity. This was achieved by varying the balance between the number of nodes supported by assumptions only, the number of nodes supported by assumptions and other nodes, and the number of nodes supported by other nodes only. As a crude measure of average connectivity we have used the average number of antecedents of a node plus the average number of nodes affected by a change in a node's label. With respect to connectivity, we observe that in randomly generated highly connected justification networks the labels of the nodes rapidly become unmanageable. This is in part due to the nogood environments being unrelated to node labels, which results in elimination of environments by nogood subsumption being rare. A consequence is that the highly connected networks have a lower degree of connectivity than we would like.

In the experiment, we have further varied the querying frequency. The querying frequency is given by the number of new justifications entered into the justification network before the next query is posed to the RMS. For each query, the queried node was chosen

at random. In addition, after all justifications had been supplied to the RMS the nogood was queried.

We compared the standard ATMS and the LazyRMS with respect to three different performance characteristics. The most interesting measure of performance used is the number of label updates performed, that is, the number of times the RMS generated a label from other labels. In addition, the percentage of nodes for which the label was fully computed was considered (assumptions excluded); note that for the standard ATMS this percentage always equals 100%. To conclude, the time spent on reason maintenance and query processing by the respective RMSs in CPU seconds was compared; the timing information given below is based on implementations of both the standard ATMS and the LazyRMS written in Common Lisp, run on a SPARC II workstation. Table 1 presents the performance results for a randomly generated justification network having the following characteristics:

- the connectivity is low ($c = 16$),
- the total number of justifications supplied to the RMS equals 901, and
- the number of nogood antecedents equals 117.

The table shows the performance characteristics of both RMSs under two querying frequencies; the low frequency corresponds with a query after every tenth justification has been entered into the network, the high frequency corresponds with a query after every hundred justifications. Note that the difference in the time spent by the standard ATMS results from the different numbers of queries.

querying frequency	*number of calls for label creation*		*percentage of labels computed*		*time CPU secs*	
	ATMS	LazyRMS	ATMS	LazyRMS	ATMS	LazyRMS
low	30562	75	100	4	462	16
high	30562	165	100	8	474	22

Table 1: Performance Results for a Network with Low Connectivity.

Table 2 presents the performance results of both the standard ATMS and the LazyRMS for a random justification network having the following characteristics:

- the connectivity is high ($c = 84$),
- the total number of justifications supplied to the RMS equals 701, and
- the number of nogood antecedents equals 329.

The querying frequencies mentioned in the table correspond to the ones mentioned in Table 1. Note that the highly connected justification network has fewer justifications than the network with the low connectivity; the labels to be computed, however, are more complex thus accounting for the differences in time spent by the ATMS for the two networks.

querying frequency	*number of calls for label creation*		*percentage of labels computed*		*time CPU secs*	
	ATMS	LazyRMS	ATMS	LazyRMS	ATMS	LazyRMS
low	25106	638	100	16	44161	379
high	25106	1129	100	24	44172	463

Table 2: Performance Results for a Network with High Connectivity.

To conclude, an additional performance test with a special type of querying strategy has been performed in which after every addition of a new node to the highly connected network every node in the network was queried. This is a rather unrealistic use of the LazyRMS but as it is a type of worst-case situation it has been included. The results of this test are shown in Table 3.

querying strategy	*number of calls for label creation*		*percentage of labels computed*		*time CPU secs*	
	ATMS	LazyRMS	ATMS	LazyRMS	ATMS	LazyRMS
all	25106	2615	100	100	44235	7081

Table 3: Performance Results for a Network with High Connectivity, All Nodes Queried.

As can be seen from the tables shown above, the LazyRMS outperforms the standard ATMS in any case. As would be expected, the most dramatic results are obtained in the situation in which the justification network is relatively disconnected and few nodes are queried: large parts of the network are never considered in this case and the savings are correspondingly large. As the connectivity of the justification network and the number of queries posed to the RMS increase, the improvement in performance of the LazyRMS over the standard ATMS shows a corresponding decrease but is still significant.

With respect to the results shown above, we would wish to emphasise that, although we have made efforts to ensure that the comparisons were fair, the figures quoted are not for highly optimised implementations of neither of the RMSs. An important point, however, is that both RMSs use exactly the same label creation procedure. All differences therefore derive from their means of controlling and choosing the way in which label updates are propagated. As with any experiment such as this, there nevertheless remains the possibility

that different implementations may well produce different results. In our opinion, though, this would not radically alter the basic outcome of the experiments.

5.2 The LazyRMS and Model-Based Diagnosis

The aim of our second experiment was to investigate the performance characteristics of the LazyRMS in a real-life application. This experiment was conducted in the context of a problem solver/RMS system for model-based diagnosis.

In model-based diagnosis, a model of the correct behaviour of a system is built and exploited for performing diagnosis. For a specific system, a diagnosis is arrived at by analysing the discrepancies between the observed behaviour of the system and the behaviour it is expected to demonstrate according to its model. The most famous framework for model-based diagnosis is the General Diagnostic Engine (GDE), developed by J. de Kleer; GDE provides a problem solver supported by an ATMS. In our experiments with the LazyRMS we have used a GDE-like framework for model-based diagnosis developed by T. van Rij, [van Rij, 1992]. In this framework, a similar approach is followed to that of GDE as described in [de Kleer & Williams, 1987]. An important difference, however, between this framework and GDE is that it makes use of so-called hierarchies, describing the decomposition of the system to be diagnosed into components. An example of a system is a full bit adder which consists of two exclusive-or-gates, two and-gates and an or-gate. At each hierarchical level of the system, model-based diagnosis is applied with the following characteristics. The diagnostic engine is driven by a constraint satisfaction system that uses test sets to arrive at a diagnosis. At a given hierarchical level, only constraints concerning components that are present in the test set associated with that specific level are applied. The first test set for example specifies all components at the highest hierarchical level. The inferences made by the diagnostic engine are passed on to the supporting RMS until an inconsistency is detected. As soon as the diagnostic engine detects an inconsistency, a nogood representing the inconsistency is created and passed on to the RMS. The environments in the label of the nogood are then taken as the new test sets. Note that the only query posed to the RMS is a query for the nogood label; no other node is ever queried directly by the diagnostic engine. The fundamental idea of this modified diagnostic engine is that it is 'selective' about its queries to the supporting RMS by concentrating entirely on the nogood: the RMS admits applying a focusing strategy (based on queries) and the diagnostic engine takes advantage of that focusing. Since in most diagnostic situations the majority of components of a system and their associated constraints are never connected with a failure of the system, the overwhelming majority of nodes in the justification network constructed during a GDE-based diagnosis are irrelevant to the operation of the system and are avoided by the framework described above.

In our experiment we have replaced the standard ATMS by the LazyRMS in the problem solver/RMS architecture outlined above. The results obtained show huge savings in the computational effort spent by the RMS. To give an idea of the savings involved in this strategy we typically found that in the diagnosis of a 100bit adder less than ten nodes from a total of several hundreds of nodes are implicated in deriving the nogood label.

That is, instead of maintaining the labels of hundreds of nodes the LazyRMS computes labels for less than ten. Although the LazyRMS has some computational overhead of two or three percent in the 'housekeeping' algorithms, this leads to startling savings in terms of computational effort. It is the focusing on nogoods only by the diagnostic engine that accounts for this result.

6 Further Research

In the foregoing, we have presented a set of algorithms designed to control the effort spent by the ATMS on label updating, thus improving on the complexity of average-case problem solving. These algorithms have been implemented in the LazyRMS. The initial experiments with the LazyRMS look promising. However, it is evident that much work needs to be done on perfecting the algorithms we have described; at present, the procedures for the LazyRMS are clear and conceptually simple but sub-optimal.

As for further research, the most interesting area appears to be the integration of focusing strategies and lazy evaluation. How the two strategies may be combined is not entirely straightforward. The problem is that the marking performed in the LazyRMS would be inappropriate under a focusing strategy: it could either be incorrect or lead to massive redundancy in computation. Consider unmarking a node as soon as the part of its label that is relevant to a given focus had been computed. This clearly leads to incorrectness when the focus changes: the label may need recomputing but this is not indicated by the node being marked. Now consider leaving the node marked in the same situation. Then it is possible that the LazyRMS is required to do the same work over and over again, recomputing exactly the same information that had been derived previously. One potential means of easing the problem is to use justifications as the basic RMS data structure; there are good efficiency reasons for this approach over and above integrating focusing strategies and lazy evaluation as it avoids the need to recompute label components for unaffected parts of a label when a justification for a node changes, [Farquhar, 1992]. With respect to the LazyRMS, marking would be of individual justification sets with labels only being computed if at least part of a set is relevant to the focus environment. The combination of the two strategies should lead to the elimination within an RMS of almost all irrelevant work, at the expense of some minor 'housekeeping' overhead.

Acknowledgement

We are indebted to Adam Farquhar for many helpful comments on an earlier draft of this paper. Tanja van Rij provided much of the information about using the LazyRMS for model-based diagnosis.

References

[de Kleer, 1986] J. de Kleer (1986). An assumption-based TMS, *Artificial Intelligence*, vol. 28, pp. 127-162.

[de Kleer, 1991] J. de Kleer (1991). *A Hybrid Truth Maintenance System*, Technical Report, Xerox Palo Alto Research Center, Palo Alto, USA.

[de Kleer and Williams, 1987] J. de Kleer and B.C. Williams (1987). Diagnosing multiple faults, *Artificial Intelligence*, vol. 32, pp. 97 - 130.

[Dechter and Pearl, 1987] R. Dechter and J. Pearl (1987). The cycle-cutset method for improving search performance in AI applications, *Proceedings of the Third IEEE Conference on AI Applications.*

[Dressler and Farquhar, 1991] O. Dressler and A. Farquhar (1991). Putting the problem solver back in the driver's seat: contextual control of the ATMS, in: J.P. Martins and M. Reinfrank (eds.) *Proceedings of the ECAI-90 Workshop on Truth Maintenance Systems*, Springer-Verlag, Heidelberg, Germany, pp. 1 - 16.

[Farquhar, 1992] A. Farquhar (1992). Personal communication.

[Forbus and de Kleer, 1988] K.D. Forbus and J. de Kleer (1988). Focusing the ATMS, *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 193 - 198.

[Goodwin, 1987] J. Goodwin (1987). *A Theory and System for Non-Monotonic Reasoning*, Linkoping Studies in Science and Technology, Dissertations no. 165, Linkoping University, Linkoping, Sweden.

[Kelleher and Bailey, 1989] G. Kelleher and J. Bailey (1989). An explanation driven architecture for a knowledge based system in post-operative care, *Proceedings of the Second European Conference on AI in Medicine*, Lecture Notes in Medical Informatics, Springer-Verlag, Heidelberg, Germany.

[Kelleher and Smith, 1988] G. Kelleher and B.M. Smith (1988). A brief introduction to reason maintenance systems, in: B.M. Smith and G. Kelleher (eds.) *Reason Maintenance Systems and their Applications*, Ellis Horwood, New Jersey, pp. 4 - 20.

[Martins, 1990] J.P. Martins (1990). The truth, the whole truth and nothing but the truth, *AI Magazine*, special issue, pp. 7 - 25.

[Morris and Feldman, 1989] P. Morris and R. Feldman (1989). *Planning and Truth Maintenance*, Technical Report, Intellicorp.

[Mott et. al., 1988] D.H. Mott, J.L. Cunningham, G. Kelleher and J.A. Gadsden (1988). Constraint-based reasoning for the generation of naval flying programmes, *International Journal of Expert Systems*, vol. 5, pp. 226 - 245.

[Provan, 1988] G.M. Provan (1988). A complexity analysis of assumption-based truth maintenance systems, in: B.M. Smith and G. Kelleher (eds.) *Reason Maintenance Systems and their Applications*, Ellis Horwood, Chichester, New Jersey, pp. 98 - 113.

[Smith and Kelleher, 1988] B.M. Smith and G. Kelleher, eds. (1988). *Reason Maintenance Systems and their Applications*, Ellis Horwood, New Jersey.

[van Rij, 1992] T. van Rij (1992). *Efficient Model Based Diagnosis*, RIKS Technical Report 9211, Research Institute for Knowledge-Based Systems, Maastricht, The Netherlands.