Appendix to the Paper: DynamicHS: Streamlining Reiter's Hitting-Set Tree for Sequential Diagnosis

Patrick Rodler PATRICK.RODLER@AAU.AT
Universitätsstr. 65-67, 9020 Klagenfurt, Austria

Abstract

In Appendix A, we give a proof of the algorithm's correctness. Appendix B details advanced techniques used by DynamicHS regarding tree management, reasoning operations and node storage, and Appendix C presents additional experimental analyses that provide a more detailed insight into the reasons for the favorable performance of DynamicHS.

All references to sections, algorithms, theorems, examples, properties, figures, tables, etc. by arabic numerals (e.g., Section 3.2.1, or Example 5) refer to the respective sections, algorithms, theorems, examples, properties, figures, tables, etc. in the paper associated with this appendix. Examples, figures, tables, etc. that can be found only in this present appendix document are referred to by roman numerals (e.g., Example II, or Figure III) in order to distinguish them from their counterparts in the paper. For convenience, we include at the end of this document a copy of both Algorithm 1 (Sequential Diagnosis) and Algorithm 3 (DynamicHS), which are presented in the paper. The line numbers mentioned throughout this document all refer to Algorithm 3 unless otherwise mentioned.

Appendix A. Algorithm Correctness: Proof of Theorem 1

We first prove the completeness and the best-first property of DynamicHS, and then we show its soundness.[1]

A.1 Proof of Completeness and of the Best-First Property

We have to show that DynamicHS, given the parameter $ld := k$, outputs the k best minimal diagnoses (according to pr). First, we make two general observations, and then we prove the completeness by contradiction.

Observation 1: Only such nodes can be deleted by PRUNE which are provably redundant (i.e., irrelevant), and, whenever existent, a suitable replacement node is extracted from $\mathbf{Q}_{dup}$ (which stores all possible replacement nodes) to replace the deleted node (cf. Sec. 3.2.5).

Observation 2: During the execution of DynamicHS given the DPI dpi_k, only diagnoses for dpi_k can be added to $\mathbf{D}_{calc}$.

Proof by Contradiction: Assume that DynamicHS returns a set $\mathbf{D}_{calc}$ with $|\mathbf{D}_{calc}| = k$ and one of the k best (according to pr) minimal diagnoses is not in $\mathbf{D}_{calc}$ (i.e., has not been computed by DynamicHS). We denote this non-found diagnosis by $\mathcal{D}'$. Since $\mathcal{D}'$ is one of the k best minimal diagnoses, we have that some of the returned k diagnoses in $\mathbf{D}_{calc}$ must have a lower probability as per pr than $\mathcal{D}'$; let us call this diagnosis $\mathcal{D}''$. Below, we will show that, for any minimal diagnosis $\mathcal{D}$ for the relevant DPI $\langle \mathcal{K}, \mathcal{B}, P \cup P', N \cup N' \rangle$ considered by DynamicHS, the following invariant

(*INV*) holds during the execution of the main while-loop (between lines 4–21) of DynamicHS: $\mathcal{D} \in \mathbf{D}_{calc}$ *or there is some node* $\mathsf{n} \subseteq \mathcal{D}$ *in* $\mathbf{Q}$. Due to the properties of pr (cf. Sec. 3.2.1) and because $\mathcal{D}'$ has a greater probability than $\mathcal{D}''$, each subset of $\mathcal{D}'$ has a greater probability than $\mathcal{D}''$. Since $\mathcal{D}' \notin \mathbf{D}_{calc}$ by assumption, we infer that $\mathsf{n} \in \mathbf{Q}$ for some node $\mathsf{n} \subseteq \mathcal{D}'$ when DynamicHS terminates. This is a contradiction to $\mathcal{D}'' \in \mathbf{D}_{calc}$ due to the sorting of $\mathbf{Q}$ in descending order of probability and the fact that only elements of $\mathbf{Q}$ can be added to $\mathbf{D}_{calc}$ in DynamicHS.

Proof of the Invariant: We now demonstrate that the invariant INV holds, by induction over the number n of times DynamicHS has already been called in Alg. 1.

(Induction Base): Assume $n = 1$, i.e., DynamicHS is called for the first time in Alg. 1. This has three implications: *(1)* At the time the while-loop is entered, the empty node [] (cf. line 3 in Alg. 1), which is a subset of any minimal diagnosis, is in $\mathbf{Q}$. Hence, INV holds from the outset. *(2)* $\mathbf{D}_{\checkmark} = \emptyset$ (cf. line 2 in Alg. 1 and note that UPDATETREE does not modify $\mathbf{D}_{\checkmark}$). As a consequence, for each node from $\mathbf{Q}$ that is processed throughout the execution of the while-loop of DynamicHS, the DLABEL function is called (because line 7 cannot be reached). *(3)* The PRUNE function (line 33) cannot be called during the execution of the while-loop (as there cannot be any non-minimal conflicts due to the soundness of FINDMINCONFLICT).

Now, assume an arbitrary minimal diagnosis $\mathcal{D}$ for the DPI dpi_0 relevant in the first call of DynamicHS. We next show that INV remains true for $\mathcal{D}$ after any node node is processed within DynamicHS's while-loop. There are two possible cases: *(a)* $\mathsf{node} \not\subseteq \mathcal{D}$ and *(b)* $\mathsf{node} \subseteq \mathcal{D}$.

Let us consider case (a) first. Since no pruning is possible as argued above, i.e., no nodes can be deleted from any node collection stored by DynamicHS except $\mathbf{Q}$ (cf. line 5), the following holds. The processing (and deletion from $\mathbf{Q}$) of node, which is in no subset-relationship with $\mathcal{D}$, *(i)* cannot effectuate an elimination of any other node from $\mathbf{Q}$ (in particular, this holds for all nodes being equal to or a subset of $\mathcal{D}$), and *(ii)* cannot modify $\mathbf{D}_{calc}$. Hence, since INV held before node was processed, INV must still hold thereafter in case (a).

Now, assume case (b). Here, we have again two cases: *(b1)* $\mathsf{node} \subset \mathcal{D}$ and *(b2)* $\mathsf{node} = \mathcal{D}$. Suppose (b1) first, i.e., let $\mathsf{node} \subset \mathcal{D}$ be processed. Because DLABEL is called for any processed node by the argumentation above, we have that line 37 is the only place where nodes can be assigned the label $valid$. Hence, if some node is assigned the label $valid$, this means that FINDMINCONFLICT in line 35 must have returned 'no conflict' for it, which is why this node is a diagnosis by the completeness of FINDMINCONFLICT. Consequently, node cannot be labeled $valid$ because it is a proper subset of a minimal diagnosis. Moreover, node cannot be labeled $nonmin$ in line 26 as there cannot be a subset of node in $\mathbf{D}_{calc}$ due to Observation 2 and the fact that node is a proper subset of a minimal diagnosis. As a result, the DLABEL function must return in either of the lines 31, 34 or 40, in each of which cases a minimal conflict set is returned. This conflict L is then used to generate a new node node_e for each $e \in L$ (lines 15–17), where $|\mathsf{node}_e| = |\mathsf{node}| + 1$ and, for some e, $\mathsf{node}_e \subseteq \mathcal{D}$ must hold (if the latter was not the case then $\mathcal{D}$ would not hit the minimal conflict L, which is a contradiction to $\mathcal{D}$ being a minimal diagnosis). For each of these nodes node_e, either a set-equal node is already in $\mathbf{Q}$ or node_e is added to $\mathbf{Q}$ (cf. lines 18–21 and note that no node set-equal to node_e can be in $\mathbf{D}_{\supset}$ due to Observation 2 and lines 24–26). Hence, in case (b1), INV remains true after node has been processed.

Finally, assume case (b2). Because DLABEL is called for any processed node by the argumentation above, it must be called for node. Since node however is equal to the minimal diagnosis $\mathcal{D}$, DLABEL will return $valid$ (this follows from Observation 2 and the fact that diagnoses are hitting sets of all conflicts). Due to lines 10–11, this means that $\mathcal{D} \in \mathbf{D}_{calc}$ will hold at the beginning of the

next iteration of the while-loop. Consequently, also in case (b2), INV still holds after the processing of node. This completes the proof of the Induction Base.

(Induction Assumption): Assume INV holds for all $n \leq k$.

(Induction Step): Now, let $n = k + 1$. That is, we consider the $(k + 1)$-th call of DynamicHS in Alg. 1. We assume again an arbitrary minimal diagnosis $\mathcal{D}$ for the DPI dpi_k relevant in this call of DynamicHS.[2] By the fact that each minimal diagnosis for dpi_k is either equal to or a superset of some diagnosis for dpi_{k-1} (Property 1.2), and since for each minimal diagnosis $\mathcal{D}'$ for dpi_{k-1} either $\mathcal{D}'$ was in $\mathbf{D}_{calc}$ or some node $\mathsf{n} \subseteq \mathcal{D}'$ was in $\mathbf{Q}$ when the k-th call of DynamicHS returned (Induction Assumption), we infer that some node corresponding to a subset of $\mathcal{D}$ is either in one of $\mathbf{D}_{\checkmark}$ or $\mathbf{D}_{\times}$ (cf. line 14 in Alg. 1 where $\mathbf{D}_{calc}$ is split into $\mathbf{D}_{\checkmark}$ and $\mathbf{D}_{\times}$) or in $\mathbf{Q}$ at the beginning of the $(k + 1)$-th execution of DynamicHS. The first steps in this execution are setting $\mathbf{D}_{calc} = \emptyset$ and calling the function UPDATETREE. Throughout UPDATETREE, some nodes might be pruned (and potentially replaced by set-equal nodes), and all non-pruned nodes from $\mathbf{D}_{\times}$ as well as all nodes from $\mathbf{D}_{\checkmark}$ are finally reinserted into $\mathbf{Q}$. Moreover, each non-pruned node from $\mathbf{D}_{\supset}$ for which there is no known diagnosis that is a subset of it is added to $\mathbf{Q}$ at the end of UPDATETREE. By Observation 1 and since $\mathcal{D}$ is a minimal diagnosis and thus relevant, we have that there must be a node $\mathsf{n} \subseteq \mathcal{D}$ in $\mathbf{Q}$ when the while-loop of the $(k + 1)$-th DynamicHS call is entered. That is, INV holds at the beginning of the while-loop.

That INV remains true for $\mathcal{D}$ after any node node is processed within the while-loop, is shown analogously (i.e., same case analysis and argumentation) as expounded for the Induction Base, except for two aspects: $\mathbf{D}_{\checkmark} \neq \emptyset$ and the PRUNE function (line 33) might be called. Consequences of these aspects are: *(1)* By Observation 1, during the execution of the while-loop of DynamicHS, the last remaining node in $\mathbf{Q}$ which is a subset of some minimal diagnosis cannot be pruned without being replaced by a set-equal node. Neither can a minimal diagnosis be removed from $\mathbf{D}_{calc}$ without being substituted by a set-equal node. Therefore, for every execution of PRUNE (line 33), if INV holds prior to it, INV holds after it finishes. *(2)* Assume $\mathsf{node} \subset \mathcal{D}$ and $\mathsf{node} \in \mathbf{D}_{\checkmark}$. Due to Observation 2 and line 14 in Alg. 1, $\mathbf{D}_{\checkmark}$ includes only diagnoses for the DPI dpi_{k-1} relevant to the preceding (k-th) call of DynamicHS in Alg. 1, and each diagnosis in $\mathbf{D}_{\checkmark}$ during the $(k + 1)$-th call of DynamicHS throughout Alg. 1 is a diagnosis for the DPI dpi_k. Hence, the assumptions $\mathsf{node} \in \mathbf{D}_{\checkmark}$ and $\mathsf{node} \subset \mathcal{D}$ are in contradiction to our assumption that $\mathcal{D}$ is a minimal diagnosis for dpi_k. Equivalently: $\mathsf{node} \subset \mathcal{D}$ implies $\mathsf{node} \notin \mathbf{D}_{\checkmark}$.

The impact of (1) and (2) on the case analysis (cf. Induction Base) is as follows: The argumentation for the case where $\mathsf{node} \not\subseteq \mathcal{D}$ is processed is analogous to case (a) for the Induction Base. The proof for the case $\mathsf{node} \subset \mathcal{D}$ is equal to case (b1) for the Induction Base since DLABEL must be called for node (due to $\mathsf{node} \notin \mathbf{D}_{\checkmark}$). Finally, the case $\mathsf{node} = \mathcal{D}$ is treated as demonstrated in case (b2) in the Induction Base because, if $\mathsf{node} \in \mathbf{D}_{\checkmark}$, then it is simply directly labeled *valid* in line 7 (no call of DLABEL)—hence, whether or not the DLABEL function is called, $\mathcal{D} \in \mathbf{D}_{calc}$ will hold after node having been processed. This completes the proof of the Induction Step, and thus the entire proof. □

A.2 Proof of Soundness

We have to show that DynamicHS outputs only minimal diagnoses. That is, we need to demonstrate that every element in $\mathbf{D}_{calc}$ satisfies the diagnosis property and the minimality property. Since

each call of DynamicHS in the course of Alg. 1 outputs one set $\mathbf{D}_{calc}$, we prove the soundness by induction over the number n of times DynamicHS has already been called in Alg. 1.

(Induction Base): Assume $n = 1$, i.e., DynamicHS is called for the first time in Alg. 1 and returns $\mathbf{D}_{calc}$. Let $\mathsf{node} \in \mathbf{D}_{calc}$. A node is added to $\mathbf{D}_{calc}$ iff it has been labeled $valid$. There are two ways a node may be labeled $valid$, i.e., *(i)* in line 7 and *(ii)* in line 9. Note that case (i) is impossible since $n = 1$ which means that $\mathbf{D}_{\checkmark} = \emptyset$ (cf. Alg. 1) and thus line 7 can never be reached. Therefore, case (ii) applies to node. That is, its label $valid$ is assigned by the DLABEL function. Hence, DLABEL must return in line 37. From this we conclude that the FINDMINCONFLICT call in line 35 returns 'no conflict' which implies that node is a diagnosis (due to the completeness of FINDMINCONFLICT). Assume there is a diagnosis $\mathcal{D}$ such that $\mathcal{D} \subset \mathsf{node}$. Since case (ii) is true for node, there cannot be any such diagnosis $\mathcal{D}$ in $\mathbf{D}_{calc}$ due to lines 24–26, because otherwise node would have been labeled $nonmin$ and line 37 could not have been reached. However, due to the completeness of DynamicHS, and since $\mathcal{D}$ must be ranked higher as per pr than node (cf. the definition of pr in Sec. 3.2.1), and since $\mathcal{D}$ is a diagnosis, $\mathcal{D}$ must already be included in $\mathbf{D}_{calc}$ when node is added. This is a contradiction. Therefore, for $n = 1$ (i.e., for the first call of DynamicHS in Alg. 1), the output $\mathbf{D}_{calc}$ contains only minimal diagnoses.

(Induction Assumption): Assume $\mathbf{D}_{calc}$ contains only minimal diagnoses for $n \leq k$.

(Induction Step): Now, let $n = k + 1$ and $\mathsf{node} \in \mathbf{D}_{calc}$. Analogously to the argumentation above, we again have the two possibilities (i) and (ii) of how node might have attained its label $valid$. Suppose case (i) first. That is, $\mathsf{node} \in \mathbf{D}_{\checkmark}$. By line 14 of Alg. 1 (ASSIGNDIAGSOKNOK, cf. Sec. 3.1.3), $\mathbf{D}_{\checkmark} \subseteq \mathbf{D}_{calc}$ where $\mathbf{D}_{calc}$ is the output of the the previous, i.e., the k-th, call of DynamicHS. Due to the Induction Assumption, we have that $\mathbf{D}_{\checkmark}$ includes only minimal diagnoses for the DPI dpi_{k-1} considered in the k-th iteration, i.e., the DPI dpi_k considered in the $(k+1)$-th iteration without the most recently added measurement. However, ASSIGNDIAGSOKNOK adds to $\mathbf{D}_{\checkmark}$ exactly those diagnoses that are consistent with the new measurement. Consequently, the diagnoses in $\mathbf{D}_{\checkmark}$ are consistent with all measurements included in dpi_k, and thus are diagnoses for dpi_k. Due to Property 1.2, no diagnosis for dpi_k can be a proper subset of any diagnosis for dpi_{k-1}. Thus, all elements of $\mathbf{D}_{\checkmark}$ must be minimal diagnoses which is why node must be a minimal diagnosis. For the other case (ii), the argumentation is exactly as for the Induction Base. □

Appendix B. Advanced Techniques in DynamicHS

DynamicHS embraces several sophisticated techniques specialized in improving its (time or space) performance, which we discuss next.

B.1 Efficient Redundancy Checking

We now detail the workings of the function REDUNDANT (called in line 43 of Alg. 3):

The definition of node redundancy given in Sec. 3.2.5 directly suggests a method for checking whether or not a node is redundant, which we call *complete redundancy check (CRC)*. It runs through all conflicts $\mathsf{nd.cs}[i]$ used as labels along the branch to node nd (i.e., $i \in \{1, \ldots, |\mathsf{nd.cs}|\}$) and calls FINDMINCONFLICT with arguments $(\langle \mathsf{nd.cs}[i] \setminus \{\mathsf{nd}[i]\}, \mathcal{B}, P \cup P', N \cup N' \rangle)$ to test if there is a witness of redundancy (see Sec. 3.2.5) for nd. A witness of redundancy exists for nd iff, for some i, this call to FINDMINCONFLICT returns a conflict X. Because, this conflict then must be a subset of $\mathsf{nd.cs}[i] \setminus \{\mathsf{nd}[i]\}$, meaning that nd is redundant. Hence, if such an X is found, then CRC is successful. In this case, the function REDUNDANT returns $isRedundant = true$ along with the

witness X, which is used as an argument passed to the subsequent call of the PRUNE method (Alg. 3, line 45). Otherwise, i.e., if all calls of FINDMINCONFLICT made by the CRC return 'no conflict', it is proven that the node is not redundant. As a result, REDUNDANT returns $isRedundant = false$ (note, in this case, the second value X returned by REDUNDANT is irrelevant and not further used).

The CRC enables sound (if CRC true, then node redundant) and complete (if node redundant, then CRC true) redundancy checking. However, a drawback of the CRC is that it requires $|\mathsf{nd}|$ calls to the (expensive) method FINDMINCONFLICT in the worst case, where $|\mathsf{nd}|$ is in $O(|\mathbf{conf}(dpi)|)$ since a node cannot hit any more than each minimal conflict for the current DPI dpi. As a remedy to that, we devised a more efficient, sound but incomplete, so-called *quick redundancy check (QRC)*, which is executed previous to the CRC and requires only a single call of FINDMINCONFLICT. The concept is that a positive QRC makes the more expensive CRC obsolete; and, in case of a negative outcome, CRC must be executed, but the overhead amounts to only a single FINDMINCONFLICT call.

To check the redundancy of nd, QRC executes FINDMINCONFLICT with arguments $(\langle U_{\mathsf{nd.cs}} \setminus \mathsf{nd}, \mathcal{B}, P \cup P', N \cup N' \rangle)$.[3] If 'no conflict' is returned, the QRC terminates negatively, which prompts the execution of the CRC (described above). Otherwise, if a conflict X is returned, QRC checks whether X is a proper subset of some conflict in $\mathsf{nd.cs}$, i.e., whether $X \subset \mathcal{C}$ for $\mathcal{C} = \mathsf{nd.cs}[k]$ for some k. In case of a positive subset-check, the QRC returns positively and it follows that nd is redundant, regardless of the particular k. The reason is that the argument $U_{\mathsf{nd.cs}} \setminus \mathsf{nd}$ passed to FINDMINCONFLICT does not include any element of nd, and hence the output conflict cannot include such elements either. Thus, if $X \subset \mathsf{nd.cs}[k]$ holds, then $X \subset \mathsf{nd.cs}[k] \setminus \mathsf{nd}$, and therefore $X \subset \mathsf{nd.cs}[k] \setminus \{\mathsf{nd}[i]\}$ for all i, and in particular for $i = k$. So, $X \subset \mathsf{nd.cs}[k] \setminus \{\mathsf{nd}[k]\}$, which is equivalent to the definition of redundancy (see Sec. 3.2.5). As a consequence, if QRC returns positively, then the function REDUNDANT directly outputs $isRedundant = true$ along with X, and CRC is not (required to be) executed.

To see why the QRC is incomplete, i.e., that nd *can* be redundant even if the outcome of the QRC is negative, consider the following example:

Example I Let $\mathsf{nd} = [1, 2]$ and $\mathsf{nd.cs} = [\langle 1, 2 \rangle, \langle 2, 3 \rangle]$. Assume that $X := \langle 2 \rangle$ is a new minimal conflict and that $\{3\}$ is not a conflict. Clearly, this implies that nd is redundant because $\mathsf{nd.cs}[1] \setminus X = \{1\}$ and $\mathsf{nd}[1] = 1$. However, $U_{\mathsf{nd.cs}} \setminus \mathsf{nd} = \{3\}$, which is not a conflict, which is why FINDMINCONFLICT given the DPI $(\langle U_{\mathsf{nd.cs}} \setminus \mathsf{nd}, \mathcal{B}, P \cup P', N \cup N' \rangle)$ as argument returns 'no conflict'. Hence, the QRC returns negatively although nd is in fact redundant. □

The crucial aspect which makes this incompleteness possible is the potential overlapping of conflicts. Exactly this overlapping effectuates in the above example that more than one element (actually even all elements) of the outdated non-minimal conflict $\langle 1, 2 \rangle$ are eliminated from $U_{\mathsf{nd.cs}} = \{1, 2, 3\}$ by deleting $\mathsf{nd} = [1, 2]$. As a consequence, the new reduced conflict $\langle 2 \rangle$ is not contained any longer in the set tested by FINDMINCONFLICT.

In fact, we can conclude that the QRC is sound *and complete* in the special cases where all minimal conflicts are pairwise disjoint or, more generally, where nd does not include any element that occurs in multiple conflicts in $\mathsf{nd.cs}$.

B.2 Lazy Updating Policy

Updating DynamicHS's hitting set tree after the detection of some witness of redundancy X involves going through all nodes of the tree and checking their redundancy wrt. X (cf. Sec. 3.2.5). To avoid these costs as much as possible, DynamicHS aims at minimizing the number of performed updates under preservation of its correctness. This can be seen from line 42, where only the set $\mathbf{D}_{\times}$ including the most "suspicious" nodes (i.e., the diagnoses invalidated by the latest added measurement) is checked for redundancy. In general, this means that we allow some differences between the tree used by DynamicHS to compute diagnoses and the one that would be obtained when building Reiter's HS-Tree for the current DPI from scratch. More specifically, we allow the presence of non-minimal conflicts that are used as node labels as well as—conditioned by these non-minimal conflicts—the presence of unnecessary nodes (while, of course, seeking to minimize the number of such occurrences; cf. PRUNE function). Still, every time a conflict is used to label a (newly processed) node, the algorithm guarantees that it is a minimal conflict for the current DPI (cf. the conflict-minimality test in the course of the conflict reuse check in DLABEL, lines 29–34).

This *lazy updating policy* takes effect, e.g., in iteration 2 of the example execution of DynamicHS shown in Fig. 3 (see Example 5). Here the, at this point, already non-minimal conflict $\mathcal{C}_{\neg min} := \langle 3, 4, 5 \rangle$ still appears as a node label, while $\mathcal{C}_{min} := \langle 4, 5 \rangle$ is now a minimal conflict for the current DPI.

Notwithstanding the correctness proof of DynamicHS given in Appendix A, we next argue briefly why the presence of such non-minimal conflicts $\mathcal{C}_{\neg min}$ does neither counteract the soundness nor the completeness of DynamicHS. To this end, we first point out that (*) only nodes can be labeled $valid$ by DynamicHS which are diagnoses for the current DPI (see line 35 in DLABEL; and line 6 along with the definition of $\mathbf{D}_{\checkmark}$):

- Assume the *completeness* is compromised. Then the processing of some minimal diagnosis must be prevented from reaching line 35 in DLABEL (note that it is unavoidable that processed nodes are recognized as diagnoses in line 6). Since the presence of all still relevant (i.e., non-redundant) nodes is not harmed by the presence of redundant nodes, which just constitute *additional* branches, each node corresponding to a minimal diagnosis $\mathcal{D}$ will sooner or later be processed by DLABEL. As $\mathcal{D}$ does not reach line 35 and since a minimal diagnosis does hit all (minimal) conflicts, a labeling of $\mathcal{D}$ by $nonmin$ (line 26) is the only possible case. That is, there must be a node labeled $valid$ (and thus stored in $\mathbf{D}_{calc}$) which is a proper subset of $\mathcal{D}$. This is a contradiction to (*).

- Assume the *soundness* is compromised. Then some node nd is labeled $valid$ which is not a minimal diagnosis. Due to (*), it must be the case that nd is a diagnosis, but a non-minimal one. First, a non-minimal diagnosis can never be identified in line 6 because of Property 1.2. Second, since the queue $\mathbf{Q}$ is always sorted such that node n is ranked prior to node n' whenever $\mathsf{n} \subset \mathsf{n}'$ (cf. Sec. 3.2.1), the non-minimal diagnosis nd can only be labeled $valid$ if some minimal diagnosis $\mathcal{D}$ ($\subset \mathsf{nd}$) —processed prior to nd— is not found to be a diagnosis (and thus not labeled $valid$ and not stored in $\mathbf{D}_{calc}$). Hence, the completeness must be compromised. This is a contradiction to the argumentation in the above bullet.

B.3 Avoidance of Expensive Reasoning

As explained in Sec. 2, DynamicHS aims at reducing the reaction *time* of a sequential diagnosis system. One crucial time-consuming and recurring operation in hitting-set-based diagnosis computation algorithms is the reasoning in terms of logical consistency checks. To optimize computation time, DynamicHS is therefore equipped with strategies that minimize the amount of and time spent for reasoning by exploiting its statefulness in terms of the hitting set tree maintained throughout its iterations. We discuss the concept behind these strategies next.

Ways of Reducing the Cost for Reasoning. In DynamicHS, the logical inference engine is called (solely) by the FINDMINCONFLICT function, which is involved in the determination of node labels in the tree expansion phase (DLABEL function) and in the evaluation of node redundancy in the tree update phase (REDUNDANT function). As discussed in Sec. 2.3, a single execution of FINDMINCONFLICT given the DPI $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$ generally requires multiple reasoner calls and their number depends critically on the size of the universe $\mathcal{K}$ from which a minimal conflict should be computed. Note, what we, for simplicity, refer to as a *reasoner call* actually corresponds to a check if some $\mathcal{C} \subseteq \mathcal{K}$ is a conflict for a DPI $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$. By the definition of a conflict (see Sec. 2.3), this means checking whether some $x \in N \cup \{\bot\}$ exists such that $\mathcal{C} \cup \mathcal{B} \cup P \models x$. Consequently, a reasoner call corresponds to a maximum of $|N|+1$ logical consistency checks. E.g., if QuickXPlain (Junker, 2004; Rodler, 2020) is used to implement the FINDMINCONFLICT function, as in our evaluations (cf. Sec. 5), then the worst-case number of consistency checks executed by a single call of FINDMINCONFLICT on the DPI $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$ is in $O(|\mathcal{K}|(|N|+1))$ (Marques-Silva, Janota, & Belov, 2013). The hardness of consistency checking tends to increase with the size of the knowledge base on which the check is performed (cf., e.g., (Gonçalves, Parsia, & Sattler, 2012)).[4] In other words, the smaller the size of $\mathcal{K} \cup \mathcal{B} \cup P$ is, the more efficient consistency checking will tend to be in the course of FINDMINCONFLICT operating on the DPI $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$.

In summary, the lower the cardinality of the first entry $\mathcal{K}$ (number of system components) of the tuple $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$ provided as an input to FINDMINCONFLICT is, the lower the hardness and the number of executed consistency checks will tend to be, and thus the faster FINDMINCONFLICT will tend to execute. Hence, there are basically three different ways of scaling down the necessary reasoning throughout DynamicHS:

- *(i)* Reducing the number and hardness of consistency checks made while FINDMINCONFLICT executes,
- *(ii)* reducing the number of FINDMINCONFLICT calls, or
- *(iii)* entirely avoiding FINDMINCONFLICT calls (and replacing them by equivalents that do not involve reasoning).

In its various stages, DynamicHS embraces all these three approaches, as we explain next.

In the tree expansion phase, any FINDMINCONFLICT call in the course of the conflict reuse check (lines 27–34) starts from an *already computed* conflict—and *not* from the entire set of system components $\mathcal{K}$—trying to verify its minimality or, alternatively, extracting a subset which constitutes a minimal conflict (for the current DPI). That is, FINDMINCONFLICT is given a set of at most $|\mathcal{C}_{\max}|$ elements as an input,[5] where $\mathcal{C}_{\max}$ is the conflict of maximal size (for the original[6] DPI, i.e., the one given as an input to Alg. 1). Note that, in many practical applications involving systems of non-negligible size, $|\mathcal{C}_{\max}|$ is significantly (if not orders of magnitude) smaller than the number of components of the diagnosed system (cf., e.g., (Horridge, Parsia, & Sattler, 2012; Shchekotykhin,

Friedrich, Fleiss, & Rodler, 2012; Shchekotykhin, Jannach, & Schmitz, 2015)). Thus, DynamicHS applies strategy *(i)* in this stage.

In the tree update phase, and particularly during redundancy detection (line 43), the quick redundancy check (QRC) is employed to potentially replace the complete redundancy check (CRC) by making only a single FINDMINCONFLICT call instead of multiple ones (cf. Appendix B.1). Moreover, any call of FINDMINCONFLICT made in the course of the redundancy detection in general involves a significantly reduced input set, as compared to the overall number of components $|\mathcal{K}|$ of the diagnosed system. The cardinality of this input set is bounded by the cardinality of the union of all minimal conflicts (for the original[7] DPI). So, during redundancy checking, both strategies *(i)* and *(ii)* are pursued.

In the tree pruning phase (PRUNE function), which constitutes a part of the tree update phase, no reasoning is required at all (i.e., no FINDMINCONFLICT calls). This is accomplished by leveraging the stored hitting set tree as well as adequate instructions operating on sets and lists (cf. Example 3 below). At this point, note that a stateless algorithm, in contrast, *has to* draw on logical reasoning to reconstruct (the still relevant) parts of the tree, and thus to achieve essentially the same as DynamicHS's pruning actions. Importantly, operations relying on a logical inference engine can be expected to have a (much) higher time complexity than the list concatenations, set-equality or subset checks performed by DynamicHS in lieu of these operations. For instance, reasoning with propositional logic is already NP-complete (Cook, 1971), not to mention more expressive languages such as Description logics (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2007), whereas set- and list-operations are (mostly linear) polynomial time operations. So, as far as the tree pruning is concerned, DynamicHS can be viewed as trading cheaper (reasoner-free) operations for expensive reasoner calls. It thus makes use of strategy *(iii)*.

Different Types of Reasoning Operations. Given the preceding discussion, we can at the core distinguish the following three categories of FINDMINCONFLICT function calls based on their input DPI $\langle X, \mathcal{B}, P, N\rangle$ (where $\langle \mathcal{K}, \mathcal{B}, P, N\rangle$ is the DPI relevant to the current iteration of DynamicHS):[8]

- *"hard"*: Size of X in the order of number of system components, i.e., $|X| \approx |\mathcal{K}|$, *and* a conflict is returned. (*multiple "hard" reasoner calls*)
- *"medium"*: Size of X in the order of number of system components, i.e., $|X| \approx |\mathcal{K}|$, *and* 'no conflict' is returned. (*single "hard" reasoner call*)
- *"easy"*: Size of X low compared to the number of system components, i.e., $|X| \ll |\mathcal{K}|$. (*few "easy" reasoner calls*)

"Hard" FINDMINCONFLICT calls are those executions of line 35 (in DLABEL) that compute a fresh conflict, and "medium" ones those which lead to the finding of a diagnosis (output 'no conflict'). In contrast, "easy" FINDMINCONFLICT invocations are those geared towards redundancy checking (line 43, UPDATETREE) and minimality testing for reused conflicts (line 29, DLABEL). In terms of this characterization, compared against HS-Tree, DynamicHS tries to substantially reduce the "hard" (and "medium") FINDMINCONFLICT operations at the cost of performing an as small as possible number of "easy" ones.

Example II Reconsider our example DPI in Tab. 1 and the evolution of the hitting set computation throughout a sequential diagnosis session for DynamicHS and HS-Tree discussed in Example 5. Tab. I shows the number of "hard", "medium" and "easy" FINDMINCONFLICT calls throughout the

Table I: Stats wrt. the number of different kinds of reasoning operations (FINDMINCONFLICTS calls) throughout the execution of DynamicHS (DHS) and HS-Tree (HST), respectively, on the example DPI from Tab. 1. "Ui" in the first column refers to the tree update performed by DynamicHS subsequent to iteration i. Note, HS-Tree does not (need to) perform any tree updates.

	# "hard"		# "medium"		# "easy"	
iteration	DHS	HST	DHS	HST	DHS	HST
1	4	4	4	4	0	0
U1	0	–	0	–	2	–
2	1	4	0	2	0	0
U2	0	–	0	–	1	–
3	1	4	1	2	0	0
U3	0	–	0	–	1	–
4	0	2	0	1	0	0
total	6	14	5	9	4	0

sequential session executed by both algorithms. The "hard" and "medium" calls are denoted by ^{C} and *, respectively, in the hitting set trees depicted by Figs. 3 and 4 ("easy" calls are not indicated as tree updates are not displayed in the figures). We can see that DynamicHS trades a significant reduction of "hard" (57 %) and "medium" (44 %) reasoner operations for some "easy" ones. □

B.4 Space-Saving Duplicate Storage and On-Demand Reconstruction

Basically, there are several options how to organize the storage of duplicate nodes. These options range from storing all of them *explicitly* to storing only a minimal set of (stubs of) duplicate nodes that *implicitly* allow all duplicates to be reconstructed on demand. Since DynamicHS performs the duplicate check at node generation time (cf. line 18), it uses the more natural way of handling duplicate storage given by the latter strategy. This means directly adding detected duplicate nodes—generated tree branches whose *set* of edge labels equals the *set* of edge labels of an active branch (node in $\mathbf{Q}$ or $\mathbf{D}_{\supset}$)—to the collection $\mathbf{Q}_{dup}$ without further extending them as the hitting set tree grows. Hence, each node stored in $\mathbf{Q}_{dup}$ is potentially only a partial duplicate node and might need to be combined with some other (partial) duplicate node or some active node in the hitting set tree to explicitly generate (or: reconstruct) a duplicate that is only implicitly stored. For instance, assume two nodes $\mathsf{n}_1, \mathsf{n}_2$ that have been detected as duplicates and added to $\mathbf{Q}_{dup}$, where $\mathsf{n}_1 = [3,2], \mathsf{n}_1.\mathsf{cs} = [\langle 1,2,3\rangle, \langle 2,4\rangle]$ and $\mathsf{n}_2 = [2,3,1], \mathsf{n}_2.\mathsf{cs} = [\langle 1,2,3\rangle, \langle 3,4\rangle, \langle 1,4\rangle]$ (cf. nodes with numbers ⑤ and ⑨ in Fig. I, discussed in more detail in Example 3). Then, an implicit duplicate constructible from n_1 and n_2 is $\mathsf{n}_{1,2} = [3,2,1], \mathsf{n}_{1,2}.\mathsf{cs} = [\langle 1,2,3\rangle, \langle 2,4\rangle, \langle 1,4\rangle]$, where the last node label (conflict $\langle 1,4\rangle$) and edge (labeled by 1) of n_2 have been appended to n_1. The rationale behind this node combination is as follows: n_1 was recognized as duplicate first, while the first part (i.e., the first two node and edge labels) of n_2 was still in the queue $\mathbf{Q}$ of open nodes. This first part of n_2 was then extended by the node label $\langle 1,4\rangle$ and the edge label 1, but was subsequently itself spotted as a duplicate. Node n_1, however, given it had still been in $\mathbf{Q}$, would have undergone the same extension. This extension is so to say "made good for" by combining n_1 with n_2 to (re)construct $\mathsf{n}_{1,2}$.

In general, to reconstruct a duplicate node $\mathsf{n}_{i,j}$ from a combination of two nodes $\mathsf{n}_i, \mathsf{n}_j$, the following criteria have to be met:

(D1) The first $|\mathsf{n}_i|$ elements of n_j interpreted as a set are equal to the elements of n_i interpeted as a set; there are no conditions on the conflict labels $\mathsf{n}_i.\mathsf{cs}$ and $\mathsf{n}_j.\mathsf{cs}$ of the combined nodes.

(D2) The reconstructed node $\mathsf{n}_{i,j}$ is built by setting[9] $\mathsf{n}_{i,j}[1..|\mathsf{n}_i|] = \mathsf{n}_i$ and $\mathsf{n}_{i,j}[|\mathsf{n}_i| + 1..|\mathsf{n}_j|] = \mathsf{n}_j[|\mathsf{n}_i| + 1..|\mathsf{n}_j|]$ as well as $\mathsf{n}_{i,j}.\mathsf{cs}[1..|\mathsf{n}_i|] = \mathsf{n}_i.\mathsf{cs}$ and $\mathsf{n}_{i,j}.\mathsf{cs}[|\mathsf{n}_i| + 1..|\mathsf{n}_j|] = \mathsf{n}_j.\mathsf{cs}[|\mathsf{n}_i| + 1..|\mathsf{n}_j|]$, i.e., the first part of $\mathsf{n}_{i,j}$ and $\mathsf{n}_{i,j}.\mathsf{cs}$, respectively, is equal to n_i and $\mathsf{n}_i.\mathsf{cs}$, to which the last part of n_j and $\mathsf{n}_j.\mathsf{cs}$ is appended.

(D3) Either *(a)* $\mathsf{n}_i, \mathsf{n}_j$ are each (reconstructed[10] or explicit) nodes from $\mathbf{Q}_{dup}$, or *(b)* node n_i is from the node combination closure $\mathbf{Q}^*_{dup}$ of $\mathbf{Q}_{dup}$ which is the union of $\mathbf{Q}_{dup}$ with the set of all nodes reconstructible[11] through *(a)*, and node n_j is from a node collection including active nodes, i.e., from one of $\mathbf{Q}$, $\mathbf{D}_{\supset}$, $\mathbf{D}_{\checkmark}$, $\mathbf{D}_{\times}$, or $\mathbf{D}_{calc}$.

In the example above, criterion *(D3)(a)* is met, where n_1 and n_2 correspond to n_i and n_j, respectively, which are both (explicit) elements of $\mathbf{Q}_{dup}$; criterion *(D1)* is satisfied as well because the first $|\mathsf{n}_i| = |\mathsf{n}_1| = 2$ elements of $\mathsf{n}_j = \mathsf{n}_2$ correspond to the set $\{2, 3\}$, which is equal to the set of elements of $\mathsf{n}_i = \mathsf{n}_1$; the validity of criterion *(D2)* can be easily verified by comparing $\mathsf{n}_{1,2}$ with n_1 and n_2. Let us consider some important remarks:

(R1) *Node reconstruction is sound and complete:* The set of all nodes constructible by means of *(D1)*, *(D2)* and *(D3)(b)* is exactly the set of all duplicates of the currently active nodes in $\mathbf{Q} \cup \mathbf{D}_{\supset} \cup \mathbf{D}_{\checkmark} \cup \mathbf{D}_{\times} \cup \mathbf{D}_{calc}$.

(R2) *Relationship between reconstructed node and combined source nodes: (D1)* implies that $|\mathsf{n}_i| \leq |\mathsf{n}_j|$. By *(D2)* node reconstruction means that the node n_i of lower (or equal) length replaces the first part of (or the complete) node n_j; we can thus call n_i the *modifying node* and n_j the *modified node*. Moreover, the reconstructed node has the same length as and is *set*-equal (wrt. *edge* labels) to node n_j. As a consequence of this, node reconstructions can never lead to nodes that are new in terms of their sets of edge labels. Hence, as far as *sets* of edge labels of nodes are concerned, $\mathbf{Q}_{dup}$ is representative of $\mathbf{Q}^*_{dup}$.

(R3) *Reconstruction of nodes only on demand:* Neither $\mathbf{Q}^*_{dup}$ (as per *(D3)(a)*) nor the set of all duplicates of active nodes (as per *(D3)(b)*) is ever exlicitly generated by DynamicHS. Instead, only a minimal number of node reconstructions necessary for the proper-functioning (completeness) of DynamicHS are performed. More specifically, node reconstructions can only take place in case one node has been pruned and a replacement node for it is sought (cf. PRUNE function, Sec. 3.2.5). And, for each pruned node, either just one replacement node is reconstructed, or none at all if no suitable replacement node exists.

(R4) *Principle of node reconstruction in the course of tree pruning:* Assume the PRUNE function is called given the minimal conflict X and finds some redundant node nd, i.e., X is a witness of redundancy for nd (cf. Sec. 3.2.5). Since there might be multiple edge and conflict labels in nd and $\mathsf{nd}.\mathsf{cs}$ due to which nd is redundant given X, let k be the maximal index such that $X \subset \mathsf{nd}.\mathsf{cs}[k]$ and $\mathsf{nd}[k] \in \mathsf{nd}.\mathsf{cs}[k] \setminus X$ (redundancy criterion, cf. Sec. 3.2.5). After deleting

nd, a replacement node for it is sought. A replacement node nd' of nd needs to be *(i)* non-redundant (as per the current knowledge, i.e., X must not be a witness of redundancy for nd') and *(ii)* set-equal to nd (cf. Sec. 3.2.5).

Due to (ii) and Remark (R2) above, the redundant node nd can be interpreted as n_j and the sought replacement node as $\mathsf{n}_{i,j}$. With that said, the task of finding a replacement node is equivalent to finding a non-redundant (as per X) node n_i in $\mathbf{Q}^*_{dup}$, see *(D3)*, such that $|\mathsf{n}_i| \geq k$ (i.e., at least the redundant part of $\mathsf{n}_j = \mathsf{nd}$ is replaced by n_i), see *(D2)*, and the set of elements of n_i is equal to the set of the first $|\mathsf{n}_i|$ elements of $\mathsf{n}_j = \mathsf{nd}$, see *(D1)*. Since $\mathbf{Q}_{dup}$ is representative of $\mathbf{Q}^*_{dup}$ in terms of the *sets* of edge labels of nodes (see Remark (R2)) and because n_i must only be suitable in terms of *set*-equality, it is sufficient to search for n_i in $\mathbf{Q}_{dup}$ as opposed to $\mathbf{Q}^*_{dup}$.

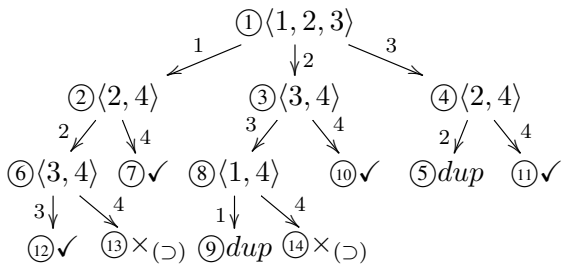
Due to (i) and since n_i is a node from $\mathbf{Q}_{dup}$, it must be provided that each node from $\mathbf{Q}_{dup}$ that qualifies as n_i in the search for a replacement node is non-redundant. This imposes two requirements, as pointed out in Sec. 3.2.5: $\mathbf{Q}_{dup}$ must be pruned previous to all other node collections (to account for case *(D3)(b)*), and nodes of $\mathbf{Q}_{dup}$ must be pruned in ascending order of their length (to account for case *(D3)(a)*[12]).

Example III We now showcase the workings of DynamicHS's tree update on a simple example, thereby also illustrating the discussed advanced techniques regarding the *efficient redundancy checking* (Appendix B.1), the *avoidance of expensive reasoning* (Appendix B.3), as well as the *space-saving duplicate storage and on-demand reconstruction* (Appendix B.4). Note that we already discussed the *lazy updating policy* (Appendix B.2) in terms of Example 5.

Consider Fig. I (with a similar notation as used in Figs. 3 and 4) which depicts the hitting set tree produced by DynamicHS (iteration 1) for some DPI dpi_0 in breadth-first order (see the node numbers ⓣ signalizing that the respective node was generated at point in time t). The sets of minimal conflicts and minimal diagnoses for dpi_0 are given by $\mathbf{conf}(dpi_0)$ and $\mathbf{diag}(dpi_0)$ in the figure. We assume that DynamicHS uses the parameter $ld := 5$, i.e., five minimal diagnoses (if existent) should be computed. Since there are only four minimal diagnoses (see $\mathbf{diag}(dpi_0)$), DynamicHS executes until the queue is empty ($\mathbf{Q} = [\,]$, see line 4 in Alg. 3), i.e., until the hitting set tree is complete. The resulting set of leading diagnoses $\mathbf{D}$ corresponds to $\mathbf{diag}(dpi_0)$ (nodes labeled by ✓ in Fig. I). Further, we assume that a (discriminating) measurement m is added to dpi_0, which leads to the new minimal conflict $\langle 3 \rangle$ for the resulting DPI dpi_1. As can be seen through a comparison of $\mathbf{diag}(dpi_0)$ with $\mathbf{diag}(dpi_1)$ in Fig, I, the leading diagnoses eliminated by the measurement m are the nodes numbered ⑦ (corresponding to the node $[1, 4]$) and ⑩ (node $[2, 4]$). These two nodes are included in the set $\mathbf{D}_\times$ given as an input argument to the second call of DynamicHS (iteration 2) in line 6 of Alg. 1.

Now, when UPDATETREE is invoked in iteration 2, the first step is the examination of the elements in $\mathbf{D}_\times$ regarding their redundancy status (see description of the UPDATETREE function in Sec. 3.2.5). For node $\mathsf{nd} = [1, 4]$, we have $\mathsf{nd.cs} = [\langle 1, 2, 3 \rangle, \langle 2, 4 \rangle]$. The QRC (see Appendix B.1) executed on nd involves calling FINDMINCONFLICT with argument[13] $(\langle U_{\mathsf{nd.cs}} \setminus \mathsf{nd}, \dots \rangle)$ which is equal to $(\langle \{1, 2, 3, 4\} \setminus \{1, 4\}, \dots \rangle) = (\langle \{2, 3\}, \dots \rangle)$. Hence, the conflict $X = \langle 3 \rangle$ is returned (cf. $\mathbf{conf}(dpi_1)$), which is a subset of $\mathsf{nd.cs}[1]$ and thus must constitute a witness of redundancy of $\mathsf{nd} = [1, 4]$.

As a next step, PRUNE is called with argument X (line 45 in Alg. 3). At first, PRUNE considers $\mathbf{Q}_{dup} = [[3, 2], [2, 3, 1]]$ (nodes numbered ⑤ and ⑨ in Fig. I) and cleans it up from redundant nodes.

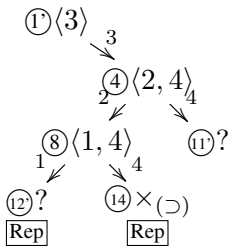


DynamicHS-Tree for dpi_0

$\mathbf{conf}(dpi_0) = \{\langle 1,2,3\rangle, \langle 2,4\rangle, \langle 3,4\rangle, \langle 1,4\rangle\}$
$\mathbf{diag}(dpi_0) = \{[1,2,3],[1,4],[2,4],[3,4]\} = \mathbf{D}$

one measurement m added $\Rightarrow$ $\mathbf{D}_{\checkmark} = \{[1,2,3],[3,4]\}, \mathbf{D}_{\times} = \{[1,4],[2,4]\}$ $\Rightarrow$ result: dpi_1

$\mathbf{conf}(dpi_1) = \{\langle 3\rangle, \langle 2,4\rangle, \langle 1,4\rangle\}$
$\mathbf{diag}(dpi_1) = \{[1,2,3],[3,4]\}$



DynamicHS-Tree after UPDATETREE

Figure I: Tree pruning and redundancy checking example.

At this, for each node $\mathsf{nd} \in \mathbf{Q}_{dup}$, the algorithm runs through the conflicts $\mathsf{nd.cs}[i]$ from small to large i and, if $X \subset \mathsf{nd.cs}[i]$, *(i)* checks if $\mathsf{nd}[i] \in \mathsf{nd.cs}[i] \setminus X$ (is X a witness of redundancy for nd?) as well as *(ii)* replaces $\mathsf{nd.cs}[i]$ by X (update of internal node labels). At the end of this traversal, the maximal index $i = k$, for which (i) is executed and true, is stored. Since, importantly, nodes are processed in ascending order of their size, the first processed node from $\mathbf{Q}_{dup}$ in this concrete example is $\mathsf{nd} = [3,2]$ with $\mathsf{nd.cs} = [\langle 1,2,3\rangle, \langle 2,4\rangle]$. For index $i = 1$, we have $\langle 3\rangle \subset \langle 1,2,3\rangle$, which is why $\mathsf{nd.cs}[1]$ is replaced by $\langle 3\rangle$ in the course of step (ii). However, since $\mathsf{nd}[1] = 3$ is not an element of $\langle 1,2,3\rangle \setminus \langle 3\rangle$, check (i) is negative (no redundancy detected). For index $i = 2$, it does not even hold that $\langle 3\rangle \subset \langle 2,4\rangle$, hence neither (i) nor (ii) are executed. The overall conclusion from this analysis is that X is not a witness of redundancy for nd. Consequently, there is no evidence up to this point that nd is redundant, which is why nd remains an element of $\mathbf{Q}_{dup}$, however, with the modified conflict labels set $\mathsf{nd.cs} = [\langle 3\rangle, \langle 2,4\rangle]$.

For the second node $[2,3,1]$ in $\mathbf{Q}_{dup}$, a similar evaluation leads to the insight that this node is redundant and $k = 1$ (because $\langle 3\rangle \subset \langle 1,2,3\rangle$ and $2 \in \langle 1,2,3\rangle \setminus \langle 3\rangle$). However, instead of only discarding the node, the algorithm seeks a replacement node that is non-redundant and *set*-equal

to $[2, 3, 1]$. To this end, it iterates through all already processed (and thus provenly non-redundant) nodes in $\mathbf{Q}_{dup}$ (in this case only the node $[3, 2]$) and tries to find some node of size l which is set-equal to the first l elements of the redundant node, for some $l \geq k$.

Indeed, since the first $2 \geq k = 1$ elements of nodes $[3, 2]$ and $[2, 3, 1]$ are equal (when considered as a set), a replacement node for the latter can be constructed. This (re)constructed node is given by $\mathsf{ndnew} = [3, 2, 1]$ with $\mathsf{ndnew.cs} = [\langle 3\rangle, \langle 2, 4\rangle, \langle 1, 4\rangle]$ (i.e., figuratively spoken, the label dup at node ⑤ is replaced by the path including nodes ⑧ and ⑨, cf. Fig. I).

Since all nodes of $\mathbf{Q}_{dup}$ have at this point been processed, the PRUNE method shifts its focus to the other collections $\mathbf{Q}$, $\mathbf{D}_{\supset}$, $\mathbf{D}_{\times}$ and $\mathbf{D}_{\checkmark}$. Essentially, the considerations made for these collections are equal to those explicated for $\mathbf{Q}_{dup}$, with the only difference that solely elements of (the already cleaned up) $\mathbf{Q}_{dup}$ are in line for being used in the construction of replacement nodes for redundant ones found in these collections. For instance, the node $\mathsf{n} = [1, 2, 3]$ with the number ⑫ is detected to be redundant ($k = 1$) when it comes to pruning $\mathbf{D}_{\checkmark}$. Since, however, the first $3 \geq k = 1$ elements of $\mathsf{ndnew} = [3, 2, 1]$ are set-equal to the first 3 elements of n, the latter is replaced by ndnew. After carrying out all pruning actions provoked by the witness of redundancy $X = \langle 3\rangle$ (detected by analysing the first node $[1, 4] \in \mathbf{D}_{\times}$), the reconstructed replacement node ndnew is now the only remaining node in the tree that corresponds to the set of edge labels $\{1, 2, 3\}$. The fact that this set does constitute a minimal diagnosis wrt. dpi_1 (cf. $\mathbf{diag}(dpi_1)$) corroborates that the storage and adequate reconstruction of duplicates is pivotal for the completeness of DynamicHS.

Note that the second node $[2, 4]$ that was originally an element of $\mathbf{D}_{\times}$ has meanwhile already been removed from $\mathbf{D}_{\times}$ in the course of the pruning actions taken. The reason is that the conflict $X = \langle 3\rangle$ that was used as a basis for pruning is also a witness of redundancy for $[1, 4]$. In fact, $\mathbf{D}_{\times} = \emptyset$ holds after conflict $X = \langle 3\rangle$ has been processed. Hence, the for-loop in line 42 of Alg. 3 terminates and no further pruning operations are conducted.

The pruned tree resulting from the execution of UPDATETREE is shown at the bottom of Fig. I. Replacement nodes (i.e., those nodes which substitute deleted redundant nodes), are marked by Rep; their node number ⓚ in the pruned tree is the number of the original deleted redundant node. Note that the second replacement node $[3, 2, 4]$ results from (the deleted) node ⑭ by a substitution of its first two edges $[2, 3]$ by the duplicate $[3, 2]$ (node ⑤). In addition, node labels changed during the pruning process are indicated by a prime (') symbol. In this concrete example, e.g., both relabeled nodes ⑪ and ⑫ originally represented minimal diagnoses for dpi_0 and were returned by the first run of DynamicHS in terms of $\mathbf{D}_{calc}$. Then, they were added to $\mathbf{D}_{\checkmark}$ (line 14 in Alg. 1) since they are consistent with the added measurement m. Finally, at the end of UPDATETREE, node ⑪ and the replacement node of node ⑫ were reinserted into the queue $\mathbf{Q}$ of unlabeled nodes (preservation of best-first property; line 59 in Alg. 3) because they "survived" the pruning.

As to the complexity of the tree pruning performed by DynamicHS in this example, the overall number of reasoner-invoking function calls amounts to merely a single "easy" call of FINDMINCONFLICT (the executed QRC), whereas the (re)construction of a tree equivalent to DynamicHS's pruned tree, carried out in iteration 2 when adopting (the stateless) HS-Tree, requires three "hard" FINDMINCONFLICT calls (computations of conflicts $\langle 3\rangle$, $\langle 2, 4\rangle$, $\langle 1, 4\rangle$). □

Appendix C. Analysis of Advanced Techniques

As the performance comparison between DynamicHS and HS-Tree at the end of Example 5 already suggested, there are two major sources for the runtime savings obtained by means of DynamicHS:

(A1) fewer expensive reasoning operations and *(A2)* saved effort for tree (re)construction. The price to pay for these reductions is *(A3)* the storage of the existing hitting set tree (including duplicate nodes) throughout the diagnosis session and *(A4)* the execution of regular tree pruning actions. In the following, we examine these four aspects more closely based on our experiment results. Appendix C.1 addresses aspect *(A1)*, Appendix C.2 analyzes aspects *(A2)* and *(A3)*, and Appendix C.3 focuses on aspect *(A4)*.

C.1 Avoidance of Expensive Reasoning and Efficient Redundancy Checking

Fig. II summarizes several statistics that illustrate aspect *(A1)* in more detail. The reduction of costly reasoning operations ("hard" FINDMINCONFLICT calls, cf. Appendix B.3) is achieved by trading them against more efficient reasoning operations ("easy" FINDMINCONFLICT calls, cf. Appendix B.3). The red circles in the figure, which show how much harder an average "hard" call is than an average "easy" one in each diagnosis scenario, attest that this strategy of swapping "hard" for "easy" calls is indeed plausible, as the former are a median of 19 times and up to 120 times as time-consuming as the latter.

The green bars in Fig. II reveal that significant savings in terms of "hard" calls are indeed consistently generated by DynamicHS. More specifically, in more than 98 % of the diagnosis scenarios, the median relative savings wrt. "hard" FINDMINCONFLICT calls in comparison to HS-Tree are higher than 30 %; over all scenarios, median and maximal savings amount to 74 % and 99 %, respectively.

Rather unsurprisingly, these savings wrt. the number of "hard" FINDMINCONFLICT calls translate to a similarly substantial reduction of runtime spent for reasoning operations (orange bars in Fig. II) manifested by DynamicHS. Savings in terms of the runtime dedicated to reasoning are achieved in 97 % of the diagnosis scenarios and amount to median and maximal values of 57 % and 78 %, respectively. The main reason why the diminution in terms reasoning time is most times lower than the decrease of "hard" reasoning operations is that DynamicHS, as opposed to HS-Tree, makes use of "easy" reasoner calls to compensate for these saved more expensive calls. That is, these "easy" calls account to a large extent for the difference between green and orange bars. As evident in some scenarios, e.g., for the knowledge base fal-cs, however, it is also possible that savings in reasoning time top savings in terms of "hard" FINDMINCONFLICT calls. This can happen, e.g., when DynamicHS saves many "medium" FINDMINCONFLICT calls against HS-Tree (cf. Example 2 and Tab. I) in addition to its savings wrt. "hard" ones.

Last but not least, also the used efficient redundancy checking technique incorporated in DynamicHS contributes to the achieved runtime savings. In fact, the adoption of the QRC (quick redundancy check, cf. Appendix B.1) attempts to minimize the ("easy") reasoning operations necessary to decide the redundancy of a particular node (wrt. a particular given minimal conflict). The white boxes in Fig. II indicate that this strategy is very powerful in that the QRC detects redundancy almost always when redundancy is actually given, thus allowing the alorithm to skip the more expensive CRC (complete redundancy check, cf. Appendix B.1). One could say that the QRC appears to be "almost complete" in our experiments. In numbers, we observe that the QRC detected *all* redundancies in 75 %, at least nine of ten redundancies in 89 %, and at least eight of ten redundancies in 99 % of the scenarios.

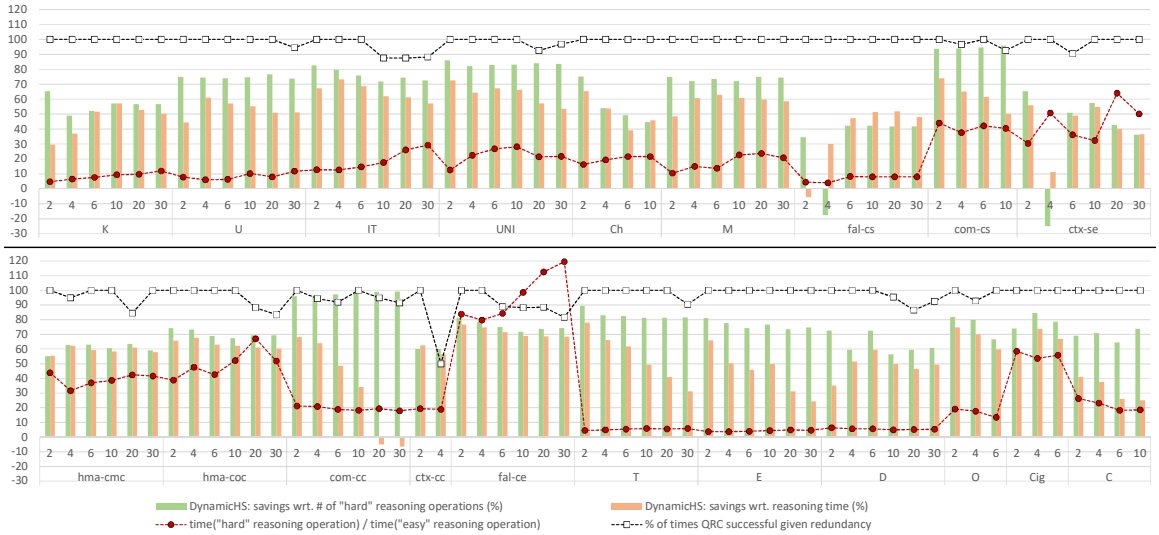


Figure II: Analysis of DynamicHS wrt. avoidance of reasoning and efficient redundancy checking: x-axis shows faulty ontologies $\mathcal{K}$ from Tab. 3 and number ld of leading diagnoses computed per call of hitting set algorithm (iteration of while-loop in Alg. 1). The plot shows data for the heuristic ENT. All values depicted are medians over the 20 sequential diagnosis sessions executed per diagnosis scenario. See Appendix B.3 for a definition of "hard" and "easy" reasoning operations; "QRC" refers to quick redundancy check, see Appendix B.1; "savings" refers to the savings over HS-Tree.

C.2 Statefulness (DynamicHS) vs. Statelessness (HS-Tree)

Let us now consider Fig. III, which allows to investigate aspects *(A2)* and *(A3)*. The figure depicts, per diagnosis scenario, the median factors how many more nodes HS-Tree had to process than DynamicHS (grey bars) and how many more nodes DynamicHS needed to store than HS-Tree (yellow bars). The fact that the majority of all bars attains values greater than 1 demonstrates that DynamicHS—expectedly—tends to trade less time for more space. It keeps the produced search tree in memory and utilizes the information contained in this tree to allow for more efficient diagnosis computation in the next sequential diagnosis iteration. We can identify from the figure that in two thirds of the diagnosis scenarios, the trade-off achieved by DynamicHS is favorable in the sense that the factor of more memory used by DynamicHS is less than the factor of more nodes processed by HS-Tree (yellow bars smaller than grey ones).

Moreover, Fig. III (yellow bars) reveals that DynamicHS required less memory than HS-Tree in 14 % of the scenarios, exhibited less than 25 % memory overhead in 35 % and less than 50 % overhead in 52 % of the scenarios, and consumed less than twice the memory of HS-Tree in 82 % of the diagnosis scenarios. Circumstances where DynamicHS can require even less memory than HS-Tree are when few or no duplicate nodes exist (e.g., when minimal conflicts are mostly disjoint), when DynamicHS's hitting set tree after tree updates is (largely) equal to the one produced by HS-Tree (cf. Appendix B.2), and when DynamicHS happens to compute or select for reuse more "favorable" conflicts than HS-Tree in the course of node labeling.[14] In less than 10 % of the scenarios, however, the memory overhead shown by DynamicHS was substantial, reaching values of 4 or more times the amount of memory consumed by HS-Tree. Hence, although not observed in our evaluation where either both or none of the algorithms ran out of memory, there may be cases where

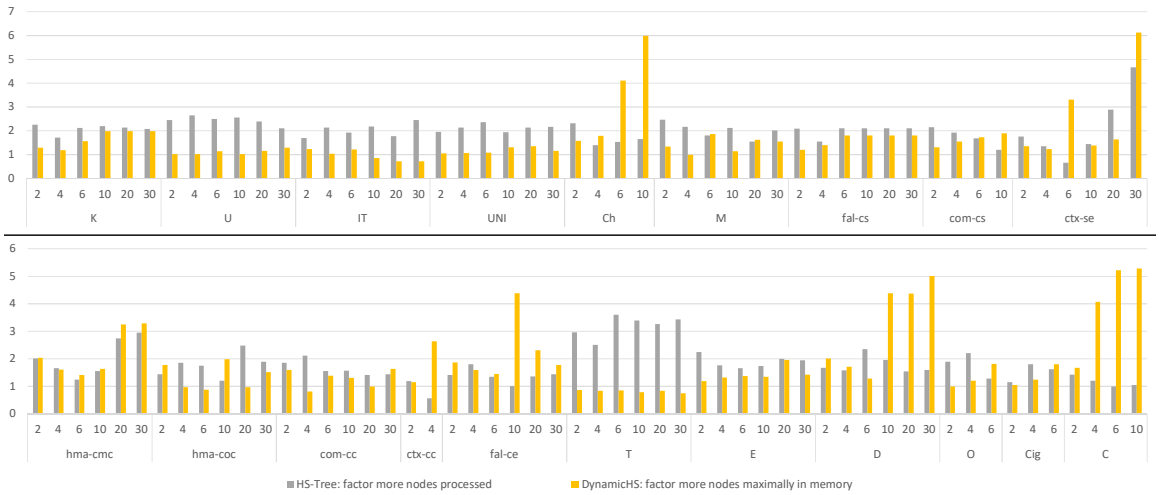


Figure III: Comparison of stateful (DynamicHS) versus stateless (HS-Tree) hitting set tree management in terms of the nodes processed (time efficiency) and the nodes maximally in memory (space efficiency): x-axis shows faulty ontologies $\mathcal{K}$ from Tab. 3 and number ld of leading diagnoses computed per call of hitting set algorithm (iteration of while-loop in Alg. 1). The plot shows data for the heuristic ENT. All values depicted are medians over the 20 sequential diagnosis sessions executed per diagnosis scenario.

DynamicHS is not applicable due to too little available memory while HS-Tree is. In general, this can be remedied by adding a mechanism to DynamicHS which simply discards the entire stored hitting set tree and starts over from scratch building a new tree whenever a certain fraction of the available memory has been exhausted. In other words, the stateful (DynamicHS) strategy can be flexibly and straightforwardly switched to a stateless (HS-Tree) one.

Fig. III (grey bars) also shows the benefit of the stateful strategy pursued by DynamicHS. That is, the overhead in terms of the time expended for tree (re)construction incurred when using HS-Tree instead of DynamicHS is significant in the majority of scenarios. In numbers, HS-Tree had to process at least 1.5 times as many nodes as DynamicHS in 78 % of the scenarios, at least twice as many in 42 %, and at least three times as many in 5 % of the diagnosis scenarios.

Finally, note that aspects *(A1)* and *(A2)*, i.e., DynamicHS's reduction of time for reasoning and its savings in terms of tree construction costs, are orthogonal in the following sense: Even if one of these aspects turns out to be unfavorable from the viewpoint of DynamicHS in comparison to HS-Tree, the other aspect is not affected by that in general. For instance, in our experiments we observed three scenarios (fal-cs, 2; com-cc, 20; com-cc, 30) where DynamicHS actually required slightly more time for reasoning than HS-Tree (cf. orange bars in Fig. II), i.e., aspect *(A1)* was unfavorable for DynamicHS. Since DynamicHS however did save node processing time in these cases, i.e., aspect *(A2)* was favorable, the overall computation time was still lower for DynamicHS. Similarly, there are cases (cf., e.g., ctx-cc, 4) where aspect *(A2)* is unfavorable while aspect *(A1)* is favorable, again resulting in overall time savings of DynamicHS.

C.3 Duplicates and Pruning

Fig. IV illuminates aspect *(A4)*. It displays the median fraction of the memory consumed by DynamicHS that is used for the storage of duplicate nodes per diagnosis scenario (blue bars). We can

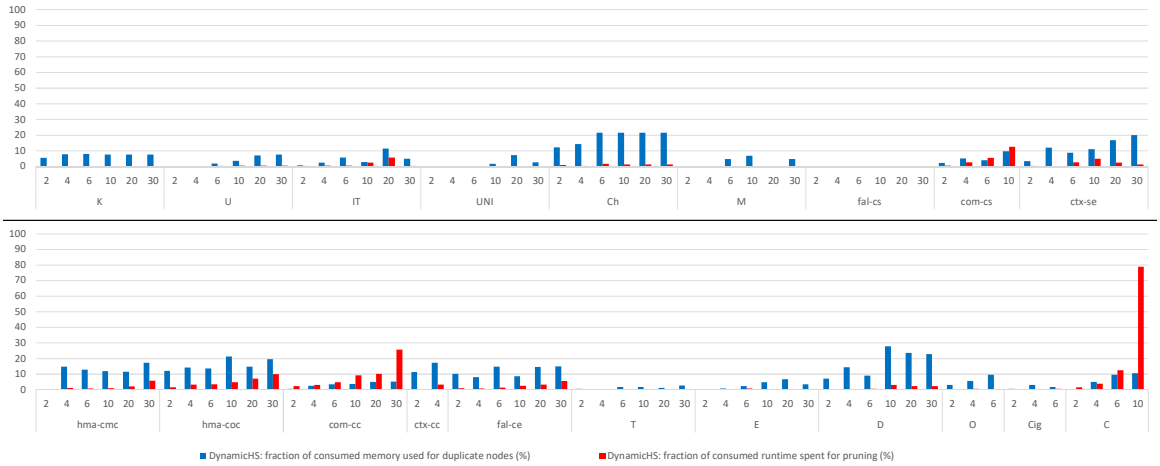


Figure IV: Analysis of DynamicHS wrt. space-efficiency of duplicate storage and time-efficiency of pruning actions: x-axis shows faulty ontologies $\mathcal{K}$ from Tab. 3 and number ld of leading diagnoses computed per call of hitting set algorithm (iteration of while-loop in Alg. 1). The plot shows data for the heuristic ENT. All values depicted are medians over the 20 sequential diagnosis sessions executed per diagnosis scenario.

see that duplicate nodes account for less than 10 % of the used memory in 69 % of the diagnosis scenarios, for less than 20 % in 92 %, and for less than 30 % in 100 % of the scenarios. In other words, the memory overhead caused by duplicate nodes does not exceed 12 % / 25 % / 39 % in 69 % / 92 % / 100 % of the scenarios. Consequently, the additional memory consumption attributable to duplicates stays within acceptable bounds in all cases. This can be due to not too many existing duplicates (e.g., if minimal conflicts tend to be disjoint) or due to DynamicHS's memory-efficient storage of duplicates (cf. Appendix B.4). The latter requires duplicates to be reconstructed on demand in the course of pruning and associated node replacement actions (cf. Appendix B.4 and Example 3). The relative computational expense of these pruning actions in comparison to the overall computation time of DynamicHS is described by the red bars in Fig. IV. These tell that the relative computation time spent for pruning is negligible (less than 1 %) in 61 % of the diagnosis scenarios, marginal (less than 5 %) in 89 %, and small (less than 10 %) in 95 % of the scenarios. This overall fairly low overhead for pruning, which already includes the time for reconstructing duplicate nodes, testifies that DynamicHS's duplicate storage and reconstruction as well as its pruning techniques that completely dispense with costly reasoner calls are reasonable strategies for dealing with the statefulness of the hitting set tree.

Only in two cases, the time for pruning exceeds 20 % of DynamicHS's total computation time; in one of these cases (C, 10) it is very high, reaching almost 80 %, a possible sign that significant portions of the tree have become out-of-date through lazy updating (cf. Appendix B.2). Note, however, that DynamicHS overall still saves time compared to HS-Tree in this scenario (cf. Fig. 5) since the time spent later for more intensive pruning is counterbalanced with time saved earlier by skipping pruning actions (lazy updating).

References

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (Eds.). (2007). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Annual ACM Symp. on Theory of Computing*.

Gonçalves, R. S., Parsia, B., & Sattler, U. (2012). Performance heterogeneity and approximate reasoning in description logic ontologies. In *Int'l Semantic Web Conf. (ISWC)*.

Horridge, M., Parsia, B., & Sattler, U. (2012). Extracting justifications from BioPortal ontologies. In *Int'l Semantic Web Conf. (ISWC)*.

Junker, U. (2004). QuickXPlain: Preferred Explanations and Relaxations for Over-Constrained Problems. In *AAAI Conf. on Artificial Intelligence (AAAI)*.

Kang, Y.-B., Li, Y.-F., & Krishnaswamy, S. (2012). Predicting reasoning performance using ontology metrics. In *Int'l Semantic Web Conf. (ISWC)*.

Karlsson, D., Nyström, M., & Cornet, R. (2014). Does SNOMED CT post-coordination scale? *Studies in Health Technology and Informatics*, *205*, 1048–1052.

Marques-Silva, J., Janota, M., & Belov, A. (2013). Minimal sets over monotone predicates in boolean formulae. In *Int'l Conf. on Computer Aided Verification*.

Rodler, P. (2015). *Interactive Debugging of Knowledge Bases*. Ph.D. thesis, Alpen-Adria Universität Klagenfurt. arXiv:1605.05950v1

Rodler, P. (2022). A formal proof and simple explanation of the QuickXplain algorithm. *Artificial Intelligence Review (https://doi.org/10.1007/s10462-022-10149-w)*.

Shchekotykhin, K., Friedrich, G., Fleiss, P., & Rodler, P. (2012). Interactive Ontology Debugging: Two Query Strategies for Efficient Fault Localization. *Web Semantics: Science, Services and Agents on the World Wide Web*, *12-13*, 88–103.

Shchekotykhin, K., Jannach, D., & Schmitz, T. (2015). MergeXPlain: Fast computation of multiple conflicts for diagnosis. In *Int'l Joint Conf. on Artificial Intelligence (IJCAI)*.

Notes

1. For additional details regarding the proof argumentation, we refer the reader to (Rodler, 2015, Sec. 12.4).
2. Please note that the original DPI considered by the first call of DynamicHS during a sequential diagnosis session is referred to as dpi_0 (cf. line 6 in Alg. 1). Hence, the DPI relevant to the k-th call of DynamicHS is denoted by dpi_{k-1}.
3. For a collection of sets Z, we denote by U_Z the union of all sets in Z.
4. Further evidence that larger knowledge bases lead to worse reasoning performance is given in (Kang, Li, & Krishnaswamy, 2012; Karlsson, Nyström, & Cornet, 2014).
5. When we say that FINDMINCONFLICT gets a set S as an input, we always mean by S the *first* argument of the 4-tuple (DPI) passed to the function as an argument.
6. Recall from Property 1.3 that minimal conflicts can only become smaller throughout a sequential diagnosis session, i.e., there cannot be any minimal conflict whose size exceeds $|\mathcal{C}_{\max}|$.

7. Cf. Footnote 6 above.

8. It is important to note that the used *intuitive* terminology "hard", "medium" and "easy" is to be understood *by tendency*, but does not allow *general* conclusions about the relative or absolute hardness of the respective FINDMINCONFLICT calls. That is, e.g., an "easy" call might not be fast or easy at all. Or a "medium" call might be faster than an "easy" one, e.g., because the latter operates on a set of logical sentences that is particularly hard to reason with (cf. (Gonçalves et al., 2012)). However, as we verified in our experimental evaluation (see Sec. 5), the used terminology does largely reflect the actual relative computation times of the FINDMINCONFLICT calls in our considered dataset. More precisely, on average "hard" calls turned out to be *always* (and at least four times and up to more than 100 times) more time-intensive than "medium" and "easy" calls; and, "easy" calls terminated faster than "medium" ones in 77 % of the studied cases.

9. Notation: Given a list n, $\mathsf{n}[k..l]$ refers to the sublist including all elements from the k-th (included) until the l-th (included).

10. Note the recursive character of this definition. That is, all combinations of explicit and already reconstructed nodes are possible. E.g., a reconstructed node n_6 can be the result of combining two explicit nodes n_1 and n_2 to reconstruct a node n_3, which in turn is combined with some explicit node n_4, which in turn is combined with a reconstructed node n_5.

11. Formally, $\mathbf{Q}^*_{dup}$ can be defined as the fixpoint S^* of the sequence of sets $S_0, S_1, \ldots$ resulting from the iterative application of the $Comb$ function starting from $S_0 := \mathbf{Q}_{dup}$ where $S_{i+1} = Comb(S_i)$ and $Comb$ is defined as $Comb(S) = S \cup \{\mathsf{n}_{i,j} \mid \mathsf{n}_{i,j}$ is the result as per *(D2)* of combining two nodes $\mathsf{n}_i, \mathsf{n}_j \in S$ which meet *(D1)*$\}$.

12. Recall from Remark (R2) that $|\mathsf{n}_i| \leq |\mathsf{n}_j|$. If $|\mathsf{n}_i| < |\mathsf{n}_j|$, then a processing of nodes in order of ascending node cardinality guarantees that all still available (non-pruned) nodes n_i are already verified non-redundant when some n_j might need to be replaced. If, on the other hand, $|\mathsf{n}_i| = |\mathsf{n}_j|$, then there are two cases: n_i is processed prior to n_j, or the opposite holds. In the former case, n_i (unless pruned) must already be verified non-redundant when n_j is considered. In the latter case, n_i is not available as a replacement node at the time n_j is addressed, but n_i itself will be processed later. Thus, if n_j is pruned and n_i non-redundant, then the latter will remain in $\mathbf{Q}_{dup}$ which means that n_i has essentially replaced n_j.

13. Note, we just mention the first element $\mathcal{K}$ of the tuple $\langle \mathcal{K}, \mathcal{B}, P, N \rangle$ passed to FINDCONFLICT and write "..." for the remaining ones for simplicity and brevity. The reason is that we did not discuss the specific DPI underlying this example and that $\mathcal{K}$ (set from which a minimal conflict is to be computed) is sufficient to understand the discussed points.

14. The order in which conflicts are computed and selected for node labeling is neither controlled by HS-Tree nor by DynamicHS. If one of the algorithms happens to compute smaller minimal conflicts that are used to label nodes at the top of the hitting set tree and when the hitting set trees produced by both algorithms are not computed to their entirety (as is the case when ld diagnoses have been computed before the tree is complete), then the tree with the smaller conflicts at the top can be smaller than the one generated by the other algorithm.

Algorithm 3 DynamicHS

Input: tuple $\langle dpi, P', N', pr, ld, \mathbf{D}_{\checkmark}, \mathbf{D}_{\times}, \mathsf{state} \rangle$ comprising

- a DPI $dpi = \langle \mathcal{K}, \mathcal{B}, P, N \rangle$
- the hitherto acquired sets of positive (P') and negative (N') measurements
- a function pr assigning a fault probability to each element in $\mathcal{K}$
- the number ld of leading minimal diagnoses to be computed
- the set $\mathbf{D}_{\checkmark}$ of all elements of the set $\mathbf{D}_{calc}$ (returned by the previous DYNAMICHS run) which are minimal diagnoses wrt. $\langle \mathcal{K}, \mathcal{B}, P \cup P', N \cup N' \rangle$
- the set $\mathbf{D}_{\times}$ of all elements of the set $\mathbf{D}_{calc}$ (returned by the previous DYNAMICHS run) which are no diagnoses wrt. $\langle \mathcal{K}, \mathcal{B}, P \cup P', N \cup N' \rangle$
- $\mathsf{state} = \langle \mathbf{Q}, \mathbf{Q}_{dup}, \mathbf{D}_{\supset}, \mathbf{C}_{calc} \rangle$ where
  - $\mathbf{Q}$ is the current queue of unlabeled nodes,
  - $\mathbf{Q}_{dup}$ is the current queue of duplicate nodes,
  - $\mathbf{D}_{\supset}$ is the current set of computed non-minimal diagnoses,
  - $\mathbf{C}_{calc}$ is the current set of computed minimal conflict sets.

Output: tuple $\langle \mathbf{D}, \mathsf{state} \rangle$ where

- $\mathbf{D}$ is the set of the ld (if existent) most probable (as per pr) minimal diagnoses wrt. $\langle \mathcal{K}, \mathcal{B}, P \cup P', N \cup N' \rangle$
- state is as described above

procedure DYNAMICHS($dpi, P', N', pr, ld, \mathbf{D}_{\checkmark}, \mathbf{D}_{\times}, \mathsf{state}$)
  $\mathbf{D}_{calc} \leftarrow \emptyset$
  $\mathsf{state} \leftarrow$ UPDATETREE($dpi, P', N', pr, \mathbf{D}_{\checkmark}, \mathbf{D}_{\times}, \mathsf{state}$)
  while $\mathbf{Q} \neq [\,] \wedge (\ |\mathbf{D}_{calc}| < ld\ )$ **do**
    $\mathsf{node} \leftarrow$ GETANDDELETEFIRST($\mathbf{Q}$) ▷ node is processed
    if $\mathsf{node} \in \mathbf{D}_{\checkmark}$ **then** ▷ $\mathbf{D}_{\checkmark}$ includes only min...
      $L \leftarrow valid$ ▷ ...diags wrt. current DPI
    else
      $\langle L, \mathsf{state} \rangle \leftarrow$ DLABEL($dpi, P', N', pr, \mathsf{node}, \mathbf{D}_{calc}, \mathsf{state}$)
    if $L = valid$ **then**
      $\mathbf{D}_{calc} \leftarrow \mathbf{D}_{calc} \cup \{\mathsf{node}\}$ ▷ node is a min diag wrt. current DPI
    else if $L = nonmin$ **then**
      $\mathbf{D}_{\supset} \leftarrow \mathbf{D}_{\supset} \cup \{\mathsf{node}\}$ ▷ node is a non-min diag wrt. current DPI
    else
      for $e \in L$ **do** ▷ L is a min conflict wrt. current DPI
        $\mathsf{node}_e \leftarrow$ APPEND(node, e) ▷ node_e is generated
        $\mathsf{node}_e.\mathsf{cs} \leftarrow$ APPEND($\mathsf{node.cs}, L$)
        if $\mathsf{node}_e \in \mathbf{Q} \vee \mathsf{node}_e \in \mathbf{D}_{\supset}$ **then** ▷ node_e is (*set-equal*) duplicate of some node in $\mathbf{Q}$ or $\mathbf{D}_{\supset}$
          $\mathbf{Q}_{dup} \leftarrow$ INSERTSORTED($\mathsf{node}_e, \mathbf{Q}_{dup}, card, <$)
        else
          $\mathbf{Q} \leftarrow$ INSERTSORTED($\mathsf{node}_e, \mathbf{Q}, pr, >$)
  return $\langle \mathbf{D}_{calc}, \mathsf{state} \rangle$

procedure DLABEL($\langle \mathcal{K}, \mathcal{B}, P, N \rangle, P', N', pr, \mathsf{node}, \mathbf{D}_{calc}, \mathsf{state}$)
  for $\mathsf{nd} \in \mathbf{D}_{calc}$ **do**
    if $\mathsf{node} \supset \mathsf{nd}$ **then** ▷ node is a non-min diag
      return $\langle nonmin, \mathsf{state} \rangle$
  for $\mathcal{C} \in \mathbf{C}_{calc}$ **do** ▷ $\mathbf{C}_{calc}$ includes conflicts wrt. current DPI
    if $\mathcal{C} \cap \mathsf{node} = \emptyset$ **then** ▷ reuse (a subset of) $\mathcal{C}$ to label node
      $X \leftarrow$ FINDMINCONFLICT($\langle \mathcal{C}, \mathcal{B}, P \cup P', N \cup N' \rangle$)
      if $X = \mathcal{C}$ **then**
        return $\langle \mathcal{C}, \mathsf{state} \rangle$
      else ▷ $X \subset \mathcal{C}$
        $\langle \mathsf{state}, \mathbf{D}_{calc} \rangle \leftarrow$ PRUNE($X, \langle \mathsf{state}, \mathbf{D}_{calc} \rangle$)
        return $\langle X, \mathsf{state} \rangle$
  $L \leftarrow$ FINDMINCONFLICT($\langle \mathcal{K} \setminus \mathsf{node}, \mathcal{B}, P \cup P', N \cup N' \rangle$)
  if L = 'no conflict' **then** ▷ node is a diag
    return $\langle valid, \mathsf{state} \rangle$
  else ▷ L is a *new* min conflict ($\notin \mathbf{C}_{calc}$)
    $\mathbf{C}_{calc} \leftarrow \mathbf{C}_{calc} \cup \{L\}$
    return $\langle L, \mathsf{state} \rangle$

Algorithm 3 DynamicHS (continued)

41: **procedure** UPDATETREE($dpi, P', N', pr, \mathbf{D}_\checkmark, \mathbf{D}_\times, \mathsf{state}$)
42: **for** $\mathsf{nd} \in \mathbf{D}_\times$ **do** ▷ search for redundant nodes among invalidated diags
43: $\langle isRedundant, X\rangle \leftarrow$ REDUNDANT(nd, dpi) ▷ if nd is redundant ($isRedundant$ is $true$), then...
44: **if** $isRedundant$ **then** ▷ ...X is a witness of redundancy for nd
45: $\langle \mathsf{state}, \mathbf{D}_\checkmark, \mathbf{D}_\times\rangle \leftarrow$ PRUNE($X, \langle \mathsf{state}, \mathbf{D}_\checkmark, \mathbf{D}_\times\rangle$)
46: **for** $\mathsf{nd} \in \mathbf{D}_\times$ **do** ▷ add all (non-pruned) nodes in $\mathbf{D}_\times$ to $\mathbf{Q}$
47: $\mathbf{Q} \leftarrow$ INSERTSORTED($\mathsf{nd}, \mathbf{Q}, pr, >$)
48: $\mathbf{D}_\times \leftarrow \mathbf{D}_\times \setminus \{\mathsf{nd}\}$
49: **for** $\mathsf{nd} \in \mathbf{D}_\supset$ **do** ▷ add all (non-pruned) nodes in $\mathbf{D}_\supset$ to $\mathbf{Q}$, which...
50: $nonmin \leftarrow false$ ▷ ...are no longer supersets of any diag in $\mathbf{D}_\checkmark$
51: **for** $\mathsf{nd}' \in \mathbf{D}_\checkmark$ **do**
52: **if** $\mathsf{nd} \supset \mathsf{nd}'$ **then**
53: $nonmin \leftarrow true$
54: **break**
55: **if** $nonmin = false$ **then**
56: $\mathbf{Q} \leftarrow$ INSERTSORTED($\mathsf{nd}, \mathbf{Q}, pr, >$)
57: $\mathbf{D}_\supset \leftarrow \mathbf{D}_\supset \setminus \{\mathsf{nd}\}$
58: **for** $\mathcal{D} \in \mathbf{D}_\checkmark$ **do** ▷ add known min diags in $\mathbf{D}_\checkmark$ to $\mathbf{Q}$ to find diags...
59: $\mathbf{Q} \leftarrow$ INSERTSORTED($\mathcal{D}, \mathbf{Q}, pr, >$) ▷ ...in best-first order (as per pr)
60: **return** state

Algorithm 1 Sequential Diagnosis

Input: DPI $dpi_0 := \langle \mathcal{K}, \mathcal{B}, P, N\rangle$, probability measure pr (to compute diagnoses probabilities), number ld (≥ 2) of minimal diagnoses to be computed per iteration, heuristic heur for measurement selection, boolean $dynamic$ that governs which diagnosis computation algorithm is used (DynamicHS if true, HS-Tree otherwise)
Output: $\{\mathcal{D}\}$ where $\mathcal{D}$ is the final diagnosis after solving SD (Problem 1)

1: $P' \leftarrow \emptyset, N' \leftarrow \emptyset$ ▷ performed measurements
2: $\mathbf{D}_\checkmark \leftarrow \emptyset, \mathbf{D}_\times \leftarrow \emptyset$ ▷ variables describing state...
3: $\mathsf{state} \leftarrow \langle [\,[\,]\,], [\,], \emptyset, \emptyset\rangle$ ▷ ...of DynamicHS tree
4: **while** $true$ **do**
5: **if** $dynamic$ **then**
6: $\langle \mathbf{D}, \mathsf{state}\rangle \leftarrow$ DYNAMICHS($dpi_0, P', N', pr, ld, \mathbf{D}_\checkmark, \mathbf{D}_\times, \mathsf{state}$)
7: **else**
8: $\mathbf{D} \leftarrow$ HS-TREE(dpi_0, P', N', pr, ld)
9: **if** $|\mathbf{D}| = 1$ **then return** $\mathbf{D}$
10: $mp \leftarrow$ COMPUTEBESTMEASPOINT($\mathbf{D}, dpi_0, P', N', pr, \mathsf{heur}$)
11: $outcome \leftarrow$ PERFORMMEAS(mp) ▷ oracle inquiry (user interaction)
12: $\langle P', N'\rangle \leftarrow$ ADDMEAS($mp, outcome, P', N'$)
13: **if** $dynamic$ **then**
14: $\langle \mathbf{D}_\checkmark, \mathbf{D}_\times\rangle \leftarrow$ ASSIGNDIAGSOKNOK($\mathbf{D}, dpi_0, P', N'$)