From Conflicts to Diagnoses: An Empirical Evaluation of Minimal Hitting Set Algorithms

Ingo Pill[1], **Thomas Quaritsch**[1], **and Franz Wotawa**[1]

[1] *Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/II, Graz, 8010, Austria*
{*pill,quaritsch,wotawa*}*@ist.tugraz.at*

ABSTRACT

Using appropriate models, minimal hitting sets of a set of conflict sets can provide a sound foundation for diagnostic reasoning. Related diagnoses can explain encountered inconsistencies between expected and experienced behavior, so that a multitude of algorithms for computing such diagnoses have been developed. Motivated by a lack of a comparative study, in this paper, we evaluate a selection of relevant algorithms in the context of synthetic and real-world test scenarios.

1 INTRODUCTION

Conflict sets are of special interest in model-based diagnosis. Designing a model around special assumptions offers us a sound foundation for diagnosis: Approaches like the general diagnostic engine GDE (de Kleer and Williams, 1987) or Reiter's "Theory of Diagnosis from first principles" (Reiter, 1987; Greiner *et al.*, 1989) exploit the fact that conflicts on these assumptions represent manifestations of behavioral inconsistencies (in the form of unsatisfiable cores). As Reiter pointed out in his theoretical framework, minimal hitting sets of the set of conflict sets are (minimal) diagnoses that can explain inconsistencies between modeled and experienced behavior. Traditionally, the tokens of a diagnosis, that is the single assumptions, address whole components, e.g. (¬)AB(axle) assumes an axle to be operating normally/abnormally. The assumptions might however also aim at component parts, single aspects of a component, or in case of temporal scenarios also at the value of a signal "request line" at time-stamp five. Other applications of hitting set algorithms include, e.g., Boolean logic, where the algorithm can be used to derive the disjunctive normal form from a conjunctive one (or vice versa), and AI planning where hitting sets of landmark sets can be exploited (Bonet and Helmert, 2010).

Given that the general search space for diagnoses is exponential in the amount of viable tokens, there is a high motivation for computing diagnoses efficiently. While so far a lot of solutions for computing minimal hitting sets have been proposed and evaluated against one or the other competing approach, we are faced with a lack of more comparative studies. Thus, in this paper, we evaluate a selection of relevant algorithms in the context of several artificial and real-world scenarios. They were implemented in two popular high-level programming languages (Java and Python), in order to rule out issues related to using a specific language.

This document is organized as follows. After formally introducing the minimal hitting set problem in Section 2, we briefly depict the underlying ideas of the evaluated approaches and state our first evaluation hypothesis in Section 3. In Section 4, we introduce our test platform as well as describe our test scenarios and state our further two hypotheses. Our experiments' results and their discussion can be found in Section 4.2, while our conclusions are drawn in Section 5.

2 THE MINIMAL HITTING SET PROBLEM

We start with a brief formal introduction to the hitting set problem, complemented by a discussion of minimality in this context. Similar to Reiter (Reiter, 1987), we formally define a hitting set as follows:

Definition 1 (Hitting set)**.** Given a set of sets *SCS*, a set $h \subseteq \bigcup_{CS_i \in SCS} CS_i$ is a hitting set for *SCS*, iff for any set $CS_i \in SCS$ the intersection with h is non-empty, i.e. $h \cap CS_i \neq \emptyset$.

The following example illustrates Definition 1.

Example 1. Let *SCS* be the set $\{\{1,2,5\}, \{2,3,4\}, \{1,2\}, \{2,4,5\}\}$. Then, the set $\{2\}$ is a hitting set of *SCS*, and so are $\{1,2\}$ as well as $\{1,4\}$.

As the set $\{2\}$ is a subset of another hitting set $\{1,2\}$, this raises the question about minimality in this context. Obviously, $\{2\}$ is the smallest possible hitting set in our example, while $\{1,2\}$ is a superset, and $\{1,4\}$ is larger with respect to cardinality as well. Considering our example, we thus might define minimal hitting sets in two distinct ways.

Definition 2 (Subset-minimal hitting set)**.** Given a hitting set h for a set of sets *SCS*, h is said to be minimal with respect to subset inclusion, iff there exists no other hitting set h' for *SCS*, such that $h' \subseteq h$.

The subset-minimal hitting sets for Example 1 would be $\{2\}$, $\{1,4\}$ and $\{1,3,5\}$. Alternatively, we can define minimality via set cardinality.

Definition 3 (Cardinality-minimal hitting set)**.** Given a hitting set h for a set of sets *SCS*, h is considered to be minimal with respect to cardinality, iff there exists no other hitting set h' for *SCS*, such that $|h'| < |h|$.

There is a single cardinality-minimal hitting set for Example 1, namely $\{2\}$.

In most applications, like model-based diagnosis, one might actually apply a combination of both definitions. While we assume from here on subset minimality when using the term minimal hitting set, we might also not want to compute all the minimal diagnoses. That is, for performance reasons, focus the search on single-, double-, or triple fault diagnoses.

Besides minimality, there are also other "optimality" criteria actually used to evaluate and order hitting sets. For example, in model-based diagnosis probabilities are of specific interest. In (de Kleer and Williams, 1987) the probability of a diagnosis Δ is defined as $\prod_{x\in\Delta} p_F(x) \cdot \prod_{x\in COMP\setminus\Delta}(1-p_F(x))$, assuming stochastic independence of faults, with p_F defining the fault probability, and *COMP* the set of components.

In the context of this paper we will focus on subset minimality, while cardinality is considered when limiting the focus to a specific maximum diagnosis size for performance reasons.

3 ALGORITHMS FOR COMPUTING MINIMAL HITTING SETS

Over the years, researchers have been proposing a multitude of approaches for computing minimal hitting sets. In this section, we will briefly introduce a selection of relevant algorithms that are capable of operating on a pre-computed set of conflict sets *SCS*. Regarding the outcome of their evaluation, one would expect one of the newer approaches (e.g. Staccato was proposed in 2009) to offer superior performance, specifically when compared to much older ones like the HS-DAG published in 1989.

Hypothesis 1. There is an algorithm with superior performance, and it is among the newer approaches.

The following characterizations of the algorithms shall only depict each ones' basic ideas. For details on the specific algorithms, we refer the reader to the corresponding publications.

3.1 The traditional HS-DAG

The basic idea of Reiter's approach (Reiter, 1987) is to calculate all minimal hitting sets guided by the structure of a tree. Each node is either a leaf representing a solution, or is labeled with a conflict set. The tree's edges are labeled by some component of the source's conflict set. Starting with a first conflict set for the root node, and constructing an outgoing edge for each of its members, the tree is iteratively extended in a breadth-first manner by checking for new labels for the destination nodes of open edges. If the edge labels on a node's path from the root represent a hitting set (that is they are consistent), the node is labeled as solution. Otherwise a new conflict set is derived, disjoint to this current set of edge labels. Several complementing mechanisms prune the tree, ensure the minimality of the final solutions, and allow the algorithm to operate also with non-minimal conflict sets. While Reiter's publication had some minor but serious flaws, Greiner and colleagues presented in (Greiner *et al.*, 1989) a corrected version, the HS-DAG, that uses a directed-acyclic-graph (reusing nodes) instead of a tree.

3.2 A variant: HST

As the pruning applied during the creation of the HS-DAG requires a lot of resources (e.g. for subset checks), Wotawa presented in (Wotawa, 2001) a competing approach, called HST, that tries to avoid the construction of unnecessary nodes in the first place. While the approach also uses a tree, it implements an idea used for subset computation. That is, it orders the elements in *COMP*, and, defined by the components encountered in the tree so far (with a special focus on the current path), the algorithm limits the outgoing edges of a node to a range in *COMP*. Thus some edges are prohibited from being constructed, but this happens iff it is ensured that the same component combinations would be constructed in other branches if they are of interest. This avoids a lot of subset checks during pruning, and Wotawa expected a serious impact on overall performance.

3.3 The BHS-Tree and the Boolean Approach

In (Lin and Jiang, 2003), the authors proposed a Binary Hitting Set Tree for computing hitting sets. Based on a special way to split (and alter) the sets, they start with the set of conflict sets *SCS* as root node, and construct a tree that splits and analyzes the given *SCS*. With a further tree traversal starting at the tree's leaves, the hitting sets can be constructed, but have to be minimized by a special function μ to ensure their minimality. The authors also showed that the idea of their tree can be implemented in a simple Boolean algorithm using bits for components. As this algorithm can work iteratively on a data structure extending to the solutions, their proposal seems very interesting in terms of performance. However, unlike the three previously introduced tree-based algorithms, it requires *SCS* to be known from the start and thus cannot work on-the-fly, i.e. with growing *SCS*.

Please note that the BHS-Tree algorithm was implemented only in Java, while for Python we implemented two versions of the Boolean approach: A standard iterative solution and a recursive one that thus implicitly constructs a tree-like structure. This way we could investigate whether the performance of the Boolean algorithm results from using an iterative approach instead of a tree-based one.

3.4 The power of the matrix: Staccato

In (Abreu and van Gemund, 2009), the authors presented an approach that represents *SCS* as a binary matrix A, where a_{ij} is true iff CS_i contains component j. Their Staccato algorithm then ranks the components using a mechanism borrowed from spectrum-based fault localization in order to identify components (and their $CS_i s$) to be removed from the matrix. The results of each step then undergo extensive subset checks, such that the minimality of solutions

is ensured. While Staccato can operate also in an approximative manner, distinctive parameters enable the computation of all solutions, so that the algorithm can be compared to the other approaches. Please note that also Staccato requires *SCS* to be known from the start.

4 EXPERIMENTS

In this section, we describe our test scenarios and report the obtained results. All the numbers presented were obtained with a 2011 generation MacBook Pro featuring an Intel Core i5 CPU with 2.3 GHz, 4GB of RAM and a solid-state drive, while running an up-to-date version of Mac OS X 10.6 .

In order to rule out any bias resulting from using a specific language, we implemented most algorithms in two popular high-level languages, namely Java (1.6.0) and Python (CPython 2.7.1). This way, we accommodate our second evaluation hypothesis:

Hypothesis 2. The performance relation between the algorithms is independent from the programming language/implementation assuming fair optimization.

Our test suite aggregates examples from seven test scenarios. Three artificial ones allow us to succinctly define and scale the features of the conflict sets. These synthetic test scenarios use positive integer numbers as components. That is, for m given components, the set of components *COMP* is the set of integers between 1 and m, and a conflict set is a set $CS \subseteq COMP$. All three scenarios offer us to parameterize the size of the set of conflict sets *SCS*. The motivation behind the artificial examples is to evaluate the algorithms' raw performance for succinctly defined conflict sets.

Our four real world scenarios are based on the ISCAS'85 benchmark suite[1] which contains net lists of ten combinatorial circuits with sizes ranging from 160 to 3512 gates, and shall show how real life example performance correlates with that for artificial settings.

Hypothesis 3. Performance characteristics for real-world diagnostics equal those for artificial scenarios.

4.1 Test Data

Test Scenario 1: Completely disjoint conflict sets. The SCS-Generator for this scenario distributes m given components over n disjoint conflict sets, where the difference in size between any $CS_i \in SCS$ is one at most. More formally, we have that

- $|SCS| = n$, where n is a given parameter
- $c_i \in CS_i \in SCS \rightarrow c_i \in COMP$
- $\forall c_i \in COMP : \exists CS_j \in SCS : c_i \in CS_j$
- $\forall CS_i, CS_j \in SCS : \big||CS_i| - |CS_j|\big| \leq 1$

Example 2. *GenerateInputTS1*$(10, 3) = \{\{1, 4, 7, 10\}, \{2, 5, 8\}, \{3, 6, 9\}\}$.

Test Scenario 2: Overlapping conflict sets. The generator *GenerateInputTS2*(m, n) derives a set of n conflict sets that share a common part $CP \subseteq COMP$ of size $m - n$. All those components in $COMP \setminus CP$ appear in any $CS_i \in SCS$ with a probability of 50 percent. Note that thus the conflict sets might also share components not in *CP*. More formally, we have that

- $|SCS| = n$, where n is a given parameter
- $c_i \in CS_i \in SCS \rightarrow c_i \in COMP$
- $\forall CS_i \in SCS : CP \subseteq CS_i$

Example 3. *GenerateInputTS2*$(4, 2) = \big\{\big\{1, \underline{3, 4}\big\}, \big\{2, \underline{3, 4}\big\}\big\}$ (the common part *CP* is underlined).

SCS-Generator 1: This generator produces n disjoint conflict sets using m components.

```
Requires: m ≥ n
 1: procedure GenerateInputTS1(m,n)
 2:     SCS ← ∅
 3:     COMP ← {1, 2, ..., m}
 4:     for i ← 0 to n do
 5:         SCS ← SCS ∪ {{pop(COMP)}}
 6:     while |COMP| > 0 do
 7:         for CS in SCS do
 8:             if |COMP| > 0 then
 9:                 CS ← CS ∪ {pop(COMP)}
10:     return SCS
```

SCS-Generator 2: This generator produces conflict sets that share a common part *CP* of size $m - n$.

```
Requires: m ≥ n
 1: procedure GenerateInputTS2(m,n)
 2:     SCS ← ∅
 3:     COMP ← {1, 2, ..., m}
 4:     CP ← ∅
 5:     for i ← 1 to m − n do
 6:         CP ← CP ∪ {pop(COMP)}
 7:     while |SCS| < n do
 8:         CS ← CP
 9:         for C in COMP do
10:             if rand(0, 1) > 0.5 then
11:                 CS ← CS ∪ {C}
12:         SCS ← SCS ∪ {CS}
13:     return SCS
```

Test Scenario 3: Completely random conflict sets. For Test Scenario 3, the generator derives n conflict sets containing m components on an entirely random basis. That is, each of the m components appears in any of the n conflict sets with a probability of 50 percent. More formally, we have that

- $|SCS| = n$, where n is a given parameter
- $c_i \in CS_i \in SCS \rightarrow c_i \in COMP$

The corresponding generator is similar to lines 7 to 12 of SCS-Generator 2, but is omitted here due to space reasons.

Test Scenarios 4 to 7: The ISCAS benchmark suite. The ISCAS'85 benchmarks were first published at the International Symposium of Circuits and

[1] Available from `http://www.cbl.ncsu.edu:16080/benchmarks/ISCAS85/` (May 30, 2011)

Fault Injector: This procedure injects m independent faults into a circuit N with gates G.

Requires: $|N| \geq m$

procedure *InjectFaults*(N, m)
  loop
    In $\leftarrow$ *createRandomInputs*(N)
    *Out** $\leftarrow \emptyset$ ▷ Freezed outputs
    Out $\leftarrow$ *calcOutputs*(N, *In*, *Out**)
    $\Delta \leftarrow \emptyset$
    $G_{left} \leftarrow$ *Gates*(N) ▷ Gates (left) to try
    while $|\Delta| < m$ **and** $|G_{left}| > 0$ **do**
      $g \leftarrow$ *popRandom*(G_{left})
      $N' \leftarrow$ *mutate*(N, g)
      Out$'$ $\leftarrow$ *calcOutputs*(N', *In*, *Out**)
      $C \leftarrow \{i | \textit{Out}_i \neq \textit{Out}'_i\}$
      if $|C| > 0$ **then**
        *Out** $\leftarrow$ *Out** $\cup C$
        $N \leftarrow N'$
        $\Delta \leftarrow \Delta \cup g$
      if $|\Delta| = m$ **then**
        return (Δ, *In*, *Out*$'$)

Systems in 1985 and contains circuits such as interrupt controllers, modules for single-error-correction (SEC), double-error-detection (DED) and arithmetic logic units (ALU) (Hansen *et al.*, 1999).

To construct a diagnosis scenario, we translated the circuit's net list into a SAT problem P, and equipped every gate with a behavioral assumption encoding whether the gate operates correctly or not. P has the form

$$P = \bigwedge_{g_i \in G} \neg \mathrm{AB}(g_i) \Rightarrow \mathrm{out}_{g_i} := f_{g_i}(\mathrm{in}^1_{g_i}, \mathrm{in}^2_{g_i}, \ldots)$$

where G is the set of gates, and out_{g_i}, $\mathrm{in}^j_{g_i}$ and f_{g_i} are a gate g_i's output-/input lines and its boolean function.

To obtain a faulty observation (i. e., a valuation of input and output lines contradicting the expected system behavior), we purposefully injected faults by altering the logic function of some gates. The procedure *InjectFaults*(N, m) injects m faults into a net N, and returns the list of altered gates Δ. The gates are chosen such that their failing do not mask each other. This is achieved by freezing all output lines that changed after altering a gate, and allowing only the remaining outputs to flip when choosing the next gate. Nevertheless, there may still be diagnoses with a cardinality lower than m, and we verified whether the "original" diagnosis Δ was among the computed ones.

Our Test Scenarios 4 to 7 were constructed from the four ISCAS circuits depicted in Table 1.

	File	In	Out	Gates	Function
TS4	c499.isc	41	32	202	32-Bit SEC
TS5	c880.isc	60	26	383	8-Bit ALU
TS6	c1355.isc	41	32	546	32-Bit SEC
TS7	c1908.isc	33	25	880	16-Bit SEC/DED

Table 1: ISCAS circuit details for TS4 to TS7

4.2 Test Results

Figures 1 to 6 show the algorithms' run-times for our artificial scenarios, and aim at identifying the algorithms' raw performance. Please note that for those figures with run-times in a sub-second range, we report an average over 20 samples. This smoothes the graphs in a sense that tendencies become more obvious, while variance due to specific instances becomes smaller. Consequently, the performance comparison for single samples might be different for two close graphs. That is, for Figure 1(a) there are also instances where the iterative Boolean implementation is actually faster than the recursive one.

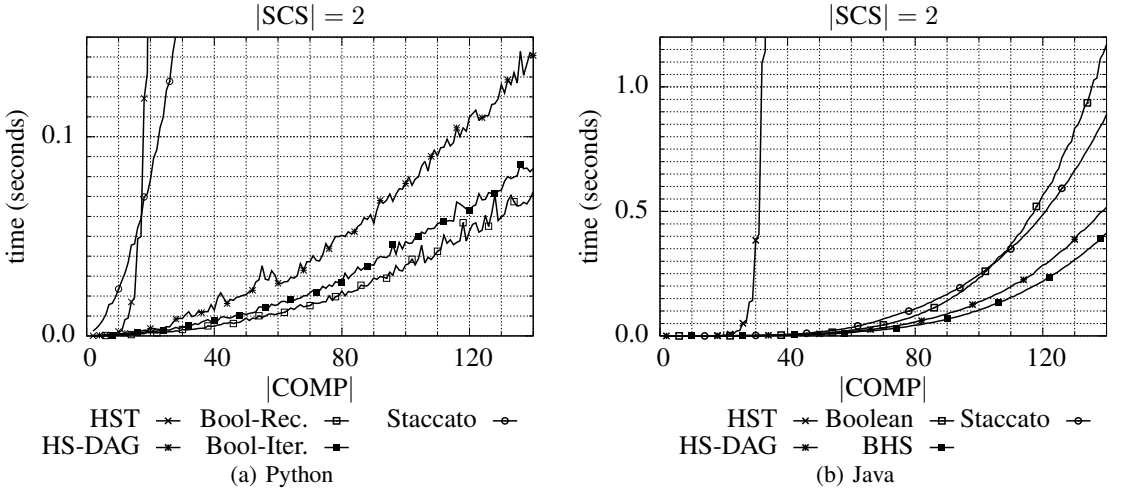
Figures 1 and 2 report the respective run-times for TS1 and a fixed $|SCS|$ but varying $|COMP|$. The graphs show that the algorithms' performance relations are inconsistent over the languages. For example, the Java version of Staccato performs well, while the Python version does not. In fact, our Python version performs so bad in all our tests that we will exclude it from further discussion. Over all other implementations, there seems to be a general advantage for Python as a language.

While HST does not perform well for TS1, it does for TS2 and TS3 (see Figures 3 to 6). For the latter test cases, HST is either on par with HS-DAG or outperforms it (sometimes even by a large margin). Thus HST cannot live up to Wotawa's expectation of outperforming HS-DAG entirely: Despite offering improvements in many cases, it seems sensitive to the completely disjoint conflict sets from TS1.

Offering good performance for TS1, the Boolean implementations show even better performance for TS2 and TS3, outperforming also their cousin BHS in almost every case. Only for TS1 and very small $|SCS|$, their tree-based cousin BHS showed better performance (see Figure 1(b)). Interestingly enough, we see that for the Boolean approaches the recursive implementation (maintaining an implicit tree) is sometimes superior to the iterative one. Thus, we assume that the performance of the Boolean approach does not originate in an iteratively refined data set, but in its specific strategy of identifying the common parts in the currently investigated conflict set parts. Substantiating our suspicion is the fact that identifying the "most" common part resulted in a general performance boost in our implementation.

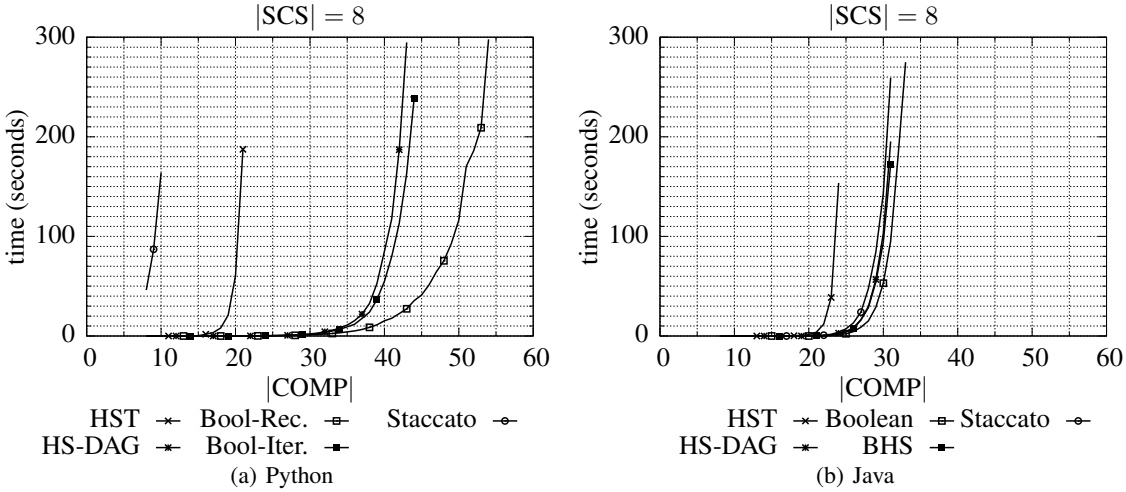
When trying to establish a ranking considering the artificial tests, the Boolean algorithm is a hot candidate for the first place. In Python its lead is almost undisputed (remember the occasional averaging over samples), and often a very comfortable one. Using Java, the Boolean approach was also either the best performing algorithm, or within scope of optimization. Thus, while in a strict sense Hypothesis 1 is refuted, considering average performance, the Boolean approach offers top performance. Please remember however that, unlike the tree-based algorithms, it cannot work on-the-fly and thus might not suit all applications.

Hypothesis 2 is strictly refuted: While we also optimized the Python implementation of Staccato, it is clearly among the worst performing implementations (sometimes by such a huge factor that we omitted it in the graph), while the Java version performs quite well.



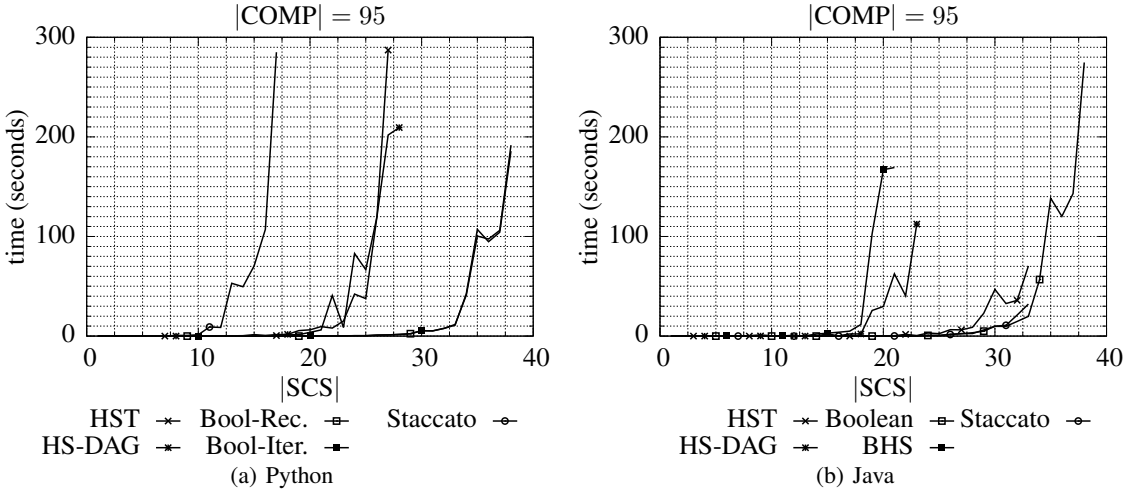
(a) Python (b) Java

Figure 1: Run-times for TS1.



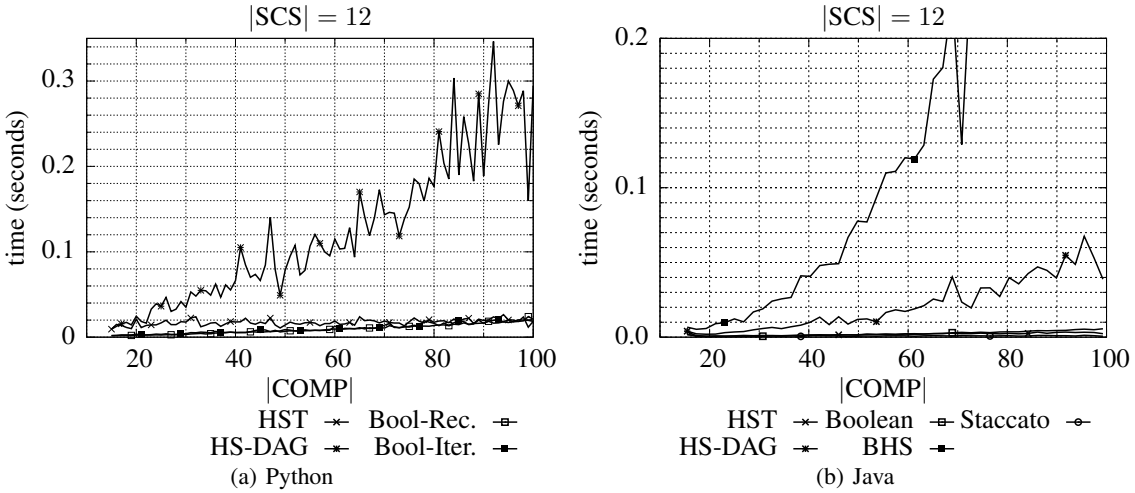
(a) Python (b) Java

Figure 2: Run-times for TS1.



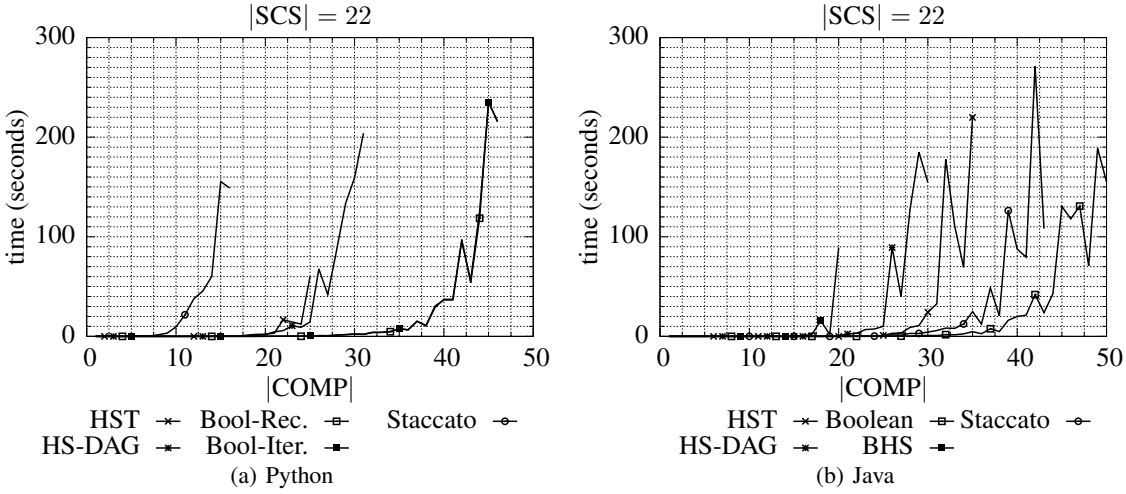
(a) Python (b) Java

Figure 3: Run-times for TS2.



(a) Python (b) Java

Figure 4: Run-times for TS2.



(a) Python (b) Java

Figure 5: Run-times for TS3.

Thus, as one might have assumed, the implementation and its language do seem to have a serious impact, a fact to be considered for prototyping and evaluating new algorithms.

Our third evaluation hypothesis motivated the following tests with real-world diagnostic scenarios, whose results are illustrated in Figures 7 to 9. Our first test runs showed that even for simple examples, it was not possible to compute all solutions within five minutes. We thus restricted the computation to single-, double-, and triple-fault diagnoses, as is usually done also in practice. The viability of this approach is evident in the fact that in case of TS5, even this restriction allowed maxima of 46/1397/12122 solutions for single-/double-/triple-faults. All Python implementations were adapted accordingly, while for Java we chose the HS-DAG implementation as reference. Figures 7 to 9 illustrate the raw algorithms' runtimes for precomputed *SCS*, averaged over ten samples. The single results' range is illustrated in the form of bars, while the labeling symbol represents the average value. Above the columns, we report numbers to be multiplied with the *normalized* values, in order to derive the actual run-times. Normalizing allowed us to use a common scale, while details are visible for both single and triple faults. Thick horizontal bars indicate the timeout of five minutes, that can be exceeded with long (unaborted) theorem-prover calls.

While we see a general advantage in favor of Python for most cases, it is also evident that the differences between the implementations are not that big as suggested by, e.g. TS3. This might result from the fact that with the restrictions the (in the Boolean case virtual) trees become not that deep, so that pruning and other optimizations cannot unfold their full potential. A confirmation for this suspicion would be the observation that specifically with increasing diagnosis size, the Boolean algorithms gain in relative performance. Interestingly enough, the performance of the old HS-DAG algorithm is quite attractive, and sometimes even sets the pace. These observations' discrepancy from the artificial case clearly contradicts Hypothesis 3.

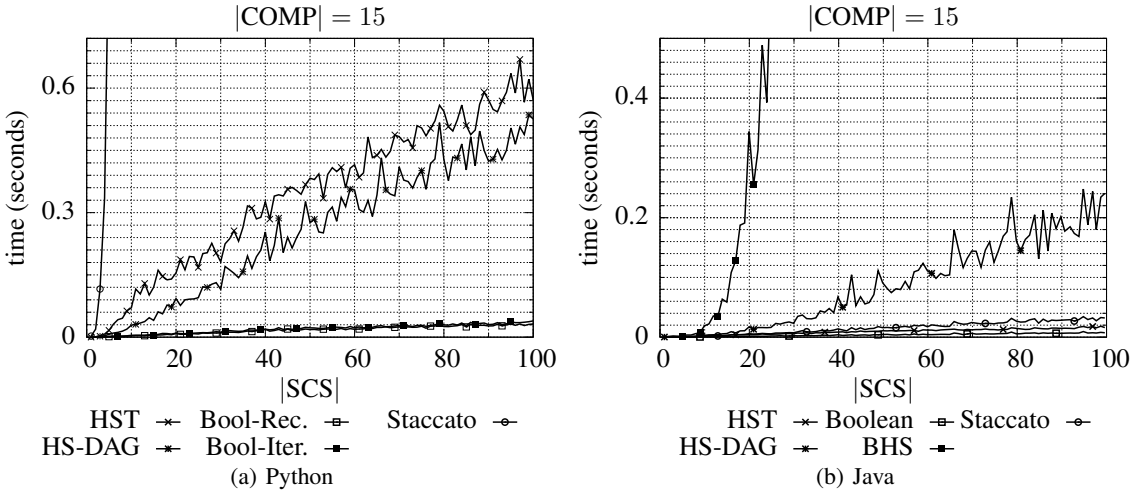


Figure 6: Run-times for TS3.

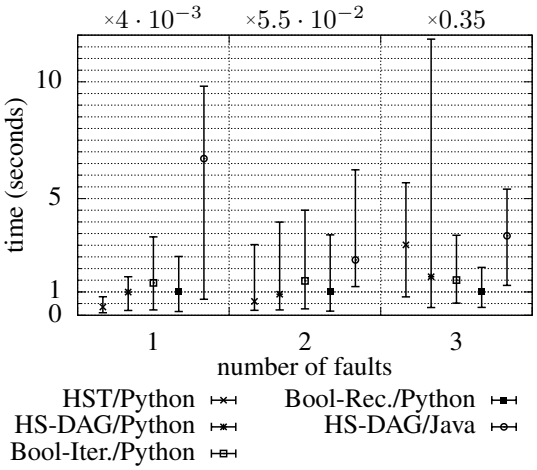


Figure 7: Run-times for TS 4.

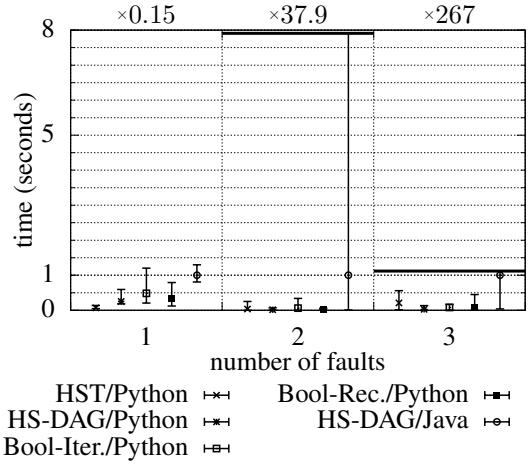


Figure 9: Run-times for TS 6.

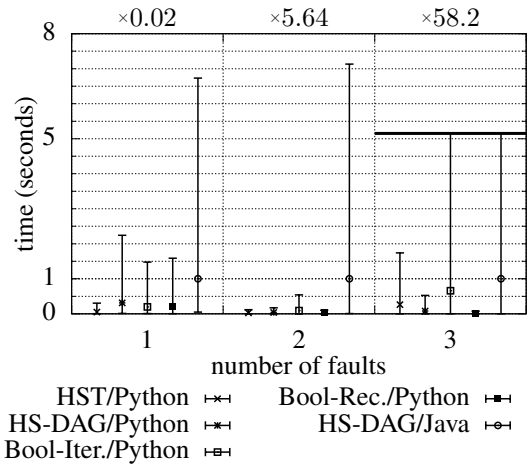


Figure 8: Run-times for TS 5.

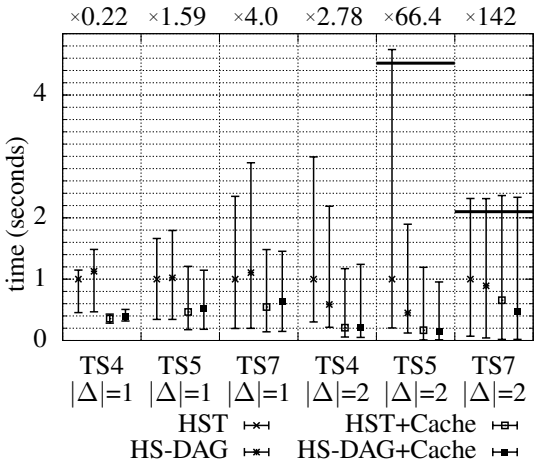


Figure 10: Impact of conflict set cache (Python).

	$\|\Delta\| = 1$			
	TS4	TS5	TS6	TS7
HS-DAG	17.3	31.4	41.1	59.7
HS-DAG+Cache	5.5	21.0	15.2	28.3
HST	17.3	31.4	41.1	59.7
HST+Cache	5.5	21.0	15.2	28.3
	$\|\Delta\| = 2$			
	TS4	TS5	TS6	TS7
HS-DAG	125.4	650.4	808.8	1892.3
HS-DAG+Cache	31.8	423.2	173.9	370.0
HST	220.3	1009.8	1899.6	2249.2
HST+Cache	31.9	423.1	173.6	478.9

Table 2: Number of TP-calls (avg. over 10 samples)

Last, but not least, we analyzed also the diagnosis problems' total computation time (see Figure 10) for on-the-fly capable HST and HS-DAG. That is, including the theorem prover calls for an on-the-fly computation of the conflict sets. For this purpose, we used the Python implementations of HS-DAG and HST, enabling and disabling a cache that allows previously computed conflict sets to be reused. In general, enabling the cache results in a significant advantage of at least a hundred percent (and often much more). Considering total time, the performance difference between the HS-DAG and HST becomes even narrower. While performance is not that different, the number of theorem prover calls (TP-calls) might differ significantly for $|\Delta| > 1$ (see Table 2), which suggests that advantages in the algorithm might be bought with more TP-calls and vice versa. In this context, it shall be noted that the TP-calls make up up to 99 percent of the total time.

5 CONCLUSIONS

In the course of the presented research, we evaluated a selection of minimal hitting set algorithms. Using several artificial and real-life scenarios, we aimed at identifying common performance characteristics.

Our tests clearly showed that, specifically for practical purposes, there is no single (old or new) superior algorithm. While the Boolean implementations were the fastest for *most* artificial examples, the real-life examples with a size limit on the desired hitting sets (diagnoses Δ) told another story. For small sensible $|\Delta|$, HS-DAG and HST beat the Boolean approach whose performance could not unfold entirely when limiting the depth of the search space. Considering total time for an on-the-fly computation (including the needed theorem-prover calls for computing *SCS*), the old HS-DAG is on par with HST, suggesting that there is a trade-off between algorithmic speedups and theorem prover calls. Our results make evident that real-life diagnostics performance (with limits on $|\Delta|$) does not match that for synthetic examples.

In order to avoid influence from a specific programming language, we implemented the algorithms in Java and Python. Despite fair optimization levels among the implementations, their performance relations evidently depend on the implementation language. This observation substantiates the natural advice of taking special caution when evaluating an algorithm using a single implementation.

Currently under investigation is whether using C (i.e. inline C-fragments in Python) significantly affects performance and observed trends. Very first tests suggest that circumventing the penalty of Python's dynamic type system with inline C-fragments allows a significant performance boost, while the main code can still be kept in more convenient Python. Further current research focuses on evaluating the influence of the order of all CS_i within an *SCS* (thus our selection of algorithms that can deal with a pre-defined *SCS*), and will expand to trying to help a theorem prover with finding the next "optimal" conflict set "on-the-fly".

Future work will include the implementation of further algorithms, the inspection of memory requirements, and adding new test scenarios. An on-the-fly version of the Boolean idea would be of special interest to us. The effects of exploiting probabilities will be another research topic. In this context we will investigate also the aforementioned general diagnosis engine (GDE) (de Kleer and Williams, 1987).

ACKNOWLEDGEMENTS

This work has been partly funded by the Austrian Science Fund (FWF) under grant P22959. The authors would like to thank Daniel Detassis and Frederix Yves for providing us with their Java implementations of some of the algorithms.

REFERENCES

(Abreu and van Gemund, 2009) R. Abreu and A. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Eighth Symposium on Abstraction Reformulation, and Approximation (SARA'09)*, July 2009.

(Bonet and Helmert, 2010) B. Bonet and M. Helmert. Strengthening landmark heuristics via hitting sets. In *19th European Conference on Artificial Intelligence (ECAI)*, pages 329–334, 2010.

(de Kleer and Williams, 1987) J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

(Greiner *et al.*, 1989) R. Greiner, B. A. Smith, and R. W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

(Hansen *et al.*, 1999) M. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design and Test*, 6:72–80, July-Sept. 1999.

(Lin and Jiang, 2003) L. Lin and Y. Jiang. The computation of hitting sets: review and new algorithms. *Inf. Process. Lett.*, 86:177–184, May 2003.

(Reiter, 1987) R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

(Wotawa, 2001) F. Wotawa. A variant of reiter's hitting-set algorithm. *Inf. Process. Lett.*, 79:45–51, May 2001.