Faster Diagnosis with Answer Set Programming

Liliana Marie Prikler
Institute of Software Technology, Graz University of Technology, Austria

Franz Wotawa
Institute of Software Technology, Graz University of Technology, Austria

Abstract

From hardware to software to human patients, diagnosis has been one of the first areas of interest in artificial intelligence, and has remained a relevant topic since. Recent research in model-based diagnosis has shown that answer set programming not only allows for an easy expression of diagnosis problems, but also efficient solving. In this paper, we improve on previous results by making use of various modern answer set programming techniques. Our experiments compare multi-shot solving, heuristics and preferences, with results indicating that heuristics provide the fastest solutions on most instances we studied.

2012 ACM Subject Classification Computing methodologies → Logic programming and answer set programming; Computing methodologies → Causal reasoning and diagnostics

Keywords and phrases Answer set programming, model-based diagnosis, performance comparison

Digital Object Identifier 10.4230/OASIcs.DX.2024.24

Category Short Paper

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.13850098` [20]

Funding This paper is part of the AI4CSM project that has received funding within the ECSEL JU in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement No.101007326. The work was partially funded by the Austrian Federal Ministry of Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK) under the program "ICT of the Future" project 877587.

1 Introduction

With the advent of new and potentially dangerous technologies such as self-driving vehicles, the time it takes to diagnose a fault is becoming increasingly relevant for systems aimed at maintaining safety and security. For example, an autonomous vehicle that drives on a highway needs to compensate failures until a safe state can be reached. For such compensating actions, diagnosis capabilities are required that are fast enough for providing relevant information, e.g., the severity of a failure and its origin. In model-based diagnosis, where we rely on knowledge representation and reasoning, we need both faster solvers and faster diagnosis algorithms to keep up with these challenges.

Previous investigations on diagnosis via SAT solvers showed that direct encodings of the diagnosis problem can outperform diagnosis algorithms [16]. Similarly, diagnosis via answer set programming is able keep up with or even outperform hitting-set based algorithms [24]. Most recently, Bayerkuhnlein and Wolter [1] introduced other codings and improvements of diagnosis utilizing answer set programming.

It is also worth noting that diagnosis has been a benchmark for solver optimization. Techniques such as heuristics [10] and preferences [2] have been studied with diagnosis as a potential application.

In this paper, we take a combined approach and look at both algorithms and solver-based approaches for efficient diagnosis. Our contributions are two-fold:

- we improve the performance of the IDIAG [17, 24] algorithm by providing an incremental implementation that makes use of multi-shot solving [9], and
- we compare IDIAG to solver optimizations [10, 2], thereby bridging a gap in research which has previously focused on solvers **or** algorithms, but not both at the same time.

The rest of this paper is structured as follows. In Section 2, we introduce the basic concepts of model-based diagnosis and answer set programming, as well as providing a short overview of alternative approaches towards model-based diagnosis through answer set programming. In Section 3, we provide an improved formulation of the prover in [24] that makes use of Clingo's multi-shot solving. In Section 4, we report the results of our comparison. Finally, we conclude the paper in Section 5.

2 Background

In this section, we lay out the basics for understanding the rest of the paper. We start with a description of model-based reasoning and the IDIAG algorithm. Then we describe answer set programming, followed by declarative approaches for obtaining optimal answer sets.

2.1 Model-based reasoning

The underlying idea of model-based reasoning [7, 21] is to solve problems by directly applying available knowledge. One starts with a description of a system from first principles, using some available (typically logical) formalism to formulate this description.

▶ **Definition 1** (Diagnosis system). *A tuple (D, C) is a diagnosis system, where D is the (logic) description of a system, and C is a set of components.*

Here, D describes the nominal behaviour of the system, assuming that all components function as expected. To make these assumptions explicit, predicates $\mathrm{ab}(c)$ and $\neg\mathrm{ab}(c)$ with a given component $c \in C$ are used to indicate that c is behaving "abnormally" (i.e. in a faulty manner) or "not abnormally" (i.e. nominally or healthy).

With this description, we start observing the system under diagnosis. We assume that several observations can be gathered from said system at a time, without impacting its normal behavior.[1] As soon as we find a discrepancy between the expected and actual behaviour of a system, we find ourselves with a diagnosis problem.

▶ **Definition 2** (Diagnosis problem). *Let (D, C) be a diagnosis system and O a set of observations. The tuple (D, C, O) is a diagnosis problem.*

Note, that this definition does not assume that our observations are inconsistent in combination with the system description. Indeed, using boolean satisfiability or answer set programming to model D, we would have to solve an NP-hard subproblem to decide, whether our observations are consistent with reasonable expectations.

Given a diagnosis problem, we want to find out which components of the system – if any – are faulty. This leads us to the actual *diagnoses* of model-based diagnosis.

▶ **Definition 3** (Diagnoses). *Let (D, C, O) be a diagnosis problem. A subset $\Delta \subseteq C$ is a diagnosis if and only if $D \cup O \cup \{ab(c)|c \in \Delta\} \cup \{\neg ab(c)|c \notin \Delta\}$ is consistent. A diagnosis is (subset-)minimal, or parsimonious, if and only if there exists no $\Delta' \subset \Delta$ that is itself a diagnosis. A diagnosis is superfluous if it is not parsimonious.*[2]

[1] This assumption may be invalid under quantum mechanics, but is assumed to be true for boolean circuits or human patients.

[2] In some literature, superfluous diagnoses may not be considered diagnoses at all.

Several algorithms exist to solve diagnosis problems [21, 12, 23, 19], but prior research suggests that direct computation of diagnoses is often just as viable. [16, 17, 24] In this paper, we focus on model-based diagnosis via answer set programming [24], which uses the procedure IDIAG and an ASP-specific prover that we discuss in the next section. Conceptually, IDIAG and Reiter's algorithm [21] are quite similar: as the number of iterations increases, so does the size of yielded diagnoses.[3]

Procedure IDIAG(D, C, O, n).

Data: a system description D, a set of components C, a set of observations O, the desired cardinality n
Result: all subset-minimal diagnoses of size up to n
Let $\Delta = \varnothing, P = D \cup O$;
for $i = 0$ **to** n **do**
  Let $\Delta_i :=$ prover(C, P, i) ;
  if $i = 0 \wedge \Delta_i = \{\varnothing\}$ **then**
    return Δ_i ;
  end
  Set $\Delta = \Delta \cup \Delta_i$;
  Set $P = P \cup \{\mathsf{ab}(c_1) \wedge \cdots \wedge \mathsf{ab}(c_i) \rightarrow \bot | \{c_1 \dots c_i\} \in \Delta_i\}$;
end
return Δ ;

2.2 Answer set programming

Answer set programming [4, 14] is a declarative problem-solving technique based on non-monotonic reasoning. Formally, we distinguish grounded answer set programs and answer set programs as they are written in practice, with predicates, variables, and other extensions.

A grounded answer set program consists of rules of the shape

$$h \leftarrow b_1 \wedge \cdots \wedge b_n \wedge \textbf{not } c_1 \wedge \cdots \wedge \textbf{not } c_m,$$

with h, b_i, c_j for $1 \leq i \leq n, 1 \leq j \leq m$ being drawn from some set of atoms $\mathcal{A}$. These rules look quite similar to Horn clauses – in fact, for $m = 0$, they are Horn clauses.

▶ **Definition 4** (Gelfond-Lifschitz transformation [11]). *Let P be a grounded answer set program and τ a set of atoms. Construct P^τ from P as follows:*

- *if a rule contains* ***not*** *x, with $x \notin \tau$, remove* ***not*** *x from that rule*
- *if a rule contains* ***not*** *x, with $x \in \tau$, remove that rule from P^τ*

By construction, the Gelfond-Lifschitz transformation arrives at a positive logic program, that is, a logic program consisting only of Horn clauses.

▶ **Definition 5** (Stable model). *Let P be a grounded answer set program and τ a set of atoms. Let P^τ be the Gelfond-Lifschitz transformation of P using τ. τ is a stable model of P if and only if τ is the smallest model of P^τ under classical (or Horn) satisfiability.*

[3] In an interesting twist, however, only IDIAG is able to yield these partial results. With an algorithm based on conflict sets, such as Reiter's algorithm, future iterations may invalidate prior results, whereas in a direct computation, minimality can be guaranteed by construction.

To account for variables, we make use of the identities

$$\exists x.\varphi \Leftrightarrow \varphi_{x=x_1} \vee \cdots \vee \varphi_{x=x_n}$$

and

$$\forall x.\varphi \Leftrightarrow \varphi_{x=x_1} \wedge \cdots \wedge \varphi_{x=x_n},$$

where x_i for $1 \leq i \leq n$ denotes all possible values the variable x may take [13].

▶ **Definition 6** (Herbrand universes and bases). *Let P be a logic program. The Herbrand universe of P, denoted $\mathcal{U}(P)$ is the set of all terms which can be formed from constants and function symbols in P.*

The Herbrand base $\mathcal{B}(P)$ is the set of all variable-free atoms, which can be constructed from predicates in P and terms in $\mathcal{U}(P)$.

By definition, the Herbrand universe holds every value that is of potential meaning for the program P. For a given rule r containing variables, we may assume that the variables in the head are implicitly qualified using $\forall$, whereas the variables appearing only in the body are implicitly qualified using $\exists$.

▶ **Definition 7** (Grounding). *A ground instance of a rule $r \in P$ is obtained by applying a substitution of the variables in r with atoms taken from $\mathcal{U}(P)$. Let $\mathcal{G}(r)$ be the set of all possible ground instances of r. Then $\mathcal{G}(P) = \bigcup_{r \in P} \mathcal{G}(r)$.*

As each variable is expanded separately from each other, we arrive at a program where $r \Leftrightarrow \mathcal{G}(r)$. Taking $\mathcal{A} = \mathcal{U}(P)$ and $\tau \subset \mathcal{B}(P)$, the stable models of an answer set program P thus become the stable Herbrand models of its grounded program $\mathcal{G}(P)$.

This technique is easy enough to understand for illustrative purposes, but in practice wasteful. Modern solvers make use of more efficient techniques such as domain-restricted grounding [18] and also allow other syntactic constructs (cf. [6, 8]). Similar to variables, these extensions are grounded before the program is solved, leading to no change in semantics for the grounded program, while aiding the programmer in formulating their program.

Before we discuss extensions for optimization, we first quickly show aggregates and cardinality constraints, which necessarily appear in diagnosis. The ASP-Core-2 standard [6], which provides a common syntax for solvers, allows for disjunctive heads of the shape

$$h_1 \vee \cdots \vee h_k$$

in which exactly one $h_i, 1 \leq i \leq k$ becomes true as a result of applying the rule, or more generally choice rules of the shape

$$\alpha_1 \prec_1 \{h_1; \ldots; h_k\} \prec_2 \alpha_2.$$

Here, α_1 and α_2 can impose a cardinality constraint on the number of h_i that hold, with $\prec_{1,2} \in \{=, \neq, <, >, \geq, \leq\}$. Either side can also be omitted to only have a single cardinality constraint. In the body, such constraints can similarly be imposed by writing

$$\alpha_1 \prec_1 \texttt{\#count}\{l_1; \ldots; l_k\} \prec_2 \alpha_2.$$

These numerical comparisons retain their intuitive mathematical meaning as far as the answer set solver allows. For the practical purposes of this paper, it suffices that operations on integers in the range $[-2^{31}, 2^{31} - 1]$ are supported.

2.3 Preferred and optimal solutions

Similar to how diagnosis algorithms are tasked to find parsimonious diagnoses, we find optimization problems in other domains where logic solvers are used. To cope with such problems, modern solvers implement methods to describe them and obtain optimal solutions. Within the Potassco solver family – to which Clingo [9], the solver we will use for our implementation, belongs – two techniques stand out as applicable to diagnosis problems: domain-specific heuristics [10] and preferences [2].

With domain-specific heuristics, users guide the internal decision making of their solver. Solvers use *conflict-driven clause learning* [15], a process that alternates between guessing the truth value of a particular literal and then propagating the effects of that guess, so that eventually either a satisfying assignment (or in the context of ASP a stable model) or a conflict is reached – this conflict is then used to derive a clause which is added to the input program. The solver then backtracks to an earlier decision and starts anew. Heuristics can guide this process in two manners: first by choosing which literals to assign, second by choosing the values to assign.

With preferences, users turn a decision problem "is there an answer set?" to an optimization problem "is there an answer set, preferably a small one?". These preferences are themselves written as a program which is evaluated on (reified) answer sets to determine which of them (if any) is preferred over the other. Several such programs are already implemented in asprin [3, 2], so that one may also use pre-defined preferences, as shown in Listing 1.

Listing 1 Stating the preference for parsimonious diagnoses.

```
#preference(p,subset) { ab(X) }.
#optimize(p).
```

Some of these preferences could be implemented in Clingo itself as weak constraints (e.g. cardinality-minimal diagnoses), but notably, parsimonious diagnoses are not one of them.

3 An incremental prover

In this section we describe our incremental implementation, which is based on the multi-shot solving of Clingo [9]. Similar to modern SAT solvers, which support incremental solving, multi-shot solving allows incremental grounding and solving of answer set programs. This opens up a tremendous opportunity for IDIAG, which for the most part only adds constraints to the underlying program.

In Listing 2, we show the necessary program parts for an incremental prover. As part of the base program, we assume that `comp/1` defines the components of the system and allow the solver to guess whether a component works as expected (in which case `nab(C)` holds), or does not work as expected (in which case `ab(C)` holds) with a choice rule. As a diagnosis consists of only the broken components, we show `ab/1`. Further, we heuristically choose `ab(C)` not to hold. This heuristic is in fact not relevant to IDIAG itself, as it needs to be explicitly enabled, but it is the main piece we will compare IDIAG against.

The *incremental* part comes with the program `idiag_step`. This program is grounded with a single argument n and corresponds to parameterization of `$n$` in [24].[4] It provides both an external literal and a constraint that becomes enabled with this external literal,

[4] Note that we use `_idiag_step_`, as it is a valid symbol that is unlikely to be bound to a constant. In clingo, whole-program constants shadow the constants given in a program parameter.

disallowing solutions of cardinality higher than the current target cardinality. The inequality is only formulated in one direction, as there ought to be no stable model in the other direction (with smaller cardinality) following the updates from IDIAG.

Listing 2 Supporting logic program for IDIAG using multi-shot solving. A choice rule is used to enforce that $ab(c) \wedge \neg ab(c)$ holds – Clingo implicitly uses $\leq$ for comparing two values.

```
#program base.
#defined comp/1.
1 { ab(C) ; nab(C) } 1 :- comp(C).
#show ab/1.
#heuristic ab(C). [1, false]

#program idiag_step(_idiag_step_).
#external max_ab(_idiag_step_).
:- max_ab(_idiag_step_), #count {C : ab(C)} > _idiag_step_.
```

With these program parts, we can now discuss the procedure prover. In collaboration with IDIAG, it maintains a representation of the diagnosis problem and the diagnoses, which were already found and ought to be ignored. The literal `max_ab(`n`)` becomes active for at most one iteration – at most one, because it might not exist.[5]

Procedure prover(C, M, n).

Data: a set of components C, a model M, a cardinality n
Invariant: no diagnosis of cardinality $n-1$ exists (maintained by `IDIAG`)
Result: all diagnoses of cardinality n
Let M' be the `clingo` representation of M ;
Ground idiag_step (n) ;
if *there exists an external literal* `max_ab(`n`)` **then**
  Assign $\top$ to `max_ab`(n) ;
  Let Σ be the answer sets of M' ;
  Let $\Delta = \{\{c_1, \ldots, c_n\} | \{\mathsf{ab}(c_1), \ldots, \mathsf{ab}(c_n)\} \in \Sigma\}$;
  `/* same as max_ab(`n`) := ⊥ for the remainder of the program */`
  Release `max_ab`(n) ;
  return Δ ;
end
else
  return $\varnothing$;

4 Experimental results

We implemented IDIAG and prover using the Clingo C API, inlining the prover into IDIAG, so that a situation in which no more diagnoses exist can be detected early (cf. [20] for the implementation and the scripts used for benchmarking). Our implementation is loosely

[5] Indeed, Clingo is wise enough to detect when it has already found all minimal diagnoses – while this is not obvious from the Python API, it is an error we need to deal with in the C API.

coupled with Listing 2 – it assumes, that an `idiag_step` program exists, which defines an external literal `max_ab`, and that the shown symbols of a model correspond to the diagnosis to report.

We compare our implementation with other techniques using the well-studied ISCAS'85 benchmark circuits [5]. In particular, we reuse the test data from [24], but rewrite it to conform stronger to the structure actually laid out in that paper.[6] For our implementation to work as intended, we need `comp(C)` to actually be established as a fact (for Listing 2), and we need `ab/1` to be the only shown predicate (as per the limitation mentioned above).

As an example, Listing 3 shows a test input before rewriting, and Listing 4 after rewriting. During this rewrite we also drop the symbol `no_ab(N)`, used to indicate the cardinality of a diagnosis. Where needed, we reintroduce this literal rather than using the `idiag_step` program, which is only handled by our implementation. Unless otherwise noted, we also provide a constraint on the maximum cardinality of *any* diagnosis, via the standard input. Regardless of how we inform the implementation of the maximum cardinality, we fix it to the number 3.

Listing 3 Example input before rewriting.

```
:- lv_1gat_0, lv_1gat_1.
:- lv_4gat_0, lv_4gat_1.
% ...

nab_118gat :- not ab_118gat.
ab_118gat :- not nab_118gat.
% ...

lv_118gat_0 :- nab_118gat, lv_1gat_1.
lv_118gat_1 :- nab_118gat, lv_1gat_0.
lv_1gat_0 :- nab_118gat, lv_118gat_1.
lv_1gat_1 :- nab_118gat, lv_118gat_0.
% ...

lv_1gat_1.
% ...
```

We run our benchmarks on a Lenovo ThinkPad T450 with an Intel Core i5-5200U processor, 8 GB of RAM and 2 GB of Swap space. While the system runs Ubuntu, we use Python 3.10.7, Clingo 5.7.1, and asprin 3.1.1-1.bc5a0cf (corresponding to the newest commit at the time of writing), as packaged in GNU Guix on commit `6d81f2a...` (see Listing 5 for the complete channel list).

We first compare our implementation to a slightly modified version of the original Python implementation [24]. To make the comparison between the two tools fair, we use Clingo's statistics in both, rather than timing the entire process, which would also take into account file I/O and grounding times, further penalizing the reference implementation. The results can be seen in Table 1 and appear to strongly favour our implementation in all but two benchmarks.

Next, we compare our tool against the approaches laid out in Section 2.3, i.e., heuristics (built into Clingo) and preferences (via asprin). It should be noted, that both solver approaches do not guarantee that diagnoses are sorted by their cardinality. Thus, when there

[6] The original tool and benchmarks can be found at `https://github.com/QAMCAS/ASP-Diagnose-Tool`.

Listing 4 Example input after rewriting.

```
:- val(V, X), val(V, Y), X < Y.

comp(gat118).
% ...

val(gat118, 0) :- nab(gat118), val(gat1, 1).
val(gat118, 1) :- nab(gat118), val(gat1, 0).
val(gat1, 0) :- nab(gat118), val(gat118, 1).
val(gat1, 1) :- nab(gat118), val(gat118, 0).
% ...

val(gat1, 1).
% ...
```

Listing 5 Guix channels used to obtain Clingo, Python, asprin, as well as the dependencies of our IDIAG implementation.

```
(list
 (channel
  (name 'guix)
  (url "https://git.savannah.gnu.org/git/guix.git")
  (branch "master")
  (commit
   "6d81f2a4ade07158fbadd560722cc386007caf68")
  (introduction
   (make-channel-introduction
    "9edb3f66fd807b096b48283debdcddccfea34bad"
    (openpgp-fingerprint
     "BBB0 2DDF 2CEA F6A8 0D1D  E643 A2A0 6DF2 A33A 54FA")))))
```

are more than the desired number of diagnoses, they may report more diagnoses of a higher cardinality – or lower quality if cardinality is a concern. We argue, however, that parsimony along with a stated maximum cardinality are likely enough quality criteria, as diagnoses need to be verified before they can be acted upon.[7]

Tables 2 and 3 show the results of our comparison with Clingo and asprin. With the exception of two outliers, Clingo is faster than both asprin and IDIAG, both on average and in the worst case. Each of asprin and IDIAG is the fastest tool once, with IDIAG on average being closer in performance to Clingo, and asprin being the slowest on average. The effects that make asprin faster than IDIAG appear to be more pronounced when looking at worst-case performance.

Finally, we look a little more closely at asprin, which also has built-in support for heuristics. Asprin provides multiple means of turning its preferences into heuristics (cf. [3, 2]). Our own results are shown in Tables 4 and 5. Interestingly, it appears as though heuristics only have a weak positive effect (lower runtime; better performance) in some instances, and a

[7] Assuming one has the means of obtaining more observations interactively and for each component individually, diagnoses of higher cardinality may even be beneficial, as they provide multiple components to investigate in parallel. One may also discover that a particular component is a likely cause of failure by counting the number of diagnoses it appears in, cf. [22].

Table 1 Average runtimes (in seconds) of the original IDIAG implementation [24] and our tool when tasked to compute subset-minimal diagnoses for the ISCAS85 benchmark circuits. The better result is highlighted using a bold font.

	reference[1,a,l)]			ours[2)]		
circuit	1	2	3	1	2	3
c432	0.5292	0.5452	0.5244	**0.1161**	**0.1184**	**0.1291**
c499	0.8898	0.8310	0.6785	**0.3136**	**0.3274**	**0.2077**
c880	1.3896	1.9994	1.9312	**0.2262**	**0.3866**	**0.3307**
c1355	**4.4549**	**4.0057**	4.2121	4.8465	14.9191	**2.0967**
c1908	11.0776	9.4301	21.0627	**3.1129**	**1.3992**	**5.6861**
c2670	10.7953	17.2354	28.2214	**0.8324**	**1.9525**	**3.5458**
c3540	37.1014	38.0905	101.4154	**4.9932**	**6.0543**	**27.2434**
c5315	116.5337	97.6262	266.2112	**4.2492**	**3.091**	**14.3082**
c6288	36.3123	63.5793	115.7475	**3.5328**	**12.5654**	**11.2871**
c7552	528.0185	759.5175	5159.9682	**11.0034**	**30.3936**	**81.2927**

1) `python idiag-ref.py --support=idiag-ref.lp --faultsize 3 --answersets 100 "$test"`
2) `idiag 100 --stats=2 idiag.lp "$test" -`
a) Maximum cardinality of diagnosis provided via `--faultsize` argument.
l) `no_ab` defined in `idiag-ref.lp`.

Table 2 Average runtimes of various tools when tasked to compute subset-minimal diagnoses for the ISCAS85 benchmark circuits. The best result is highlighted with a bold font. An italic font is used to highlight, that asprin is second in terms of performance rather than third.

	asprin[1,u,N)]			clingo[2,u)]			IDIAG[3,p)]		
circuit	1	2	3	1	2	3	1	2	3
c432	3.0025	2.6269	2.4364	**0.086**	**0.0929**	**0.0964**	0.1161	0.1184	0.1291
c499	3.8399	2.9064	1.4139	**0.2319**	0.4069	**0.1866**	0.3136	**0.3274**	0.2077
c880	4.1091	6.2516	5.7019	**0.1532**	**0.1899**	**0.1544**	0.2262	0.3866	0.3307
c1355	9.0238	**8.2308**	7.4812	**1.9398**	37.8516	**1.2266**	4.8465	14.9191	2.0967
c1908	13.7109	15.781	15.7449	**0.5426**	**0.5067**	**1.0541**	3.1129	1.3992	5.6861
c2670	12.6344	17.4196	17.9126	**0.5566**	**0.5437**	**0.5851**	0.8324	1.9525	3.5458
c3540	21.9377	28.6404	34.1104	**0.9637**	**0.9309**	**5.7958**	4.9932	6.0543	27.2434
c5315	30.5597	21.8695	43.8268	**1.5116**	**1.4836**	**2.0253**	4.2492	3.091	14.3082
c6288	35.8091	39.6309	39.4172	**1.9525**	**1.6256**	**1.5504**	3.5328	12.5654	11.2871
c7552	67.5605	74.2026	*73.0008*	**2.6636**	**2.4526**	**3.1381**	11.0034	30.3936	81.2927

1) `asprin 100 --stats idiag.lp preference.lp "$test" -`
2) `clingo 100 --stats=2 --enum-mode=domRec --heuristics=domain idiag.lp "$test" -`
3) `idiag 100 --stats=2 idiag.lp "$test" -`
p) Reports diagnoses in a partially-ordered manner (by cardinality).
u) Reports diagnoses in a seemingly unordered manner.
N) May report more than the desired number of diagnosis (due to non-optimal solutions)

Table 3 Maximal runtimes of various tools when tasked to compute subset-minimal diagnoses for the ISCAS85 benchmark circuits. The best result is highlighted with a bold font. An italic font is used to highlight, that asprin is second in terms of performance rather than third.

	asprin[1,u,N)]			clingo[2,u)]			IDIAG[3,p)]		
circuit	1	2	3	1	2	3	1	2	3
c432	5.426	3.927	3.746	**0.099**	**0.124**	**0.113**	0.136	0.153	0.195
c499	4.595	4.278	4.129	**0.481**	1.041	**0.284**	0.858	**0.965**	0.302
c880	8.099	7.794	6.751	**0.180**	**0.402**	**0.168**	0.434	1.582	1.111
c1355	15.886	**9.646**	8.969	**15.382**	232.803	**4.253**	18.113	82.934	5.994
c1908	22.879	18.766	19.498	**1.272**	**0.619**	**3.048**	12.844	3.724	16.156
c2670	29.583	22.347	27.956	**0.715**	**0.720**	**0.785**	2.161	5.625	16.682
c3540	33.875	34.000	*60.765*	**1.756**	**1.082**	**45.566**	21.039	18.122	107.792
c5315	53.090	39.037	78.912	**2.502**	**3.582**	**5.012**	15.348	5.075	72.425
c6288	40.874	42.954	42.745	**2.160**	**1.920**	**1.679**	7.663	35.284	26.621
c7552	73.987	*90.521*	*115.544*	**3.754**	**4.630**	**9.238**	36.467	136.745	242.663

1) `asprin 100 --stats idiag.lp preference.lp "$test" -`
2) `clingo 100 --stats=2 --enum-mode=domRec --heuristics=domain idiag.lp "$test" -`
3) `idiag 100 --stats=2 idiag.lp "$test" -`
p) Reports diagnoses in a partially-ordered manner (by cardinality).
u) Reports diagnoses in a seemingly unordered manner.
N) May report more than the desired number of diagnosis (due to non-optimal solutions)

significantly negative effect (much larger runtime; much worse performance) in others. These findings contrast prior research suggesting that heuristics to have a pronounced positive effect – we were not able to discover such an effect in our data.

Combining our comparison results, it appears as though structural properties in some test cases favour one family implementations over others. In particular, the c1355 benchmarks show good performance in asprin and the reference implementation, both of which use multiple solver calls, whereas most other benchmarks are faster using Clingo (single-shot) or our multi-shot IDIAG, which keep intermediate information.

Closer inspection of our test data reveals three test cases

- `c1355_tc_1_33`,
- `c1355_tc_2_33`, and
- `c1355_tc_2_34`,

which do not have a diagnosis of cardinality 3, but do have a diagnosis of higher cardinality. We imagine that our test results would favour asprin and the reference implementation more strongly if more inputs shared this behavioural pattern. We leave the investigation as to why this pattern causes such huge influence on some configurations, but not others, to future research.

5 Conclusion

We have implemented an incremental prover for the previously described IDIAG algorithm, based on the Clingo answer set solver. Our experiments indicate, that this incremental prover is significantly faster than the reference, but also, that Clingo itself is faster than both of them at determining parsimonious diagnoses.

We have further identified a problematic pattern, in which the performance of Clingo and our IDIAG implementation is worse than that of state-of-the-art alternatives. Future work may investigate these patterns more closely and find more efficient ways of handling them.

Table 4 Average runtimes of asprin with various heuristics. The best result is highlighted with a bold font.

	asprin[1)]			asprin$_s$[2,n)]			asprin$_l$[3,n)]		
circuit	1	2	3	1	2	3	1	2	3
c432	3.0025	2.6269	2.4364	2.8272	2.7462	2.5235	**2.1994**	**2.5593**	**2.4048**
c499	3.8399	**2.9064**	**1.4139**	3.7184	2.9622	1.429	**3.7165**	2.9736	1.4403
c880	4.1091	6.2516	**5.7019**	4.050	6.2695	5.7221	**3.9736**	**6.110**	5.740
c1355	**9.0238**	**8.2308**	**7.4812**	9.1101	8.3522	8.4739	10.1812	18.2236	9.5113
c1908	13.7109	**15.781**	**15.7449**	15.3369	16.5029	16.6365	**11.6994**	17.0772	22.685
c2670	12.6344	17.4196	17.9126	12.8349	18.9981	18.1386	**9.637**	**16.6658**	**17.0139**
c3540	**21.9377**	**28.6404**	**34.1104**	22.0679	29.2287	37.9311	22.3426	28.5987	50.9506
c5315	30.5597	**21.8695**	43.8268	31.1048	23.0021	56.2116	**25.9356**	23.452	**36.5648**
c6288	35.8091	39.6309	**39.4172**	36.6559	39.2868	40.0324	**34.2615**	**38.4413**	51.6775
c7552	67.5605	74.2026	**73.0008**	73.4326	94.2245	102.8467	**64.2992**	**61.5337**	79.2451

1) `asprin 100 --stats idiag.lp preference.lp "$test" -`
2) `asprin 100 --stats --dom-heur -1 sign idiag.lp preference.lp "$test" -`
3) `asprin 100 --stats --dom-heur 1 level idiag.lp preference.lp "$test" -`
n) Stripped heuristic from `idiag.lp` to use internal heuristics from asprin.

Table 5 Maximal runtimes of asprin with various heuristics. The best result is highlighted with a bold font.

	asprin[1)]			asprin$_s$[2,n)]			asprin$_l$[3,n)]		
circuit	1	2	3	1	2	3	1	2	3
c432	5.426	3.927	3.746	5.361	3.925	3.911	**3.498**	**3.368**	**3.730**
c499	4.595	4.278	4.129	**4.306**	4.602	4.110	4.868	**4.247**	**3.929**
c880	8.099	7.794	6.751	7.990	7.522	6.686	**7.495**	**7.265**	**6.423**
c1355	15.886	**9.646**	8.969	**14.226**	9.926	**8.905**	14.717	64.170	11.622
c1908	22.879	**18.766**	**19.498**	28.714	18.790	28.001	**16.622**	29.100	56.778
c2670	29.583	22.347	**27.956**	28.675	24.659	28.132	**19.662**	**20.250**	28.685
c3540	33.875	**34.000**	**60.765**	**33.066**	34.784	73.620	44.448	38.024	268.888
c5315	53.090	**39.037**	78.912	58.880	43.743	169.542	**42.327**	56.649	**74.325**
c6288	**40.874**	**42.954**	**42.745**	44.732	43.429	49.227	42.546	43.081	186.843
c7552	**73.987**	**90.521**	**115.544**	100.091	177.680	181.315	85.318	117.532	326.509

1) `asprin 100 --stats idiag.lp preference.lp "$test" -`
2) `asprin 100 --stats --dom-heur -1 sign idiag.lp preference.lp "$test" -`
3) `asprin 100 --stats --dom-heur 1 level idiag.lp preference.lp "$test" -`
n) Stripped heuristic from `idiag.lp` to use internal heuristics from asprin.

References

1 Moritz Bayerkuhnlein and Diedrich Wolter. Model-based diagnosis with ASP for non-groundable domains. In Arne Meier and Magdalena Ortiz, editors, *Foundations of Information and Knowledge Systems*, pages 363–380, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-56940-1_20.

2 Gerhard Brewka, James Delgrande, Javier Romero, and Torsten Schaub. A general framework for preferences in answer set programming. *Artificial Intelligence*, 325:104023, 2023. doi:10.1016/j.artint.2023.104023.

3 Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub. asprin: Customizing answer set preferences without a headache. In *AAAI*, pages 1467–1474. AAAI Press, 2015. doi:10.1609/AAAI.V29I1.9398.

4 Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, December 2011. doi:10.1145/2043174.2043195.

5 Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target simulator in fortran. In *Proceedings of the 1985 International Symposium on Circuits and Systems (ISCAS'85)*, pages 695–698, June 1985. Special Session on ATPG and Fault Simulation.

6 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309, 2020. doi:10.1017/S1471068419000450.

7 Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. doi:10.1016/0004-3702(87)90063-4.

8 Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract Gringo. *Theory and Practice of Logic Programming*, 15(4-5):449–463, 2015. doi:10.1017/S1471068415000150.

9 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019. doi:10.1017/S1471068418000054.

10 Martin Gebser, Benjamin Kaufmann, Javier Romero, Ramón Otero, Torsten Schaub, and Philipp Wanko. Domain-specific heuristics in answer set programming. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, pages 350–356. AAAI Press, 2013. doi:10.1609/AAAI.V27I1.8585.

11 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

12 Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989. doi:10.1016/0004-3702(89)90079-9.

13 Jacques Herbrand. *Recherches sur la théorie de la démonstration.* Doctoral thesis, Faculté des sciences de Paris, 1930.

14 Vladimir Lifschitz. *Answer Set Programming.* Springer, 2019. doi:10.1007/978-3-030-24658-7.

15 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

16 Iulia Nica, Ingo Pill, Thomas Quaritsch, and Franz Wotawa. The route to success - A performance comparison of diagnosis algorithms. In Francesca Rossi, editor, *IJCAI 2013,*

Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013, pages 1039–1045. IJCAI/AAAI, 2013. URL: http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6597.

17 Iulia Nica and Franz Wotawa. Condiag - computing minimal diagnoses using a constraint solver. In *Proceedings of the 23rd International Workshop on Principles of Diagnosis*, 2012.

18 Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999. doi:10.1023/A:1018930122475.

19 Ingo Pill and Thomas Quaritsch. Rc-tree: A variant avoiding all the redundancy in reiter's minimal hitting set algorithm. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Gaithersburg, MD, USA, November 2-5, 2015*, pages 78–84. IEEE Computer Society, 2015. doi:10.1109/ISSREW.2015.7392050.

20 Liliana Marie Prikler and Franz Wotawa. Replication package for: Faster diagnosis with answer set programming. The replication package contains our implementation of IDIAG and the scripts we used to benchmark them as two gzipped tar archives. It does not include the test data. doi:10.5281/zenodo.13850098.

21 Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987. doi:10.1016/0004-3702(87)90062-2.

22 Roni Stern, Meir Kalech, Shelly Rogov, and Alexander Feldman. How many diagnoses do we need? *Artificial Intelligence*, 248:26–45, 2017. doi:10.1016/j.artint.2017.03.002.

23 Franz Wotawa. A variant of reiter's hitting-set algorithm. *Information Processing Letters*, 79(1):45–51, 2001. doi:10.1016/S0020-0190(00)00166-6.

24 Franz Wotawa and David Kaufmann. Model-based reasoning using answer set programming. *Applied Intelligence*, 52(15):16993–17011, 2022. doi:10.1007/S10489-022-03272-2.