UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Incremental SAT-based Detection of Core and Dead Features in Configuration Models

LUIS CAMBELO GUTIÉRREZ

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Itinerario: Ingeniería de Software - 31105151

Directores:
Rubén Heradio Gil
David Fernández Amorós

Resumen

Los sistemas de software son cada vez más configurables. Un claro ejemplo es el Kernel de Linux, que puede adaptarse a una extraordinaria variedad de dispositivos de hardware (teléfonos inteligentes, computadoras portátiles, clústeres de computadoras, etc.) gracias a las miles de características configurables que admite.

Un problema central en el análisis de este tipo de sistema altamente configurable es la detección automática de características esenciales e inactivas. Las características esenciales son aquellas que deben incluirse en cada configuración. Por el contrario, las características muertas son aquellas que, debido a sus incompatibilidades con otras funciones, no se pueden activar en ninguna configuración y, por lo tanto, deben eliminarse durante el mantenimiento del sistema.

En la literatura de ingeniería de software, y particularmente en el área de las línea de productos software, las características esenciales y muertas se identifican típicamente llamando masivamente a un *SAT-solver* para analizar una fórmula proposicional que codifica el modelo configurable. En la medida de nuestro conocimiento, esta tesis es el primer trabajo que establece una conexión entre las características centrales/muertas y el *backbone* de las fórmulas proposicionales, mostrando su total equivalencia. Gracias a esta equivalencia, esta tesis proporciona una implementación funcional de varios algoritmos de última generación para la detección de backbones y prueba su notable escalabilidad para detectar características esenciales y muertas en modelos configurables.

Nuestra implementación se basa en la interfaz IPASIR, que es una forma estándar de interactuar con los *SAT-solvers* de forma incremental. De esta manera, nuestro código se desacopla de cualquier solucionador *SAT-solver* específico (es decir, funciona con cualquier solucionador que implemente el estándar)

Palabras clave: SAT solver, característica esencial, característica muerta, backbone, IPASIR, minibones, EDUCIBone, modelo de configuración.

Abstract

Software systems are becoming increasingly configurable. A clear example is the Linux Kernel, which can be adapted for an extraordinary variety of hardware devices (smartphones, laptops, computer clusters, etc.) thanks to the thousands of configurable features it supports.

One central problem in analyzing this kind of highly configurable system is the automated detection of *core* and *dead* features. Core features are those that must be included in every configuration and thus are entirely essential. In contrast, dead features are those that, because of their incompatibilities with other features, cannot be activated in any configuration and thus should be removed during the system maintenance.

In the software engineering literature, and particularly in the software product line field, core and dead features are typically identified by calling a SAT-solver massively to analyze a propositional formula that encodes the configurable model. To the extent of our knowledge, this thesis is the first work that makes a connection between core/dead features and the backbone of propositional formulas, showing their total equivalence. Thanks to this equivalence, this thesis provides the functional implementation of several state-of-the-art algorithms for detecting backbones, and tests their remarkable scalability to detect core and dead features in configurable models.

Our implementation is based on the IPASIR interface, which is a standard way to interact with SAT-solvers incrementally. This way, our code is decoupled from any specific SAT-solver (i.e., it works with any solver that implements the standard).

Keywords: SATsolver, dead feature, core feature, variability model, configuration model, backbone, IPASIR, minibones, EDUCIBone.

Acknowledgements

A Charo, David y Sara

Luis Cambelo Gutiérrez
Madrid
February 2023

Contents

List of Tables

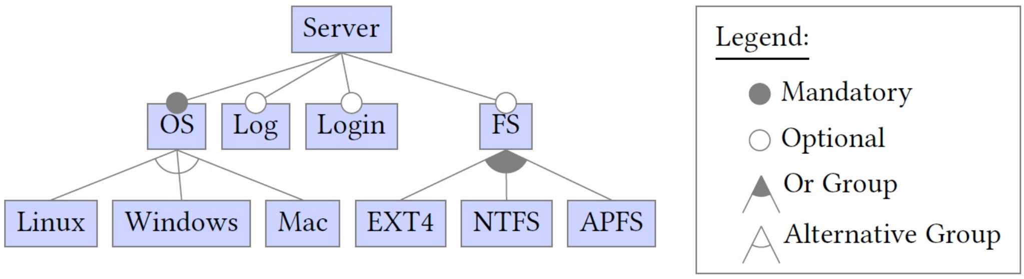
List of Figures

1

Chapter 1: Introduction

The number of highly configurable software systems is increasing. A very illustrative example is the Linux Kernel, which can be configured for an immense variety of hardware devices thanks to the existing thousands of configurable features it currently supports and the capacity to add new ones. Devices supported by some kind of Linux distribution range from smartphones, desktop or laptop computers, and even 100% of the *top-500* most powerful supercomputers. These software systems evolve over time, and new features are added to the feature base, so they can be selected or deselected to produce a final configuration. This evolution leads to intermediate scenarios where some features are mandatory (also known as *core* features) and some other features became obsolete or incompatible and, therefore, can not be chosen (*dead* features). Once these core and dead features have been identified, software engineers can continue selecting and deselecting additional features into the desired configuration. Note this process is incremental and, most times bidirectional: the designer might choose to add a new feature and observe that feature, in turn, will require other features to be selected or even be deselected due to incompatibilities.

Managing this process when the number of features is small (Figure 1.1, taken from [Krieter et al., 2021]), is a tractable problem, but for highly configurable systems (the Linux Kernel has more than 13,000 features), the problem becomes intractable.



$$(Windows \rightarrow NTFS) \land (Mac \rightarrow APFS) \land (Linux \lor NTFS \lor APFS) \land (Server \rightarrow \neg Login)$$

Figure 1.1: An example of feature model taken from [Krieter et al., 2021]

Efficiently identifying those core and dead features while working with a configuration model, either as a background task or in an interactive setting is an active research stream. There are several lines of research (refer to Chapter 2), some of them are: Binary Decision Diagrams (BDD), Strong Dependencies, SAT-solvers, every one with its own highlights and deficiencies.

[Batory, 2005] showed the equivalence between feature models and propositional logic, which supports the automated analysis of models using SAT-solver. A SAT-solver can be called with a particular configuration and it will return *satisfiable* if the configuration is valid, and *unsatisfiable* if the configuration is not valid. But, before starting with the configuration model, it is key to know which ones are the core and the dead features in order to set them and avoid starting with an invalid configuration. Once that initial configuration has been set, more features can be incrementally selected or deselected in an interactive way.

The problem described is not uniquely applicable to the Linux Kernel. Other highly configurable software systems in areas like automotive, financial systems, software testing, and chip testing also experience it (refer to [Krieter et al., 2021] and section 2.3.1).

1.1 Objective

Within this work, we take abstract algorithms available in previous literature and produce an equivalent implementation based on the IPASIR incremental interface, which allows not only a first identification of the dead and core features but also further addition of rules and whatever feature assumptions (selected/deselected) in an incremental and efficient way. We call our implementation IPASIRBones.

Then, the two following research questions will be answered:

- RQ1: What is the best IPASIRBones and SAT combination implementation?
- RQ2: How IPASIRBones performs when compared to state-of-the-art tools?

We believe IPASIRBones will help in creating new solutions for interactive model configuration.

1.2 Concepts and Definitions

This section provides some concepts and definitions, which will be used in the following chapters.

Definition (Boolean variable). A Boolean variable x has two possible values: True or False.

Definition (Literal). A literal can be either a Boolean variable x (positive literal) or its negation $\overline{x}$ (negative literal.

Hence, we will denote the set of Boolean variables by $X = \{x_1, x_2, ..., x_n\}$ and the set of literals over X as $L = \{x_i, \overline{x}_i | x_i \in X, 1 \leq i \leq n\}$

Note that most implementations of Boolean literals represent them as x_i for the True assignment and $-x_i$ for the False assignment of variable x_i.

Definition (Clause). A clause is a disjunction (or = $\vee$) of literals.

Definition (CNF-Formula). A formula ψ is in Conjunctive Normal Form (CNF) when it is expressed as a conjunction (and = $\wedge$) of clauses.

Definition (Assignment). Given a CNF formula ψ over a set of variables X, an assignment is a mapping from each variable x_i to {`True`, `False`}.

Definition (Satisfiable Formula). Given a CNF formula ψ over a set of variables X, ψ is satisfiable if and only if, for each variable x_i, there exists an assignment that makes formula ψ True. If every possible variable assignment makes the formula False, then formula ψ is unsatisfiable

Definition (Backbone). There are several definitions of the backbone of a satisfiable formula, but the most generally used is the one by [Kilby et al., 2005]: *The backbone of a propositional formula is the set of literals which are true in every satisfying truth assignment*. An alternative definition by [Janota et al., 2015] defines the backbone as the set of necessary assignments: *If a literal l is in the backbone of ψ, any assignment satisfying ψ must set l to true.*

Definition (Core Literal). Given a literal x_i from the backbone of the formula ψ, x_i is a Core Literal if the assignment satisfying formula ψ is x_i.

Definition (Dead Literal). Given a literal x_i from the backbone of the formula ψ, x_i is a Dead Literal if the assignment satisfying formula ψ is $\overline{x}_i$.

Definition (Feature Model). A feature model [Batory, 2005] is a hierarchically arranged set of features. Relationships between apparent (or compound) features and their child features (or subfeatures) are categorized as:

- And — all subfeatures must be selected,
- Alternative — only one subfeature can be selected,
- Or — one or more can be selected,
- Mandatory — features that required, and
- Optional — features that are optional

Feature models, in turn, can be translated into propositional formulas [Mannion, 2002] and this connection allows us to use satisfiability solvers or SAT-solvers

CNF-Formulas (Feature Models, Models) are typically stored in a standard format created by the Center for Discrete Mathematics and Theoretical Computer Science, called DIMACS [SAT Challenge, 1993]. DIMACs are textual files with the following types of lines:

- **Comment lines**: These lines start with a lowercase "c" and can contain any informative text, which is expected to be ignored by any program reading the file.
- **Problem line**: This line starts with a lowercase "p" character and has the format:

```
p FORMAT NUM_VARIABLES NUM_CLAUSES
```

where `FORMAT` must be `"CNF"`, a confirmation that following lines encode a cnf formula, `NUM_VARIABLES` is the number of variables of the formula described and `NUM_CLAUSES` is the number of clauses of the formula.

- **Clause lines**: Must be placed after the problem line. The format is as follows:
  - Every literal is represented by its variable number, with a negative sign in case of a negated literal.
  - A clause is a sequence of literals, separated by spaces and ended with a 0.
  - A formula is a sequence of clauses.
  - There are no restrictions to line splitting. A clause can be split in multiple lines, provided that it is properly ended with a 0. On the other side, multiple formulas can be stored, separated by 0, in a single line.

As an example, given the formula from [Perez-Morago et al., 2015]:

$$\psi = (x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5 \vee x_6) \wedge (\overline{x}_2 \vee x_3) \wedge (\overline{x}_3 \vee x_1) \wedge (\overline{x}_4 \vee x_3) \wedge$$
$$(\overline{x}_5 \vee x_3) \wedge (\overline{x}_6 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2) \wedge (\overline{x}_4 \vee \overline{x}_5) \wedge (\overline{x}_4 \vee \overline{x}_6)$$

The corresponding DIMACS file is listed in Listing 1.1.

Listing 1.1: Example DIMACS file

```
c This is the CNF corresponding to the example from Perez-Morago 2015 article
c f1 is a core feature - included in every derivative
c f2 is a dead feature - missing in every derivative
p cnf 6 9
1 2 3 4 5 6 0
-2 3 0
1 -3 0
3 -4 0
3 -5 0
3 -6 0
-1 -2 0
-4 -5 0
-4 -6 0
```

1.3 Document Structure

The rest of this document is structured as follows: Chapter 2 reviews literature relevant to this work related to feature models, SAT-solvers, backbones and IPASIR, including a brief introduction to its interface. Chapter 3 describes the main contribution of our work: an IPASIR-based implementation of diverse algorithms to compute the backbone of propositional formulas. Chapter 4, reports an in-depth empirical evaluation of our IPASIR programs presented in Chapter 3, ending with a comparison to other state-of-the-art solvers. Finally, Chapter 5 outlines the conclusions and suggest future work.

Chapter 2: Related Work

This chapter reviews academic literature related to the subject discussed in the following chapters: Feature Models, SAT-solvers and backbones.

2.1 Feature Models and SAT-solvers

Batory's seminal paper [Batory, 2005] showed the equivalence between feature models and propositional logic, which supports the automated analysis of models using SAT-solvers. This paper has originated fruitful literature on how to solve variated feature models' problems employing SAT-solvers.

SAT solving literature can be traced back to [Davis and Putnam, 1960], often named as *DP procedure*, and its extension [Davis et al., 1962], named *DPLL procedure*, as the first SAT-solvers. Modern SAT-solvers include additional heuristics to the DPLL procedure and some others are based on conflict-driven clause learning (CDCL) [Marques-Silva et al., 2021].

Eventually, SAT solving became an active research area, with SAT Competitions[1] organized bi-yearly or yearly since 2002, usually as a satellite event to the SAT Conference (International Conference on Theory and Ap-

[1] http://www.satcompetition.org/

plications of Satisfiability Testing) have produced, over the years new algorithms and better heuristics and implementation techniques.

On more advanced topics, [Alyahya et al., 2022] analyzes existing SAT-solvers' literature, looking into underlying structural measures such as backbones, backdoors and others which might help in defining SAT structure.

There are other authors that make different proposals on feature modeling. [Krieter et al., 2021] uses *implication graphs* instead of a SAT-solver within FeatureIDE, a feature-oriented software development framework.

2.2 IPASIR

IPASIR, is the reverse acronym for "Re-entrant Incremental Satisfiability Application Program Interface" [Balyo, 2017], was first presented at the 2015 SAT race [Balyo et al., 2016] to unify the interface for the different incremental SAT-solvers, and since then it has been a competition track for each following SAT competition.

Some SAT-solvers with IPASIR interface are Picosat [Biere, 2008], Lingeling [Biere, 2014], Cadical [Biere et al., 2020], Minisat [Eén and Sörensson, 2004] and Glucose [Audemard and Simon, 2017] which is based on Minisat.

IPASIR aims to provide a universal SAT-solver interface, which can be easily implemented by every SAT-solver and used to build applications in every domain without knowing the underlying implementation of each solver and allowing changing the solver used in the application at compile time without any change in code.

IPASIR interface is composed of the nine functions in Listing 2.1.

Listing 2.1: IPASIR Interface

```
const char *ipasir\_signature();
   // Returns solver name and version
void *ipasir\_init();
    // Initiliazes solver instance and returns a pointer to it
void ipasir\_release(void *solver);
    // Releases (Destroys) the solver instance
void ipasir_set_terminate(void *solver, void *state,
                          int(*terminate) (void *state));
   // Sets a call-back function for aborting solving process when required
void ipasir_add(void *solver, int lit_or_zero);
   // Adds a literal to the current clause or finalize it
void ipasir_assume(void *solver, int lit);
   // Assumes a literal for the next solver call
int ipasir_solve(void *solver);
   // Solves the formula and returns:
   // 10 if SATisfiable, 20 if UNSATisfiable
int ipasir_val(void *solver, int lit);
   // Retrieves a variable truth value (SAT case)
int ipasir_failed(void *solver, int lit);
   // Checks for a failed assumption (UNSAT case)
```

As IPASIR is central for our work, an example of how to use it is provided in Listing 2.2. This example uses the CNF formula in Listing 1.1 as input. Lines 1-3 link the `ipasir.h` interface to the current program, and Line 4 prints the solver name (e.g. "minisat220"). Then, Line 6 returns a pointer to the `solver` instance newly created. Lines 9-11 show how to add the literals of the clause in Line 12 from 1.1 to the formula in the solver: each literal is added with a call to `ipasir_add` and, when all literals from the clause are added, the clause is *added* to the formula by calling `ipasir_add` with a 0 value. Lines 13-15 repeat the same process for the clause in Line 13. This process is intended to be done with a loop or a function reading those values from the DIMACS file.

Suppose that our program has properly added all literals and clauses from 1.1, the call to `ipasir_solve` in Line 18 will return 10[2] in the `res` variable since the formula is satisfiable. Line 22 makes the temporary

[2] `ipasir_solve` returns 10 or 20, meaning SAT or UNSAT, respectively.

assumption that variable x_1 takes the -1 value (i.e., false). In this case, the new call to `ipasir_solve` in Line 23 will now return 20 in the `res` variable as the formula is unsatisfiable under the assumption that variable x_1 takes the literal -1. If the solver returns the UNSAT state, then it can be queried to confirm which assumption variable caused this state. A call to `ipasir_failed`, asking about variable x_1, will return 1, meaning that the previous assumption caused SAT-solver to move into UNSAT. Note that `ipasir_failed` can be only called when the solver is in UNSAT state, and only the variables used in previous calls to `ipasir_assume` can be queried. If the assumed literal does not make the formula unsatisfiable, `ipasir_failed` returns 0.

Listing 2.2: IPASIR Interface example

```
1  extern "C" {
2          #include "ipasir.h"
3  }
4  printf("c Solver: %s\n", ipasir_signature());
5
6  void *solver = ipasir_init();
7
8  // Omitted adding clauses in Lines 5 to 11
9  ipasir_add(solver, -4);
10 ipasir_add(solver, -5);
11 ipasir_add(solver, 0);
12
13 ipasir_add(solver, -4);
14 ipasir_add(solver, -6);
15 ipasir_add(solver, 0);
16
17 // This call to solve will return 10 (SAT)
18 res=ipasir_solve(solver);
19
20 // Assuming variable x1 takes literal value -1
21 // Now solve call will return 20 (UNSAT)
22 ipasir_assume(solver, -1);
23 res= ipasir_solve(solver);
24 failed = ipasir_failed(solver, 1);
25
26 // This new assumption is SAT
27 ipasir_assume(solver, -2);
28 res= ipasir_solve(solver);
```

After any satisfiable call to `ipasir_solver`, the solver internally stores a solution that satisfies the assumptions and formula in the state at the time of the call. This solution can be queried by calling to `ipasir_val( solver, lit)`, with the number of the desired variable and function will return `lit` if the satisfying literal is True and `-lit` if the satisfying literal is False. The IPASIR documentation states that the function may return 0 if the found assignment is satisfying for both valuations of `lit`. Note that `lit` argument name in the function call can be misleading as it is a positive integer between 1 and the number of variables,

Assumptions are temporal and automatically cleared after any call to `ipasir_solve`. In fact, after making the assumption that variable x_2 takes the literal -2 (Line 27), the call to `ipasir_solve` in Line 28 will return 10 (Satisfiable), without being interfered with by the assumption made before the previous solver call (Lines 22 and 23).

New clauses can be added at any time, in the same way as done above (Lines 9 -11 and 13-15) and, unlike assumption variables, they are permanent during the solver instance lifetime.

2.3 Backbones

While there are several definitions of the backbone of a satisfiable SAT problem, the most generally used is the one by [Kilby et al., 2005] (refer to Section 1.2).

But the backbone term was first defined and some of its properties were enumerated at [Monasson et al., 1999] while experimenting on random k-CNF instances. In a recent survey, [Alyahya et al., 2022] provides an overview of structural measures related to the Satisfiability Problem, like the backbone size, strong backdoor size, weak backdoor size, frequency of variables in a weak backdoor, LS backdoor size, LSR backdoor size, and backbone/backdoor variable overlap, by using models ranging from random, crafted, and industrial benchmarks, but the evidence was inconclusive in relation to the backbone.

There have been many attempts to use innovative solutions for obtaining the backbone of a formula. For example, [Guo et al., 2019] uses a heuristic backbone algorithm which provides significant time improvement when compared to the one test per time algorithm.

A different approach is followed by [Perez-Morago et al., 2015], by using a Binary Decision Diagram (BDD) instead of using a SAT-solver to identify features of the *product platform* which must part of every *derivative* and those to be excluded of it.

On the other side, other authors used machine learning techniques. For example, [Wu, 2017] used a logistic regression model in conjunction with a Monte-Carlo approach achieving an accuracy of 78 percent in identifying backbones. Similarly, [Liang et al., 2020] applies ID3 machine learning algorithm [Quinlan, 1986], reaching an accuracy of 75 percent or more, while still resorting to a SAT-solver to complete the backbone. Fully solving the backbone variable based on ID3_algorithm is still an open problem.

[Previti et al., 2017] provides two generic algorithms to compute generalized backbones, that is, formulas defined over-generalized domains, not limited to Boolean values. Another proposal, with practical application in bounded model checking, analysis of hardware circuits, static analysis, and test generation is made by [Previti and Järvisalo, 2018].

The most extensive and deepest work was done at [Janota et al., 2015], as an extension of their previous work at [Marques-Silva et al., 2010] and [Janota et al., 2012]. This work describes seven algorithms to calculate backbones and different performance results, which were implemented in the *minibones* tool and made available publicly.

EDUCIBone, presented at [Zhang et al., 2018], implements three strategies, COV, WHIT, and 2LEN to improve backbone computing. Authors claim that *EDUCIBone* requires 18% less runtime than minibones-cb10.

EDUCIBone and *minibones*, will be used later, during the Experimental Evaluation (Chapter 4) to complete an extended performance evaluation by comparing them with our best-performing programs based on IPASIR.

2.3.1 Applications of backbones

Some examples of backbone applications are the localization of faults in silicon integrated circuits [Zhu et al., 2011], knowledge representation and reasoning (KRR) [Previti and Järvisalo, 2018], vessel stowage [Kroer, 2012] and [Janota, 2010] for interactive model configuration.

Chapter 3: Computing Backbones

This chapter describes the main contribution of our work: an IPASIR-based implementation of diverse algorithms to compute the backbone of propositional formulas (which may encode configuration or any other kind of model). In Section 3.1, the seven algorithms described in [Janota et al., 2015] are implemented with the IPASIR interface. These algorithms are state-of-the-art in backbone computing. Section 3.2 proposes additional heuristics to improve the backbone computation. Finally, Section 3.3 provides details about additional improvements. After describing each algorithm's pseudocode and our corresponding IPASIR implementation, an execution sample will be provided using the `buildroot.cnf` configuration model, taken from [Fernandez-Amoros et al., 2023]. The next Chapter 4 will report an in-depth performance analysis of all the algorithms.

3.1 Backbone computation using IPASIR

As described in Section 2.2, IPASIR is a C/C++ interface to create a uniform interface allowing developers to access compatible SAT-solvers without requiring knowledge of their internal structure.

This section describes fully-functioning IPASIR implementations of seven backbone algorithms abstractly described in [Janota et al., 2015]. In contrast with dedicated backbone calculation tools, our implementation will help integrate backbone calculation into other tools, such as interactive configurators.

The "template" we used to write our code is `genipabones.cpp`, an implementation available at [Balyo, 2017] of the algorithm in Section 3.1.2. Our programs produce an output similar to other tools like *minibones* [Janota et al., 2015] and *EDUCIBone* [Zhang et al., 2020], but redirected to the `stderr` stream to facilitate its ulterior processing. After the description of each algorithm throughout this section, an execution sample will be provided.

3.1.1 Algorithm 1: Enumeration-based

This algorithm enumerates all the implicants, one by one, and updates the backbone in every iteration (Figure 3.1).

Algorithm 1: Enumeration-based backbone computation

Input : Satisfiable formula ϕ
Output: Backbone of ϕ, ν_R

$\nu_R \leftarrow \{x \mid x \in \text{var}(\phi)\} \cup \{\bar{x} \mid x \in \text{var}(\phi)\}$ `// Initial backbone upper bound`
while $\nu_R \neq \emptyset$ **do**
  $(\text{outc}, \nu) \leftarrow \text{SAT}(\phi)$ `// SAT solver call`
  if outc $=$ false **then**
    return ν_R `// Terminate if no more implicants`
  $\nu_R \leftarrow \nu_R \cap \nu$ `// Update backbone estimate`
  `// Block implicant`
  $\omega_B \leftarrow \bigvee_{l \in \nu} \bar{l}$
  $\phi \leftarrow \phi \cup \omega_B$
assert$(\nu_R = \emptyset)$ `// Backbone estimate became empty before enumeration finished`
return ν_R

Figure 3.1: Algorithm 1 - Enumeration-based backbone computation

As a first step, it establishes the set of all literals as the initial backbone upper bound (Line 1). Then a search is performed until that upper bound is not empty, either because that literal was identified as a backbone member or because that literal is not appearing in every SAT solution calculated (the latter is the definition of the backbone). In every loop, a SAT-solver call is performed, causing the loop to finish if that call is not satisfiable. If the SAT call is satisfiable, the upper bound set is filtered to contain only those literals which are also appearing in the new SAT solution returned. As a performance aid and in order to prevent the algorithm to calculate an implicant already found, it uses the *blocking clause* heuristics, by adding it to the formula. A blocking clause for an implicate v is defined as the clause $\bigvee_{l \in v} \bar{l}$.

In our IPASIR implementation (Listing 3.1), the initial backbone upper bound is set as two arrays (Lines 1 and 2), one for positive and one for negative literals. After every SAT call (Line 11), resulting SAT solution literals are saved (Lines 19 to 21), used first negated to filter backbone upper bound (Lines 23 to 35) and then used to compute the block implicant (Lines 37 to 43) to be added as a new clause to the formula. Literals from the SAT solution must be saved to a temporary variable, since IPASIR (and most SAT-solvers) cannot mix calls to *ipasir_val* (which will move SAT-solver from either INPUT, SAT, or UNSAT to INPUT state) and calls to *ipasir_add* (which require SAT-solver to be in SAT state)

The screenshot in Figure 3.2 shows the execution results of Algorithm 1 using `buildroot.cnf` model.

Listing 3.1: IPASIRBones-1

```
int* pos_literals = new int[maxVar];  // upper bound - positive literals
int* neg_literals = new int[maxVar];  // upper bound - negative literals
for (int i=0; i<maxVar; i++) {
    pos_literals[i] =   i+1;
    neg_literals[i] = -(i+1);
}
int vr_upper_count = 2 * maxVar;
int* sat_sol = new int[maxVar];

while (vr_upper_count != 0) {
    res = ipasir_solve(solver);
    satCalls++;
    if (vr_upper_count != vr_cpy) {
        vr_cpy = vr_upper_count;
    }
    if (res==UNSAT) {
        break;                          // return VR upper_bound;
    }
    for (int lit=1; lit<=maxVar; lit++) {
        sat_sol[lit-1] = ipasir_val(solver, lit);
    }
    // VR <- VR ^ v
    for (int lit=1; lit<=maxVar; lit++) {
        if (sat_sol[lit-1]>0) {
            if (neg_literals[lit-1] != 0) {
                neg_literals[lit-1] = 0;
                vr_upper_count--;
            }
        } else if (sat_sol[lit-1]<0) {
            if (pos_literals[lit-1] != 0) {
                pos_literals[lit-1] = 0;
                vr_upper_count--;
            }
        }
    }
    // now computing block implicant
    for (int i=0; i<maxVar; i++) {
        if (neg_literals[i] !=0)
            ipasir_add(solver, -neg_literals[i]);
        if (pos_literals[i] !=0)
            ipasir_add(solver, -pos_literals[i]);
    }
    ipasir_add(solver, 0);
}
```

```
0 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971 53972 53973 53979 53
983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015
54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 5403
3 54034 54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54
075 54077 54078 54080
c App-Solver: bin/ipasirbones1-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver            : minisat220
c SAT solver calls :    7815
c Formula Variables:   54080
c Backbone size    :   16834

real    1m9.564s
user    1m9.432s
sys     0m0.130s
```

Figure 3.2: Running Algorithm 1 on `buildroot.cnf`

3.1.2 Algorithm 2: Iterative testing - Two tests per variable

This algorithm performs an iterative loop for all variables in the model, checking in sequence both, the negative literal and the positive literal. In every SAT call (Figure 3.3), an assumption, negating the literal under evaluation is added to the current SAT stage. Once that SAT call is completed, those assumptions previously made are automatically cleared by the SAT-solver. This algorithm performs a total of $2n$ sequential calls to the SAT-solver, one for each literal of each variable. An additional performance improvement heuristic consists in adding those backbones, after they are found, as single literal clauses to the formula.

Algorithm 2: Iterative algorithm (two tests per variable)

Input : Satisfiable formula ϕ
Output: Backbone of ϕ, ν_R

$\nu_R \leftarrow \emptyset$ `// Initial backbone lower bound`
foreach $x \in \mathrm{var}(\phi)$ **do**
    $(\mathsf{outc}_1, \nu) \leftarrow \mathtt{SAT}(\phi \cup \{x\})$
    $(\mathsf{outc}_0, \nu) \leftarrow \mathtt{SAT}(\phi \cup \{\bar{x}\})$
    assert ($\mathsf{outc}_1 = \mathsf{true}$ **or** $\mathsf{outc}_0 = \mathsf{true}$)   `//` ϕ `must be satisfiable`
    if $\mathsf{outc}_1 = \mathsf{false}$ **then**
        $\nu_R \leftarrow \nu_R \cup \{\bar{x}\}$ `//` $\bar{x}$ `is backbone`
        $\phi \leftarrow \phi \cup \{\bar{x}\}$
    if $\mathsf{outc}_0 = \mathsf{false}$ **then**
        $\nu_R \leftarrow \nu_R \cup \{x\}$ `//` x `is backbone`
        $\phi \leftarrow \phi \cup \{x\}$
return ν_R

Figure 3.3: Algorithm 2 - Iterative algorithm - Two tests per variable

`genipabones.cpp` is a preliminary IPASIR implementation of this algorithm, which is available at [Balyo, 2017]. Our refactored version of `genipabones.cpp` is showed in Listing 3.2.

Listing 3.2: IPASIRBones-2

```
int* backbones = new int[maxVar];
for (int i=0; i<maxVar; i++) backbones[i] = 0;

for (int lit=1; lit<=maxVar; lit++) {

    ipasir_assume(solver, lit);
    int res1 = ipasir_solve(solver);
    satCalls++;
    if (res1 == UNSAT) {
        bbonesFound++;
        backbones[lit-1] = -lit;
        ipasir_add(solver, -lit);
        ipasir_add(solver, 0);
    }

    ipasir_assume(solver, -lit);
    int res2 = ipasir_solve(solver);
    satCalls++;
    if (res2 == UNSAT) {
        bbonesFound++;
        backbones[lit-1] = lit;
        ipasir_add(solver, lit);
        ipasir_add(solver, 0);
    }

    if (res1==UNSAT && res2==UNSAT) {
        printf("UNSAT formula (literal: %d), exiting...\n", lit);
        exit(-1);
    }

}
```

The screenshot in Figure 3.4 shows the result of running Algorithm 2 on `buildroot.cnf`. A noticeable observation with respect to Algorithm 1's execution, shown in Figure 3.2, is the higher number of SAT calls, leading to a longer execution time.

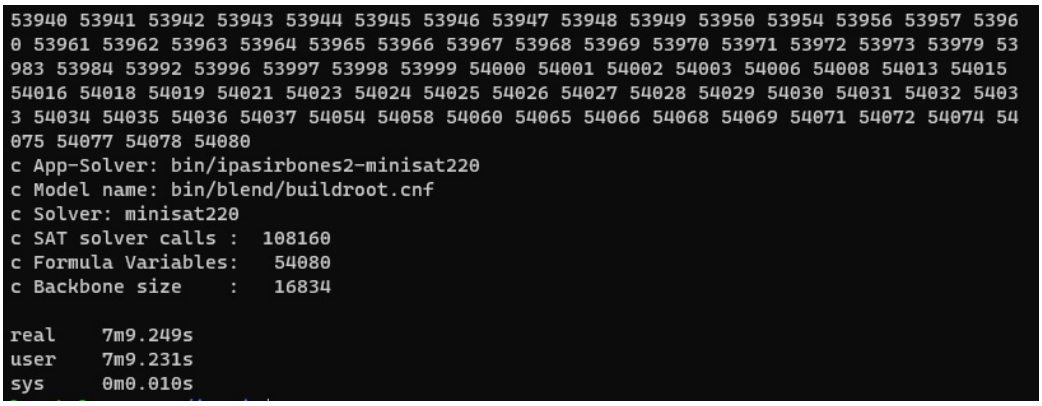

```
53940 53941 53942 53943 53944 53945 53946 53947 53948 53949 53950 53954 53956 53957 5396
0 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971 53972 53973 53979 53
983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015
54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 5403
3 54034 54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54
075 54077 54078 54080
c App-Solver: bin/ipasirbones2-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c SAT solver calls :  108160
c Formula Variables:   54080
c Backbone size    :   16834

real    7m9.249s
user    7m9.231s
sys     0m0.010s
```

Figure 3.4: Running Algorithm 2 on `buildroot.cnf`

3.1.3 Algorithm 3: Iterative testing - One test per variable

The definition of backbone itself provides a clue on how to improve the backbone computation performance by reducing the number of SAT calls: backbone variables must be present in every satisfiable solution always with the same literal. The third algorithm in Figure 3.5 takes advantage of this fact, by first computing a satisfiable solution and then performing an iterative test for each of those particular solution literals. In each step of the loop, the SAT-solver is called with the complementary of the literal (Line 6). If that instance is not satisfiable, then the literal is added to the backbone estimate (Line 8), removed from the candidate list (Line 9), and added as one unit clause to the formula (Line 10). If the SAT call is satisfiable, for each variable the literal from the current backbone estimate is checked with the solution literal. If the literals from both sides are different, then the literal is removed from the backbone estimate.

Algorithm 3: Iterative algorithm (one test per variable)

Input : Satisfiable formula ϕ
Output: Backbone of ϕ, ν_R

```
1  (outc, ν) ← SAT(φ)
2  Λ ← ν                                         // SAT tests planned
3  ν_R ← ∅                                       // Initial backbone lower bound
4  while Λ ≠ ∅ do
5      l ← pick a literal from Λ                 // Pick a literal to test
6      (outc, ν) ← SAT(φ ∪ {l̄})                  // Test if l is a backbone
7      if outc = false then
           // Backbone identified
8          ν_R ← ν_R ∪ {l}                       // Add l to the backbone estimate
9          Λ = Λ ∖ {l}                           // l does not need to be tested anymore
10         φ ← φ ∪ {l}
11     else
12         Λ ← Λ ∩ ν                             // Literal filtering
13 return ν_R
```

Figure 3.5: Algorithm 3 - One test per variable

That way, the resulting IPASIR-based code is listed at 3.3. Note that the code has been accommodated to store the backbone in a dedicated array (Line 1), while also performing a SAT call to identify an initial *upper bound* (Line 4) as done with previous algorithms. This algorithm performs a maximum of n+1 SAT calls, that is, the initial one (Line 4) to set up the upper bound plus one more for each literal of the upper bound (Line 15). This algorithm also implements *literal filtering* (Lines 23 to 28), comparing all literals from the upper backbone estimate pending to check with the current SAT solution and discarding those which are different.

Listing 3.3: IPASIRBones-3

```
1  int* backbones = new int[maxVar];
2  for (int i=0; i<maxVar; i++) backbones[i] = 0;
3
4  ipasir_solve(solver);
5  satCalls++;
6  int* sat_solution = new int[maxVar];
7  for (int lit = 1; lit <= maxVar; lit++) {
8      sat_solution[lit-1] = ipasir_val(solver, lit);
9  }
10
11 for (int i = 0; i < maxVar; i++) {
12     int candidate = sat_solution[i];
13     if (candidate == 0) continue;
14     ipasir_assume(solver, -candidate);
15     int res = ipasir_solve(solver);
16     satCalls++;
17     if (res == UNSAT) {
18         bbonesFound++;
19         backbones[i] = candidate;
20         ipasir_add(solver, candidate);
21         ipasir_add(solver, 0);
22     } else {
23         for (int lit = i+1; lit < maxVar; lit++) {
24             if ( (sat_solution[lit] != 0)
25                 && (sat_solution[lit] != ipasir_val(solver, lit+1)) ) {
26                 sat_solution[lit] = 0;
27             }
28         }
29     }
30 }
```

Figure 3.6 shows the execution of Algorithm 3 on `buildroot.cnf`. It reveals a dramatic reduction in the number of SAT calls when compared to Algorithm 2 (Figure 3.4), with 180.160 SAT calls for Algorithm 2 versus 22,158 SAT calls for Algorithm 3. This is a result of the combined effect of reducing the initial upper bound backbone estimate to half plus the literal filtering. This lower number of SAT calls also leads to shorter execution times.

```
53940 53941 53942 53943 53944 53945 53946 53947 53948 53949 53950 53954 53956 53957 5396
0 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971 53972 53973 53979 53
983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015
54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 5403
3 54034 54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54
075 54077 54078 54080
c App-Solver: bin/ipasirbones3-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c SAT solver calls :   22158
c Formula Variables:   54080
c Backbone size    :   16834

real    0m15.743s
user    0m15.732s
sys     0m0.010s
```

Figure 3.6: Running Algorithm 3 on `buildroot.cnf`

3.1.4 Algorithm 4: Iterative algorithm with the complement of backbone estimate

This algorithm, shown in Figure 3.7, also starts populating the initial backbone estimate from the solution of a first SAT call (Lines 1-2), being it the *upper bound* of the backbone. Then it loops until there are no more elements in the initial backbone estimate to test. In every loop, the SAT-solver is called adding the complement of the backbone at the time as an additional clause (Line 4). If the SAT call is not satisfiable, the current backbone estimated is returned as the backbone (Line 6). Otherwise, *literal filtering* is applied to the backbone estimate.

Listing 3.4 shows our IPASIR implementation of Algorithm 4. First, the initial backbone estimate is computed (Lines 2 to 9), followed by the iterative loop. In order to temporarily add the backbone estimate to the

Algorithm 4: Iterative algorithm with complement of backbone estimate

Input : Satisfiable formula ϕ
Output: Backbone of ϕ, ν_R

$(\mathsf{outc}, \nu) \leftarrow \mathtt{SAT}(\phi)$
$\nu_R \leftarrow \nu$ // Initial backbone estimate
while $\nu_R \neq \emptyset$ **do**
  $(\mathsf{outc}, \nu) \leftarrow \mathtt{SAT}(\phi \cup \{ \bigvee_{l \in \nu_R} \bar{l} \})$
  if $\mathsf{outc} = \mathsf{false}$ **then**
    return ν_R // Terminate if unsatisfiable
  $\nu_R \leftarrow \nu_R \cap \nu$ // Refine backbone estimate
return ν_R

Figure 3.7: Algorithm 4 - Complement of backbone estimate

formula (Lines 15 to 20), an additional variable is added to that clause (Line 19) so that variable is first assumed with the complementary literal (Line 21) before calling SAT and then it is set with the actual literal, therefore making that temporary clause always true and not affecting any later SAT-solver calculation. As with every one of these backbone complement's SAT calls a new dummy variable must be used, a `roll_back` variable (Line 11) allocates new formula variables past the actual formula variables. After the SAT call, *literal lifting* and *variable lifting* are applied.

Figure 3.8 shows the execution of Algorithm 4 on `buildroot.cnf`. A key observation is the reduced number of SAT calls (only 8,139) compared to Algorithm 3 (22,158 calls). But this fact does not help in reducing the processing time, which is approximately five times higher. This is because of the higher complexity of adding large temporary clauses and rolling them back.

Listing 3.4: IPASIRBones-4

```
// initial backbone estimate
int res = ipasir_solve(solver);
if (res == UNSAT)
    exit(-1);
satCalls++;
int* sat_solution = new int[maxVar];
for (int lit = 1; lit <= maxVar; lit++) {
    sat_solution[lit-1] = ipasir_val(solver, lit);
}

int roll_back = 1;
// looping with the complement of backbone estimate
while (true) {
    // adding backbone complement clause
    for (int i = 0; i < maxVar; i++) {
        if (sat_solution[i] != 0)
            ipasir_add(solver, -sat_solution[i]);
    }
    ipasir_add(solver, maxVar + roll_back);
    ipasir_add(solver, 0);
    ipasir_assume(solver, -(maxVar + roll_back));
    res=ipasir_solve(solver);

    if (res == UNSAT) {
        break; // terminate loop if UNSAT
    }
    // refine backbone estimate
    for (int lit = 1; lit <= maxVar; lit++) {
        if (sat_solution[lit-1] != 0) {
            if (sat_solution[lit-1] != ipasir_val(solver, lit))  {
                sat_solution[lit-1] = 0;
            }
        }
    }
    ipasir_add(solver, maxVar + roll_back);
    ipasir_add(solver, 0);
    roll_back++;
}
```

```
53940 53941 53942 53943 53944 53945 53946 53947 53948 53949 53950 53954 53956 53957 5396
0 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971 53972 53973 53979 53
983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015
54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 5403
3 54034 54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54
075 54077 54078 54080
c App-Solver: bin/ipasirbones4-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c SAT solver calls :    8139
c Formula Variables:   54080
c Backbone size    :   16834

real    1m16.573s
user    1m16.563s
sys     0m0.010s
```

Figure 3.8: Running Algorithm 4 on `buildroot.cnf`

3.1.5 Algorithm 5: Chunking

While the previous algorithm picked up one single literal, negated it, and called the SAT-solver to check satisfiability or perform literal filtering, the chunking algorithm in Figure 3.9 picks several literals (Lines 5-6), a *chunk*, negates them and calls the SAT-solver (Line 7). With the response, it performs literal filtering if SAT-solver returns satisfiable (Line 13). In the less probable event that SAT call returns unsatisfiable, then all literals in the chunk used for the call are part of the backbone (Lines 8 to 11). Note the SAT call requires temporarily adding a clause with the negation of each literal in the chunk.

Algorithm 5: Chunking algorithm

Input : Satisfiable formula ϕ, with variables X; $K \in \mathbb{N}^+$ chunk size
Output: Backbone of ϕ, ν_R

1 $(\mathsf{outc}, \nu) \leftarrow \mathtt{SAT}(\phi)$
2 $\nu_R \leftarrow \emptyset$ `// Initial backbone lower bound`
3 $\Lambda \leftarrow \nu$ `// Initial literals to test`
4 **while** $\Lambda \neq \emptyset$ **do**
5     $k \leftarrow \min(K, |\Lambda|)$
6     $\Gamma \leftarrow$ pick k literals from Λ
7     $(\mathsf{outc}, \nu) \leftarrow \mathtt{SAT}(\phi \cup \{\bigvee_{l \in \Gamma} \bar{l}\})$
8   **if** $\mathsf{outc} = \mathsf{false}$ **then**
        `// All literals in chunk are backbones`
9         $\nu_R \leftarrow \nu_R \cup \Gamma$ `// Add` Γ `to lower bound.`
10         $\Lambda \leftarrow \Lambda \setminus \Gamma$ `// Literals in` Γ `do not need to be tested anymore.`
11         $\phi \leftarrow \phi \cup \{\{l\} \mid l \in \Gamma\}$
12   **else**
13         $\Lambda \leftarrow \Lambda \cap \nu$
14 **return** ν_R

Figure 3.9: Algorithm 5 - Chunking Algorithm

Listing 3.5 shows our IPASIR implementation of Algorithm 5, which follows a similar approach to Algorithm 4 to solve the issue by calling the SAT-solver with a temporary clause consisting of the *or* of the negation of each literal in the chunk.

Listing 3.5: IPASIRBONES-5

```
int chunk_size = 100;
if (argc > 2 ) {
    chunk_size = atoi(argv[2]);
} else {
    printf("Using default chuck size: %d", chunk_size);
    printf(" => Add as command Line argument any other value.\n");
}
int roll_back = maxVar + 1;
int pos = 0;
for (int i = 0; vars_tested < maxVar; ) {
   // printf("Vars tested: %d. Rollback: %d \n", vars_tested, roll_back+1);
   roll_back++;
   for (int k = 0; (k < chunk_size) && (pos < maxVar); pos++) {
        if (sat_solution[pos] != 0) {
            ipasir_add(solver, -sat_solution[pos]);
            k++;
        }
    }
    ipasir_add(solver, roll_back);
    ipasir_add(solver, 0);
    // roll_back clause:
    ipasir_assume(solver, -roll_back);
    int res = ipasir_solve(solver);
    satCalls++;
    if (res == UNSAT) { // all literals in the chunk are backbones
        pos = i;
        // printf("UNSAT: i=%d, pos=%d, Found= %d => ", i, pos, bbonesFound);
        for (int k = 0; (k < chunk_size) && (pos < maxVar); pos++) {

            if (sat_solution[pos] != 0) {
                bbonesFound++;
                // printf("BB= %d, ", sat_solution[pos]);
                backbones[pos] = sat_solution[pos];
                ipasir_add(solver, sat_solution[pos]) ;
                ipasir_add(solver, 0);
                sat_solution[pos] = 0;
                vars_tested++;
                k++;
            }

        }
        // printf("UNSAT: i=%d, pos=%d, Found=%d, Tested=%d\n",
        //                              i, pos, bbonesFound, vars_tested);
    } else { // SAT
        // check below lit = i
        // printf("SAT: i=%d, pos=%d\n", i, pos);
        for (int lit = i; lit < maxVar; lit++) {
            if (sat_solution[lit] != 0) {
                // below includes val return 0 valid for lit and -lit
                if (sat_solution[lit] != ipasir_val(solver, lit+1))  {
                    sat_solution[lit] = 0;
                    vars_tested++;
                }
            }
        }
    }
    // rolling back:
    ipasir_add(solver, roll_back);
    ipasir_add(solver, 0);
    i = pos ;
    if (i >= maxVar) {
        i = 0;
        pos = 0;
    }
}
```

This is solved by adding a *roll-back* variable (Line 9). Negated literals in the chunk are added as a clause into the formula (Lines 15 to 20). Then, the *roll-back* variable is added (Line 21) before storing the clause into the SAT-solver (Line 22). Before performing the call, that variable is negated as done with others in the chunk (Line 24), but using an `assume`. After the SAT call, at the end of the loop, the *roll-back* variable is added to the solver (Lines 61-62) to cancel the temporarily added clause effectively.

The screenshot in Figure 3.10 shows the execution results of Algorithm 5, the chunking algorithm, using `buildroot.cnf` model. So far, this algorithm needed the lowest number of SAT calls, but the time to complete still is bigger than Algorithm 3.

```
0 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971 53972 53973 53979 53
983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015
54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 5403
3 54034 54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54
075 54077 54078 54080
c App-Solver: bin/ipasirbones5-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c SAT solver calls :    6417
c Formula Variables:   54080
c Backbone size    :   16834

real    0m27.191s
user    0m27.160s
sys     0m0.030s
```

Figure 3.10: Running Algorithm 5 on `buildroot.cnf`

3.1.6 Algorthm 6: Core-based Algorithm

Algorithm 6 in Figure 3.11 uses the idea of flipping all and each literal pending to test and adding it to the solver as a single literal clause before every SAT-solver call. This is a similar approach to Algorithm 3, iterative - one test per variable (Figure 3.5), which takes one literal each time. Core-based, instead, takes all literals pending to check (Lines 5-7). If the result is satisfiable, literal filtering is applied (Lines 8-10). Otherwise, the literal from the core is added to the backbone lower bound, removed from the pending list, and added as a unit clause to the formula (Lines 13-17). The algorithm includes a provision in case SAT-solver is not able to properly identify the core (Lines 18-20).

Algorithm 6: Core-based Algorithm

Input : Satisfiable formula ϕ
Output: Backbone of ϕ, ν_R

1 $(\mathsf{outc}, \nu, C) \leftarrow \mathtt{SAT}(\phi)$
2 $\nu_R \leftarrow \emptyset$ `// Initial backbone lower bound`
3 $\Lambda \leftarrow \nu$ `// Initial literals to test`
4 **while** $\Lambda \neq \emptyset$ **do**
5     $\omega_N \leftarrow \{\bar{l} \mid l \in \Lambda\}$
6   **while** true **do**
7         $(\mathsf{outc}, \nu, C) \leftarrow \mathtt{SAT}(\phi \cup \{\, \{\, l \,\} \mid l \in \omega_N \})$
8   **if** $\mathsf{outc} = \mathsf{true}$ **then**
9             $\Lambda \leftarrow \Lambda \cap \nu$
10   **break** `// Move onto a different set of literals to flip`
11   **else**
12   assert$(C \cap \omega_N \neq \emptyset)$ `//` ϕ `must be satisfiable`
13   **if** $C \cap \omega_N = \{\, l \,\}$ **then**
                `// The core contains a single literal from` ω_N
14                 $\nu_R \leftarrow \nu_R \cup \{\, \bar{l} \,\}$
15                 $\Lambda \leftarrow \Lambda \setminus \{\, \bar{l} \,\}$
16                 $\phi \leftarrow \phi \cup \{\, \bar{l} \,\}$
17             $\omega_N \leftarrow \{p \mid p \in \omega_N \wedge \{p\} \notin C\}$ `// Remove from` ω_N `literals that appear in the core`
18   **if** $\omega_N = \emptyset$ **then**
19   test literals in Λ by another algorithm
20   **return** ν_R
21 **return** ν_R

Figure 3.11: Algorithm 6 - Core based

To make no modification to the formula/model, then that variable has to be added as a single-clause literal to the formula to make sure that added clause is always true and does not make any change in the formula. This method has two main issues:

- It requires more SAT calls, more complicated because of the number of clauses

- It makes SAT computations harder, as those added fake clauses might require more effort from SAT-solver.

However, this approach does not achieve good performance, as we will see in Chapter 4. Additionally, all SAT-solvers tested with IPASIR returned only one conflicting literal from the core, which limits the possibilities for performance improvement. Listing 3.6 shows our IPASIR implementation.

Note two IPASIR function calls to `ipasir_failed` and `ipasir_add` near in the code. According to IPASIR, `ipasir_failed` can only be called when SAT-solver is in UNSAT state, which is the case in the code (Line 34), but a call to `ipasir_add`, would change that state. This is why the loop is broken at Line 53 after the first failed literal has been found (Line 47). *ipasir_failed* return value in other SAT stages is not specified otherwise.

The screenshot in Figure 3.12 shows the execution results of Algorithm 6 on `buildroot.cnf`. It involves a higher number of SAT calls (22,158) when compared to other algorithms, even worse than Algorithm 3.2, which took 1 minute and 9.56 seconds for 7,815 SAT calls.

Listing 3.6: IPASIRBones-6

```
int* backbones = new int[maxVar];
for (int b=0; b<maxVar; b++) {
    backbones[b] = 0;
}

int res = ipasir_solve(solver);
satCalls++;
int* sat_solution = new int[maxVar];
for (int lit = 1; lit <= maxVar; lit++) {
    sat_solution[lit-1] = ipasir_val(solver, lit);
}


////////////////////////////////////////////

for (int i = 0; i < maxVar; i++) {
    if (sat_solution[i] == 0) continue;

    for (int j=i; j < maxVar; j++) {
        if (sat_solution[i] != 0)
            ipasir_assume(solver, -sat_solution[i]);
    }

    int res = ipasir_solve(solver);
    satCalls++;
    if (res == SAT) {
        for (int lit = i+1; lit < maxVar; lit++) {
            if (sat_solution[lit] != 0) {
                if (sat_solution[lit] != ipasir_val(solver, lit+1))  {
                    sat_solution[lit] = 0;
                }
            }
        }
    } else {
        // checking for core. IPASIR returns 1
        for (int lit = i; lit < maxVar; lit++) {
            if ( sat_solution[lit] != 0) {
                if ( ipasir_failed(solver, lit+1)==1 ) {
                    printf("%d => %d => %d\n", i, lit, ipasir_failed(solver, lit+1));
                }
            }
        }
        for (int lit = i; lit < maxVar; lit++) {
            if ( sat_solution[lit] != 0) {
                // printf("Pos= %6d, Lit=%6d, Failed=%2d, Value=%6d\n",
                //            i, lit, ipasir_failed(solver, lit+1), sat_solution[lit] );
                if ( ipasir_failed(solver, lit+1)==1 ) {
                    bbonesFound++;
                    backbones[lit] = sat_solution[lit];
                    ipasir_add(solver, sat_solution[lit]);
                    ipasir_add(solver, 0);
                    sat_solution[lit] = 0;

                    break;
                }
            }
        }
    }
}
```

```
0 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971 53972 53973 53979 53
983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015
54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 5403
3 54034 54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54
075 54077 54078 54080
c App-Solver: bin/ipasirbones6-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c SAT solver calls :    22158
c Formula Variables:    54080
c Backbone size     :   16834

real    0m17.909s
user    0m17.898s
sys     0m0.010s
```

Figure 3.12: Running Algorithm 6 on `buildroot.cnf`

3.1.7 Algorthm 7: Core-based Algorithm with Chunking

Algorithm 7 in Figure 3.13 is basically a mix of Algorithm 5 and Algorithm 6, so instead of flipping all pending literals at once, it only flips a fixed

Algorithm 7: Core-based Algorithm with Chunking

Input : Satisfiable formula ϕ; $K \in \mathbb{N}^+$ chunk size
Output: Backbone of ϕ, ν_R

```
(outc, ν, C) ← SAT(φ)
ν_R ← ∅                                   // Initial backbone lower bound
Λ ← ν                                     // Initial literals to test
while Λ ≠ ∅ do
    k ← min(K, |Λ|)
    Γ ← pick k literals from Λ
    ω_N ← {l̄ | l ∈ Γ}
    while true do
        (outc, ν, C) ← SAT(φ ∪ { {l} | l ∈ ω_N })
        if outc = true then
            Λ ← Λ ∩ ν
            break // Done with the chunk
        else
            if C ∩ ω_N = {l} then
                // The core contains a single literal from ω_N.
                ν_R ← ν_R ∪ {l̄}
                Λ ← Λ ∖ {l̄}
                φ ← φ ∪ {l̄}
            ω_N ← {p | p ∈ ω_N ∧ {p} ∉ C}     // Remove from ω_N literals that appear in the core.
            if ω_N = ∅ then
                test literals in Γ by another algorithm
                Λ = Λ ∖ Γ
                break // Done with the chunk
return ν_R
```

Figure 3.13: Algorithm 7 - Core based with chunking

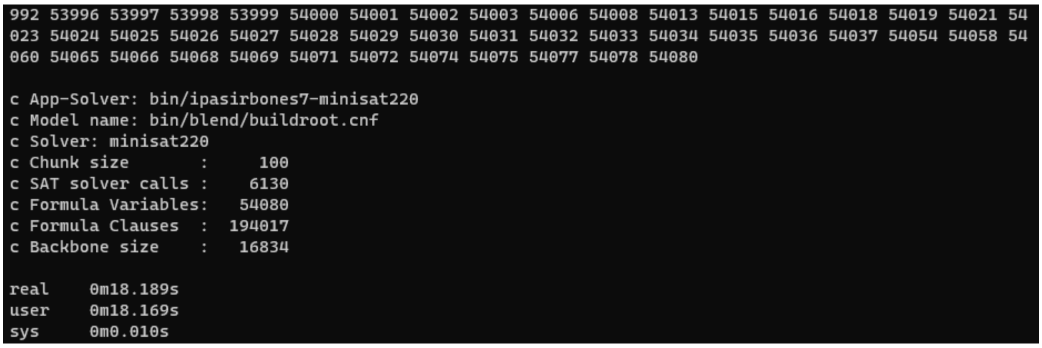
amount of them (Lines 5 and 6), therefore also working with chunks as Algorithm 5.

Listing 3.7 shows our IPASIR implementation, requiring a roll back clause so all literals in the block can be negated and added as single clause *or* clause, and later this clause can be deactivated with an *assume* call.

Listing 3.7: IPASIRBones-7

```
int chunk_size = 100;
int pending = maxVar;
if (argc > 2 ) {
    chunk_size = atoi(argv[2]);
    printf("Using supplied chunk size: %d\n", chunk_size);
} else {
    printf("Using default chunk size: %d\n", chunk_size);
}
int roll_back = 1;
while (pending != 0) {
    for (int i = 0; i < maxVar; i++) {
        if (sat_solution[i] == 0) continue;
        for (int lit = i; (lit<maxVar) && (lit<i+chunk_size); lit++) {
            if (sat_solution[lit] != 0)
                ipasir_add(solver, -sat_solution[lit]);
        }
        ipasir_add(solver, maxVar + roll_back);
        ipasir_add(solver, 0);
        ipasir_assume(solver, -(maxVar + roll_back));
        int res = ipasir_solve(solver);
        satCalls++;
        if (res == SAT) {
            for (int lit = 0; lit<maxVar; lit++) {
                if (sat_solution[lit] != 0) {
                    if (sat_solution[lit] != ipasir_val(solver, lit+1))  {
                        sat_solution[lit] = 0;
                        pending--;
                    }
                }
            }
        } else {
            for (int lit=i; (lit<maxVar) && (lit<i+chunk_size); lit++) {
                if ( sat_solution[lit] != 0) {
                    bbonesFound++;
                    backbones[lit] = sat_solution[lit];
                    ipasir_add(solver, sat_solution[lit]);
                    ipasir_add(solver, 0);
                    sat_solution[lit] = 0;
                    pending--;
                }
            }
        }
        ipasir_add(solver, maxVar + roll_back);
        ipasir_add(solver, 0);
        roll_back++;
    }
}
```

The screenshot in Figure 3.14 shows the execution results of Algorithm 7, the core-based with chunking algorithm, using `buildroot.cnf` model. Note that chunk size has been set to 100 since this is the default value in minibones [Janota et al., 2015] and EDUCIBones [Zhang et al., 2020].



```
992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015 54016 54018 54019 54021 54
023 54024 54025 54026 54027 54028 54029 54030 54031 54032 54033 54034 54035 54036 54037 54054 54058 54
060 54065 54066 54068 54069 54071 54072 54074 54075 54077 54078 54080

c App-Solver: bin/ipasirbones7-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c Chunk size        :     100
c SAT solver calls :    6130
c Formula Variables:   54080
c Formula Clauses  :  194017
c Backbone size    :   16834

real    0m18.189s
user    0m18.169s
sys     0m0.010s
```

Figure 3.14: Running Algorithm 7 on `buildroot.cnf`

3.2 Heuristics

This section describes several heuristics to improve Algorithms 1-7 performance. Although some of them were already introduced in the previous section, they are not bound to any specific algorithm, and thus they could be used in new algorithms. These heuristics target at performing **backbone filtering**, equivalently *implicant reduction*, that is, identifying variables or literals which are not backbone candidates and can be skipped during testing, so the number of SAT calls and computation effort in evaluating them is reduced. Examples described below are the literal filtering [Janota et al., 2015], and also the identification of one-literal clauses during the CNF/DIMACS file load as backbones, which is a heuristic we have not found in the backbone literature.

The insertion of the backbone into the formula is not properly a reduction of the implicant size nor does identify a backbone, but it does improve performance.

3.2.1 Insertion of the backbone into the formula

A heuristic to speed up backbone computation is inserting the backbone literal into the formula (SAT-solver object) once it has been found as such (Listing 3.8). This is done by calling `ipasir_add` function with the newly identified backbone literal first and then calling again with 0. This will have the same effect as adding a clause to the CNF formula of the DIMACS file only composed of the literal number plus the zero:

Listing 3.8: Heuristic: Adding backbones to the formula

```
if (res == UNSAT) {
    bbonesFound++;
    backbones[i] = new_backbone_literal;
    ipasir_add(solver, new_backbone_literal);
    ipasir_add(solver, 0);
}
```

3.2.2 Literal filtering

During the iterative process of checking if each literal is in the backbone or not, the SAT-solver is called in every loop. When the result is satisfiable, a new solution is available, which might be different from the ones obtained before. Checking literals from that new solution and comparing them to the existing upper bound will help reduce the number of checks. If the literal obtained for the new satisfiable solution is different from the literals obtained in previous satisfiable solutions (upper bound) then that variable cannot be part of the backbone. Listing 3.9 shows an IPASIR-based code. Note that this check loop is only needed for variables not yet processed in order to improve performance, as the goal is to identify which ones of the pending variables are backbone candidates. Variables found as not a valid candidate for the backbone are identified with a 0 value, meaning it can be skipped during further variable checks, therefore saving a SAT call in that case.

Listing 3.9: Heuristic: Code for literal filtering

```
for (int lit = i+1; lit < maxVar; lit++) {
    if ( (sat_solution[lit] != 0) &&
        (sat_solution[lit] != ipasir_val(solver, lit+1)) ) {
            sat_solution[lit] = 0;
    }
}
```

For this heuristics to work, a `sat_solution` array is kept, which stores the results of the first satisfiable call performed. Then, after every SAT call with satisfiable result, variables are checked and updated to 0 when they are no longer backbone candidates (upper bound).

3.2.3 Identification of one-literal clauses

When reading the CNF/DIMACS file, identify those clauses consisting of a single literal. Therefore they are part of the backbone (if the formula/-model is satisfiable), so no need to make any checks with them. Adding those literals to the formula will identify backbones beforehand without performing any SAT call (Listing 3.10). The empirical analysis of formulas for configuration models shows a high percentage of backbone literals appearing as one-literal clauses in the original formula. For example, the *`buildroot.cnf`* model, used in the previous section to illustrate the algorithms' execution, has 16.783 unary clauses out of a total of 16.834 backbone literals. Additional model analysis is provided at table 3.1.

Listing 3.10: Heuristic: Identification of one-literal clauses from CNF/DIMACS file

```
// add to the solver
ipasir_add(solver, num);
if (num==0) {
    if (numcount==1) {
        // a clause with only one literal, then it is a backbone
        bblist.push_back(lastnum);
    }
    numcount=0;
} else {
    lastnum = num;
    numcount++;
}
```

3.2.4 Cascading CNF literals

A step forward from the previous heuristics is to perform a cascaded analysis of the literals of the CNF formula as they are read from the DIMACS file. The process will consist of several loops, performing the following tasks until no change is made in a loop:

- Select a clause.
- If the clause has a single literal clause, add the literal to the backbone list.
- If the clause has several literals, for each one check if the complementary literal is a backbone.
- If all complementary literals are backbones except one, add that literal to backbone.

This heuristic was coded in a Ruby script (Listing 3.11) to perform an empirical evaluation of its impact.

Listing 3.11: Cascading Literals

```
Dir.glob('*.cnf') do |dimacs|
model_time = Time.now
input_lines = File.read(dimacs)
puts "Model: #{dimacs}"
backbones = Array.new(0)
bb_count = 0
bb_candidates_count = 0
bb_candidate = 0
num_loop = 0

while true
    new_backbones = 0
    num_loop += 1
    input_lines.each_line do |line|
      if line =~ /^[c]/
        # skip
      elsif line =~ /^p cnf/
        problem = line.split(" ")
        num_vars = problem[2]
        num_clauses = problem[3]
        if num_vars == 0
          puts("Num cnf variables: #{num_vars}")
          puts("Num cnf clauses  : #{num_clauses}")
        end
      else
        literals = line.split(" ")
        literals.each do |lit|
          int_lit = lit.to_i
          if int_lit != 0
            if backbones[int_lit.abs].nil? and bb_candidates_count==0
              bb_candidate = int_lit
              bb_candidates_count += 1
            elsif  backbones[int_lit.abs] == int_lit*(-1)
              # this candidate is the negation of a backbone, good to go
            elsif (bb_candidates_count > 0) and backbones[int_lit.abs].nil?
              break   # more than one candidate, not useful...
            else
              break   # this lit is already in backbone
            end
          elsif int_lit == 0
            if bb_candidates_count == 1
              backbones[bb_candidate.abs] = bb_candidate
              bb_count += 1
              new_backbones += 1
            end
          end
        end
        bb_candidates_count = 0
        bb_candidate = 0
      end
    end
    puts "Loop #{num_loop} => #{new_backbones} backbones found."
    break if new_backbones == 0
end
puts "Backbone count: #{bb_count}"
puts "Processing time: #{Time.now - model_time}s."
backbones.each { |bb| print "#{bb} " unless bb.nil? }
print "\n"
end
```

Table 3.1 analyzes the result of identifying backbones directly from the CNF/DIMACS file at the time of file reading. [Fernandez-Amoros et al., 2023] used this model set in configuration management and software engineering domain.

Table 3.1: Direct backbone identification from CNF formula

Model	Backbone Size	One-lit Clauses	Cascading CNF Literals				
		Backbones	Loop 1	Loop 2	Loop 3	Backbones	Time (s.)
axtls	127	127	127	0	0	127	0,0120
buildroot	16.834	16783	16.790	0	0	16.790	1,4350
busybox	762	713	714	0	0	714	0,1018
coreboot	20.966	16012	16.020	0	0	16.020	2,7228
embtoolkit	4.422	4383	4.389	0	0	4.389	0,5606
fiasco	111	93	93	0	0	93	0,0064
freetz	10.093	9504	9.504	0	0	9.504	1,8116
linux	27.239	23368	22.306	7	0	22.313	6,1762
toybox	74	74	74	0	0	74	0,0054
uClibc	383	381	383	0	0	383	0,0322

3.2.5 Coding and performance

Despite the improvement obtained by the different algorithms and the heuristics above, some authors have also identified other means to improve performance. [Mitchell, 2005] identified improvements factors in the range from 3 to 8 by using cache aware implementations. Some directions provided are:

- Reduce the memory footprint
- Use arrays instead of pointers.
- Store data in memory in the same sequence it will be accessed.

Our IPASIR implementation follows these directions and implements required data structures in arrays instead of C++ vectors, which make an extensive use of pointer. In addition, those data structures will be later accessed in a sequential way.

3.3 Tweaking the Algorithms

This section provides improved versions of the algorithms in Section 3.1. While the algorithms themselves are not changing dramatically, the heuristics and code enhancements significantly reduce the computing time:

- Identification as backbones all those literals from clauses with that one literal. This is done while reading the source CNF/DIMACS file.
- Backbone insertion: Adding backbones as a single literal clause to the formula when it has been identified as such after the SAT call returns.
- Literal lifting: After a satisfiable SAT-solver call, compare all variables pending for backbone checking with the results of the SAT-solver. If, for a given variable, its literal from the last SAT-solver solution differs from the literal from the initial SAT solution, any of the two literals for that variable can be part of the backbone.

3.3.1 Enhancing Algorithm 3 - Version a

This version of Algorithm 3 includes the three following heuristics (Listing 3.12):

Listing 3.12: IPASIRBones-3a

```
int* backbones = new int[maxVar];
for (int i=0; i<maxVar; i++) backbones[i] = 0;

ipasir_solve(solver);
satCalls++;
int* sat_solution = new int[maxVar];
for (int lit = 1; lit <= maxVar; lit++) {
    sat_solution[lit-1] = ipasir_val(solver, lit);
}

for (size_t b=0; b<bblist.size(); b++) {
    if (backbones[abs(bblist[b])-1] == 0) {
        bbonesFound++;
        backbones[abs(bblist[b])-1] = bblist[b];
        sat_solution[abs(bblist[b])-1] = 0;
    }
}
printf("\nc Initializing %d unary clauses as backbones\n", bbonesFound);

for (int i = 0; i < maxVar; i++) {
    int candidate = sat_solution[i];
    if (candidate == 0) continue;
    ipasir_assume(solver, -candidate);
    int res = ipasir_solve(solver);
    satCalls++;
    if (res == UNSAT) {
        bbonesFound++;
        backbones[i] = candidate;
        ipasir_add(solver, candidate);
        ipasir_add(solver, 0);
    } else {
        for (int lit = i+1; lit < maxVar; lit++) {
            if ( (sat_solution[lit] != 0) && (sat_solution[lit]
                != ipasir_val(solver, lit+1)) ) {
                sat_solution[lit] = 0;
            }
        }
    }
}
```

Figure 3.15 shows a notable reduction of SAT calls' (5.375 from 22.158) when compared to Algorithm 3 (Listing 3.3). While these SAT calls seem to be easy, as it is reflected only with a small execution time reduction ($\sim -1s.$). As these are only preliminary evaluations, a complete analysis using other solvers will be done in Chapter 4.

```
0 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971 53972 53973 53979 53
983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013 54015
54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 5403
3 54034 54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54
075 54077 54078 54080
c App-Solver: bin/ipasirbones3a-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c SAT solver calls :    5375
c Formula Variables:   54080
c Backbone size    :   16834

real    0m14.550s
user    0m14.549s
sys     0m0.000s
```

Figure 3.15: Execution of Algorithm 3a with `buildroot.cnf` model

3.3.2 Enhancing Algorithm 7 - Version a

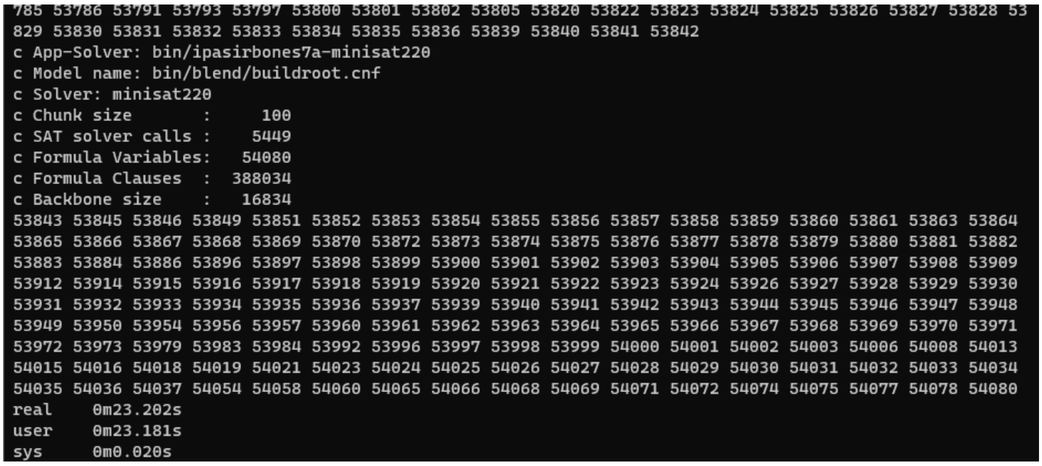
Algorithm 7 can be tweaked in the same way that Algorithm 3 (3.5): adding detection of backbones by identifying one-literal clauses at the time of reading the DIMACS file. Listing 3.13 shows this enhanced version.

In this case, our preliminary evaluations (Figure 3.16) do not show any time improvement when used with the *minisat220* SAT-solver.

Interestingly, when the *cadicalsc2020* SAT-solver is used, the time reduction is dramatic (Figure 3.17).

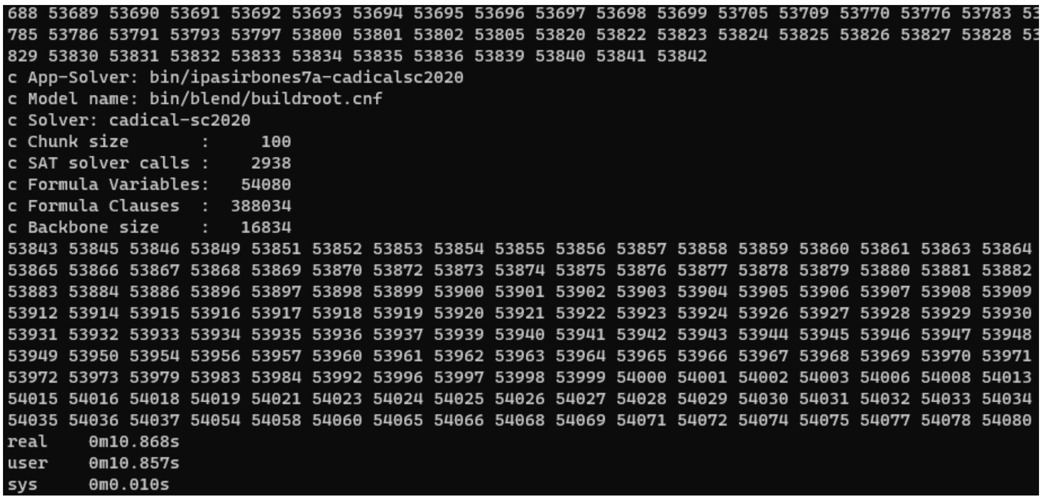
Listing 3.13: IPASIRBones-7a

```
for (size_t b=0; b<bblist.size(); b++) {
    if (backbones[abs(bblist[b])-1] == 0) {
        bbonesFound++;
        pending--;
        backbones[abs(bblist[b])-1] = bblist[b];
        sat_solution[abs(bblist[b])-1] = 0;
    }
}
printf("\nc Initializing %d unary clauses as backbones\n", bbonesFound);

while (pending != 0) {
    for (int i = 0; i < maxVar; i++) {
        if (sat_solution[i] == 0) continue;

        for (int lit = i; (lit<maxVar) && (lit<i+chunk_size); lit++) {
            if (sat_solution[lit] != 0)
                ipasir_add(solver, -sat_solution[lit]);
        }
        ipasir_add(solver, maxVar + roll_back);
        ipasir_add(solver, 0);
        ipasir_assume(solver, -(maxVar + roll_back));

        int res = ipasir_solve(solver);
        satCalls++;
        if (res == SAT) {
            for (int lit = 0; lit<maxVar; lit++) {
                if (sat_solution[lit] != 0) {
                    if (sat_solution[lit] != ipasir_val(solver, lit+1))  {
                        sat_solution[lit] = 0;
                        pending--;

                    }
                }
            }
        } else {
            for (int lit=i; (lit<maxVar) && (lit<i+chunk_size); lit++) {
                if ( sat_solution[lit] != 0) {
                    bbonesFound++;
                    backbones[lit] = sat_solution[lit];
                    ipasir_add(solver, sat_solution[lit]);
                    ipasir_add(solver, 0);
                    sat_solution[lit] = 0;
                    pending--;
                }
            }
        }
        ipasir_add(solver, maxVar + roll_back);
        ipasir_add(solver, 0);
        roll_back++;
    }
}
```



```
785 53786 53791 53793 53797 53800 53801 53802 53805 53820 53822 53823 53824 53825 53826 53827 53828 53
829 53830 53831 53832 53833 53834 53835 53836 53839 53840 53841 53842
c App-Solver: bin/ipasirbones7a-minisat220
c Model name: bin/blend/buildroot.cnf
c Solver: minisat220
c Chunk size       :     100
c SAT solver calls :    5449
c Formula Variables:   54080
c Formula Clauses  :  388034
c Backbone size    :   16834
53843 53845 53846 53849 53851 53852 53853 53854 53855 53856 53857 53858 53859 53860 53861 53863 53864
53865 53866 53867 53868 53869 53870 53872 53873 53874 53875 53876 53877 53878 53879 53880 53881 53882
53883 53884 53886 53896 53897 53898 53899 53900 53901 53902 53903 53904 53905 53906 53907 53908 53909
53912 53914 53915 53916 53917 53918 53919 53920 53921 53922 53923 53924 53926 53927 53928 53929 53930
53931 53932 53933 53934 53935 53936 53937 53939 53940 53941 53942 53943 53944 53945 53946 53947 53948
53949 53950 53954 53956 53957 53960 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971
53972 53973 53979 53983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013
54015 54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 54033 54034
54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54075 54077 54078 54080
real    0m23.202s
user    0m23.181s
sys     0m0.020s
```

Figure 3.16: Execution of Algorithm 7a with `buildroot.cnf` model



```
688 53689 53690 53691 53692 53693 53694 53695 53696 53697 53698 53699 53705 53709 53770 53776 53783 53
785 53786 53791 53793 53797 53800 53801 53802 53805 53820 53822 53823 53824 53825 53826 53827 53828 53
829 53830 53831 53832 53833 53834 53835 53836 53839 53840 53841 53842
c App-Solver: bin/ipasirbones7a-cadicalsc2020
c Model name: bin/blend/buildroot.cnf
c Solver: cadical-sc2020
c Chunk size       :     100
c SAT solver calls :    2938
c Formula Variables:   54080
c Formula Clauses  :  388034
c Backbone size    :   16834
53843 53845 53846 53849 53851 53852 53853 53854 53855 53856 53857 53858 53859 53860 53861 53863 53864
53865 53866 53867 53868 53869 53870 53872 53873 53874 53875 53876 53877 53878 53879 53880 53881 53882
53883 53884 53886 53896 53897 53898 53899 53900 53901 53902 53903 53904 53905 53906 53907 53908 53909
53912 53914 53915 53916 53917 53918 53919 53920 53921 53922 53923 53924 53926 53927 53928 53929 53930
53931 53932 53933 53934 53935 53936 53937 53939 53940 53941 53942 53943 53944 53945 53946 53947 53948
53949 53950 53954 53956 53957 53960 53961 53962 53963 53964 53965 53966 53967 53968 53969 53970 53971
53972 53973 53979 53983 53984 53992 53996 53997 53998 53999 54000 54001 54002 54003 54006 54008 54013
54015 54016 54018 54019 54021 54023 54024 54025 54026 54027 54028 54029 54030 54031 54032 54033 54034
54035 54036 54037 54054 54058 54060 54065 54066 54068 54069 54071 54072 54074 54075 54077 54078 54080
real    0m10.868s
user    0m10.857s
sys     0m0.010s
```

Figure 3.17: Execution of algorithm 7a (cadicalsc2020 solver) with `buildroot.cnf` model

4

Chapter 4: Experimental Validation

This chapter reports an in-depth empirical evaluation of our IPASIR programs, presented in Chapter 3. First, Section 4.2 evaluates each IPASIR program with a variety of SAT-solvers, thus identifying (i) what solver works best for each program, and (ii) what program/solver has the best performance. Later, Section 4.3 compares our best program/solver with two state-of-the-art backbone detection tools: *minibones* [Janota et al., 2015] and *EDUCIBone* [Zhang et al., 2020].

4.1 Experimental Setup

Our evaluation targets two Research Questions:

RQ1: Best IPASIRBONES/SAT combination *What combination of IPASIRBONES program and SAT-solver achieves the best time performance?*

RQ2: IPASIRBONES *vs.* state-of-the-art tools *What is the IPASIRBONES' time performance compared to minibones and EDUCIBone?*

To do so, we started developing our IPASIRBONES' prototypes on a PC. The IPASIR environment was set up using the distribution available from [Balyo, 2017], which is the one used in most SAT competitions. As this distribution is Linux-based, it was installed in an Ubuntu instance of the

Windows Subsystem for Linux 2 (WSL2), running under Windows 11. Once the prototypes were tested in a PC, their performance was evaluated in a cluster provided by the UNED GISS[1] research group, which is equipped with an Intel^{TM} Xeon^{TM} CPU E5-2660 v4 2.00GHz with 28 physical cores with 2 threads each one and 220.3 GiB of available RAM memory for the operating system, an Ubuntu Release 20.04.5 LTS 64-bit with Kernel Linux 5.4.0-135 generic x86_64. Note that the captures and values used in Chapter 3 were taken from the development machine, whereas the captures and values in this current chapter were taken from the GISS cluster.

Our benchmark was composed of two sets of configuration models taken from relevant literature on software engineering and software product lines:

1. MIG: 116 configuration models proposed in [Krieter et al., 2018] [Krieter et al., 2021], and also used in [Plazar et al., 2019].
2. FA: 10 configuration models provided in [Fernandez-Amoros et al., 2023].

The standard IPASIR distribution includes, by default, interfaces with the following SAT-solvers:

- *lingelingbcj*
- *minisat220*
- *picosat961*

Additionally, the following SAT-solver interfaces were selected for evaluation:

[1] http://www.issi.uned.es/giss

- From SAT Competition 2020, *cadicalsc2020* [Biere et al., 2020]
- From SAT Competition 2017, *glucose4* [Audemard and Simon, 2017]

abcdsat_i20 [Balyo et al., 2020] was also evaluated, but after some preliminary executions, we noticed its performance with IPASIR was out of range when compared to the other solvers. Table 4.1 shows the average execution time (100 loops) of IPASIRBones-3 for the MIG set of models using the different SAT-solvers. As a result, *abcdsat_i20* was discarded.

Table 4.1: abcdsat_i20 compared to other SAT-solvers with IPASIRBones-3

Program	SAT-solver	Average Time (s.)
IPASIRBones-3	abcdsat_i20	79.87948
IPASIRBones-3	cadicalsc2020	0.01967
IPASIRBones-3	glucose4	0.08300
IPASIRBones-3	lingelingbcj	0.17038
IPASIRBones-3	minisat220	0.07710
IPASIRBones-3	picosat961	0.22023

The actual backbone computation was managed via an R script (Listing 4.1) , which reads a configuration file indicating the number of loops (executions of individual backbone programs solver and model combinations), the source path for the backbone programs and the source path for the model set. Note that all the loops for the 5 backbone programs for a particular algorithm were executed with a single script call on all the models from the provided model set. The computation part of every model-backbone program combination, taking advantage of the high number of cores of the computer, was performed in parallel, therefore saving time during evaluation. The time elapsed for both MIG and FA model sets was actually more than 10 times smaller than the overall total

CPU CORE time in all algorithms. Executions for minibones and EDUCIBone had a lower parallel multiplier due to they were only two solvers in the set while each algorithm had five SAT-solvers to run. Table 4.2 shows the elapsed time for each algorithm, the aggregated CPU time used for computing the backbones during that elapsed time, and the multiplier for those timings.

Listing 4.1: run_tests.R

```
library(doParallel)
library(foreach)
library(iterators)
library(tidyverse)

print(str_c("Started: ", Sys.time()))
# Reading configuration settings from file
args = commandArgs(trailingOnly=TRUE)
if (is.na(args[1])) {
   stop("Missing configuration file!")
} else {
    # Read configuration file
    config_str <- read_file(args[1])
    num_loops <- str_extract(config_str, "num_loops\\s*=\\s*(\\d+)", group=1)
    cpu_cores <- str_extract(config_str, "cpu_cores\\s*=\\s*(\\d+)", group=1)
    model_path <- str_trim(str_extract(
      config_str, "model_path\\s*=\\s*(.+)(\\s*\\n)", group=1))
    solver_path <- str_trim(str_extract(
      config_str, "solver_path\\s*=\\s*(.+)(\\s*\\n)", group=1))
    num_loops <- as.numeric(num_loops)
    cpu_cores <- as.numeric(cpu_cores)
}

# Function processing and getting time/results from IPASIRBones
get_run_time <- function(
    solver = ""   # backbones-SAT-solver executable, include full path
    , model = ""          # model file name, include .cnf or .dimacs extension
    , arg_str = ""        # argument string
    , solver_path = ""    # path to solver executable
    , model_path =  ""    # path to model file
    , get_cmd_out = FALSE  # return all output from command

  ) {
  # Optimized for Linux. Check: decimal point in "real time" and bash/cmd:
  runcmd <- str_c( "-c \"time ", solver_path, "/", solver, arg_str, " "
                 , model_path, "/", model, "> /dev/null \"" )
  # cat("Running: ", model, "with:",  runcmd, "\n")
  system2_out <- system2("bash", runcmd, stdout=FALSE, stderr=TRUE)
  cmd_out <- str_flatten(system2_out, collapse="\n")
  mins <- str_extract(cmd_out, "real\t(\\d+)m(\\d+)\\,(\\d+)s", group=1)
  secs <- str_extract(cmd_out, "real\t(\\d+)m(\\d+)\\,(\\d+)s", group=2)
  millis <- str_extract(cmd_out, "real\t(\\d+)m(\\d+)\\,(\\d+)s", group=3)
  millis <- as.numeric(mins) * 60 + as.numeric(secs) + as.numeric(millis) / 1000
  model_vars <- str_extract(cmd_out
                           , "(c Formula Variables\\s*:\\s*)(\\d+)", group=2)
  model_clauses <- str_extract(cmd_out
                              , "(c Formula Clauses\\s*:\\s*)(\\d+)", group=2)
  sat_calls <- str_extract(cmd_out
                          , "(c SAT-solver calls\\s*:\\s*)(\\d+)", group=2)
```

```
  backbone_size <- str_extract(cmd_out
                             , "(c Backbone size\\s*:\\s*)(\\d+)", group=2)
  list("millis"=as.numeric(millis)
        , "solver" = solver
        , "model" = model
        , "sat"=attributes(system2_out)$status
        , "sat_calls" = as.numeric(sat_calls)
        , "variables"= as.numeric(model_vars)
        , "clauses"  = as.numeric(model_clauses)
        , "backbones"= as.numeric(backbone_size)
        , "cmd_out" = if (get_cmd_out) cmd_out else ""
  )
}

######################## main ##########################

out_file <- str_replace(args[1], ".cfg$", ".csv")
process_time <- 0

# starting paralell setup
registerDoParallel(cpu_cores, cores=cpu_cores)

cat("\n")
cat("Executing ", num_loops, "loops\n")
cat("Model folder: ", model_path, "\n")
cat("Solver folder: ", solver_path, "\n")
cat("CPU cores. Req/Act: ", cpu_cores, "/", getDoParWorkers(), "\n")
cat("Backend name/vers: ", getDoParName(), " ", getDoParVersion(), "\n")
cat("\n")

# Headers for csv file
results <- tibble(solvername= "", modelname="", numvars=0, numclauses=0
                 , arguments="" , chunksize = 0, milliseconds=0, backbones=0
                 , cmd_out= "", .rows=0)
write_csv(results, out_file, append=FALSE, col_names=TRUE)

# preparing iterators
models <- list.files(path=model_path, full.names= FALSE, recursive = TRUE)
solvers <- list.files(path=solver_path, full.names= FALSE, recursive = TRUE)

# main loop
for (solver_name in solvers) {
    cat("Running solver>", solver_name, "...\n")
    for (model in models) {
        results <- foreach(s=1:num_loops, .combine=rbind
                           , .packages=c('stringr', 'glue')) %dopar% {
            res <- get_run_time(solver=solver_name, model=model
                                , "", solver_path, model_path)
            data.frame(res[["solver"]], res[["model"]],
                       res[["variables"]], res[["clauses"]], ""
                       , 0, res[["millis"]], res[["backbones"]], res[["cmd_out"]])
        }
        process_time <<- process_time + sum(results$milliseconds)
        write_csv(results, out_file, append=TRUE, col_names=FALSE)
        TRUE
    }
}
warnings()
print(str_c("Ended: ", Sys.time()))
```

Table 4.2: Elapsed computing time vs. CPU CORE Time (seconds)

Algorithm	Elapsed time	CPU CORE Time	Parallel multiplier
IPASIRBones-1	19.335	203.096	10,50
IPASIRBones-2	79.088	805.557	10,19
IPASIRBones-3	2.732	30.432	11,14
IPASIRBones-4	16.104	171.561	10,65
IPASIRBones-5	4.113	48.039	11,68
IPASIRBones-6	2.976	33.057	11,11
IPASIRBones-7	2.859	33.282	11,64
IPASIRBones-3a	2.633	29.574	11,23
IPASIRBones-7a	2.832	33.074	11,68
minibones + EDUCIBone	1.943	5.796	2,98

4.2 RQ1: Best IPASIRBones/SAT combination

This section addresses the individual performance of each IPASIRBones program by using all five SAT-solvers to identify which solver is the best suited for each algorithm. In all cases, the same two sets of models were used: MIG and FA, for easy and hard instances, respectively. In order to get better statistical relevance, easy models from the MIG set will be run in a loop of 100 repetitions, and harder models from the FA set will be run 10 times.

4.2.1 IPASIRBones-1

Figure 4.1 compares the number of variables of the model to the backbone computing time on average. With IPASIRBones-1, all SAT-solvers have a similar linear response in relation to the number of model variables, except for some particular hard models. It is clearly visible that the SAT-solver with the best performance is *cadicalsc2020*, while *lingelingbcj* gets the longest execution times.

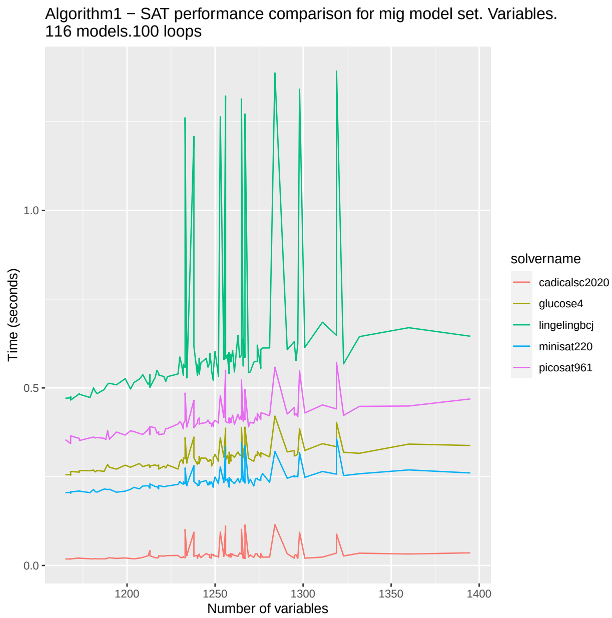


Figure 4.1: IPASIRBONES1 - Time vs. variables for MIG

On the other side, Figure 4.2 performs the comparison for the FA model set. It shows a step increase in the computation time as the number of variables increases, more visible for *lingelingbcj* and *picosat961* SAT-solvers.

4.2.2 IPASIRBONES-2

IPASIRBONES-2 is not so uniform as the one for the implicant listing algorithm, with a noticeable jitter in the graph (Figure 4.3. Glucose4 SAT-solver now is not the best performer, but there are other 2 SAT-solvers with similar performance (*cadicalsc2020* and *minisat220*).

FA results show a more stepped increase of computing time for larger models, but with a more linear response with respect to the number of variables (Figure 4.4).

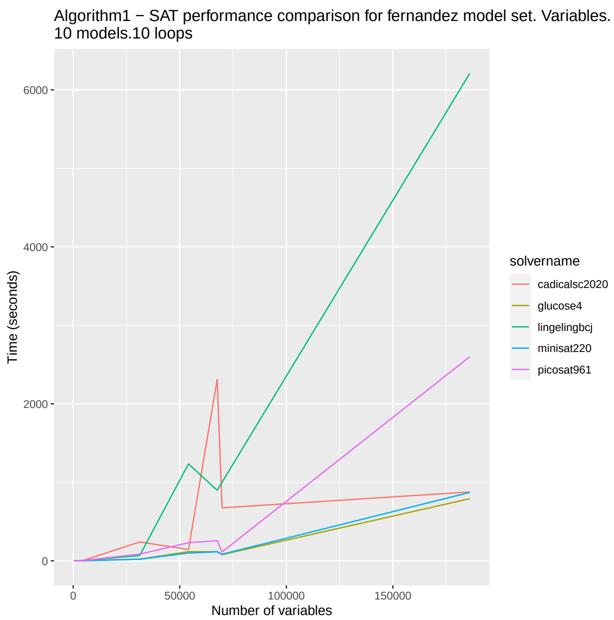


Figure 4.2: IPASIRBones1 - Time vs. variables for FA

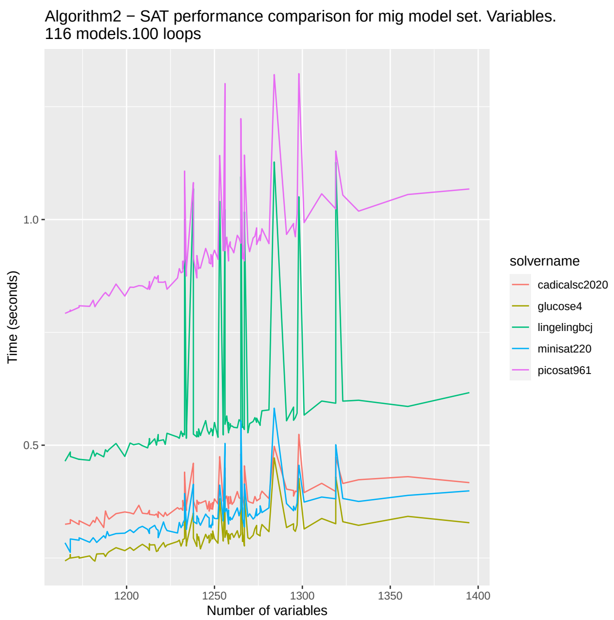


Figure 4.3: IPASIRBones-2 - Time vs. variables for MIG

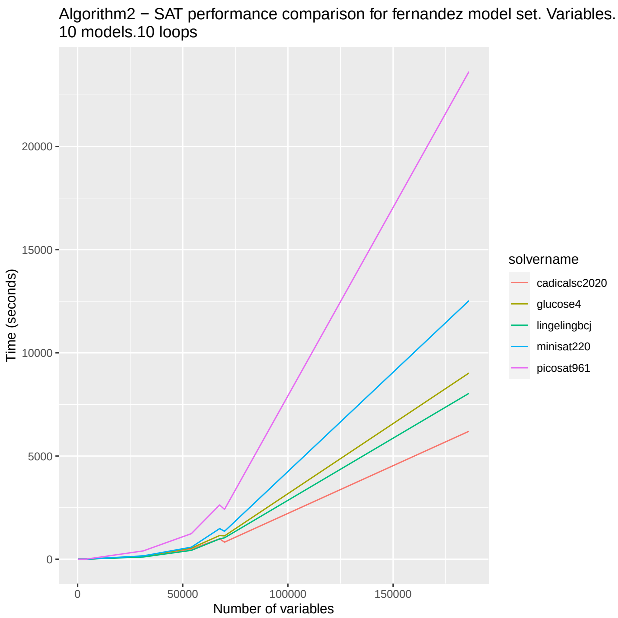


Figure 4.4: IPASIRBones-2 - Time vs. variables for FA

4.2.3 IPASIRBones-3

As in previous cases, every model from the *MIG* set was executed 100 times, while big models from *FA* set were executed 10 times. Total accumulated execution times for each model set and each SAT-solver are listed in Table 4.3. The first conclusion from that table is that *cadicalsc2020* is the faster SAT-solver from the selected ones providing the IPASIR interface. In addition, *lingelingbcj* performs comparatively worse with the smaller models from the MIG set. All samples from this set are uniformly slower, without any particular case accountable for such deviation.

Figures 4.5 and 4.6 show the execution results for the MIG set, correlating the effect of the number of variables and the size of the backbone, respectively. When comparing SAT-solvers, it is clearly visible that the best performing for this algorithm and the small and medium size of

Table 4.3: SAT time comparison for IPASIRBones-3 (seconds)

SAT-solver	MIG set	FA set
cadicalsc2020	22.2	1.921
glucose4	75.5	3.649
lingelingbcj	148.0	3.778
minisat220	70,6	4.080
picosat961	212.0	10.325

the models of this set is the *cadicalsc2020* SAT-solver. Another conclusion also visible is that both, the number of variables and the size of the backbone are not the only dimensions driving the time required to compute the backbone list of a model.

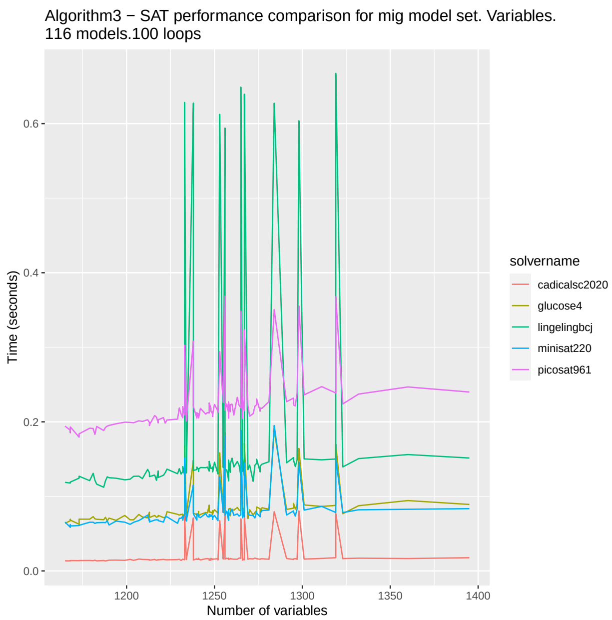


Figure 4.5: IPASIRBones-3 - Time vs. variables for MIG

Figures 4.7 and 4.8 show the effect of the number of variables and the size of the backbone, respectively for the FA model set. When comparing SAT-solvers, it is also clearly visible that the best performing for this al-

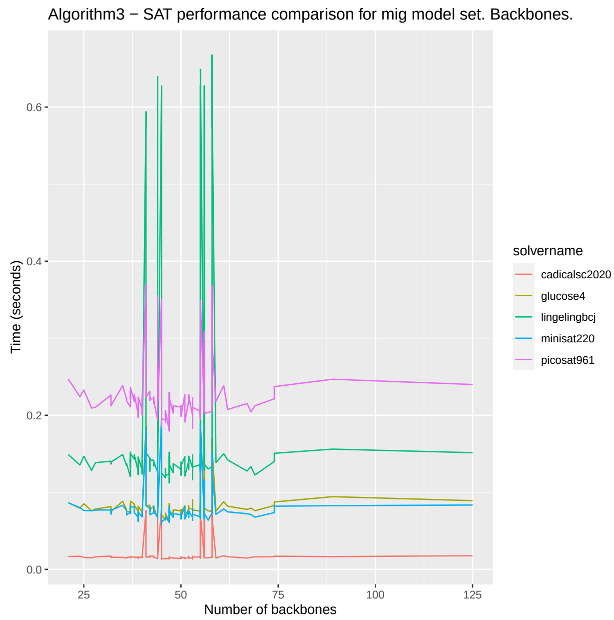


Figure 4.6: IPASIRBones-3 - Time vs. backbones for MIG

gorithm now for the big size of the models of this set is the *cadicalsc2020* SAT-solver. For bigger models, time increase is not so pronounced as in IPASIRBones-2, specially for the best performer SAT-solver, the green one.

4.2.4 IPASIRBones-4

For IPASIRBones-4, cadicalsc2020 is again the best SAT-solver for MIG, followed by glucose4 and minisat220. It also presents some small peaks for the same models that made it harder for previous algorithms (Figure 4.5). Lingelingbcj is the worst performer for this algorithm, with higher peaks on harder MIG models. For bigger models from FA set (Figure 4.7), glucose4 and minisat220 have slightly lower times than cadicalsc2020.

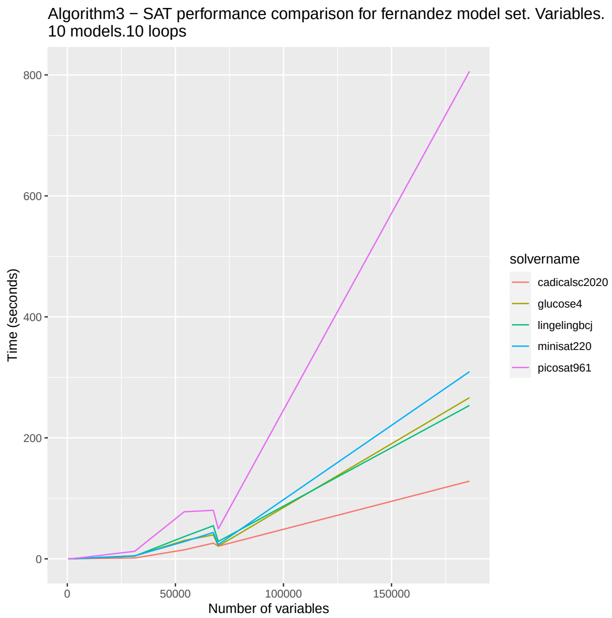


Figure 4.7: IPASIRBones-3 - Time vs. variables for FA

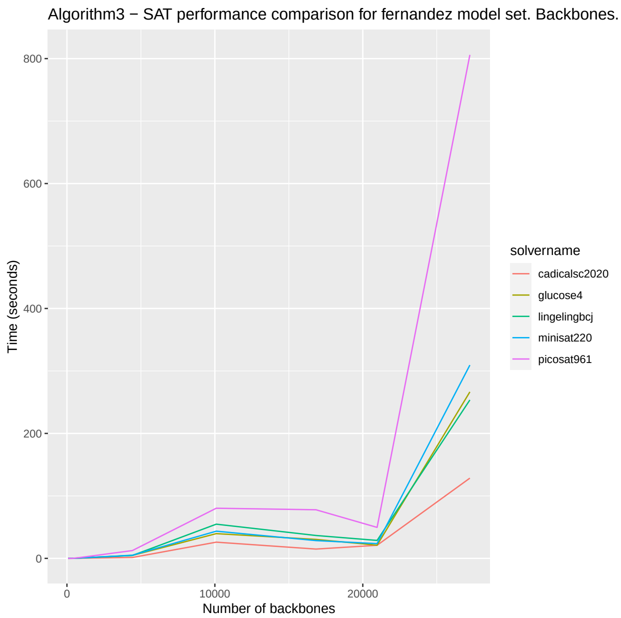


Figure 4.8: IPASIRBones-3 - Time vs. backbones for FA

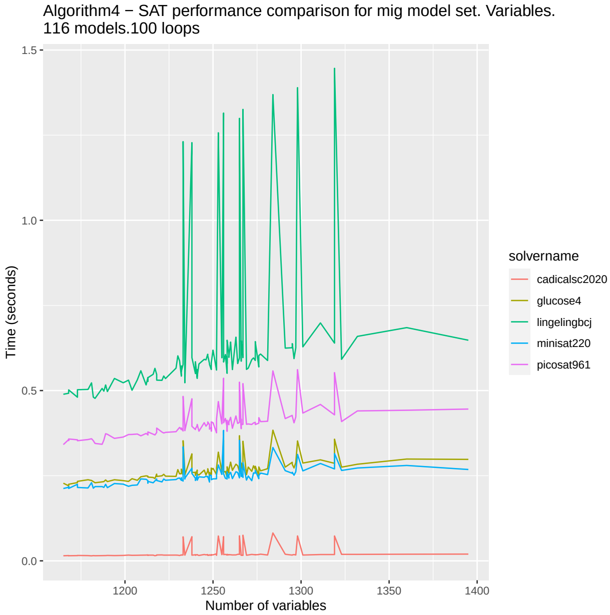


Figure 4.9: IPASIRBONES-4 - Time vs. variables for MIG

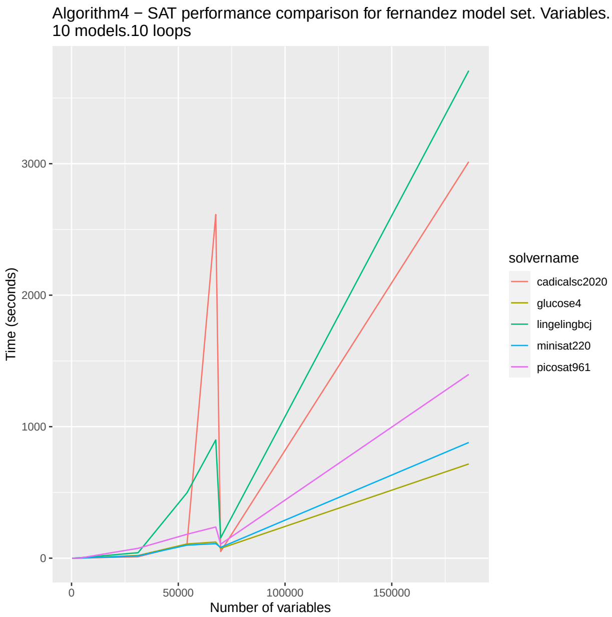


Figure 4.10: IPASIRBONES-4 - Time vs. variables for FA

4.2.5 IPASIRBones-5

IPASIRBones-5 presents a flat response time (Figure 4.11) when executed with *MIG* model set, having models in the range from 1.100 to 1.400 variables. Best SAT performer, *cadicals2020* also provides lower jitter in computing time, while *lingelingbcj* presents higher variances for some models. In respect to harder models from *fernandez* model set, again *cadicalsc2020* SAT-solver presents the flattest computation time (Figure 4.14) and *picosat961* quickly increases required computation time.

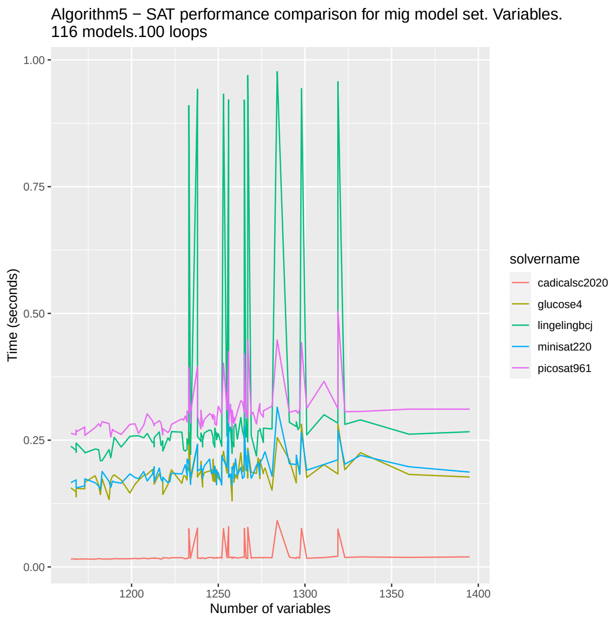


Figure 4.11: IPASIRBones-5 - Time vs. variables for MIG

4.2.6 IPASIRBones-6

For IPASIRBones-6, SAT-solvers *cadicalsc2020* and *lingelingbcj* must be discarded since, despite the algorithm being the same for all SAT-solvers,

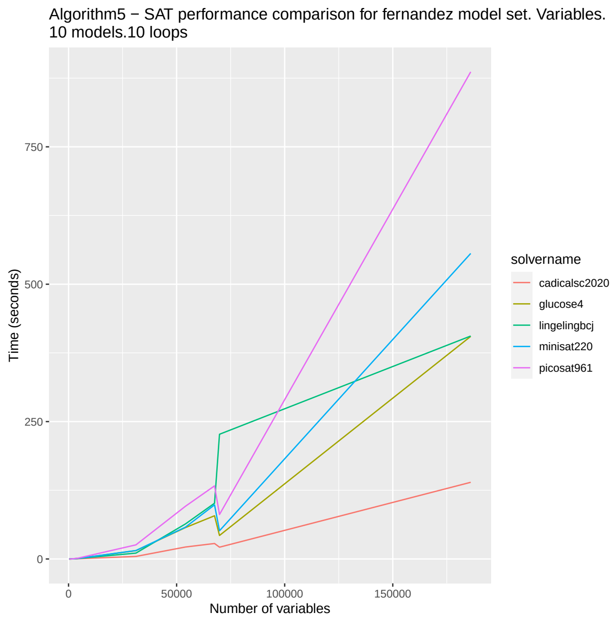


Figure 4.12: IPASIRBones-5 - Time vs. variables for FA

these two produce wrong results. This failure is due to the call made to `ipasir_failed()` used in combination with a previous call to `ipasir_assume()` does not produce correct results.

Therefore, for IPASIRBones-6, best SAT-solvers are *minisat220*, followed by *glucose4*.

4.2.7 IPASIRBones-7

SAT-solvers for IPASIRBones-7 follow the same pattern as in the previous ones, with cadicalsc2020 as the best performer for all models (Figures 4.15 and 4.16).

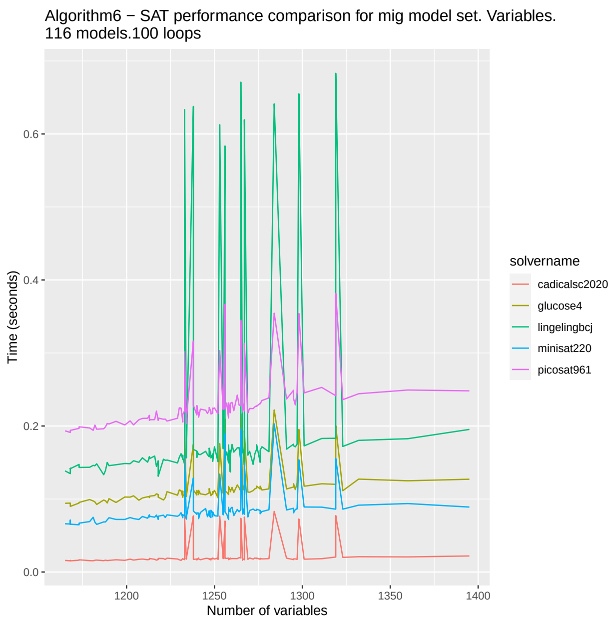


Figure 4.13: IPASIRBones-6- Time vs. variables for MIG

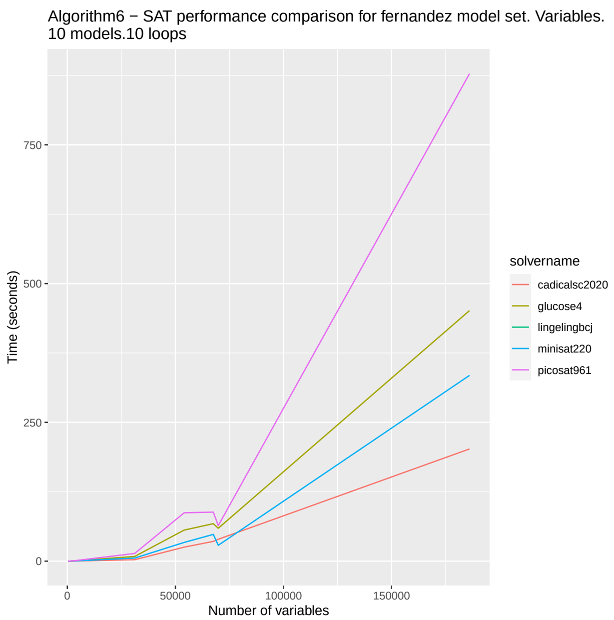


Figure 4.14: IPASIRBones-6- Time vs. variables for FA

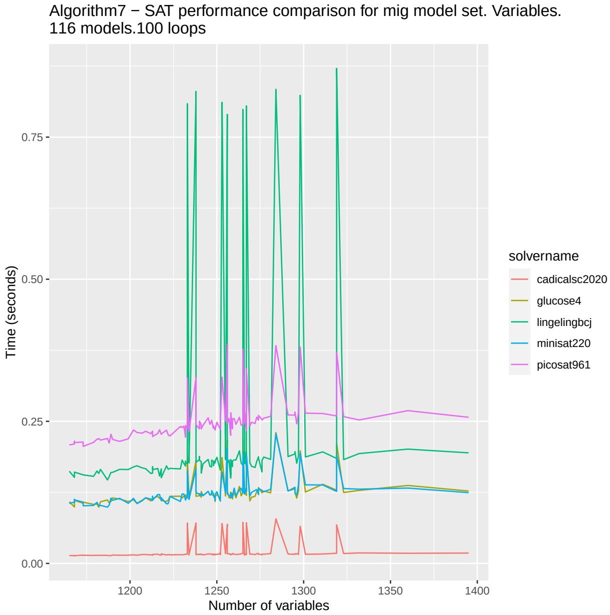


Figure 4.15: IPASIRBONES-7 - Time vs. variables for MIG

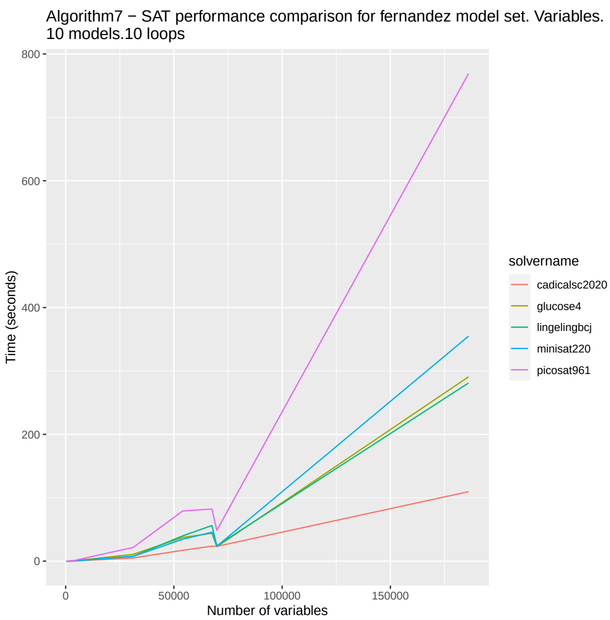


Figure 4.16: IPASIRBONES-7 - Time vs. variables for FA

4.2.8 Enhanced Algorithms: IPASIRBONES-3a&7a

The SAT-solvers for these programs follow the same pattern as in the previous ones, with cadicalsc2020 as the best performer for all models. Plots for algorithm 3a are displayed in Figures 4.5 and 4.7. Plots for algorithm 7a are shown in Figures 4.15 and 4.16.

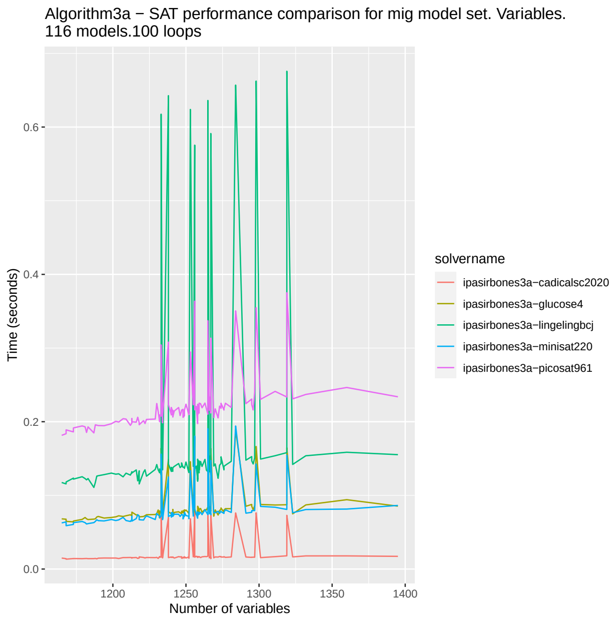


Figure 4.17: IPASIRBONES-3a- Time vs. variables for MIG

4.2.9 Conclusions on individual algorithms

Our experimental results show a high degree of variability among the algorithms, SAT-solvers, and model sizes, which makes it difficult to select a unique IPASIRBONES/SAT-solver combination fitting all scenarios. Table 4.5 shows a detailed overview of those results. Each row provides the total time for the MIG set and the FA model set plus the total computation time.

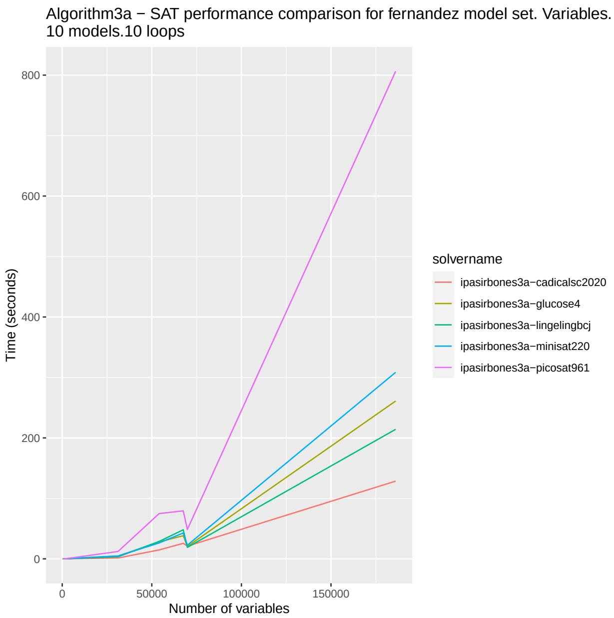


Figure 4.18: IPASIRBones-3a- Time vs. variables for FA

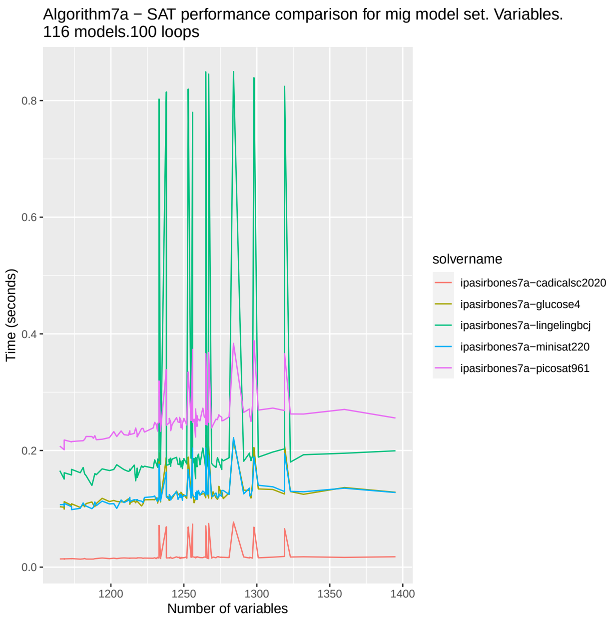


Figure 4.19: IPASIRBones-7a - Time vs. variables for MIG

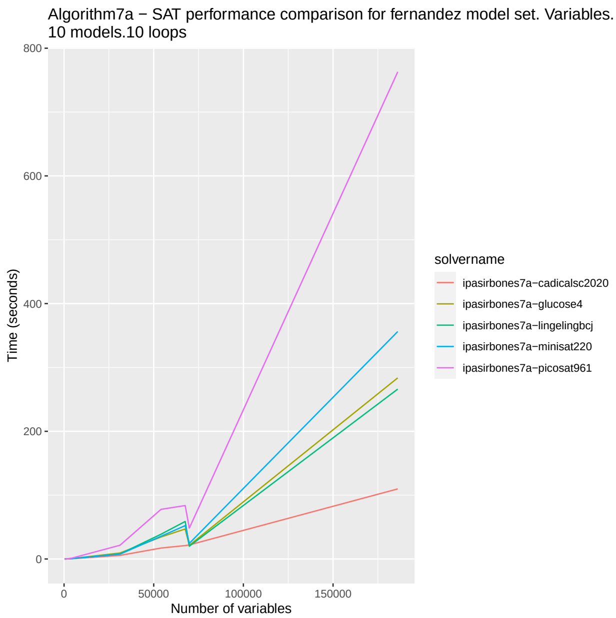


Figure 4.20: IPASIRBONES-7a - Time vs. variables for FA

Each total is calculated as the sum of the average time to compute the backbone of each model in the set. Models for MIG were computed 100 times each before calculating the individual model average and models for FA were also computed 10 times each before calculating that average. Times are expressed in seconds.

With respect to each individual program, Table 4.4 shows the best solver times.

Overall best algorithm and SAT-solver combination are the *IPASIRBONES-7a* and *IPASIRBONES-7* algorithms working with *cadicalsc2020* SAT-solver. Anyhow, the combination of the IPASIRBONES-3 and IPASIRBONES-3a algorithms together with *cadicalsc2020* SAT-solver also achieved similar performance. Note that SAT-solver *lingelingbcj* was not considered for IPASIRBONES-6as this SAT-solver fails as described in 4.2.6.

Table 4.4: Best solver per algorithm (seconds)

Algorithm	SAT-solver	MIG	FA	Total
IPASIRBones-1	minisat220	2765	11911	14676
IPASIRBones-2	cadicalsc2020	4359	86384	90743
IPASIRBones-3	cadicalsc2020	233	1920	2153
IPASIRBones-3a	cadicalsc2020	235	1917	2153
IPASIRBones-4	glucose4	3086	10441	13527
IPASIRBones-5	cadicalsc2020	259	2162	2421
IPASIRBones-6	cadicalsc2020	259	3056	3314
IPASIRBones-7	cadicalsc2020	233	1793	2026
IPASIRBones-7a	cadicalsc2020	235	1760	1995

4.3 RQ2: IPASIRBones *vs.* state-of-the-art tools

There are other specialized tools in computing the backbone from a given formula in CNF format. The most outstanding one [Janota et al., 2015] is *minibones*, and another one is *EDUCIBone* [Zhang et al., 2020]. This section will perform a comparison between the best performers IPASIRBonesprograms described in Section 4.2.9 and these two.

Minibones and EDUCIBone have also been evaluated under the same conditions and model sets as the IPASIRBones programs, both have been evaluated to separately compute *MIG* and *fernandez* model sets, executing a loop of 100 repetitions for each model from *MIG* model set and execution a loop of 10 repetitions for each model from *fernandez* model set. Then, the average for the executions of each model was calculated, and finally, all those averages for each model set were summed. Algorithms 5 and 7, the ones based on chunks, require an input value as the chunk size. Given that both, Minibones and EDUCIBone use 100 as the default value for the chunk size, our IPASIRBones programs have been also setup to use same chunk size value by default.

Table 4.5: Total times for the different algorithms and solvers (seconds)

algorithm	solvername	MIG	FA	total
IPASIRBones-1	minisat220	2765	11911	14676
IPASIRBones-1	glucose4	3511	11270	14781
IPASIRBones-1	picosat961	4743	32889	37632
IPASIRBones-1	cadicalsc2020	368	42480	42848
IPASIRBones-1	lingelingbcj	7178	85980	93158
IPASIRBones-2	cadicalsc2020	4359	86384	90743
IPASIRBones-2	lingelingbcj	6620	105937	112558
IPASIRBones-2	glucose4	3469	119817	123285
IPASIRBones-2	minisat220	3973	161067	165040
IPASIRBones-2	picosat961	10799	303131	313930
IPASIRBones-3	cadicalsc2020	233	1920	2153
IPASIRBones-3	glucose4	983	3629	4612
IPASIRBones-3	minisat220	916	4106	5022
IPASIRBones-3	lingelingbcj	2010	3790	5800
IPASIRBones-3	picosat961	2572	10273	12845
IPASIRBones-3a	cadicalsc2020	235	1917	2153
IPASIRBones-3a	glucose4	970	3529	4499
IPASIRBones-3a	minisat220	919	4055	4974
IPASIRBones-3a	lingelingbcj	2020	3140	5161
IPASIRBones-3a	picosat961	2562	10226	12787
IPASIRBones-4	glucose4	3086	10441	13527
IPASIRBones-4	minisat220	2892	11897	14789
IPASIRBones-4	picosat961	4670	20004	24674
IPASIRBones-4	cadicalsc2020	250	57965	58215
IPASIRBones-4	lingelingbcj	7312	53045	60357
IPASIRBones-5	cadicalsc2020	259	2162	2421
IPASIRBones-5	glucose4	2162	5998	8160
IPASIRBones-5	minisat220	2236	7802	10037
IPASIRBones-5	lingelingbcj	3567	8096	11664
IPASIRBones-5	picosat961	3522	12235	15757
IPASIRBones-6	lingelingbcj	2277	201	2478
IPASIRBones-6	cadicalsc2020	259	3056	3314
IPASIRBones-6	minisat220	994	4515	5508
IPASIRBones-6	glucose4	1338	6436	7774
IPASIRBones-6	picosat961	2653	11330	13983
IPASIRBones-7	cadicalsc2020	233	1793	2026
IPASIRBones-7	glucose4	1450	4065	5515
IPASIRBones-7	minisat220	1447	4672	6119
IPASIRBones-7	lingelingbcj	2604	4098	6702
IPASIRBones-7	picosat961	2898	10021	12919
IPASIRBones-7a	cadicalsc2020	235	1760	1995
IPASIRBones-7a	glucose4	1460	3967	5427
IPASIRBones-7a	minisat220	1451	4769	6220
IPASIRBones-7a	lingelingbcj	2625	3923	6548
IPASIRBones-7a	picosat961	2927	9957	12884

Visualizations resulting from this experimental evaluation are shown in Figure 4.21 and 4.22.

A visual inspection of these two plots unveils a similar pattern to the ones seen for the IPASIRBones programs: time evolution for the MIG set (with the number of variables ranging from 1100 to 1400), is mostly flat or slightly increasing, except for a reduced number of models, apparently harder than the others. In relationship to the FA model set, *EDUCIBone* shows the same exponential increase pattern as seen in the IPASIRBones programs but, on the other side *minibones* is able to manage a high number of variables with a small linear increase pattern instead of an exponential time increase.

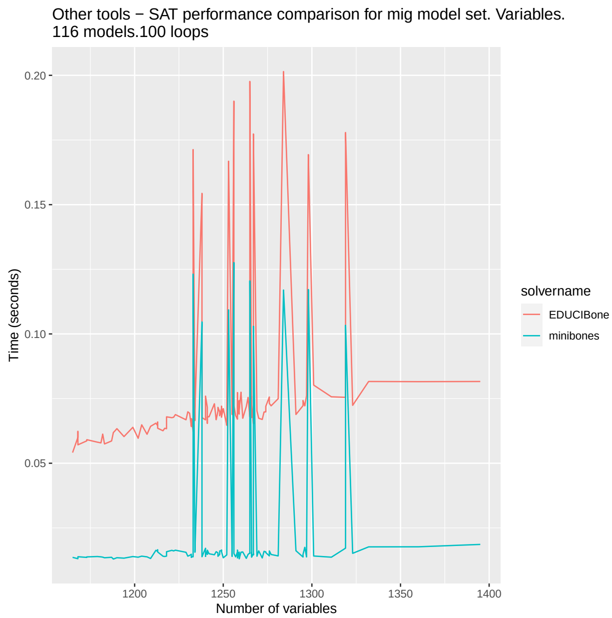


Figure 4.21: Other tools - Time vs. variables for MIG

Figure 4.6 provides the total computing time for these tools, in seconds. *Minibones* can be clearly identified as the best performer backbone

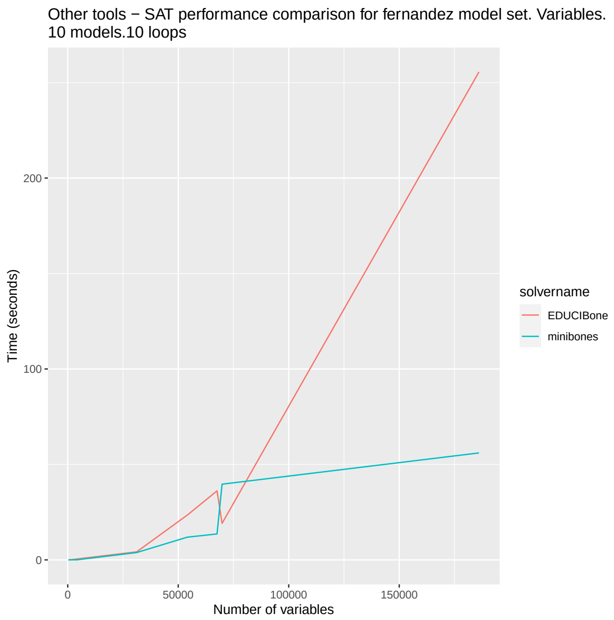


Figure 4.22: Other tools - Time vs. variables for FA

Table 4.6: Other tools performance (seconds)

Backbone tool	MIG set	FA set	Total
EDUCIBone	892	3391	4283
minibones	262	1251	1513

computation tool. Although some IPASIRBONES programs provide better performance for the *MIG* model set than the *minibones* tool, overall all other tools except *minibones* do not perform so well on very big models. *EDUCIBone* times are far from *minibones* and the best *ipasirbones* algorithms-solver combinations.

Figure 4.7 shows the final ranking, reflecting the facts discussed above, with *minibones* at the top, followed by most IPASIRBONES programs and EDUCIBone. The list ends with the worst IPASIRBONES programs (the ones based on IPASIRBONES-4, IPASIRBONES1 and IPASIRBONES-2).

Table 4.7: Final performance comparison table (seconds)

Solver name	MIG set	FA set	Total
minibones	262	1251	1513
IPASIRBONES-7a-cadicalsc2020	235	1760	1995
IPASIRBONES-7-cadicalsc2020	233	1793	2026
IPASIRBONES-3-cadicalsc2020	233	1920	2153
IPASIRBONES-3a-cadicalsc2020	235	1917	2153
IPASIRBONES-5-cadicalsc2020	259	2162	2421
IPASIRBONES-6-cadicalsc2020	259	3056	3314
EDUCIBone	892	3391	4283
IPASIRBONES-4-glucose4	3086	10441	13527
IPASIRBONES-1-minisat220	2765	11911	14676
IPASIRBONES-2-cadicalsc2020	4359	86384	90743

5

Chapter 5: Conclusions and Future Work

5.1 Conclusions

Our work has provided IPASIRBones, an IPASIR-based implementation of diverse algorithms to incrementally compute the *backbone* of propositional formulas, which may encode configuration or any other kind of model. As shown in Chapter 1, backbones are what in the software product line literature is called the *core* and the *dead* features of a configuration model. As our implementation works incrementally, processing the model can continue after the backbone has been computed, for example by adding new clauses to the formula or new assumptions. This makes this approach suitable for interactive solutions or being embedded into other applications.

Thanks to the IPASIR interface, IPASIRBones can take advantage of any SAT-solver that complies with the standard. So if a new SAT-solver is designed and the interface is provided, any application previously developed can be linked to that new SAT-solver without requiring re-coding.

The reported experimental validation identifies the best-performing configurations of IPASIRBones (underlying algorithm + SAT-solver) and compared them to two state-of-the-art backbone computing tools. The results show that IPASIRBones performs better than *EDUCIBone*, but *minibones* still beats IPASIRBones in huge industrial models.

5.2 Future Work

We envision two main lines of future work:

- IPASIRBones has been tested with the most relevant IPASIR-compatible SAT-solvers. Nevertheless, there are many other SAT-solvers that can be adapted to use this interface. In addition, every year new solvers are submitted to SAT competitions, which are a source of new developments. Some other SAT-solvers have native parallel capabilities, and none have been used here. Regardless of new solvers, backbone computation algorithms and heuristics can be improved.
- IPASIR-based implementations are well suited, not only to be used in providing back-end support for visual feature modeling tools, but also to be used to automate feature modeling tasks in general. In particular, future work can be directed toward managing the tasks following the identification of those core and dead features, like checking new features and their dependencies or conflicts while still using the same solver instance used for the backbone.

References

[Alyahya et al., 2022] Alyahya, T. N., Menai, M. E. B., and Mathkour, H. (2022). On the structure of the boolean satisfiability problem: A survey. *ACM Comput. Surv.*, 55(3).

[Audemard and Simon, 2017] Audemard, G. and Simon, L. (2017). Glucose and syrup in the sat'17. *Proceedings of SAT Competition 2017*, pages 16–17.

[Balyo, 2017] Balyo, T. (2017). IPASIR: The Standard Interface for Incremental Satisfiability Solving. https://github.com/biotomas/ipasir.

[Balyo et al., 2016] Balyo, T., Biere, A., Iser, M., and Sinz, C. (2016). SAT Race 2015. *Artificial Intelligence*, 241:45–65.

[Balyo et al., 2020] Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., and Suda, M., editors (2020). *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki, Finland.

[Batory, 2005] Batory, D. (2005). Feature models, grammars, and propositional formulas. In Obbink, H. and Pohl, K., editors, *Software Product Lines*, pages 7–20, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Biere, 2008] Biere, A. (2008). Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97.

[Biere, 2014] Biere, A. (2014). Lingeling essentials, a tutorial on design and implementation aspects of the the sat solver lingeling. In Berre, D. L., editor, *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, page 88. EasyChair.

[Biere et al., 2020] Biere, A., Fazekas, K., Fleury, M., and Heisinger, M. (2020). CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., and Suda, M., editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53. University of Helsinki.

[Davis et al., 1962] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397.

[Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *J. ACM*, 7(3):201–215.

[Eén and Sörensson, 2004] Eén, N. and Sörensson, N. (2004). An extensible sat-solver. In Giunchiglia, E. and Tacchella, A., editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Fernandez-Amoros et al., 2023] Fernandez-Amoros, D., Heradio, R., Mayr-Dorn, C., and Egyed, A. (2023). Scalable sampling of highly-configurable systems: Generating random instances of the linux kernel. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA. Association for Computing Machinery.

[Guo et al., 2019] Guo, S., He, J., Song, X., and Liu, W. (2019). Backbone solving algorithm based on heuristic thinking. In *Proceedings of the 3rd International Conference on Machine Learning and Soft Computing*, ICMLSC 2019, page 44–48, New York, NY, USA. Association for Computing Machinery.

[Janota, 2010] Janota, M. (2010). SAT solving in interactive configuration. Ph.D. dissertation, University College Dublin.

[Janota et al., 2012] Janota, M., Lynce, I., and Marques-Silva, J. (2012). Experimental analysis of backbone computation algorithms. In *International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion (RCRA)*, pages 15–20.

[Janota et al., 2015] Janota, M., Lynce, I., and Marques-Silva, J. (2015). Algorithms for computing backbones of propositional formulae. *AI Communications*, 28(2):161 – 177.

[Kilby et al., 2005] Kilby, P., Slaney, J. K., Thiébaux, S., and Walsh, T. (2005). Backbones and backdoors in satisfiability. In Veloso, M. M. and Kambhampati, S., editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1368–1373. AAAI Press / The MIT Press.

[Krieter et al., 2021] Krieter, S., Arens, R., Nieke, M., Sundermann, C., Heß, T., Thüm, T., and Seidl, C. (2021). Incremental construction of modal implication graphs for evolving feature models. In *25th ACM International Systems and Software Product Line Conference (SPLC)*, pages 64–74, Leicester, United Kingdom.

[Krieter et al., 2018] Krieter, S., Thüm, T., Schulze, S., Schröter, R., and Saake, G. (2018). Propagating configuration decisions with modal implication graphs. In *40th International Conference on Software Engineering (ICSE)*, pages 898–909, Gothenburg, Sweden.

[Kroer, 2012] Kroer, C. (2012). SAT and SMT-based interactive configuration for container vessel stowage planning. *Master's Thesis, IT University of Copenhagen*.

[Liang et al., 2020] Liang, T., Wang, X., Wang, S., and Wang, X. (2020). An improved ID3 classification algorithm for solving the backbone of proposition formulae. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing*, pages 386–391.

[Mannion, 2002] Mannion, M. (2002). Using first-order logic for product line model validation. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, page 176–187, Berlin, Heidelberg. Springer-Verlag.

[Marques-Silva et al., 2010] Marques-Silva, J., Janota, M., and Lynce, I. (2010). On computing backbones of propositional theories. *Frontiers in Artificial Intelligence and Applications*, 215:15 – 20.

[Marques-Silva et al., 2021] Marques-Silva, J., Lynce, I., and Malik, S. (2021). Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*.

[Mitchell, 2005] Mitchell, D. (2005). A sat solver primer. *Bulletin of the EATCS*, 85:112–132.

[Monasson et al., 1999] Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., and Troyansky, L. (1999). Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137.

[Perez-Morago et al., 2015] Perez-Morago, H., Heradio, R., Fernandez-Amoros, D., Bean, R., and Cerrada, C. (2015). Efficient identification of core and dead features in variability models. *IEEE Access*, 3:2333–2340.

[Plazar et al., 2019] Plazar, Q., Acher, M., Perrouin, G., Devroey, X., and Cordy, M. (2019). Uniform sampling of sat solutions for configurable systems: Are we there yet? In *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*.

[Previti et al., 2017] Previti, A., Ignatiev, A., Jarvisalo, M., and Marques-Silva, J. (2017). On computing generalized backbones. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1050–1056, Los Alamitos, CA, USA. IEEE Computer Society.

[Previti and Järvisalo, 2018] Previti, A. and Järvisalo, M. (2018). A preference-based approach to backbone computation with application to argumentation. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 896–902, New York, NY, USA. Association for Computing Machinery.

[Quinlan, 1986] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.

[SAT Challenge, 1993] SAT Challenge (1993). Satisfiability suggested format. Rutgers University.

[Wu, 2017] Wu, H. (2017). Improving sat-solving with machine learning. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, page 787–788, New York, NY, USA. Association for Computing Machinery.

[Zhang et al., 2020] Zhang, Y., Zhang, M., and Pu, G. (2020). Optimizing backbone filtering. *Science of Computer Programming*, 187:102374.

[Zhang et al., 2018] Zhang, Y., Zhang, M., Pu, G., Song, F., Li, J., Fontaine, P., Kaliszyk, C., Schulz, S., and Urban, J. (2018). Towards backbone computing: A greedy-whitening based approach. *AI Commun.*, 31(3):267–280.

[Zhu et al., 2011] Zhu, C. S., Weissenbacher, G., and Malik, S. (2011). Post-silicon fault localisation using maximum satisfiability and backbones. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 63–66.