# Driving CDCL Search

Carmine Dodaro[1], Philip Gasteiger[2], Nicola Leone[1],
Benjamin Musitsch[2], Francesco Ricca[1], and Konstantin Schekotihin[2]

[1] Università della Calabria, Rende CS, IT
{lastname}@mat.unical.it
[2] Alpen-Adria-Universität Klagenfurt, 9020 Klagenfurt, AT
{firstname.lastname}@aau.at

**Abstract.** The CDCL algorithm is the leading solution adopted by state-of-the-art solvers for SAT, SMT, ASP, and others. Experiments show that the performance of CDCL solvers can be significantly boosted by embedding domain-specific heuristics, especially on large real-world problems. However, a proper integration of such criteria in off-the-shelf CDCL implementations is not obvious. In this paper, we distill the key ingredients that drive the search of CDCL solvers, and propose a general framework for designing and implementing new heuristics. We implemented our strategy in an ASP solver, and we experimented on two industrial domains. On hard problem instances, state-of-the-art implementations fail to find any solution in acceptable time, whereas our implementation is very successful and finds all solutions.

## 1 Introduction

The Conflict Driven Clause Learning (CDCL) algorithm [25] is the leading solution adopted by state-of-the-art solvers for Boolean Satisfiability (SAT) [7], Satisfiability Modulo Theories (SMT) [23], and Answer Set Programming (ASP) [9] to mention a few. Notably, CDCL solvers have been applied with success for solving several real-world problems ranging from hardware and software model checking, planning, equivalence checking, bioinformatics, configuration problems, hardware and software test, software package dependencies, cryptography and more [7].

As a matter of fact the performance of a CDCL solver heavily depends on the adoption of heuristics that drive the search for solutions. Among these, the heuristic for the selection of the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation) can dramatically affect the overall performance of an implementation [7]. State-of-the-art CDCL implementations feature very good general purpose heuristics belonging to the family of VSIDS [6, 12, 22]. Since their introduction, VSIDS heuristics proved to be a key ingredient for solving many relevant problems [6]. Nonetheless, no general heuristic is known to be the best possible choice for all problems [21]. Pioneering work on employing domain-specific heuristics corroborates the validity of that idea. In particular, Beame et al. demonstrated the utility of a domain-specific branching heuristic for solving the pebbling formulas with a CDCL solver [5]; and Rintanen proposed to replace the standard VSIDS by a domain-heuristics for efficiently solving planning as satisfiability [24]. Moreover, Gerhard Friedrich in his

joint invited talk at CP-ICLP 2015 described a number of experiences in which "domain-specific heuristics turned out to be the key component in several industrial applications of problem solvers" [14]. However, as argued in [24]: "the main challenge in defining a variable selection scheme is its integration in the CDCL algorithm in a productive way". Indeed, domain experts can quickly provide promising heuristic-criteria based on properties of solutions, but they may not have the knowledge and the experience to plug them in CDCL implementations.

The goal of the paper is to ease the design and the evaluation of new heuristics for CDCL solvers as well as to reduce the efforts needed to provide effective implementations. To this end we distill the key ingredients that drive the search of CDCL solvers, and propose a general framework for designing and implementing new heuristics. Useless to say that one could invent new heuristics based on other different "ingredients". Nevertheless, our framework is quite rich and flexible, and indeed it covers the definition of the most popular heuristics for CDCL, as well as enables the definition of interesting problem-specific heuristics. Moreover, further features could be easily incorporated. A number of examples are provided to demonstrate the applicability of the proposed framework for defining a variety of known heuristic criteria from the literature [6, 24]. Pragmatic evidence of the benefits of our framework in devising problem-specific heuristics is provided by reporting on a use case where two hard industrial domains defined by Siemens, the Partner Units Problem (PUP) [26] and Combined Configuration Problem (CCP) [20], are solved efficiently.

In our use case, the framework has been implemented by extending the ASP solver WASP [1]. Our implementation offers a wide range of possibilities to end users for developing new heuristics. In particular, it offers multi-language support including scripting languages for fast prototyping, an embedded C++ interface for performance-oriented implementations, as well as heuristic definition by means of declarative predicate-based paradigms.

Experiments on PUP and CCP confirm the viability of the framework in real-world scenarios, and the effectiveness of our implementation that is able to solve successfully hard problem instances which were not solvable by state-of-the-art ASP solvers. Furthermore, we were able to solve hardest CCP instances for which no solution was known before.

## 2 An overview of the CDCL algorithm

In this section we provide some basic knowledge about the Conflict-Driven Clause Learning (CDCL) algorithm [25] for solving the satisfiability problem (SAT) [6]. The Conflict-Driven Clause Learning (CDCL) [25] backtracking search algorithm is a de facto standard for the satisfiability problem (SAT) [4, 6]. During the recent years, CDCL has been extended in order to fit specific requirements of formalisms in the neighborhood of SAT, e.g. Satisfiability Modulo Theories and Answer Set Programming. In the following, we first introduce some basic concepts of the satisfiability problem and then we describe the CDCL algorithm.

*SAT.* Let $\mathcal{A}$ be a fixed, countable set of propositional atoms including $\bot$. A literal $\ell$ is either an atom $a$ or its negation $\neg a$, and $atom(\ell) = a$ denotes the atom associated

to $\ell$. The complement of $\ell$ is denoted by $\overline{\ell}$, where $\overline{a} = \neg a$ and $\overline{\neg a} = a$ for an atom $a$. For a set $L$ of literals, $\overline{L} := \{\overline{\ell} \mid \ell \in L\}$, $L^+ := L \cap \mathcal{A}$, and $L^- := \overline{L} \cap \mathcal{A}$. A (partial) interpretation is a set of literals $I$ containing $\neg\bot$. $I$ is inconsistent if $I^+ \cap I^- \neq \emptyset$, otherwise $I$ is consistent. $I$ is total if $I^+ \cup I^- = \mathcal{A}$. Given an interpretation $I$, a literal $\ell$ is true if $\ell \in I$; is false if $\overline{\ell} \in I$, and is undefined otherwise. A clause $\varphi$ is a set of literals, and a formula $\Gamma$ is a set of clauses. An interpretation $I$ satisfies a clause $\varphi$ if $\varphi \cap I \neq \emptyset$. A consistent, total interpretation $I$ is a model of a formula $\Gamma$ if all clauses in $\Gamma$ are satisfied. A formula $\Gamma$ is said to be consistent if it admits some model, inconsistent otherwise. Given a formula $\Gamma$, the SAT problem consists in determining whether $\Gamma$ is consistent.

*Example 1.* Consider the formula $\Gamma = \{\{a, b, \neg c\}, \{a\}, \{\neg b\}, \{c, d\}, \{c, \neg d\}\}$. $I = \{a, \neg b, c, d\}$ is a model of $\Gamma$.

*CDCL algorithm.* The CDCL algorithm takes as input a formula $\Gamma$, and determines whether it is consistent. The first step of the algorithm consists of the simplification of $\Gamma$. Polynomial algorithms for clause rewriting and variable elimination are applied on $\Gamma$, which strengthen and/or remove redundant clauses [11]. After the simplifications step, the partial interpretation $I$ is set to $\{\neg\bot\}$, and the backtracking search starts. First, $I$ is extended with all the literals that can be deterministically inferred by applying some inference rule (*propagation* step). The main inference rule of a SAT solver is *unit propagation* that extends $I$ by an undefined literal $\ell$ whenever there is a clause $\varphi \in \Gamma$ such that $\ell \in \varphi$ and $\varphi \setminus \ell \subseteq \overline{I}$. Three cases are possible after a propagation step is completed: $(i)$ $I$ is consistent but not total. In that case, an undefined literal $\ell$ (called *branching literal*) is chosen according to some heuristic criterion, and is added to $I$. Then, a propagation step is performed that infers the consequences of this choice. $(ii)$ $I$ is inconsistent, thus there is a conflict, and $I$ is analyzed. The reason of the conflict is modeled by a fresh clause $\varphi$ that is added to $\Gamma$ (clause *learning*), e.g. by applying the *first Unique Implication Point (UIP)* technique [28]. Moreover, the algorithm backtracks (i.e. choices and their consequences are undone) until the consistency of $I$ is restored. Then the algorithm propagates inferences starting from the fresh clause $\varphi$. Otherwise, if the consistency of $I$ cannot be restored, the algorithm terminates returning INCONSISTENT. Finally, in case $(iii)$ $I$ is consistent and total, i.e. $I$ is a model, the algorithm terminates returning CONSISTENT.

*Example 2.* Consider the formula $\Gamma$ of Example 1. The search starts with $I := \{\neg\bot\}$. Unit propagation extends $I$ with $a$ and $\neg b$ to satisfy clauses $\{a\}$ and $\{\neg b\}$. No other inferences can be done, thus a choice is performed among $c$, $\neg c$, $d$, and $\neg d$. Choosing $\neg c$ would lead to an inconsistency, while all other choices would directly lead to a model.

*Heuristics.* It is well-known that the selection the branching literal plays a crucial role for efficiency. Nowadays, state-of-the-art solvers, such as LINGELING [6] and GLUCOSE [4], implement branching criteria belonging to the VSIDS [22] family that are a variant of the MINISAT [12] heuristic. Thus, in the following, the MINISAT [12] heuristic is referred to as the *default branching heuristic* of a CDCL solver. The MINISAT heuristic is based on the *activity* value for each atom in the input formula $\Gamma$, which is initially set to 0. The activity of $atom(\ell)$ is incremented by a value $inc$ when a literal $\ell$ occurs

<div align="center">

**CDCL $\longrightarrow$ Driver: Events**

</div>

| |
|---|
| **ONSEARCH($\Gamma'$, $\mathcal{A}'$)** |
| *Triggered when the backtracking search starts. $\Gamma'$ and $\mathcal{A}'$ are the formula and the atoms after the simplifications, resp.* |

| |
|---|
| **ONINCOCHOICE($\ell$)** |
| *Triggered when the choice $\ell$ led to an inconsistency.* |

| |
|---|
| **ONCONFLICT($\ell$)** |
| *Triggered when a conflict is detected after the choice of $\ell$.* |

| |
|---|
| **ONLEARNCLAUSE($\varphi$)** |
| *Triggered when the clause $\varphi$ is learned.* |

| |
|---|
| **ONLITINCONFLICT($\ell$)** |
| *Triggered when the literal $\ell$ is involved in the creation of a clause to learn, e.g., while applying the first UIP schema.* |

| |
|---|
| **ONDELETION($\varphi$)** |
| *Triggered when the clause $\varphi$ has been removed.* |

| |
|---|
| **ONRESTART()** |
| *Triggered when a restart of the search occurs.* |

| |
|---|
| **ONUNROLLLIT($\ell$)** |
| *Triggered during backtracking if the literal $\ell$ is set to undefined.* |

<div align="center">

Table 1: Events notified by the CDCL solver.

</div>

in a learned clause. Then, after learning the clause, the value of $inc$ is multiplied by a constant slightly greater than 1, to give more and more importance to variables that occur in recently-learned clauses. Once a choice is needed, the literal $\neg at$ is chosen, where $at$ is the undefined atom having the highest activity value (ties are broken randomly).

For the sake of completeness, we mention that the CDCL is usually complemented with heuristics that control the number of learned clauses, and restart the computation to explore different branches of the search tree [7].

## 3   Driving CDCL Search

In this section we present the results of our studies pinpointing the key ingredients needed to drive the search of CDCL solvers. These were distilled by considering the needs of well-known heuristics, as well as by applying the lessons learned while developing domain-specific heuristics in the case study reported in the next section.

In our framework we identify two main actors: the *solver* and the *driver*. The first is, basically, an implementation of the CDCL algorithm, and the second implements an external branching heuristic.

Selection of a branching literal may depend (or not depend) on properties of the input, on the (current) solver state, on employed optimization techniques (e.g., input

rewritings), as well as on the status of variables that directly model some properties of the problem, etc. This information is possibly communicated by the solver to the driver by means of *events*. Events can be seen as asynchronous messages sent to the driver when specific points of the computation are reached, as detailed in Table 1. The driver could exploit this information to implement the heuristic criteria, e.g., updating counters or modifying its status. None of the events are mandatory for the driver (we expect that an efficient implementation smartly sends only those required by the driver).

We have also identified two main moments in the computation when the solver gets directions from the driver. These are modeled in our framework by means of *requests*, that are detailed in Table 2 together with the *responses* the solver expects to receive. Requests are synchronous, i.e., the solver waits for a response of (i.e., a command from) the driver.

The request GETATOMSTOBEFROZEN($\mathcal{A}$) is sent before starting the simplifications of the input formula. In fact, preprocessing can remove atoms that the heuristic strategy may want to keep (they may play a role in a domain-specific heuristic). The response #FREEZE($A \subseteq \mathcal{A}$) informs the solver to avoid eliminating the variables in $\mathcal{A}$.

The GETCHOICE($I$) request is the main one, and is made by the solver when a non-deterministic choice is required. A number of responses are possible besides the obvious one where the driver provides the solver with an ordered list of branching literals to be chosen. The list can hold just one element as in many well-known heuristics, or can provide in one call several choices so to reduce communication overheads in domain-heuristics that may provide at once an entire plan of choices. Alternative responses were inspired by domain-specific criteria. For example, the driver is likely to recognize specific paths in the search space that cannot lead to a solution, so it can issue a UNROLL($\ell$) to cause a backtrack up to the last choice made before $\ell$. Moreover, the driver could add some additional domain knowledge modeled by a clause $\varphi$ in the solver by answering #ADDCLAUSE($\varphi$). Finally, there are cases (see next section) in which it makes sense to blend custom heuristics with the default heuristic, to exploit the strengths of both. To this end the driver can respond #FALLBACK($n, \mathcal{I}, \mathcal{F}, \mathcal{S}$) so to force the solver to use its default heuristic for the next $n$ choices. The parameters of #FALLBACK($.$) allow to customize this interaction by (possibly) tuning the activity values and the multiplying factors used by the default heuristic.

In the following we provide a number of examples that show how to encode several well-known conflict-based and domain-specific heuristics. This way the reader can appreciate the generality of our proposal that cover the needs of a wide range of conflict-based heuristics as well as domain-heuristics proposed in the literature. Nevertheless, we remark that our framework can be easily extended by adding new events/requests to fit the needs of future applications.

*Example 3 (MINISAT heuristic).* We first consider the MINISAT heuristic as described in [12]. This criterion can be implemented in our framework by exploiting the events ONSTARTSEARCH($\Gamma', \mathcal{A}'$) and ONLEARNCLAUSE($\varphi$), as well as the GETCHOICE($I$) request. The first is used to initialize the activity values of atoms in the input; the second causes an increment the activity of atoms in the learned clause $\varphi$. Finally, the request GETCHOICE($I$) is answered with #CHOICE($\{(a, n)\}$) by the driver, where $a$ is the undefined atom in $I$ with the highest value of activity.

**CDCL ⟷ Driver: Requests & Responses**

| **GetAtomsToBeFrozen($\mathcal{A}$)** |
|---|
| *Asks the heuristic strategy to provide a list of atoms $A \subseteq \mathcal{A}$ that must not be removed by simplifications.* |
| Allowed Response: |

| **#Freeze($A$)** |
|---|
| *Asks to not remove atoms in $A$ during preprocessing.* |

| **GetChoice($I$)** |
|---|
| *Asks the heuristic strategy to provide the next branching literal(s) given the current interpretation $I$.* |
| Allowed Response: |

| **#Choice($\mathcal{L}$)** |
|---|
| *Asks the solver to perform the choices in list $\mathcal{L}$. $\mathcal{L}$ contains tuples of the form $(at, sign)$, where $at \in \mathcal{A}'$ and $sign \in \{p, n, f\}$. Given tuple $(at, sign)$, the solver chooses $at$ if $sign = p$, $\neg at$ if $sign = n$, and if $sign = f$ the CDCL is left free to choose the sign.* |

| **#Unroll($\ell$)** |
|---|
| *Asks the solver to backtrack until $\ell$ becomes undefined. To trigger a restart set $\ell = \bot$.* |

| **#Fallback($n, \mathcal{I}, \mathcal{F}, \mathcal{S}$)** |
|---|
| *Asks the solver to use the default heuristics for the next $n$ choices. The default heuristic is enable permanently if $n \leq 0$. Parameters $\mathcal{I}, \mathcal{F}, \mathcal{S}$ initialize the default heuristics as follows: $\mathcal{I} : A \to \mathbb{N}$ provides the initial activity of atoms; $\mathcal{F} : A \to \mathbb{N}$ associates to atoms an amplifying factor; $\mathcal{S} : A \to \{p, n\}$ to provide a priority on the sign of literals.* |

| **#AddClause($\varphi$)** |
|---|
| *Asks the solver to add the clause $\varphi$ to the formula.* |

Table 2: Requests and responses.

*Example 4 (Pigeonhole problem heuristic).* Consider the famous pigeonhole problem: given $n$ pigeons and $m$ holes, assign each pigeon to exactly one hole in such a way that two different pigeons do not share the same hole. It can be verified that state-of-the-art SAT solvers are not able to solve this problem efficiently when $n > m$, despite it is known that no solution exists in that case. A domain-specific heuristic based on this observation only uses the requests and no event. To GetAtomsToBeFrozen($\mathcal{A}$) the driver answers to not remove any atom (providing #Freeze($\mathcal{A}$)), and computes $n$ and $m$ by interpreting the atoms in $\mathcal{A}$. Once the first choice is requested, if $n > m$ the heuristic replies with #AddClause($\{\bot\}$) causing the termination of the CDCL returning inconsistent; otherwise, #fallback($0, \emptyset, \emptyset, \emptyset$) is sent to go on with the default heuristic.

*Example 5 (Rintanen's heuristic).* In [24], a heuristic for solving the planning problem modeled as SAT problem is proposed that intuitively forces the solver to prefer short and simple plans. This criterion can be implemented in our framework by using the event OnSearch($\Gamma', \mathcal{A}$), and the request GetChoice($I$). In fact, OnSearch(.) allows

the driver to identify *actions* and *goals* literals; while the answer to GETCHOICE(.) is computed by running the algorithm in Figure 3 of [24].

We mention that, one can easily obtain all the VSIDS-like heuristics described in [6] by applying small modifications to the solution presented in Example 3.

## 4   Case study: Answer Set Programming

In this section we report on a case study that provides pragmatical evidence of the applicability of our framework. The case study is developed in the context of Answer Set Programming (ASP) [9]. ASP is a declarative problem solving paradigm proposed in the area of logic programming and non-monotonic reasoning. ASP is an ideal test bed for our framework for two main reasons: ($i$) ASP solvers are nowadays efficient CDCL implementations; ($ii$) ASP has been applied to a variety of complex problems including hard industrial ones. In the following we first describe the implementation of our framework in a well-known ASP solver and, then, we report on the solution of two real-world configuration problems occurring in practice of Siemens: the Partner Units Problem (PUP) and the Combined Configuration Problem (CCP). These appeared to be among the hardest industrial benchmarks from the ASP Competition 2015 [17, 18]; indeed, on hard problem instances state-of-the-art ASP solvers were not effective. Given the practical importance of these two problems, many researchers studied them in detail [13, 2, 3, 26, 10, 20]. Their investigations resulted in a number of domain-specific heuristic approaches that, in some cases, significantly outperformed existing ASP solvers. We experimented with these heuristics, thus applying our framework in practice, to see whether these can be integrated successfully in a CDCL solver.

### 4.1   Implementation of the framework

The framework has been implemented as an extension of the ASP solver WASP. WASP implements a variant of CDCL devised for evaluating ASP programs, thus featuring additional inference rules required for the evaluation of ASP programs, such as *unfounded-based* and *aggregate-based* inferences [16]. The framework for specifying heuristics has been implemented in *C++* and it offers an infrastructure for easy specification and testing of heuristic strategies. The user can adopt several languages for implementing external heuristics that range from imperative to declarative ones. In particular, the heuristic strategies can be implemented in *perl* and *python* for obtaining fast prototypes and in *C++* for obtaining better performances. (In our tests, we observed that *C++* implementations are often faster than equivalent *perl* and *python* counterparts, which are usually easier to develop instead). In case the user would like to use a declarative language to implement a heuristic, our implementation provides a *predicates-based* interface extending the one proposed in [15]. We next sketch the idea by assuming that (on the lines of in [15]) one encodes the heuristic strategy in ASP by providing a logic program $\Pi$. Each request of the CDCL algorithm causes an evaluation of $\Pi$ together with *facts* modeling the request (intuitively, a fact is an atom that is always true). The evaluation of $\Pi$ with the aforementioned facts produces a set of atoms (i.e., the

answer set produced by a solver) interpreted as responses for the request. For instance, consider Example 4, the request GETFROZENATOMS($\mathcal{A}$) is represented by the facts $get\_frozen\_atoms()$ and $atom(p(i,j))$ for each $p_{i,j} \in \mathcal{A}$. A call to an external solver produces an answer set containing $freeze(p(i,j))$, for each $p_{i,j} \in \mathcal{A}$, that is interpreted as #FREEZE($\mathcal{A}$). Other request and corresponding responses are handled in a similar manner. Our implementation currently supports any external solver compliant with ASP solver output. The predicate-based interface can easily support variants of ASP (e.g., with *preferences* [8], *CASP* [19], etc.) an could be adapted, in principle, also to support other languages based on predicate logic.

### 4.2 Partner Units Problem

The PUP comes from the railway safety domain [13], but has a variety of other applications including security monitoring systems, peer-to-peer networks, etc. [26]. In order to ensure the safety of train movements, railway tracks (see Figure 1) are equipped with hardware *sensors* $s_1, \ldots, s_6$ which register when a wagon passes by. All registered events are forwarded from sensors to *control units* $u_1, \ldots, u_3$ belonging to a safety system. The latter has to prevent wagons of different trains from entering the same safety *zone* of the tracks $z_1, \ldots, z_{24}$, hence avoiding unwanted collisions. To solve the problem one needs to (1) assign every zone and sensor to a control unit such that every unit is connected to at most UCAP sensors and at most UCAP zones. Moreover, (2) if a sensor contributes to a zone, it must be placed on the same or on a connected unit (partner unit). However, every unit can be a partner of at most IUCAP other units.

*Heuristics.* For our evaluation we selected the QUICKPUP heuristic and its derivatives, since search algorithms using these heuristics were able to outperform best ASP solvers.

  – QUICKPUP [26] generates an order of zones and sensors in the first step by selecting some zone as a root and traversing the input zone-sensor relations breadth-first. Then, a depth-first search selects the next unassigned zone or sensor in the order and assigns it to a new unit. If the assignment cannot be done due to a violation of requirement (2), then the already used units are tested starting from the last created one. In case QUICKPUP is not able to assign a zone or sensor to any unit it unrolls the last made assignment and tries another possibility. Finally, if no more assignments are possible, QUICKPUP falls back to the default heuristic.
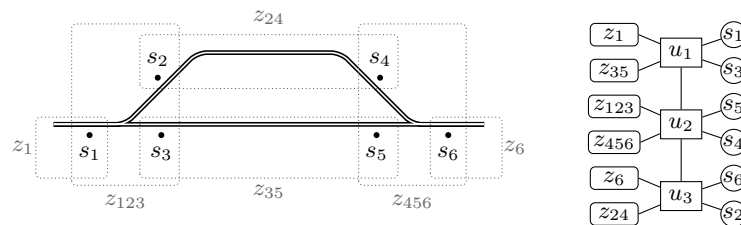


Fig. 1: A sample railway station layout and a solution of a corresponding PUP instance with UCAP = IUCAP = 2

- QUICKPUP* is a modification of QUICKPUP that tests already used units first and a new one afterwards.
- PRED [10] tries first to connect an unassigned zone or sensor to one of the units that already have connections to neighbour zones or sensors, i.e. reachable over at most two input zone-sensor relations. If this fails, PRED tries a new unit first and used ones subsequently.

*Implementation details.* We implemented the four heuristics in WASP using the C++ interface given in Section 3. A heuristic is initialized as ONSEARCH($\Gamma'$, $\mathcal{A}'$) event is invoked by the solver. Next, the ordering of vertices is generated and used to determine sets of chosen literals $L$ in response to GETCHOICE($I$) requests. In addition, a heuristic registers ONUNROLLLIT($\ell$) events in order to synchronize its inner state with decisions made by WASP between two subsequent GETCHOICE($I$) requests. If a heuristic finds that the current partial interpretation is inconsistent, i.e. no further assignments can be made, it backtracks and sends #UNROLL($\ell$) to WASP. In this case, $\ell$ is one of the literals that were selected by a heuristic at the previous choice event. Finally, the fallback from QUICKPUP-based heuristics to the default one was implemented using #FALLBACK($0, \emptyset, \emptyset, \emptyset$) response.

*Setup.* The evaluation was done by using the set of instances provided by Siemens[3]. This package comprises instances of four types: (1) double, (2) double-variant, (3) triple and (4) grid. The first three types are instances representing topologies occurring frequently in practice, whereas grids are parts of real railway systems (see [2] for details). The instances were tested with two encodings: PUP-E1 – the original straightforward encoding published in [2] (Section 3.1, p. 6) and PUP-E2 – a complex encoding comprising symmetry breaking and ordering rules used in the ASP Competition 2015. Moreover, in all experiments with WASP using heuristics, we removed rule 9 (line 14) from PUP-E2. This rule is a symmetry breaker that does not allow our heuristics to select the starting zone. For each encoding we grounded all instances with GRINGO (version 4.5.3). Then each solver was run for 900 sec. on a system equipped with i7-3030K CPU, 64GB RAM and Ubuntu 11.10.

*Results.* A summary of our evaluation results is presented in Figures 2 and 3. The results of CLASP [15], which was used as a reference system, show that it was able to solve at most 23 instances using the tested encodings. Moreover, in both experiments CLASP failed to find a model for most of the double as well as for some double-variant and triple instances. Also WASP with QUICKPUP solved only 12 and 15 instances using PUP-E1 and PUP-E2 respectively. The main reason for this discrepancy between our results and the ones obtained by [26] is due to inability of the underlying ASP solver to generate new units on-the-fly. Therefore, the number of ground literals denoting available units is determined in the grounding step and cannot be changed later. [26], instead, try to quickly build a solution by using new units whenever possible. This strategy appears to be successful for greedy algorithms, but works poorly as a heuristic for WASP.

The two remaining heuristics assign zones and sensors to already existing units first. This allowed WASP to outperform CLASP and QUICKPUP in all experiments. The best
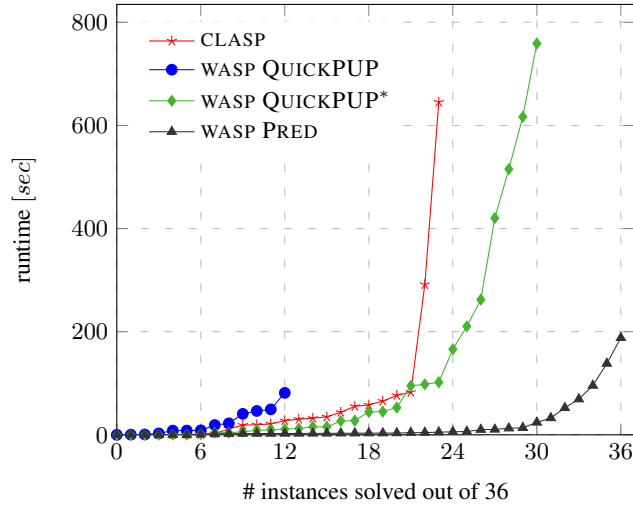
---

Fig. 2: Experimental results for 36 PUP instances [27] with PUP-E1 encoding

result was observed for WASP with PRED, which solved *all* instances independently of the encoding. Moreover, PUP-E2 allowed WASP to reduce its average solving time to only 27 sec. Finally, we would like to emphasize the fact that PRED allowed WASP to find solutions for all instances with the PUP-E1 encoding. This is an important feature of our approach. It allows a programmer to easily model a problem using a declarative language, like ASP-Core, and then to scale performance of the solver using heuristics defined in a procedural way.

### 4.3 Combined Configuration Problem

Similarly to the PUP, this problem was derived from a number of real-world problems including railway interlocking systems, safety automation, resource distribution, etc. The CCP is an abstract problem that demonstrates an important case occurring in practice of Siemens when a complex problem can be represented as a set of subproblems [20].

A CCP instance is defined by a directed acyclic graph $G = (V, E)$ representing a track layout (see Figure 4). Each vertex has a type, e.g. $b$ and $p$, with an associated size, e.g. 1 and 3. Furthermore, an instance includes two disjoint paths $P_1$ and $P_2$, shown in red and green, as well as several sets of border elements, e.g. $BE_1 = \{b_1, b_2, b_3\}$ and $BE_2 = \{b_2, b_4, b_5\}$. Finally, it defines parameters specifying a maximal number of border elements per safety area $M$, a number of colors $C$, a number of available bins $B$ per color and capacity of a bin $K$. The goal is to find a solution that satisfies all requirements of the following subproblems:

**(P1) Coloring** Every vertex must have exactly one color.
**(P2) Bin-Packing** For every set $V_i \subseteq V$ comprising all vertices of one color, assign each vertex $v_j \in V_i$ to exactly one bin such that for each bin the sum of the vertex sizes is less or equal to $K$ and at most $B$ bins are used.
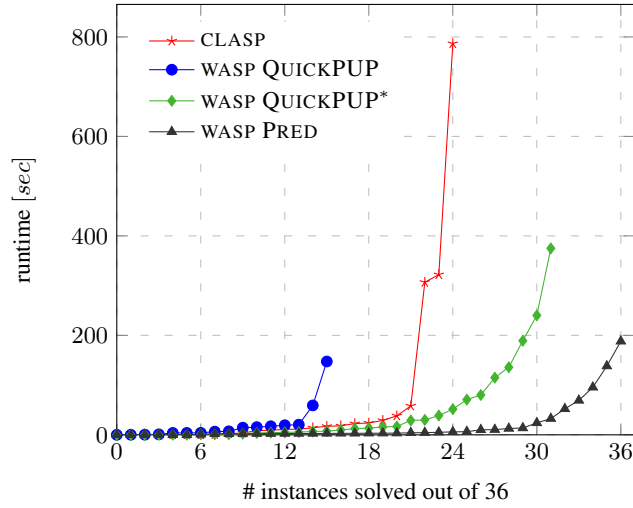
Fig. 3: Experimental results for 36 PUP instances [27] with PUP-E2 encoding
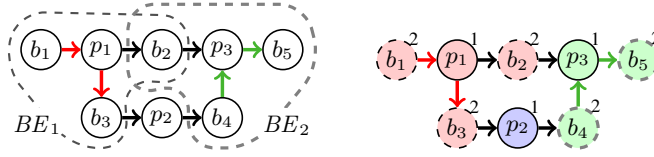


Fig. 4: Sample CCP instance (left) and its solution (right).

**(P3) Disjoint Paths** Colors of any two vertices $v_i \in P_1$ and $v_j \in P_2$ must be different.

**(P4) Matching** For every input set of border elements $BE_i$ find a safety area $A_i \subseteq BE_i$ such that all border elements of one area are colored in one color, $|A_i| \leq M$ and every border element is exactly in one area.

**(P5) Connectedness** Two vertices of the same color must be connected via a path that comprises only vertices of this color.

*Example 6.* Let the sample CCP instance comprise the following parameters: $B = 2$, $K = 3$, $C = 3$ and $M = 3$. The solution presented in Figure 4 assigns 3 colors to vertices, places them into 2 bins, shown in subscript of each vertex, and defines two safety areas $A_1 = \{b_1, b_2, b_3\}$ and $A_2 = \{b_4, b_5\}$.

*Heuristics.* The CCP has a number of heuristics suggested in [20] which can be used to obtain a partial solution of a problem instance. It turned out that it is quite simple to find a heuristic that solves one or some of the subproblems. However, no complete heuristics, like QUICKPUP or PRED for PUP, are known for CCP.

The heuristics suggested in [20] were designed to be used with the heuristic interface of CLASP. This interface allows a programmer to define a heuristic function in the program using *_heuristic*/4 predicate. If a ground "heuristic" atom over this predicate

becomes true in a partial interpretation, then terms of the atom are used by CLASP to modify behavior of the VSIDS heuristic. Therefore, given a CCP instance, the proposed approach first runs a heuristic search method to find a partial solution of the instance. Next, CLASP is forced to start the search from the found partial solution by expending the CCP instance with facts comprising "heuristic" atoms representing it. Gebser et al. [20] use three heuristics:

- A1 solves the matching problem (P4). In every iteration it assigns an unassigned border element to a safety area which comprises the minimum number of already assigned elements.
- A2. This heuristic is designed to solve three subproblems (P1), (P2) and (P5). First, it sets the color to 1 and adds some vertex $v \in V$ to a queue $Q$. Next, it retrieves an element $v_i \in Q$, assigns it the current color, places it to a bin according to the best-fit heuristic and adds all uncolored neighbors of $v_i$ to $Q$. The process continues, until no further vertices can be placed to bins. In this case the algorithm increases the value of current color and adds some uncolored node $v_j$ to $Q$.
- A1A2. This is a combination of the previous two which first applies A1 and then uses A2 to extend the output of A1.

All original heuristics were designed to run only once prior to solving whereas in our solver a heuristic is queried multiple times. Therefore, we suggest the following modifications:

- A2F is based on A2 which is extended with a fallback to a default solver heuristic if (i) a partial solution is found or (ii) the 10 seconds timeout is reached or (iii) A2 cannot make a choice. The latter situation might occur if all atoms used in subproblems (P1), (P2) and (P5) already have assigned truth values.
- A2FO. Originally A2F processes the vertices of $G$ in no specific order. To increase its performance we extended A2F with an ordering that assigns a score of 2 to all vertices with only incoming or outgoing edges, a score of 1, it is in a path, and 3, if both applies. Then, A2FO runs A2F such that in every iteration it sorts the unprocessed vertices in the decreasing order of their scores.
- A2AFO. This version of A2FO uses a fallback strategy which alternates between A2FO and a default solver heuristic. Thus, A2AFO runs A2FO with the first ordering until it cannot make any further choices. Then, it switches to the default heuristic for the next 10 seconds. If solver was unable to get a complete solution within this time, A2AFO creates a new ordering and executes A2FO once more. In the case all possible orderings starting with vertices with a score higher than zero has been used, A2AFO falls back to the default heuristic permanently. The idea is to allow the default heuristic of the solver to acquire enough representative conflicts and hence to improve its performance.

*Implementation details.* The heuristics were implemented similarly to the ones used for the PUP using the C++ version of the WASP interface. When A2AFO decides to use the default solver heuristic, it starts a timer and responses all GETCHOICE($I$) requests with #FALLBACK($1, \emptyset, \emptyset, \emptyset$). When the timeout is reached, A2AFO generates another ordering, restarts the search using #UNROLL($\bot$) and answers all subsequent requests itself.
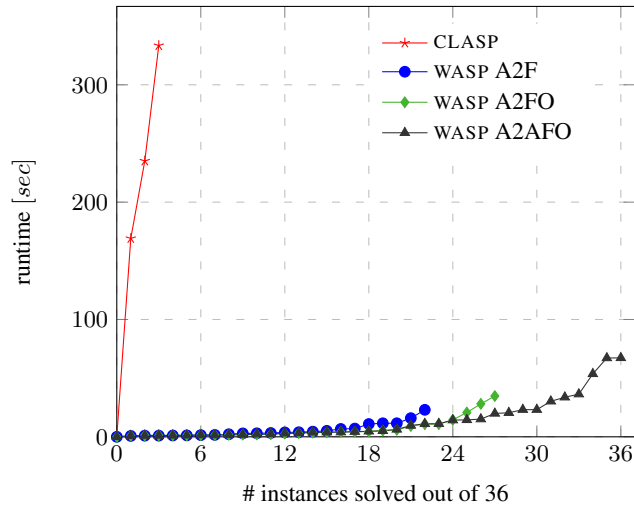
Fig. 5: Evaluation results for 36 CCP instances with [20] encoding

*Setup.* We evaluated A2F, A2FO and A2AFO on the set of hard instances[4] and the encoding submitted to the ASP Competition 2015. The package contains a set of real-world instances used in the evaluation of [20] as well as a number of new instances. The latter have a grid-like graph structure in which all vertices are of the same type. The evaluation was done in the same settings as the PUP and similar to [20]. The average grounding time was about 2 seconds and is excluded from results.

*Results.* The evaluation results, presented in Figure 5, show that the reference system could solve 3 out of 36 instance with an average solution time of 246 sec. Our approach using A2F, A2FO and A2AFO heuristics was able to solve 22, 27 and 36 instances with an average solution time 6, 7, and 14 sec. resp. A2AFO heuristic has a greater solving time only because it was able to solve the hardest instances. Our approach shows also better results in comparison with [20], which solved 6 out of 16 real-world instances.

Moreover, since A2AFO is based on an alternating usage of two heuristics we performed an additional evaluation. The goal was to test whether conflict learning is essential or the heuristic has to find a "correct" ordering as it was often the case for PUP heuristics. Therefore, we saved the last ordering A2AFO used before the last fallback and then started a new experiment in which this ordering was used first. The results show that in none of the experiments this strategy allowed WASP to find a solution with only one fallback. In all test cases A2AFO run A2 at least twice, i.e. generates at least two orderings, to find a solution. This indicates that learned clauses are essential.

---

[4] http://isbi.aau.at/hint/images/ccp/ccp.tgz

# 5 Conclusion

The branching heuristic is a key ingredient of the CDCL algorithm. Despite several effective heuristics have been proposed in the literature, their integration in existing solvers is usually nontrivial. The paper proposed a novel framework that simplifies the design of new heuristics for CDCL solvers by distilling the key ingredients useful to drive the search. The proposal has been validated in a case study where our framework was employed for solving effectively two hard industrial problems. Results are positive: hard instances that were not solvable by previous approaches are efficiently solved by our implementation. Ongoing work concerns the implementation of our framework in other well-known solvers. Moreover, we plan to apply it in further practical use cases.

## Acknowledgements

## References

1. Alviano, M., Dodaro, C., Leone, N., Ricca, F: Advances in WASP. In LPNMR 2015. pp. 40–54. (2015)
2. Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., Thorstensen, E.: Optimization Methods for the Partner Units Problem. In: CPAIOR 2011. pp. 4–19. (2011)
3. Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., Thorstensen, E.: Tackling the Partner Units Configuration Problem. In: IJCAI 2011. pp. 497–503. (2011)
4. Audemard, G., Simon, L.: Predicting Learnt Clauses Quality in Modern SAT Solvers. In: IJCAI 2009. pp. 399–404. (2009)
5. Beame, P., Kautz, H.A., Sabharwal, A.: Towards Understanding and Harnessing the Potential of Clause Learning. J. Artif. Intell. Res. (JAIR) 22, 319–351 (2004)
6. Biere, A., Fröhlich, A.: Evaluating CDCL Variable Scoring Schemes. In: SAT 2015. pp. 405–422. (2015)
7. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
8. Brewka, G., Delgrande, J.P., Romero, J., Schaub, T.: asprin: Customizing Answer Set Preferences without a Headache. In: AAAI 2015. pp. 1467–1474. (2015)
9. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
10. Drescher, C.: The Partner Units Problem a Constraint Programming Case Study. In: ICTAI 2012. pp. 170–177. (2012)
11. Eén, N., Biere, A.: Effective Preprocessing in SAT Through Variable and Clause Elimination. In: SAT 2005. pp. 61–75. (2005)
12. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: SAT 2003. pp. 502–518. (2003)

13. Falkner, A., Haselböck, A., Schenner, G.: Modeling Technical Product Configuration Problems. In: Workshop on Configuration, ECAI 2010. pp. 40–46. (2010)
14. Friedrich, G.: Industrial success stories of ASP and CP: What's still open? (2015), joint invited talk at ICLP and CP 2015
15. Gebser, M., Kaufmann, B., Romero, J., Otero, R., Schaub, T., Wanko, P.: Domain-Specific Heuristics in Answer Set Programming. In: AAAI 2013. (2013)
16. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187, 52–89 (2012)
17. Gebser, M., Maratea, M., Ricca, F.: The design of the sixth answer set programming competition report. In: LPNMR 2015. pp. 531–544. (2015)
18. Gebser, M., Maratea, M., Ricca, F.: The sixth answer set programming competition web site. http://aspcomp2015.dibris.unige.it/
19. Gebser, M., Ostrowski, M., Schaub, T.: Constraint Answer Set Solving. In: ICLP 2009. pp. 235–249. (2009)
20. Gebser, M., Ryabokon, A., Schenner, G.: Combining Heuristics for Configuration Problems Using Answer Set Programming. In: LPNMR 2015. pp. 384–397. (2015)
21. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: Methods & evaluation. Artif. Intell. 206, 79–111 (2014)
22. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001. pp. 530–535. (2001)
23. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL($T$). J. ACM 53(6), 937–977 (2006)
24. Rintanen, J.: Planning as satisfiability: Heuristics. Artif. Intell. 193, 45–86 (2012)
25. Silva, J.P.M., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)
26. Teppan, E.C., Friedrich, G., Falkner, A.A.: QuickPup: A Heuristic Backtracking Algorithm for the Partner Units Configuration Problem. In: IAAI 2012. (2012)
27. Heuristic Wasp Website. http://yarrick13.github.io/hwasp/
28. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: ICCAD 2001. pp. 279–285. (2001)