TLP **16** (5–6): 653–669, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000284

Combining Answer Set Programming and domain heuristics for solving hard industrial problems (Application Paper)

CARMINE DODARO[1], PHILIP GASTEIGER[2], NICOLA LEONE[1],
BENJAMIN MUSITSCH[2], FRANCESCO RICCA[1],
and KOSTYANTYN SHCHEKOTYKHIN[2]

[1]*Department of Mathematics and Computer Science, University of Calabria, Italy*
(*e-mail:* {lastname}@mat.unical.it)
[2]*Alpen-Adria-Universität Klagenfurt, Austria*
(*e-mail:* {firstname.lastname}@gmail.com)

submitted 6 May 2016; revised 8 July, 2016; accepted 22 August, 2016

Abstract

Answer Set Programming (ASP) is a popular logic programming paradigm that has been applied for solving a variety of complex problems. Among the most challenging real-world applications of ASP are two industrial problems defined by Siemens: the Partner Units Problem (PUP) and the Combined Configuration Problem (CCP). The hardest instances of PUP and CCP are out of reach for state-of-the-art ASP solvers. Experiments show that the performance of ASP solvers could be significantly improved by embedding domain-specific heuristics, but a proper effective integration of such criteria in off-the-shelf ASP implementations is not obvious. In this paper the combination of ASP and domain-specific heuristics is studied with the goal of effectively solving real-world problem instances of PUP and CCP. As a byproduct of this activity, the ASP solver WASP was extended with an interface that eases embedding new external heuristics in the solver. The evaluation shows that our domain-heuristic-driven ASP solver finds solutions for all the real-world instances of PUP and CCP ever provided by Siemens.

KEYWORDS: Answer Set Programming, Domain Heuristics, Industrial Applications

1 Introduction

Answer Set Programming (ASP) (Brewka *et al*. 2011) is a programming paradigm which has been proposed in the area of logic programming and non-monotonic reasoning. ASP has become a popular paradigm for solving complex problems since it combines high knowledge-modeling power (Dantsin *et al*. 2001) with robust solving technology. ASP has been applied in the areas of Artificial Intelligence (Balduccini *et al*. 2001; Gaggl *et al*. 2015; Baral and Uyan 2001; Erdem *et al*. 2013), Bioinformatics (Erdem and Öztok 2015; Koponen *et al*. 2015; Campeotto *et al*. 2015), and Databases (Marileo and Bertossi 2010; Manna *et al*. 2015), to mention a few. ASP has also been attracting the interest of companies (Grasso *et al*. 2011; Friedrich 2015).

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

Among the most challenging industrial applications of ASP is the solution of two problems defined by Siemens, namely the Partner Units Problem (PUP) (Falkner *et al*. 2011) and the Combined Configuration Problem (CCP) (Gebser *et al*. 2015c). The PUP originates from the railway safety domain and it generalizes a variety of other industrial problems in the area of security monitoring, peer-to-peer networking, etc. (Teppan *et al*. 2012). The CCP abstracts a family of configuration problems recurring in the practice of Siemens with application in systems of railway interlocking, safety automation, resource distribution, etc. Given their practical importance, PUP and CCP have been the subject of research in the last few years (Falkner *et al*. 2011; Aschinger *et al*. 2011b; Teppan *et al*. 2012; Drescher 2012; Gebser *et al*. 2015c).

The solution of these problems using ASP was quite successful from the perspective of modeling. Indeed, both natural and optimized logic programs for PUP and CCP were proposed in (Aschinger *et al*. 2011a; Gebser *et al*. 2015c; Calimeri *et al*. 2016). However, state-of-the-art ASP solvers (including portfolios) fail to compute solutions of the hardest real-world instances provided by Siemens in a reasonable amount of time (where finding a solution is the main real-world requirement). Indeed, the instances of PUP and CCP were among the hardest benchmarks of the recent ASP Competitions (Gebser *et al*. 2015b; Calimeri *et al*. 2016). One possible approach for boosting the performance of an ASP solver on specific problems is to provide a domain-specific heuristic (Gebser *et al*. 2013). This idea can be particularly effective for solving real-world industrial problems. As stated by Gerhard Friedrich in his joint invited talk at CP-ICLP 2015, "domain-specific heuristics turned out to be the key component in several industrial applications of problem solvers" (Friedrich 2015). The first attempts of solving CCP instances by combining domain-heuristics and an off-the-shelf ASP implementation have been very promising (Gebser *et al*. 2015c). Nonetheless, the challenge of solving the hardest real-world instances using ASP remained open (Gebser *et al*. 2015c).

In this paper, the combination of ASP and domain-specific heuristics is studied with the goal of providing effective solutions to PUP and CCP. Several heuristic criteria are considered, including the ones already proposed, as well as novel ones. The heuristics were implemented by extending the ASP solver WASP (Alviano *et al*. 2015), and their performance is empirically evaluated on the real-world instances of PUP and CCP provided by Siemens.

One of the lesson learned dealing with the solution of PUP and CCP is that finding an effective combination of domain-heuristics with ASP is not obvious. On the one hand, heuristic criteria are usually invented by domain experts studying the properties of solutions. Thus, it is likely that such criteria do not fit the working principles of solvers and are not always embeddable in ASP encodings. On the other hand, despite the fact that an average developer could easily come up with an implementation of a heuristic criterion, the final embedding of an external algorithm in ASP solvers requires in depth knowledge of the internals of ASP implementations, which are nowadays very optimized and sophisticated.

Therefore, the paper provides a pragmatic contribution in this respect. Indeed, as a byproduct of the research on PUP and CCP, WASP was extended with an interface that makes it easier for developers to embed new external heuristics in the solver. The interface provides simplified access to the procedures that repeatedly needed modifications in our work. The extension of WASP presented in this paper offers a wide range of possibilities to

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

end users for developing new heuristics by implementing the above-mentioned interface. In particular, it offers multi-language support including scripting languages for fast prototyping, as well as an embedded C++ interface for performance-oriented implementations. In our experience the implementation of several heuristics resulted to be much easier (with respect to the development of several ad-hoc modification) after WASP was extended with the new interface.

The contributions of this paper can be summarized as follows:

1. The combination of ASP and domain heuristics for solving PUP and CCP is studied and effective solutions are presented (see Section 3).
2. An extension of WASP with an interface that eases the design and implementation of new external heuristics is presented (see Section 4).
3. An experiment is carried out to both study the behavior of WASP equipped with several domain heuristics and compare it to state-of-the-art solutions (see Section 5).

The results of the experiments are positive: A combination of ASP and domain heuristics can be used to solve effectively both PUP and CCP. Notably, WASP empowered with the best considered domain-heuristic was able to solve all the hardest real-world instances ever provided by Siemens including those for which no solution was known before.

2 Preliminaries

In this section, Answer Set Programming (ASP) language and contemporary solving techniques are over-viewed first; then the reader is provided with a description of the problems approached in this paper, namely the Partner Units (PUP) and the Combined Configuration Problem (CCP). Hereafter the reader is assumed to be familiar with ASP (Brewka *et al.* 2011).

2.1 Answer Set Programming

Syntax and Semantics. Let $\mathscr{A}$ be a fixed, countable set of propositional atoms including $\bot$. A literal ℓ is either an atom a (a positive literal), or an atom preceded by the negation as failure symbol $\sim$ (a negative literal). The atom associated to ℓ is denoted by $atom(\ell) = a$. The complement of ℓ is denoted by $\overline{\ell}$, where $\overline{a} = {\sim}a$ and $\overline{{\sim}a} = a$ for an atom a. For a set L of literals, $\overline{L} := \{\overline{\ell} \mid \ell \in L\}$, $L^+ := L \cap \mathscr{A}$, and $L^- := \overline{L} \cap \mathscr{A}$. A program Π is a finite set of rules. A rule is an implication $a \leftarrow l_1, \ldots, l_n$, where a is an atom, and $l_1, \ldots, l_n$ are literals, $n \geqslant 0$. For a rule r, $H(r) = \{a\}$ is called the head of r and $B(r) = \{l_1, \ldots, l_n\}$ is called the body of r. A rule r is a fact if $B(r) = \emptyset$, and is a constraint if $H(r) = \{\bot\}$. A (partial) interpretation is a set of literals I containing ${\sim}\bot$. I is inconsistent if $I^+ \cap I^- \neq \emptyset$, otherwise I is consistent. I is total if $I^+ \cup I^- = \mathscr{A}$. Given an interpretation I, a literal ℓ is true if $\ell \in I$; is false if $\overline{\ell} \in I$, and is undefined otherwise. An interpretation I satisfies a rule r if $I \cap (H(r) \cup \overline{B(r)}) \neq \emptyset$. Let Π be a program, a model I of Π is a consistent and total interpretation that satisfies all rules in Π. The reduct of Π w.r.t. I is the program Π^I obtained from Π by (i) deleting all rules r having $B(r)^- \cap I \neq \emptyset$, and (ii) deleting the negative body from the remaining rules (Gelfond and Lifschitz 1991). A model I of a program Π is an answer set if there is

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

no model J of Π^I such that $J^+ \subset I^+$. A program Π is coherent if it admits answer sets, otherwise it is incoherent.

Answer Set Computation. The computation of answer sets can be carried out by employing an extended version of the Conflict-Driven Clause Learning (CDCL) algorithm, which was introduced for SAT solving (Biere *et al*. 2009). The algorithm takes as input a program Π, and outputs an answer set if Π is consistent, otherwise it outputs $\{\bot\}$.

The computation starts by applying polynomial simplifications to strengthen and/or remove redundant rules on the lines of methods employed by SAT solvers (Biere *et al*. 2009). After the simplifications step, the partial interpretation I is set to $\{\sim\bot\}$, 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). In particular, in ASP three propagation rules are applied: unit, support and unfounfed-free. Roughly, unit propagation is as in SAT whereas support propagation and unfounded-free propagation are ASP-specific (Gebser *et al*. 2015a; Alviano *et al*. 2015). Three cases are possible after a propagation step is completed: (i) I is consistent but not total. In that case, an undefined literal ℓ (called branching literal) is chosen according to some heuristic criterion, and is added to I. Subsequently, 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 constraint r that is added to Π (learning). Moreover, the algorithm backtracks (i.e. choices and their consequences are undone) until the consistency of I is restored. The algorithm then propagates inferences starting from the fresh constraint r. Otherwise, if the consistency of I cannot be restored, the algorithm terminates returning $\{\bot\}$. Finally, in case (iii) I is consistent and total, the algorithm terminates returning I.

State-of-the-art CDCL implementations usually employ a heuristic for choosing the branching literal that can be seen as a variant of the MINISAT (Eén and Sörensson 2003) heuristic. The MINISAT heuristic is based on the *activity* value, that is initially set to 0 for each atom. Whenever a literal ℓ occurs in a learned constraint, the activity of $atom(\ell)$ is incremented by a value *inc*. Then, after each learning step, the value of *inc* is multiplied by a constant slightly greater than 1, to promote variables that occur in recently-learned constraints. Once a choice is needed, the literal $\sim a$ is chosen, where a is the undefined atom having the highest activity value (ties are broken randomly). In the following, the MINISAT (Eén and Sörensson 2003) heuristic is referred to as the default branching heuristic of a solver.

For the sake of completeness, we mention that the CDCL algorithm is usually complemented with heuristics that control the number of learned constraints, and restart the computation to explore different branches of the search tree (Biere *et al*. 2009).

2.2 Partner Units Problem (PUP)

The PUP originates from the railway safety domain (Falkner *et al*. 2011) and has a variety of other applications, including security monitoring systems and peer-to-peer networks. (Teppan *et al*. 2012). To ensure the safety of the train traffic, railway tracks (see Figure 1) are equipped with hardware *sensors*, e.g. $s_1, \ldots, s_6$, registering wagons passing by. These sensors are organized in safety *zones*, e.g. $z_1, \ldots, z_{24}$. In order to prevent wagons

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

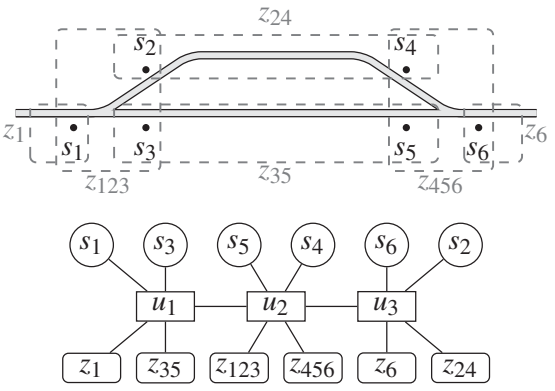


Fig. 1. Sample PUP instance (top) and a solution with $\text{UCAP} = \text{IUCAP} = 2$ (bottom).

of different trains from entering the same zone simultaneously, the safety system has a set of *control units*, e.g. u_1, u_2, u_3, that enforce safety requirements on connected zones and sensors. Consequently, every PUP instance consists of (i) a layout of sensors S and zones Z represented as an undirected bipartite graph $G = (S, Z, E)$, where $S \cup Z$ are vertices connected by edges $E \subseteq S \times Z$; (ii) a set of available units U; (iii) a natural number UCAP equal to the maximum number of sensors/zones connected to a unit; and (iv) a natural number IUCAP equal to the maximum number of inter-unit connections.

A solution of a PUP instance is an assignment of zones and sensors to units and an interconnection of units, such that (i) every unit is connected to at most UCAP sensors and at most UCAP zones; (ii) every unit is connected to at most IUCAP *partner units*; and (iii) if a sensor s is part of a zone z, then s must be connected to the same or a partner unit of the unit connected to z.

2.3 Combined Configuration Problem (CCP)

The CCP abstracts a number of real-world problems including railway interlocking systems, safety automation, and resource distribution. It occurs in practical applications at Siemens, where a complex problem is composed of a set of subproblems (Gebser *et al.* 2015c).

A CCP instance (see Figure 2) consists of (i) a directed acyclic graph $G = (V, E)$, where each vertex has a type with an associated size, e.g. b and p with sizes 1 and 3 resp.; (ii) two disjoint paths P_1 and P_2, e.g. edges highlighted in red and green; (iii) a set of safety areas and their border elements, e.g. $A_1 = \{b_1, b_2, b_3, p_1\}$ and $A_2 = \{b_2, b_4, b_5, p_3\}$ with border elements $BE_1 = \{b_1, b_2, b_3\}$ and $BE_2 = \{b_2, b_4, b_5\}$; (iv) the maximum number M of assigned border elements per area, e.g. $M = 3$; (v) the number C of colors, e.g. $C = 3$; (vi) the number B of bins available per color, e.g. $B = 2$; and (vii) the capacity K of each bin, e.g. $K = 3$. A solution is an assignment of colors to vertices, vertices to bins and border elements to areas, such that the following subproblems are solved:

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

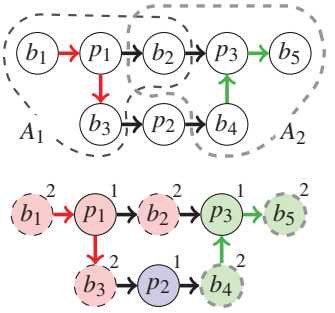


Fig. 2. Sample CCP instance (top) and its solution (bottom).

(P1) *Coloring:* Every vertex must have exactly one color.
(P2) *Bin-Packing:* For each set $V_i \subseteq V$ of color i, assign each $v \in V_i$ to exactly one bin s.t. for each bin the sum of vertex sizes is not greater than K, and at most B bins are used.
(P3) *Disjoint Paths:* The color of any two vertices $u \in P_1$ and $v \in P_2$ must be different.
(P4) *Matching:* For each area A assign a set of border elements BE, such that all border elements have the same color and A has at most M border elements. Additionally, each border element must be assigned to exactly one area.
(P5) *Connectedness:* Two vertices of the same color must be connected via a path that comprises only vertices of this color.

3 Domain Heuristics

In this section, we study several domain-heuristics for solving PUP and CCP and their embedding in an ASP solver. The implementation of different domain-heuristics required to extend the CDCL computation more or less the same in some specific points. The interface of methods reported in Table 1 (and described below) abstracts some of these points and allows us to factor out the details of the heuristics from the main CDCL algorithm.

Heuristics Interface. The implementation of a heuristic may require information regarding the current state of the search. To this end, *events* (see Table 1 upper part) are triggered when certain points of the CDCL algorithm are reached. The status of the heuristic can be thus initialized or updated by handling events. For example, the OnSearch event may be used to initialize some heuristic counters at the beginning of the search. The OnLitTrue and OnUnrollLit events can be used to keep track of the current partial interpretation. Whereas OnIncoChoice, OnConflict, and OnRestart events might be used to handle conflicts and restarts, respectively. The OnLearnConstraint event could be used to keep track of changes to the logic program that occur during the learning process. To drive the search for a solution, the heuristic issues commands (see Table 1 lower part). This is done whenever a new branching literal is required by the CDCL algorithm, i.e. whenever the OnChoiceRequired event is triggered. In more detail, the command #Choose(ℓ) instructs the solver to use ℓ as the next branching literal. Since domain-specific information may allow to recognize a dead end in advance, the heuristics may add a new constraint c (issuing #AddConstraint(c)) or backtrack to an earlier choice

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

Table 1. *Events of the solver and commands issued by the heuristic*

	CDCL ⟶ Heuristic: Events
ONSEARCH($\Pi', \mathscr{A}'$)	*Triggered when the backtracking search starts. Π' and $\mathscr{A}'$ are the program and the atoms after the simplifications, respectively.*
ONINCOCHOICE(ℓ)	*Triggered when the choice ℓ led to an inconsistency.*
ONCONFLICT(ℓ)	*Triggered when a conflict is detected. After analyzing the conflict the consistency is restored, i.e. choices and their consequences are undone. ℓ is the latest valid branching literal after the analysis.*
ONLEARNCONSTRAINT(c)	*Triggered when the constraint c is learned.*
ONLITTRUE(ℓ)	*Triggered when a literal ℓ is set to true.*
ONRESTART( )	*Triggered when a restart of the search occurs.*
ONUNROLLLIT(ℓ)	*Triggered during backtracking if the literal ℓ is set to undefined.*
ONCHOICEREQUIRED( )	*Triggered when a choice is required.*
	Heuristics ⟶ CDCL: Commands
#CHOOSE(ℓ)	*Choose ℓ as next branching literal.*
#UNROLL(ℓ)	*Unroll the previous choices until ℓ becomes undefined. To trigger a restart set $\ell = \bot$.*
#FALLBACK($n, \mathscr{I}, \mathscr{F}, \mathscr{S}$)	*Use the default heuristics for the next n choices. The default heuristic is enabled permanently if $n \leqslant 0$. Parameters $\mathscr{I}, \mathscr{F}, \mathscr{S}$ initialize the default heuristics as follows: $\mathscr{I} : A \to \mathbb{N}$ provides the initial activity of atoms; $\mathscr{F} : A \to \mathbb{N}$ associates to atoms an amplifying factor; $\mathscr{S} : A \to \{pos, neg\}$ to provide a priority on the sign of literals.*
#ADDCONSTRAINT(c)	*Add the constraint c to the program.*

ℓ (issuing #UNROLL(ℓ)). Finally, when the heuristic is unable to compute further choices, the #FALLBACK($n, \mathscr{I}, \mathscr{F}, \mathscr{S}$) command instructs the solver to use the default heuristic for n steps.

Note that one may choose a different interface to implement heuristics in CDCL solvers. The ones reported above are expressive enough to implement the domain heuristics for PUP and CCP as detailed below, and to provide a high-level description of our implementations.

3.1 PUP Heuristics

A number of PUP heuristics are suggested in the literature (Aschinger *et al*. 2011b; Ryabokon 2015). We consider the QUICKPUP (Teppan *et al*. 2012) heuristic and its derivatives, since ad hoc algorithms based on them outperform the best off-the-shelf ASP solvers:

- QUICKPUP first generates an order of zones and sensors. To do so, given the input graph $G = (S, Z, E)$, the heuristic traverses the edges in E breadth-first, starting with

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

some zone $z \in Z$. Each time a new branching literal is requested, the next unassigned zone or sensor with respect to the order is selected. Afterwards, the selected zone or sensor is assigned to a unit, preferring new units (i.e. units without any zones or sensors assigned). If no assignment is possible, the last one performed is unrolled and another assignment is tried.

- QUICKPUP* is a variation of QUICKPUP that prefers used units instead of new units.
- PRED (Drescher 2012) is a variation of QUICKPUP that prefers units assigned to neighbors of the considered vertex (i.e. units with zones or sensors that are reachable within two edges in the input graph starting with the current zone or sensor).

The integration of the PUP heuristic in the CDCL algorithm is done using the interface presented in Table 1. When ONSEARCH is triggered, the heuristic is initialized. This is done by parsing the atoms that represent sensors, zones, the zone-to-sensor relation, available units, UCAP, and IUCAP. Furthermore, the order of vertices is generated according to one of the heuristic criteria described above. Whenever a branching literal is required, the order is used to determine the next unassigned zone or sensor. This zone or sensor is then assigned to a unit by returning the #CHOOSE(ℓ) command, where ℓ is the literal corresponding to the appropriate zone-to-unit or sensor-to-unit atom. The implementation keeps track of the already done assignments in a list. If the solver retrieves a conflict, the ONCONFLICT event is triggered and the heuristic restores a consistent state. In addition, ONUNROLLLIT and ONLITTRUE are used to synchronize the internal state of the heuristics with the one of the CDCL algorithm. Finally, the heuristic may recognize that the current partial assignment cannot be completed to a valid solution. In this case, a restart is performed by returning the #UNROLL($\bot$) command and subsequently adding a constraint c preventing the partial assignment using the #ADDCONSTRAINT(c) command.

3.2 CCP Heuristics

Various heuristics for computing partial solutions of the CCP have been suggested in (Gebser *et al*. 2015c). However, in contrast with PUP, no complete heuristic for CCP is known. There are heuristics for the subproblems, e.g. best-fit for bin packing or minimal degree for graph coloring (Garey and Johnson 1979), but the task of combining them is non-trivial.

The following heuristics are proposed in (Gebser *et al*. 2015c):

- Algorithm 1 (A1) solves the matching subproblem (P4). It iterates over the set of border elements and assigns them to the area with the lowest number of border elements so far.
- Algorithm 2 (A2) solves the subproblems (P1), (P2) and (P5). It starts by creating a queue Q, initially filled with some arbitrary vertex $v \in V$ and setting the initial color c. Afterwards, the first vertex v in Q is colored with c and placed into a bin according to the *best-fit* heuristic. If successful, all uncolored neighbors of v are added to Q; otherwise color of v is removed. The process is repeated until the queue is empty. In this case, the algorithm gets the next color and adds some arbitrary uncolored vertex to Q.
- A1A2 applies both of the previous heuristics, extending the output of A1 by using A2.

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

We elaborate in our study the following variants:

- A2F is based on A2. A fallback to the default heuristic is done when (i) a partial solution is found, (ii) the heuristic is used for 10 seconds, or (iii) the heuristic can not make further choices. The latter case might occur if all atoms used in the subproblems have been assigned or ignored.
- A2FO extends A2F with a descending ordering of all vertices in G as well as in Q w.r.t. a score. Vertices having either only incoming or only outgoing edges are rated with a score of 1, all other vertices are rated with a score of 0.
- A2AFO is a variant of A2FO with an alternating fallback strategy between A2FO and the solvers default heuristic. The heuristic strategy is switched every 10 seconds. When A2FO is restarted, a new order is created. If all possible orders have been tried, A2FO falls back to the default heuristic permanently. Intuitively, the default heuristic of the solver may become more effective when a number of representative conflicts are acquired.

The heuristics given above are implemented similarly to the ones for the PUP. First, when the ONSEARCH event is triggered, (i) the input graph is read, and (ii) an empty queue Q is created, which is used to extract the next choices. In addition, the A2FO and A2AFO heuristics assign a score to the vertices of the input graph, as described above (all other heuristics assign a score of 0 to all vertices). The set of all vertices (order) is sorted with respect to their score. Whenever a branching literal is requested, a vertex v is selected, where v is the first element of the queue Q. If the queue is empty, v is the first uncolored and unassigned vertex among all vertices. At this point, the heuristic computes a color or bin assignment for v, prioritizing the first one. The atom ℓ corresponding to the obtained assignment is returned to the CDCL algorithm using the #CHOOSE(ℓ) command. If v was successfully colored or assigned to a bin, (i.e. no conflict is implied by the choice), Q is updated as follows: (i) v is removed from Q, (ii) all non-processed neighbors of v are added to Q, (iii) vertices in Q are ordered according to their score. Otherwise, if the solver finds a conflict (event ONCONFLICT), the heuristic restores a consistent state of the queue, order, and choices. The color and bin assignments made by the CDCL algorithm are tracked by the heuristics using the ONUNROLLLIT and ONLITTRUE events. When the queue is empty and all vertices in the order have been considered, the control is given back to CDCL by using the #FALLBACK$(0,\emptyset,\emptyset,\emptyset)$ command to complete the partial assignment. Finally, in the variants requiring alternation of heuristics with the default strategy, the #FALLBACK$(1,\emptyset,\emptyset,\emptyset)$ and #UNROLL($\bot$) commands are issued every 10 seconds.

4 Implementation

We implemented an infrastructure for easy specification and testing of heuristic strategies as an extension of the ASP solver WASP (Alviano *et al*. 2015). WASP accepts as input a propositional program encoded in the numerical format produced by the grounder GRINGO (Gebser *et al*. 2011) and uses a CDCL-like algorithm for computing answer sets (see Section 2.1). Our extended version of WASP can additionally ask for guidelines from an external module implementing a new heuristic. The communication with external heuristics follows a synchronous message passing protocol. The protocol is implemented (as

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

customary in object-oriented languages) by means of method calls. Basically, an external heuristic implementation must be compliant with a specific interface. The methods of the interface correspond to the events abstracted in Table 1. Whenever a specific point of the computation is reached the corresponding event is triggered, i.e., a method of the heuristic is called. Some of the methods of the interface are allowed to return values that are interpreted by WASP as commands. For instance, whenever the next choice must be provided from the external heuristic (i.e., the event *onChoiceRequired* is triggered) a method is called on the heuristic that computes and returns the next branching literal.

Our implementation supports heuristic strategies implemented in (i) PERL and PYTHON for obtaining fast prototypes and (ii) C++ in case better performances are needed. Note that C++ implementations must be integrated in the WASP binary at compile time, whereas PERL and PYTHON can be specified by means of text files given as parameters for WASP, thus scripting-based heuristic does not require changes and recompilation of WASP.

The source code and the documentation are available on the branch *plugins* at `https://github.com/alviano/wasp`.

5 Experimental Analysis

In this section, we present an experiment assessing the performance of the described heuristics on the industrial instances of PUP and CCP provided by Siemens. All instances, encodings and binaries can be downloaded from `http://yarrick13.github.io/hwasp/`.

The evaluation was done on a system equipped with an i7-3030K CPU, 64GB RAM and Ubuntu 11.10. The solvers were run for at most 900 seconds on instances grounded by GRINGO (Gebser *et al*. 2011). The grounding time is not reported, as we focus on solver performance only. On average, the grounder required 1 and 5 seconds to ground PUP and CCP instances, respectively. We compared our approach with CLASP (Gebser *et al*. 2015a, v. 3.1.3), CLASPFOLIO (Hoos *et al*. 2014, v. 2.2.0) and ME-ASP (Maratea *et al*. 2014, v. 2015). The solvers were executed using the following configurations: (1) standard CLASP with 1 thread; (2) standard CLASP with 10 threads; (3) manually configured CLASP portfolio with 10 different configured threads, e.g. configurations like handy or crafty, as well as two random decision configurations; (4) CLASPFOLIO with 10 threads; and (5) ME-ASP with the 2015 ASP Competition configuration. For the sake of simplification, for each instance the best result obtained by one of the solvers mentioned above is given by the *Virtually Best Solver* (VBS).

The PUP instances are categorized in *double*, *double-variant*, *triple* and *grid*. The *grid* instances comprise parts of real-world railway systems, whereas all others are synthetic. Nevertheless, the instances of the first three types represent interesting topologies occurring frequently in the practice of Siemens (Aschinger *et al*. 2011a). We used two different encodings ENC1 (Aschinger *et al*. 2011a) (Sect. 3.1, p. 6) and ENC2 (ASP Competition 2015). The first one is natural and can be written by a trained student or even automatically generated from UML diagrams by tools like OOASP (Falkner *et al*. 2015). ENC2 is a complex encoding comprising symmetry breaking and ordering rules resulting in a significant performance improvement on some PUP instances. Some deterministic choices made by these heuristics interfere with our approach. Thus, we removed rule 9 (line 14) from ENC2 only in executions of WASP variants.

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

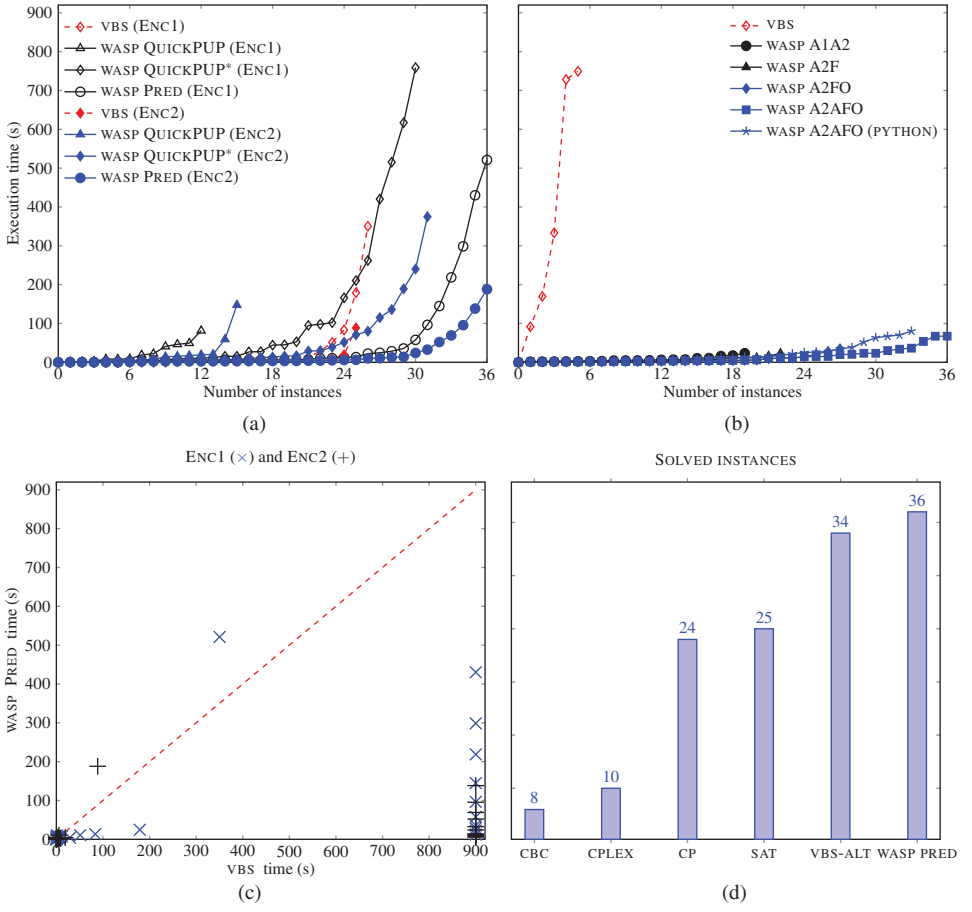


Fig. 3. Experimental analysis for PUP (a) and CCP (b) instances. Instance by instance comparison for PUP using ENC1 and ENC2 encodings (c). Solved instances by approaches beyond ASP on PUP instances (d).

The CCP instances are categorized in *easy*, *moderate* and *hard*. We focused on *hard* instances since these have not been solved in previous work. There are two types of hard instances: (i) a set of 20 real-world instances by (Gebser *et al*. 2015c) and (ii) 16 new instances with grid-like input graph structure (*grids*). The encoding used for CCP is the same of (Gebser *et al*. 2015c).

Partner Units Problem. The evaluation results concerning PUP are summarized by means of a cactus plot in Figure 3a. Recall that in a cactus plot a line is reported for each compared method, and there is a point (x, y) in one of such lines whenever the corresponding method finds a solution for the x^{th} fastest instance within y seconds.

VBS was able to solve 26 and 25 instances using ENC1 and ENC2, respectively. VBS failed for most of the *double* as well as for some *double-variant* and *triple* instances. Looking at Figure 3a, note that WASP with QUICKPUP solved only 12 and 15 instances using ENC1 and ENC2 respectively. The main reason for the discrepancy between our results and the ones obtained by (Teppan *et al*. 2012) is due to the inability of the underlying

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

ASP solver to generate new units on-the-fly. The number of available units is determined in the grounding step and cannot be changed later. The original QUICKPUP implementation, on the other hand, tries to quickly build a solution by creating and using new units whenever possible and reducing their number afterwards. This strategy appears to be successful for greedy algorithms, but works poorly as a heuristic for WASP. The remaining two heuristics, namely QUICKPUP* and PRED, are able to outperform VBS with respect to the number of solved units. Our best heuristic, PRED, solved *all* instances independently of the encoding. Furthermore, the solving time was reduced to at most 27 seconds using ENC2. An instance-wise comparison of VBS with our best solution (i.e., PRED) is reported in the scatter plot in Figure 3c, comparing results obtained with ENC1 (points of shape "×") and ENC2 (points of shape "+"). From the figure it is evident that WASP with PRED outperforms VBS in all instances but one, no matter the encoding used. In fact, the outlier instance (`2-triple-60.dl`) was solved in 188 and 88 seconds by CLASP with 10 threads using ENC1 and CLASPFOLIO with 10 threads using ENC2, whereas PRED needed 521 and 350 seconds, respectively.

We would like to emphasize the fact that PRED allowed WASP to find solutions for all instances with the "simple" encoding ENC1. This is an important feature of our approach. It indicates that a programmer can use a non-optimized ASP modeling of a problem, and (if needed) scale the performance of the solver by embedding a domain-heuristic defined by a procedure.

For the sake of completeness, we provide here a comparison of our best solution (PRED) with approaches alternative to ASP proposed in the literature. In particular, we consider the same setting of (Aschinger *et al*. 2011a), where encodings for propositional satisfiability (SAT), constraint programming (CP), and integer programming (CPLEX and CBC) were proposed. Results of the comparison are shown in Figure 3d, where the number of solved instances is reported for each approach. First of all, SAT and CP outperform the approach based on integer programming, solving 25 and 24 instances, respectively. The virtual best combination of all the approaches alternative to ASP (VBS-ALT) allows to solve all instances but two within ten minutes. This result is explained by the complementary behavior of SAT and CP. In fact, the former was effective on *grid* instances but did not perform well on *double*, and the other way round for CP.

Combined Configuration Problem. The evaluation results concerning CCP are summarized by means of a cactus plot in Figure 3b. Here, the reference system VBS solved 5 out of 36 instances. Instead, WASP solved 22, 27 and all 36 instances with an average solving time of 6, 7, and 14 seconds using A2F, A2FO and A2AFO, respectively. Moreover, since A2AFO alternates between two heuristics, we performed an additional evaluation. The goal was to test whether conflict learning is essential for finding a "correct" ordering as it was often the case for PUP heuristics. Therefore, we saved the last ordering created by A2AFO before a solution was found. This ordering was used as initial one for another run of the solver. In our experiments at least one more ordering, in addition to the saved one, was needed to find a solution. Thus, we conclude that the interplay of the heuristic with the learning feature of CDCL is essential in that case.

Scripting Interface Assessment. In Figure 3b we also report the results obtained by running a PYTHON implementation of A2AFO. It solves all instances but three within the timeout,

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

and is on average slower than the performance-oriented C++ implementation. The difference is due to several factors: (i) the overhead of communication, (ii) a slower PYTHON *interpreter* (than compiled code), and (iii) some differences in the implementation and tuning of system libraries Nonetheless, the PYTHON version behaves acceptably and far better than VBS. Moreover, the PYTHON version was *considerably faster to develop than the* C++ *one*. Indeed, according to the Function Point (FP) effort estimation metric (Albrecht 1979), which is based on Lines of Code (LOC), PYTHON and PERL require about 21 LOC/FP, whereas C++ requires about 53 LOC/FP. This is confirmed in our implementation where C++ required about twice as much LOC compared to PYTHON (1396 vs 631).

6 Related Work

Several ways of combining domain heuristics and ASP are proposed in the literature.

In (Balduccini 2011), a technique which allows learning of domain-specific heuristics in DPLL-based solvers is presented. The basic idea is to analyze off-line the behavior of the solver on representative instances from the domain to learn and use a heuristic in later runs. In our approach, the heuristics are provided by domain-experts, nonetheless our extended WASP could be considered as development platform for porting the ideas of (Balduccini 2011) in a CDCL solver.

A declarative approach to definition of domain specific heuristics in ASP is presented in (Gebser *et al*. 2013). The suggested HCLASP framework extends the GRINGO language with a special *_heuristic* predicate. Atoms over this predicate allow one to influence the choices made by the default heuristic of CLASP. In fact, a user can provide initial weights, importance factors, decision levels and sign selection for atoms involved in non-deterministic decisions. HCLASP supports definition of dynamic heuristics by considering only those atoms over *_heuristic* predicate that are true in the current interpretation. In opposite to static heuristics, where heuristic decisions are encoded as facts, dynamic ones comprise normal rules with a *_heuristic* atom in the head. As argued in (Gebser *et al*. 2015c), the grounding of programs comprising definitions of dynamic heuristics can be expensive as it impacts on grounding speed and size. The reason is that a grounder needs to output a rule for every possible heuristic decision. Therefore, the authors avoid application of dynamic heuristics and use only a static one providing initial weights for a set of atoms. Roughly, the idea of (Gebser *et al*. 2015c) is to compute a partial solution for an instance using a greedy algorithm. The obtained solution is then transformed to a set of heuristic atoms used to initialize the counters of the default heuristics. Thus, CLASP tries the partial solution first. A similar behavior could be obtained in our implementation by properly defining the functions $\mathscr{I}$, $\mathscr{F}$, $\mathscr{S}$ of the command #FALLBACK$(0, \mathscr{I}, \mathscr{F}, \mathscr{S})$ (see Table 1).

The main drawback of this approach is that the applied heuristic is static and cannot be updated during the computation. On the contrary, our variants of the same heuristics (e.g., compare A1A2 with A2AFO in Section 3.2) have been designed to be dynamic, i.e. they are queried multiple times during the execution of the algorithm so to consider different partial solutions to start with. The source code of (Gebser *et al*. 2015c) is unavailable, thus a direct comparison with their implementation via re-evaluation was not possible. Nonetheless, since their empirical evaluation was done on a similar hardware and on a subset of the instances we considered, we report that in (Gebser *et al*. 2015c) the implementation was

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

able to solve 6 out of the 20 *real-world instances* (see Section 5) whereas WASP with A2AFO solved all of them.

Additionally, we also evaluated the performance of HCLASP on dynamic heuristics. Therefore, we implemented an ASP program for the best-fit heuristic (Garey and Johnson 1979) of the bin packing problem used in A2. Evaluation on the CCP instances presented in Section 5 showed impracticality of this approach and confirmed similar observation of (Gebser *et al*. 2015c). For the sake of completeness, we report that the grounding step alone requires up to 4 minutes (1 minute on average), but resulted in ground programs up to 2 Gb (300 Mb on average). The vast majority of ground rules in these programs represent all possible heuristic decisions for any partial interpretation. Since such tabular representation of the best-fit part of the A2 heuristic alone is already very large, we discontinued this experiment. Next, we researched on possible improvements due to CLASP `--dom-mod` option, which also introduces domain modifications to the default heuristic. The best result in our test – solving one instance – was obtained using the value `(3,16)`, which forces the internal CLASP heuristic to assign *true* to all atoms specified with the `#show` directive of the GRINGO language.

PUP was introduced in (Falkner *et al*. 2011) where a number of different approaches were considered: Alloy, ASP, CSP, Constraint Handling Rules as well as problem-specific algorithm based on graph partitioning. (Falkner *et al*. 2011) reports that all these approaches are not effective on large industrial instances. A thorough modeling and evaluation of PUP was continued in (Aschinger *et al*. 2011a). The results reported in Section 5 confirm the findings of (Aschinger *et al*. 2011a): None of the applied general methods are able to solve all the instances. This result pushed the research towards investigation of domain-specific heuristics. These efforts resulted in development of the QUICKPUP heuristic (Teppan *et al*. 2012). The suggested greedy algorithm is based on backtrack search combined with QUICKPUP for variable orderings and time-based restarts. Later, results of (Drescher 2012) witnessed that the performance of declarative approaches can be also improved using similar heuristics. The author suggested PRED heuristic and implemented it using both ECL^{i}PS^e Prolog and the CSP solver MINION. The first approach included a custom propagator for the partner unit constraint and a Prolog definition of the heuristic. The second one utilized a very efficient implementation of a *global cardinality constraint* as it was unfeasible to implement PRED in MINION. Therefore, the solvers were extended with specific implementations of procedural propagation or decision-making techniques. The source code of (Drescher 2012) is not available, thus also in this case a re-evaluation was not possible. However, (Drescher 2012) report that some of the triple and grid with IUCAP =4 were not solved, whereas our approach solved all of them.

7 Conclusion

In this paper the combination of ASP and domain-specific heuristics is studied for solving two challenging industrial applications of ASP: the Partner Units and the Combined Configuration problems. Several known heuristic criteria and also novel ones are considered and implemented on the top of the ASP solver WASP. As a by-product of this work, WASP was extended with an interface that eases the development of new external heuristics. The

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

interface was effective and simplified the development of the heuristics presented in this work. The benefits in terms of development time (w.r.t. the first ad-hoc modifications) were sensible; for instance the PYTHON version of heuristic A2AFO has been implemented in only one working day. Moreover, the performances of our implementations have been evaluated on all the real-world instances of PUP and CCP ever provided by Siemens. Results are positive: WASP solves all the considered instances within few minutes outperforming all known alternatives. As future and ongoing work we are investigating the application of ASP and domain-heuristics for solving a hard resource assignment problem modeling a requirement of a Siemens project, for solving the allotment problem (Dodaro *et al*. 2015), and for performing complex reasoning on combinatorial auctions (Fionda and Greco 2013).

8 Acknowledgement

This work was partially supported by the Austrian Science Fund (FWF) contract number I 2144 N-15, the Carinthian Science Fund (KWF) contract KWF-3520/26767/38701, the Italian Ministry of University, Research under PON project "Ba2Know (Business Analytics to Know) Service Innovation – LAB", No. PON03PE_00001_1, and by the Italian Ministry of Economic Development under project "PIUCultura (Paradigmi Innovativi per l'Utilizzo della Cultura)" n. F/020016/01–02/X27.

Supplementary materials

For supplementary material for this article, please visit http://dx.doi.org/10.1017/S1471068416000284

References

ALBRECHT, A. 1979. Measuring Application Development Productivity. In IBM Application Development Symp., I. B. M. Press, Ed. *Proc. of IBM Application Development Symp.*, 83–92.

ALVIANO, M., DODARO, C., LEONE, N. AND RICCA, F. 2015. Advances in WASP. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 40–54.

ASCHINGER, M., DRESCHER, C., FRIEDRICH, G., GOTTLOB, G., JEAVONS, P., RYABOKON, A. AND THORSTENSEN, E. 2011a. Optimization methods for the partner units problem. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings*, T. Achterberg and J. C. Beck, Eds. Lecture Notes in Computer Science, vol. 6697. Springer, 4–19.

ASCHINGER, M., DRESCHER, C., GOTTLOB, G., JEAVONS, P. AND THORSTENSEN, E. 2011b. Tackling the partner units configuration problem. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, T. Walsh, Ed. IJCAI/AAAI, 497–503.

BALDUCCINI, M. 2011. Learning and using domain-specific heuristics in ASP solvers. *AI Commun. 24*, 2, 147–164.

BALDUCCINI, M., GELFOND, M., WATSON, R. AND NOGUEIRA, M. 2001. The usa-advisor: A case study in answer set planning. In *Logic Programming and Nonmonotonic Reasoning, 6th*

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings, T. Eiter, W. Faber, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 439–442.

BARAL, C. AND UYAN, C. 2001. Declarative specification and solution of combinatorial auctions using logic programming. In *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, T. Eiter, W. Faber, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer, 186–199.

BIERE, A., HEULE, M., VAN MAAREN, H. AND WALSH, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

BREWKA, G., EITER, T. AND TRUSZCZYNSKI, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

CALIMERI, F., GEBSER, M., MARATEA, M. AND RICCA, F. 2016. Design and results of the fifth answer set programming competition. *Artif. Intell. 231*, 151–181.

CAMPEOTTO, F., DOVIER, A. AND PONTELLI, E. 2015. A declarative concurrent system for protein structure prediction on GPU. *J. Exp. Theor. Artif. Intell. 27,* 5, 503–541.

DANTSIN, E., EITER, T., GOTTLOB, G. AND VORONKOV, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33,* 3, 374–425.

DODARO, C., LEONE, N., NARDI, B. AND RICCA, F. 2015. Allotment problem in travel industry: A solution based on ASP. In *Web Reasoning and Rule Systems - 9th International Conference, RR 2015, Berlin, Germany, August 4-5, 2015, Proceedings*, B. ten Cate and A. Mileo, Eds. Lecture Notes in Computer Science, vol. 9209. Springer, 77–92.

DRESCHER, C. 2012. The partner units problem a constraint programming case study. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*. IEEE Computer Society, 170–177.

EÉN, N. AND SÖRENSSON, N. 2003. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, E. Giunchiglia and A. Tacchella, Eds. Lecture Notes in Computer Science, vol. 2919. Springer, 502–518.

ERDEM, E. AND ÖZTOK, U. 2015. Generating explanations for biomedical queries. *Theory and Practice of Logic Programming 15,* 1, 35–78.

ERDEM, E., PATOGLU, V., SARIBATUR, Z. G., SCHÜLLER, P. AND URAS, T. 2013. Finding optimal plans for multiple teams of robots through a mediator: A logic-based approach. *Theory and Practice of Logic Programming 13,* 4-5, 831–846.

FALKNER, A. A., HASELBÖCK, A., SCHENNER, G. AND SCHREINER, H. 2011. Modeling and solving technical product configuration problems. *AI EDAM 25,* 2, 115–129.

FALKNER, A. A., RYABOKON, A., SCHENNER, G. AND SHCHEKOTYKHIN, K. M. 2015. OOASP: connecting object-oriented and logic programming. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 332–345.

FIONDA, V. AND GRECO, G. 2013. The complexity of mixed multi-unit combinatorial auctions: Tractability under structural and qualitative restrictions. *Artif. Intell. 196*, 1–25.

FRIEDRICH, G. 2015. Industrial success stories of ASP and CP: What's still open? Joint invited talk at ICLP and CP 2015 - `http://booleconferences.ucc.ie/iclp2015speakers`.

GAGGL, S. A., MANTHEY, N., RONCA, A., WALLNER, J. P. AND WOLTRAN, S. 2015. Improved answer-set programming encodings for abstract argumentation. *Theory and Practice of Logic Programming 15,* 4-5, 434–448.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press

GEBSER, M., KAMINSKI, R., KAUFMANN, B., ROMERO, J. AND SCHAUB, T. 2015a. Progress in clasp series 3. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 368–383.

GEBSER, M., KAMINSKI, R., KÖNIG, A. AND SCHAUB, T. 2011. Advances in *gringo* series 3. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 345–351.

GEBSER, M., KAUFMANN, B., ROMERO, J., OTERO, R., SCHAUB, T. AND WANKO, P. 2013. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA.*, M. desJardins and M. L. Littman, Eds. AAAI Press.

GEBSER, M., MARATEA, M. AND RICCA, F. 2015b. The sixth answer set programming competition web site. `http://aspcomp2015.dibris.unige.it/`.

GEBSER, M., RYABOKON, A. AND SCHENNER, G. 2015c. Combining heuristics for configuration problems using answer set programming. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 384–397.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9,* 3/4, 365–386.

GRASSO, G., LEONE, N., MANNA, M. AND RICCA, F. 2011. ASP at work: Spin-off and applications of the DLV system. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, M. Balduccini and T. C. Son, Eds. Lecture Notes in Computer Science, vol. 6565. Springer, 432–451.

HOOS, H., LINDAUER, M. T. AND SCHAUB, T. 2014. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming 14,* 4-5, 569–585.

KOPONEN, L., OIKARINEN, E., JANHUNEN, T. AND SÄILÄ, L. 2015. Optimizing phylogenetic supertrees using answer set programming. *Theory and Practice of Logic Programming 15,* 4-5, 604–619.

MANNA, M., RICCA, F. AND TERRACINA, G. 2015. Taming primary key violations to query large inconsistent data via ASP. *Theory and Practice of Logic Programming 15,* 4-5, 696–710.

MARATEA, M., PULINA, L. AND RICCA, F. 2014. A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming 14,* 6, 841–868.

MARILEO, M. C. AND BERTOSSI, L. E. 2010. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng. 69,* 6, 545–572.

RYABOKON, A. 2015. Knowledge-based (Re)configuration of Complex Products and Services. Ph.D. thesis, Alpen-Adria-University Klagenfurt.

TEPPAN, E. C., FRIEDRICH, G. AND FALKNER, A. A. 2012. Quickpup: A heuristic backtracking algorithm for the partner units configuration problem. In *Proceedings of the Twenty-Fourth Conference on Innovative Applications of Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, M. P. J. Fromherz and H. Muñoz-Avila, Eds. AAAI.

https://doi.org/10.1017/S1471068416000284 Published online by Cambridge University Press