Multi-Criteria Optimization in ASP and its Application to Linux Package Configuration∗

Martin Gebser and Roland Kaminski and Benjamin Kaufmann and Torsten Schaub†
Institut für Informatik, Universität Potsdam

submitted [n/a]; revised [n/a]; accepted [n/a]

Abstract

We elaborate upon new strategies and heuristics for solving multi-criteria optimization problems via Answer Set Programming (ASP). In particular, we conceive a new solving algorithm, based on conflict-driven learning, allowing for non-uniform descents during optimization. We apply these techniques to solve realistic Linux package configuration problems, thereby showing how transparently such problems can be modeled in ASP. Finally, we describe the Linux package configuration tool *aspcud* and compare its performance with systems pursuing alternative approaches.

1 Introduction

Upgrading and maintaining complex software systems constitutes a major challenge in modern software architectures. This problem is addressed in the mancoosi project (mancoosi), having a particular focus on GNU/Linux distributions. To this end, the consortium organizes an international competition of solvers for package installation and upgrade problems. In formal terms, this amounts to solving multi-criteria optimization problems. Such problems are of great interest in various application domains because they allow for identifying the best solutions among all feasible ones. The quality of a solution is often associated with costs or rewards subject to minimization and/or maximization, respectively.

In what follows, we are interested in solving Linux package configuration problems by appeal to the multi-criteria optimization capacities of Answer Set Programming (ASP; (Baral 2003)). To this end, we develop novel general-purpose strategies and heuristics in the context of modern (conflict-driven learning) ASP solving (Gebser et al. 2007). In particular, we conceive a new optimization algorithm allowing for non-uniform descents during optimization. In multi-criteria optimization, this enables us to optimize criteria in the order of significance, rather than pursuing a rigid lexicographical descent. We illustrate the impact of our contributions by appeal to the Linux package configuration tool *aspcud* and its performance in comparison with alternative approaches.

Pioneering work in this area was done by Tommi Syrjänen in (1999; 2000), using ASP for representing and solving configuration problems for the Debian GNU/Linux system.

∗ This draft extends a Pragmatics of SAT 2011 (PoS'11) contribution of the same title, which in turn extends the short paper "Multi-Criteria Optimization in Answer Set Programming", presented at the International Conference on Logic Programming 2011 (ICLP'11), by detailing ASP-based Linux package configuration.

† Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

In fact, ASP allows for defining such problems through sequences of cost functions represented by (multi)sets of literals with associated weights. For instance, in the approach taken by *smodels* (Simons et al. 2002), cost functions are expressed through a sequence of #minimize (and #maximize) statements. Optimal models are then computed via a branch-and-bound extension to *smodels*' enumeration algorithm. Similarly, *dlv* (Leone et al. 2006) offers so-called weak constraints, serving the same purpose.

2 Background

We only briefly introduce the syntax and semantics of ground (extended) logic programs. For further details, we refer the reader to (Simons et al. 2002). Likewise, first-order representations, commonly used to encode problems in ASP, are informally introduced by need in the remainder of this paper. See, e.g., (Syrjänen) and (Gebser et al.) for detailed descriptions of the input languages of the grounders *lparse* and *gringo*, respectively.

A ground (extended) rule r is an expression of the form

$$h \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n.$$

The head h of r is either an atom a, a choice $\{a\}$, or the special symbol $\bot$. If h is $\{a\}$, we call r a choice rule, and an integrity constraint if h is $\bot$; we skip $\bot$ when writing integrity constraints below. For $1 \leq i \leq n$, each b_i in the body of r is an atom a or a #sum constraint of the form $L\ \#\mathtt{sum}[\ell_1 = w_1, \ldots, \ell_k = w_k]U$. In the latter, $\ell_j = a$ or $\ell_j = not\ a$ is a literal and w_j an integer for $1 \leq j \leq k$; L and U are integers providing a lower and an upper bound. A #sum constraint holds wrt a set X of atoms if $L \leq \sum_{1 \leq j \leq k, \ell_j = a, a \in X} w_j + \sum_{1 \leq j \leq k, \ell_j = not\ a, a \notin X} w_j \leq U$. Either or both of L and U can be omitted, in which case they are identified with the (trivial) bounds $-\infty$ and $+\infty$, respectively. A body literal b_i (or $not\ b_i$) holds wrt X if b_i holds (or does not hold) wrt X, where an atom a holds if $a \in X$. If $n = 0$, i.e., r has an empty body, we call r a fact and simply write $h.$ in the sequel. A rule r is satisfied wrt X if some body literal of r does not hold wrt X, h is a choice, or $h \in X$. Note that an integrity constraint is unsatisfied if all literals in its body hold wrt X.

A ground logic program Π is a set of ground rules. A set X of atoms is a model of Π if each $r \in \Pi$ is satisfied wrt X. An answer set of Π is a model X of Π such that every atom in X is derivable from Π. Roughly speaking, the latter means that, for each $a \in X$, Π contains a rule r with head $h = a$ or $h = \{a\}$ such that all body literals of r hold wrt X; see (Simons et al. 2002) for further details.

In addition to rules, a logic program can contain #minimize statements of the form

$$\#\mathtt{minimize}[\ell_1 = w_1@L_1, \ldots, \ell_n = w_n@L_n].$$

Besides literals ℓ_i and integer weights w_i for $1 \leq i \leq n$, a #minimize statement includes integers L_i providing priority levels (Gebser et al. 2011). The #minimize statements in Π distinguish optimal answer sets of Π in the following way. For any set X of atoms and integer L, let Σ_L^X denote the sum of weights w_i such that $\ell_i = w_i@L$ occurs in some #minimize statement in Π and ℓ_i holds wrt X. We also call Σ_L^X the utility of X at priority level L. An answer set X of Π is dominated if there is an answer set Y of Π such that $\Sigma_L^Y < \Sigma_L^X$ and $\Sigma_{L'}^Y = \Sigma_{L'}^X$ for all $L' > L$, and optimal otherwise. Note that greater priority levels are more significant than smaller ones, which allows for representing sequences of several

optimization criteria. Finally, letting $\overline{\ell_i}$ denote the complement of a literal ℓ_i, the following can be used as a synonym for a `#minimize` statement: `#maximize`$[\overline{\ell_1} = w_1@L_1, \ldots, \overline{\ell_n} = w_n@L_n]$.

3 Approach

We start by describing the package configuration problem along with our ASP encoding of it. The formal problem specification is oriented at (Argelich et al. 2010), yet extended here to reflect the most recent optimization criteria assessed in the mancoosi project (mancoosi).

To begin with, we define the constituents of a package description.

Definition 1 (*Package Description*)
A package description (p_n^v, D, E, R) consists of

- a package identifier p_n^v associated with a package of name n in version v,
- a set D of dependency clauses,
- a set E of exclusion clauses, and
- a set R of recommendation clauses,

where clauses of each type are sets of package identifiers.

Note that each package is identified via a pair consisting of a name and a version. Furthermore, there are three kinds of package relations: dependencies, exclusions, and recommendations. An exclusion clause means that a package must not be installed jointly with any of the excluded packages. A dependency clause expresses the requirement that some package in it needs to be installed along with the dependent package. Finally, the intention of recommendations is similar to dependencies, but unlike the latter a recommendation clause specifies the (soft) desire that some of its packages should be installed.

Given a set of package descriptions, called universe, along with install goals, the installability problem is about identifying installations such that all goals are fulfilled and the hard requirements (dependencies and exclusions) are satisfied for all packages.

Definition 2 (*Valid Installation Profile*)
Given a universe U of package descriptions and a set I of install clauses, where each install clause is a set of package identifiers, some $P \subseteq \{p_n^v \mid (p_n^v, D, E, R) \in U\}$ is a valid installation profile if

- $P \cap i \neq \emptyset$ for each $i \in I$,
- $P \cap d \neq \emptyset$ for each $d \in (\bigcup_{(p_n^v, D, E, R) \in U, p_n^v \in P} D)$, and
- $P \cap e = \emptyset$ for each $e \in (\bigcup_{(p_n^v, D, E, R) \in U, p_n^v \in P} E)$.

Note that install clauses and dependencies act positively by requiring some contained package to be installed, while exclusions prohibit the installation of any of their packages. Rather unsurprisingly, deciding whether there is a valid installation profile is NP-complete (Di Cosmo et al. 2006), already if there is a single install clause of one element.

In the life cycle of a software system, an existing installation is frequently extended by installing new components or updating existing ones. In such a setting, it is desirable that a follow-up installation is "as close as possible" to its predecessor. To address this, a number of optimization criteria have been proposed in the mancoosi project. They rely on counting the elements of the sets defined next.

$unit(n_1, 1).$	$unit(n_1, 2).$	$unit(n_1, 3).$ $latest(n_1, 3).$
$satisfies(n_1, 1, c_5).$	$satisfies(n_1, 2, c_5).$	$satisfies(n_1, 3, c_5).$
$depends(n_1, 1, c_1).$	$depends(n_1, 2, c_2).$	$conflicts(n_1, 3, c_3).$
$unit(n_2, 1).$	$unit(n_2, 2).$ $latest(n_2, 2).$	$unit(n_3, 1).$ $latest(n_3, 1).$
$satisfies(n_2, 1, c_1).$	$satisfies(n_2, 2, c_1).$	$satisfies(n_3, 1, c_3).$
$satisfies(n_2, 1, c_2).$	$conflicts(n_2, 2, c_2).$	$unit(n_4, 1).$ $latest(n_4, 1).$
	$recommends(n_2, 2, c_4).$	$satisfies(n_4, 1, c_4).$
$installed(n_2, 1).$	$installed(n_3, 1).$	$requested(c_5).$

Fig. 1: Facts describing a package configuration instance.

Definition 3 (*Utilities*)
Given a universe U of package descriptions and two sets O, P of package identifiers, define

$$\begin{aligned}
\mathcal{N}_O^P &= \{n \mid p_n^v \in P, \nexists w : p_n^w \in O\},\\
\mathcal{D}_O^P &= \{n \mid p_n^v \in O, \nexists w : p_n^w \in P\},\\
\mathcal{C}_O^P &= \{n \mid p_n^v \in (P \setminus O) \cup (O \setminus P)\},\\
\mathcal{U}_U^P &= \{n \mid p_n^v \in P, p_n^{\max\{w \mid (p_n^w, D, E, R) \in U\}} \notin P\}, \text{ and}\\
\mathcal{R}_U^P &= \{(p_n^v, r) \mid (p_n^v, D, E, R) \in U, p_n^v \in P, r \in R, P \cap r = \emptyset\}.
\end{aligned}$$

Viewing O as the existing installation and P as its follow-up, $\mathcal{N}_O^P$ is the collection of package names n such that some version v of n belongs to P, while O contains no version of n; that is, a package of name n is new in P. Similarly, $\mathcal{D}_O^P$ and $\mathcal{C}_O^P$ collect the names of packages that are deleted or changed, respectively, where change means that some version is new or deleted. The sets $\mathcal{U}_U^P$ and $\mathcal{R}_U^P$ investigate P relative to the universe U. A package name n belongs to $\mathcal{U}_U^P$ if some version v of n is in P, but not the latest version w of n; that is, packages of name n are not up-to-date. Finally, a pair (p_n^v, r) in $\mathcal{R}_U^P$ points to an unsatisfied recommendation clause of a package of name n in version v that is contained in P. In fact, recommendations impose soft constraints that have not been considered in the description of valid installation profiles in Definition 2.

Lexicographically ordered combinations of utilities give rise to multi-criteria optimization problems. In a recent trial-run[1] of the competition organized by mancoosi, the following criteria have been applied:

paranoid: Minimize first $|\mathcal{D}_O^P|$ and second $|\mathcal{C}_O^P|$.
trendy: Minimize first $|\mathcal{D}_O^P|$, second $|\mathcal{U}_U^P|$, third $|\mathcal{R}_U^P|$, and fourth $|\mathcal{N}_O^P|$.
user: Arbitrary sequence of objectives (minimization or maximization) over utilities.

The *paranoid* criteria aim at avoiding, first, package deletions and, second, changes in general. The *trendy* criteria also penalize package deletions in the first place, but then aim at an up-to-date installation, satisfaction of recommendations, and few new packages. Note that the *paranoid* and *trendy* criteria have particular practical counterparts, viz., a server system or a desktop environment, respectively.

After specifying package configuration problems, we now describe their representation in ASP. A particular package configuration instance is provided in terms of facts,

[1] `http://www.mancoosi.org/misc-live/20101126/`

paranoid: $\mathit{utility}(\mathit{delete}, -2).\ \mathit{utility}(\mathit{change}, -1).$
trendy: $\mathit{utility}(\mathit{delete}, -4).\ \mathit{utility}(\mathit{update}, -3).\ \mathit{utility}(\mathit{recomm}, -2).\ \mathit{utility}(\mathit{newpkg}, -1).$

Fig. 2: Facts describing the *paranoid* and *trendy* optimization criteria.

like the ones shown in Figure 1. Pairs of a package name and version are given via instances of the predicate $\mathit{unit}/2$; the respective facts represent the package identifiers $p^1_{n_1}$, $p^2_{n_1}$, $p^3_{n_1}$, $p^1_{n_2}$, $p^2_{n_2}$, $p^1_{n_3}$, and $p^1_{n_4}$. The identifiers of latest versions, $p^3_{n_1}$, $p^2_{n_2}$, $p^1_{n_3}$, and $p^1_{n_4}$, are provided by means of the predicate $\mathit{latest}/2$. Facts of the form $\mathit{depends}(n, v, c)$, $\mathit{conflicts}(n, v, c)$, and $\mathit{recommends}(n, v, c)$ express dependencies, exclusions, and recommendations, respectively, where a package identified via p^v_n is related to a clause c. The members of such a clause are specified in terms of the predicate $\mathit{satisfies}/3$; e.g., $\mathit{satisfies}(n_1, 1, c_5)$, $\mathit{satisfies}(n_1, 2, c_5)$, and $\mathit{satisfies}(n_1, 3, c_5)$ represent that $p^1_{n_1}$, $p^2_{n_1}$, and $p^3_{n_1}$ belong to the clause labeled c_5. The aforementioned predicates allow for providing the package descriptions (cf. Definition 1) of a universe (U in Definition 2). Finally, an existing installation (O in Definition 3) and install clauses (I in Definition 2) are specified via facts over $\mathit{installed}/2$ and $\mathit{requested}/1$, respectively.

Optimization criteria are declared via facts of the form $\mathit{utility}(u, l)$, where $u \in \{\mathit{newpkg}, \mathit{delete}, \mathit{change}, \mathit{update}, \mathit{recomm}\}$ refers to $\mathcal{N}^P_O$, $\mathcal{D}^P_O$, $\mathcal{C}^P_O$, $\mathcal{U}^P_U$, or $\mathcal{R}^P_U$ in Definition 3, respectively, and l is an integer representing a priority level as well as an objective (minimization or maximization). In fact, if l is negative, it means that the cardinality of the set associated with u is subject to minimization, while a positive l can be used to express maximization. The *paranoid* and *trendy* criteria, described by the facts in Figure 2, rely solely on minimization. Hence, all priority levels are negative, and a smaller priority level takes precedence over greater levels (with smaller absolute values).

The described facts represent a package configuration problem along with objectives. Corresponding (optimal) follow-up installations can be encoded in ASP as shown in Figure 3. Here, the first block of rules abstracts relations from particular package versions to their common name, provided that all respective versions share a relation.[2] For instance, these rules allow us to derive $\mathit{satisfies}(n_1, c_5)$ in view of $\mathit{satisfies}(n_1, 1, c_5)$, $\mathit{satisfies}(n_1, 2, c_5)$, and $\mathit{satisfies}(n_1, 3, c_5)$, given by the facts in Figure 1. The second block of rules starts with a choice rule saying that any instance of $\mathit{unit}/2$ may be installed in a follow-up installation, in which case $\mathit{install}/2$ holds. A similar abstraction as before is applied by deriving $\mathit{install}/1$, omitting the version of a package to be installed, from $\mathit{install}/2$. The following rules propagate relations of packages to be installed to clauses, identifying the ones that need to be satisfied ($\mathit{include}/1$), must not be satisfied ($\mathit{exclude}/1$), and those that are satisfied ($\mathit{satisfy}/1$). Based on this, the three integrity constraint at the end of the second block stipulate all exclusion clauses to be unsatisfied and all dependency as well as install clauses to be satisfied. These checks ensure that a follow-up installation corresponds to a valid installation profile (cf. Definition 2). The rules of the third block identify members of $\mathcal{N}^P_O$, $\mathcal{D}^P_O$, $\mathcal{C}^P_O$, $\mathcal{U}^P_U$, or $\mathcal{R}^P_U$ (cf. Definition 3), respectively, provided that associated objectives are declared via facts over $\mathit{utility}/2$. Finally, the objectives rep-

[2] The "`:`" connective expands to the list of all instances of its left-hand side such that corresponding instances of literals on the right-hand side hold (Syrjänen; Gebser et al.).

$$
\begin{aligned}
&\% \text{ Lift interdependencies to names of packages}\\
&\mathit{installed}(N) \leftarrow \mathit{installed}(N,V).\\
&\mathit{aux\_depends}(N,D) \leftarrow \mathit{depends}(N,V,D).\\
&\mathit{depends}(N,D) \leftarrow \mathit{aux\_depends}(N,D), \mathit{depends}(N,V,D) : \mathit{unit}(N,V).\\
&\mathit{aux\_conflicts}(N,E) \leftarrow \mathit{conflicts}(N,V,E).\\
&\mathit{conflicts}(N,E) \leftarrow \mathit{aux\_conflicts}(N,E), \mathit{conflicts}(N,V,E) : \mathit{unit}(N,V).\\
&\mathit{aux\_satisfies}(N,S) \leftarrow \mathit{satisfies}(N,V,S).\\
&\mathit{satisfies}(N,S) \leftarrow \mathit{aux\_satisfies}(N,S), \mathit{satisfies}(N,V,S) : \mathit{unit}(N,V).\\
&\% \text{ Generate valid installation profile}\\
&\{\mathit{install}(N,V)\} \leftarrow \mathit{unit}(N,V).\\
&\mathit{install}(N) \leftarrow \mathit{install}(N,V).\\
&\mathit{include}(D) \leftarrow \mathit{install}(N), \mathit{depends}(N,D).\\
&\mathit{include}(D) \leftarrow \mathit{install}(N,V), \mathit{depends}(N,V,D).\\
&\mathit{exclude}(E) \leftarrow \mathit{install}(N), \mathit{conflicts}(N,E).\\
&\mathit{exclude}(E) \leftarrow \mathit{install}(N,V), \mathit{conflicts}(N,V,E).\\
&\mathit{satisfy}(S) \leftarrow \mathit{install}(N), \mathit{satisfies}(N,S).\\
&\mathit{satisfy}(S) \leftarrow \mathit{install}(N,V), \mathit{satisfies}(N,V,S).\\
&\leftarrow \mathit{exclude}(E), \mathit{satisfy}(E).\\
&\leftarrow \mathit{include}(D), \mathit{not\ satisfy}(D).\\
&\leftarrow \mathit{requested}(I), \mathit{not\ satisfy}(I).\\
&\% \text{ Optimize installation profile}\\
&\mathit{violate}(\mathit{newpkg},N) \leftarrow \mathit{utility}(\mathit{newpkg},L), \mathit{install}(N), \mathit{not\ installed}(N).\\
&\mathit{violate}(\mathit{delete},N) \leftarrow \mathit{utility}(\mathit{delete},L), \mathit{installed}(N), \mathit{not\ install}(N).\\
&\mathit{violate}(\mathit{change},N) \leftarrow \mathit{utility}(\mathit{change},L), \mathit{installed}(N,V), \mathit{not\ install}(N,V).\\
&\mathit{violate}(\mathit{change},N) \leftarrow \mathit{utility}(\mathit{change},L), \mathit{install}(N,V), \mathit{not\ installed}(N,V).\\
&\mathit{violate}(\mathit{update},N) \leftarrow \mathit{utility}(\mathit{update},L), \mathit{install}(N), \mathit{latest}(N,V), \mathit{not\ install}(N,V).\\
&\mathit{violate}(\mathit{recomm},r(N,V,R)) \leftarrow \mathit{utility}(\mathit{recomm},L), \mathit{recommends}(N,V,R),\\
&\qquad\qquad \mathit{install}(N,V), \mathit{not\ satisfy}(R).\\
&\#\texttt{minimize}[\mathit{violate}(U,T) = 1 \mathbin{@} -L : \mathit{utility}(U,L) : L < 0].\\
&\#\texttt{maximize}[\mathit{violate}(U,T) = 1 \mathbin{@} \quad L : \mathit{utility}(U,L) : L > 0].
\end{aligned}
$$

Fig. 3: ASP encoding of package configuration.

resented via negative priority levels are subject to a #minimize statement, and positive ones to a #maximize statement.

As packages of the names n_2 and n_3 are installed according to facts in Figure 1, the relevant parts of #minimize statements for criteria-specific priority levels L are:

$$
\begin{aligned}
&\#\texttt{minimize}[\mathit{violate}(\mathit{newpkg},n_1) = 1 \mathbin{@} L, \mathit{violate}(\mathit{newpkg},n_4) = 1 \mathbin{@} L]\\
&\#\texttt{minimize}[\mathit{violate}(\mathit{delete},n_2) = 1 \mathbin{@} L, \mathit{violate}(\mathit{delete},n_3) = 1 \mathbin{@} L]\\
&\#\texttt{minimize}[\mathit{violate}(\mathit{change},n_1) = 1 \mathbin{@} L, \mathit{violate}(\mathit{change},n_2) = 1 \mathbin{@} L,\\
&\qquad\qquad \mathit{violate}(\mathit{change},n_3) = 1 \mathbin{@} L, \mathit{violate}(\mathit{change},n_4) = 1 \mathbin{@} L]\\
&\#\texttt{minimize}[\mathit{violate}(\mathit{update},n_1) = 1 \mathbin{@} L, \mathit{violate}(\mathit{update},n_2) = 1 \mathbin{@} L,\\
&\qquad\qquad \mathit{violate}(\mathit{update},n_3) = 1 \mathbin{@} L, \mathit{violate}(\mathit{update},n_4) = 1 \mathbin{@} L]\\
&\#\texttt{minimize}[\mathit{violate}(\mathit{recomm},r(n_2,2,c_4)) = 1 \mathbin{@} L]
\end{aligned}
$$

Note that $recommends(n_2, 2, c_4)$ is the single recommendation in Figure 1. Hence, no further atoms regarding recommendations are subject to the last `#minimize` statement.

4 Algorithm

As detailed in (Simons et al. 2002), `#maximize` statements can be turned into `#minimize` statements, literals with negative weights be transformed such that weights become positive,[3] and multiple priority levels be collapsed into a single one by scaling the weights of literals, where all such transformations keep the optimal answer sets intact. However, while the elimination of `#maximize` statements and negative weights can be done locally, collapsing priority levels may lead to very large weights and also disguises an original multi-criteria optimization problem. Hence, we assume here that optimization criteria are represented in terms of a `#minimize` statement over literals associated with non-negative weights and, notably, priority levels; i.e., priorities are not eliminated. The restriction to non-negative weights has the advantages that the sum of weights is monotonically increasing the more literals are assigned to true and that 0 is a (trivial) lower bound of the optimum at each priority level.

As mentioned in the introduction, multi-criteria optimization can in principle be accomplished by extending a standard enumeration algorithm, like the one of *smodels* (Simons et al. 2002), in the following way: for every solution, memorize its vector of utilities, backtrack, and check (during propagation) that assignments generated in the sequel induce a lexicographically smaller vector of utilities (otherwise backtrack). This simple approach requires only the most recent utility vector to be stored, and optimality of the last solution is proven once the residual problem turns out to be unsatisfiable. But the simplicity comes along with the drawback that the number of intermediate solutions, encountered before an optimal one, is completely up to "luck" of the underlying enumeration algorithm. In fact, if no additional measures are taken, such multi-criteria optimization is logically identical to optimization of a single priority level along with scaled weights of literals.

The observation that plenty intermediate solutions improving only at low-priority utilities can gravely obstruct the convergence towards a global optimum gave the main impetus to our new approach to multi-criteria optimization in ASP. As noted in (Argelich et al. 2009) for Maximum Satisfiability (MaxSAT) and Pseudo-Boolean Optimization (PBO), a better idea is to optimize priority levels stepwise in the order of significance, rather than to optimize all priority levels at once. Thereby, we adhere to the strategy of successively improving upper bounds given by intermediate solutions. On the one hand, focusing on one priority level after the other settles the issue of intermediate solutions improving only at low-priority levels. On the other hand, it leads to the situation that, before optimization proceeds to the next priority level, optimality at the current level must be verified by proving unsatisfiability wrt an infeasible upper bound. Beyond the fact that accomplishing such unsatisfiability proofs can be a bottleneck (cf. (Argelich et al. 2010)), they imply that too strong bounds need to be taken back before optimization can proceed at the next level. In particular, with solvers like *clasp* (Gebser et al. 2007), exploiting conflict-driven learning,

[3] Since optimization statements merely serve the evaluation of answer sets, all preference-preserving transformations are admissible and cannot incur semantic problems (cf. (Ferraris 2005)).

also the learned constraints that rely on an infeasible upper bound must be retracted. To this end, we make use of assumptions assigned at a solver's root level (Eén and Sörensson 2003), i.e., unbacktrackable literals allowing for the selective (de)activation of constraints. In fact, a speculative upper bound is imposed via an assumption such that a corresponding constraint is not satisfied by making the assumption. If the upper bound turns out to be infeasible, the respective constraint and all learned information relying on it can then easily be discarded by irrevocably assigning the complement of the former assumption. Likewise, if the upper bound is feasible, the former assumption can be fixed, so that constraints involving it may be simplified and apply unconditionally in the sequel. In the following, we detail how dedicated multi-criteria optimization can be accomplished in modern (conflict-driven learning) Boolean constraint solvers, thereby exploiting assumptions to circumvent the need of a relaunch after an unsatisfiability proof.

Our algorithm augmenting conflict-driven learning (cf. (Darwiche and Pipatsrisawat 2009; Marques-Silva et al. 2009)) with multi-criteria optimization is shown in Algorithm 1. The sequence $\langle \mathbf{L}_1, \ldots, \mathbf{L}_{low} \rangle$ determined in the first line contains the priority levels of the input `#minimize` statement in decreasing order of significance. The counters $assm$, $prio$, and $step$, initialized to 1 in the second line, are used to generate new assumptions on demand, to identify the current priority level to be optimized, and to determine the amount by which the upper bound ought to be decreased when a solution is found. The latter is always 1, thus yielding a linear decrease, if the input $leap$ flag is $false$, while an exponential scheme (described below) is applied otherwise. Furthermore, the lower bound lb, set to 0 in the third line, stores the greatest value such that unsatisfiability has been proven for smaller bounds at the current priority level. In fact, the optimization of a priority level is finished once the utility of a solution matches the lower bound. In the loop in Line 4–45, the optimization-specific information, kept in counters and the lower bound, is used to guide conflict-driven search. As usual, the loop starts in Line 5 with a deterministic PROPAGATE step, assigning literals implied by the current assignment. Afterwards, one of the following is the case: a conflict (Line 6–23), a solution (Line 24–44), or a heuristic decision (Line 45). While the latter simply leads to reentering the loop, the first two cases deserve more attention. We describe next the reaction to a solution and then the one to a conflict.

Upon encountering a solution, we start by checking whether its objective value at the current priority level provides us with a new (non-speculative) upper bound. This is clearly the case if the current solution is the first one, as tested via $assm = 1$ in Line 25, and setting $recd$ to $true$ informs our algorithm that the upper bound needs to be recorded before proceeding to the next priority level. On the other hand, if a speculative upper bound $ub_{prio} - step$ has already been imposed, the current solution witnesses that this bound is feasible. Hence, a respective optimization constraint is made unconditional by fixing the former assumption $\overline{\alpha_{assm}}$ in Line 27. In view of this, adding another constraint before proceeding to the next priority level is required only if the current solution's objective value is smaller than $ub_{prio} - step$, as tested in Line 28. The sequence $\langle ub_1, \ldots, ub_{low} \rangle$ of upper bounds given by the current solution is memorized in Line 30 and printed along with an answer set of the input program Π in Line 31. Then, the loop in Line 33–37 proceeds to the next priority level to optimize, depending on whether the condition $ub_{prio} = lb$ holds in Line 33. If so, it means that the upper bound witnessed by the solution at hand matches the lower bound at a priority level, so that no further improvement is possible.

Algorithm 1: CDNL-OPT

Input: A logic program Π, a statement $\#\texttt{minimize}[\ell_1 = w_1@L_1, \ldots, \ell_n = w_n@L_n]$, and a flag $\mathit{leap} \in \{\mathit{true}, \mathit{false}\}$.

$\langle \mathbf{L}_1, \ldots, \mathbf{L}_{low} \rangle \leftarrow \langle \max(\{L_1, \ldots, L_n\} \setminus \{\mathbf{L}_1, \ldots, \mathbf{L}_{m-1}\}) \rangle_{1 \leq m \leq |\{L_1, \ldots, L_n\}|}$
$\mathit{assm} \leftarrow \mathit{prio} \leftarrow \mathit{step} \leftarrow 1$ *// assumption, priority, and step counter*
$\mathit{lb} \leftarrow 0$ *// lower bound*
loop
  PROPAGATE *// deterministically assign implied literals*
  if *conflict* **then**
    if at root level **then** *// unsatisfiability modulo optimization constraint*
      if $\mathit{assm} = 1$ **then exit**
      ASSIGN α_{assm} *// deactivate old optimization constraint*
      $\mathit{lb} \leftarrow (ub_{prio} - \mathit{step}) + 1$
      while $\mathit{prio} \leq \mathit{low}$ **and** $ub_{prio} = \mathit{lb}$ **do**
        if $\mathit{recd} = \mathit{true}$ **then** ADD $\#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]\mathit{lb}$
        $\mathit{lb} \leftarrow 0$
        $\mathit{recd} \leftarrow \mathit{true}$
        $\mathit{prio} \leftarrow \mathit{prio} + 1$
      if $\mathit{prio} > \mathit{low}$ **then exit**
      $\mathit{step} \leftarrow 1$
      $\mathit{assm} \leftarrow \mathit{assm} + 1$
      ADD $\big(\alpha_{assm} \vee \#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]ub_{prio} - \mathit{step}\big)$
      ASSUME $\overline{\alpha_{assm}}$ *// activate new optimization constraint*
    else
      ANALYZE *// analyze conflict and add (violated) conflict constraint*
      BACKJUMP *// unassign literals until conflict constraint is unviolated*
  else if *solution* **then**
    if $\mathit{assm} = 1$ **then** $\mathit{recd} \leftarrow \mathit{true}$ *// upper bound of witness yet unrecorded*
    else
      ASSIGN $\overline{\alpha_{assm}}$ *// fix old optimization constraint*
      if $(\Sigma_{1 \leq i \leq n, L_i = \mathbf{L}_{prio}, \ell_i \text{ assigned to true}} w_i) < ub_{prio} - \mathit{step}$ **then** $\mathit{recd} \leftarrow \mathit{true}$
      else $\mathit{recd} \leftarrow \mathit{false}$
    $\langle ub_1, \ldots, ub_{low} \rangle \leftarrow \langle \Sigma_{1 \leq i \leq n, L_i = \mathbf{L}_m, \ell_i \text{ assigned to true}} w_i \rangle_{1 \leq m \leq low}$
    print answer set along with $\langle ub_1, \ldots, ub_{low} \rangle$
    $\mathit{prio}' \leftarrow \mathit{prio}$
    while $\mathit{prio} \leq \mathit{low}$ **and** $ub_{prio} = \mathit{lb}$ **do**
      if $\mathit{recd} = \mathit{true}$ **then** ADD $\#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]\mathit{lb}$
      $\mathit{lb} \leftarrow 0$
      $\mathit{recd} \leftarrow \mathit{true}$
      $\mathit{prio} \leftarrow \mathit{prio} + 1$
    if $\mathit{prio} > \mathit{low}$ **then exit**
    if $\mathit{prio} = \mathit{prio}'$ **and** $\mathit{leap} = \mathit{true}$ **then** $\mathit{step} \leftarrow \min\{2 * \mathit{step}, \lceil (ub_{prio} - \mathit{lb})/2 \rceil\}$
    else $\mathit{step} \leftarrow 1$
    $\mathit{assm} \leftarrow \mathit{assm} + 1$
    ADD $\big(\alpha_{assm} \vee \#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{prio}]ub_{prio} - \mathit{step}\big)$
    ASSUME $\overline{\alpha_{assm}}$ *// activate new optimization constraint*
    BACKJUMP *// unassign literals until optimization constraint is unviolated*
  else DECIDE *// non-deterministically assign some literal*

Furthermore, if the current upper bound still needs to be recorded, a corresponding $\#\texttt{sum}$ constraint, as available in ASP input languages (Syrjänen; Gebser et al.), is added to the constraint database of the solver in Line 34; this makes sure that future solutions cannot exceed the lower bound lb at a forsaken priority level. Also note that lb is set to the minimum 0 in Line 35, so that proceeding by more than one priority level is possible only if some upper bound given by the solution at hand is trivially optimal. After finishing the loop in Line 33–37, multi-criteria optimization has been accomplished if the test $prio > low$ succeeds in Line 38, meaning that the utilities $\langle ub_1, \ldots, ub_{low} \rangle$ cannot be improved. Otherwise, an amount by which the current upper bound ought to be decreased is determined in Line 39–40. If the priority level has not been changed and the $leap$ flag is $true$, we take the minimum of the double former $step$ size and half of the gap between the lower and upper bound as the amount by which to decrease the upper bound. This exponential scheme aims at balancing two objectives: try to skip non-optimal intermediate solutions while decreasing the upper bound, but do not provoke many unnecessary (and potentially hard) proofs of unsatisfiability. Given the next $step$ size, an optimization constraint, being the disjunction of a fresh literal α_{assm} and a $\#\texttt{sum}$ constraint enforcing the new (speculative) upper bound, is added to the constraint database of the solver in Line 42, and $\overline{\alpha_{assm}}$ is assumed in Line 43, so that any further solution must fall below the speculative upper bound $ub_{prio} - step$. Finally, backjumping in Line 44 retracts literals (but not $\overline{\alpha_{assm}}$ assumed at the root level) in order to re-enable the search for solutions satisfying the new optimization constraint.

In case of a conflict, we distinguish whether it is encountered at the root level or beyond it. The latter means that the conflict is related to decisions made previously (in Line 45), so that regular conflict analysis and backjumping (cf. (Darwiche and Pipatsrisawat 2009; Marques-Silva et al. 2009)) can in Line 22–23 be applied to identify a reason in terms of a conflict constraint and to resume search at a point where the conflict constraint yields an implication. On the other hand, a conflict at the root level indicates unsatisfiability. Provided that $assm = 1$ does not hold in Line 8, i.e., if Π has some answer set, there is no solution meeting the upper bound $ub_{prio} - step$. This bound is imposed by the most recently added optimization constraint, which is in Line 9 retracted by assigning α_{assm}, thus withdrawing the former assumption and unconditionally satisfying the optimization constraint (as well as all conflict constraints relying on it). Furthermore, the unsatisfiability relative to the upper bound provides us with the lower bound $(ub_{prio} - step) + 1$, assigned to lb in Line 10. As in the case of a solution, the loop in Line 11–15 proceeds to the next priority level to optimize, where a gap between the lower and upper bound leaves room for improvements. If such a level $prio$ exists, i.e., $prio > low$ does not hold in Line 16, the $step$ size is reduced to 1 in Line 17, and the next optimization constraint along with a fresh assumption are put into effect in Line 18–20. By reducing the $step$ size to the smallest value that would still improve ub_{prio}, we reset the exponential scheme applied if the input $leap$ flag is $true$. This directs search to first check whether improvements are possible at all before reattempting to decrease the upper bound more aggressively.

For illustrating multi-criteria optimization via Algorithm 1, reconsider the package configuration problem represented by the facts in Figure 1 along with the encoding in Figure 3. A valid installation profile is given by atoms of the predicate $install/2$ that belong to an answer set, while instances of $violate/2$, indicating optimization criteria violations,

Witness	Util.	α	Optimization Constraint
$install(n_1, 3)$	$\langle 2, 3\rangle$		$\alpha_2 \vee$ $\#\mathtt{sum}[violate(delete, n_2), violate(delete, n_3)]1$
$install(n_1, 1), install(n_2, 2),$ $install(n_3, 1), install(n_4, 1)$	$\langle 0, 3\rangle$	$\overline{\alpha_2}$	$\#\mathtt{sum}[violate(delete, n_2), violate(delete, n_3)]0$
ditto	$\langle \mathbf{0}, 3\rangle$		$\alpha_3 \vee$ $\#\mathtt{sum}[violate(change, n_1), violate(change, n_2),$ $violate(change, n_3), violate(change, n_4)]2$
$install(n_1, 1), install(n_2, 1),$ $install(n_3, 1)$	$\langle \mathbf{0}, 1\rangle$	$\overline{\alpha_3}$	$\alpha_4 \vee$ $\#\mathtt{sum}[violate(change, n_1), violate(change, n_2),$ $violate(change, n_3), violate(change, n_4)]0$
—	$\langle \mathbf{0}, \mathbf{1}\rangle$	α_4	$\#\mathtt{sum}[violate(change, n_1), violate(change, n_2),$ $violate(change, n_3), violate(change, n_4)]1$

Table 1: *Run of* CDNL-OPT *on the instance in Fig. 1 using the paranoid criteria in Fig. 2.*

are counted in an associated utility vector. The criteria specified by the facts in Figure 2, *paranoid* and *trendy*, penalize package deletions and changes or, respectively, package deletions, missing latest versions, unsatisfied recommendations, and new packages (listed in the order of priority). Note that the value of the $leap$ flag is inconsequential for the example computations described next; the obtained utilities are not yet large enough for admitting $step$ sizes greater than 1 if $leap$ is $true$.

Table 1 provides a sequence of witnesses, as it can be generated upon optimizing the *paranoid* criteria. The first column shows the atoms representing valid installation profiles, the second column provides associated utility vectors, the third column displays literals corresponding to former assumptions as they become irrevocably assigned (not necessarily as assumed beforehand), and the fourth column gives optimization constraints added to impose upper bounds. For the first witness, according to which only the package of name n_1 in version 3 is to be installed, we have that packages of the names n_2 and n_3 are deleted, so that the first component of the utility vector is 2. For decreasing this upper bound, the optimization constraint containing α_2 is added, while $\overline{\alpha_2}$ is assumed. In fact, the utility vector of the second witness is $\langle 0, 3\rangle$, given that packages of the names n_1, n_2, and n_4 are changed. Since the upper bound imposed by the first optimization constraint has been shown to be feasible, the former assumption $\overline{\alpha_2}$ can now be fixed. In addition, the first priority level has been optimized in view of hitting the (trivial) lower bound 0; as it undercuts the imposed upper bound 1, we also add an (unretractable) optimization constraint to make sure that future solutions do not involve deleted packages. Then, the first priority level is completely processed, and the next optimization constraint, including α_3, tackles the improvement of the number of changed packages. The final witness, according to which packages of the names n_1, n_2, and n_3 in version 1 are to be installed, only changes n_1, which shows that the upper bound 1 is feasible at the second priority level. Hence, we fix the former assumption $\overline{\alpha_3}$ and add the optimization constraint containing α_4 in order to decrease the upper bound further to 0. In view of the request to install some package of the name n_1, while no such package belongs to the existing installation, 0 changes are infeasible. As a consequence, the last optimization constraint along with the assumption $\overline{\alpha_4}$ impose an unsatisfiable problem; since the complement α_4 is irrevocably assigned in reaction to the root-level conflict, the (too strong) optimization constraint is satisfied in the sequel and can thus be discarded. Furthermore, the conflict shows that the lower bound

Witness	Util.	α	Optimization Constraint
$install(n_1, 3)$	$\langle 2, 0, 0, 1\rangle$		$\alpha_2 \vee$ $\#\mathtt{sum}[violate(delete, n_2), violate(delete, n_3)]1$
$install(n_1, 1), install(n_2, 1),$ $install(n_3, 1), install(n_4, 1)$	$\langle 0, 2, 0, 2\rangle$	$\overline{\alpha_2}$	$\#\mathtt{sum}[violate(delete, n_2), violate(delete, n_3)]0$
ditto	$\langle \mathbf{0}, 2, 0, 2\rangle$		$\alpha_3 \vee$ $\#\mathtt{sum}[violate(update, n_1), violate(update, n_2),$ $violate(update, n_3), violate(update, n_4)]1$
$install(n_1, 1), install(n_2, 2),$ $install(n_3, 1)$	$\langle \mathbf{0}, 1, 1, 1\rangle$	$\overline{\alpha_3}$	$\alpha_4 \vee$ $\#\mathtt{sum}[violate(update, n_1), violate(update, n_2),$ $violate(update, n_3), violate(update, n_4)]0$
—	$\langle \mathbf{0}, \mathbf{1}, 1, 1\rangle$	α_4	$\alpha_5 \vee$ $\#\mathtt{sum}[violate(recomm, r(n_2, 2, c_4))]0$
$install(n_1, 1), install(n_2, 2),$ $install(n_3, 1), install(n_4, 1)$	$\langle \mathbf{0}, \mathbf{1}, \mathbf{0}, 2\rangle$	$\overline{\alpha_5}$	$\alpha_6 \vee$ $\#\mathtt{sum}[violate(newpkg, n_1), violate(newpkg, n_4)]1$
—	$\langle \mathbf{0}, \mathbf{1}, \mathbf{0}, \mathbf{2}\rangle$	α_6	$\#\mathtt{sum}[violate(newpkg, n_1), violate(newpkg, n_4)]2$

Table 2: *Run of* CDNL-OPT *on the instance in Fig. 1 using the trendy criteria in Fig. 2.*

is 1, thus hitting the upper bound given by the witness. That is, the second priority level has been optimized as well, and Algorithm 1 may add a final optimization constraint recording the lower bound at the second level before it terminates in view of having processed all levels. Note that the last witness comprises an optimal valid installation profile, consisting of packages of the names n_1, n_2, and n_3 in version 1.

For further illustration, Table 2 provides a sequence of witnesses obtainable upon optimizing the *trendy* criteria. Note that the utility vectors now consist of four components and that the optimization of the first component is accomplished analogously to the *paranoid* criteria considered above. However, the second *trendy* criterion, aiming at latest versions of packages to be installed, is different. For the second witness, according to which packages of the names n_1, n_2, n_3, and n_4 in version 1 are to be installed, the criterion is not met by n_1 and n_2, so that the given upper bound is 2. The optimization constraint containing α_3 along with the assumption $\overline{\alpha_3}$ lead us to the third witness, where only n_1 is not up-to-date. After fixing $\overline{\alpha_3}$, decreasing the upper bound further to 0 yields an unsatisfiable problem, given that the package of name n_3 in version 1 must not be deleted, which in turn implies that the latest version 3 of n_1 cannot be installed. Thus, the assumption $\overline{\alpha_4}$ is withdrawn and α_4 assigned before proceeding to the third priority level, addressing unsatisfied recommendations. Note that no new optimization constraint needs to added because the lower bound 1 at the second level has already been imposed via assigning $\overline{\alpha_3}$. The last witness avoids the previously unsatisfied recommendation by incorporating the package of name n_4 in version 1. This reduces the upper bound at the third priority level to 0, and assigning $\overline{\alpha_5}$ makes sure that it cannot be exceeded later on. Finally, the addition of the optimization constraint containing α_6 along with the assumption $\overline{\alpha_6}$ yield a root-level conflict, which shows that the witness at hand is optimal at the fourth priority level, dealing with new packages. Since the complement α_6 is assigned in reaction, an optimization constraint recording the lower bound 2 at the fourth level may be added before Algorithm 1 terminates. The last witness provides an optimal valid installation profile, consisting of packages of the names n_1, n_3, and n_4 in version 1 as well as the package of name n_2 in (the latest) version 2.

Multi-criteria optimization via Algorithm 1 is implemented in *clasp* from version 2.0.0

on. We do not detail the implementation here, but mention matters of interest. To begin with, note that *clasp* stores a statement $\#\texttt{minimize}[\ell_1 = w_1@L_1, \ldots, \ell_n = w_n@L_n]$ in a single optimization constraint, using as data-structure a two-dimensional array of size $|\{L_1, \ldots, L_n\}| * |\{\ell_1, \ldots, \ell_n\}|$ with $w_1, \ldots, w_n$ as its (non-zero) entries. Furthermore, the vector $\langle ub_m \rangle_{1 \leq m \leq |\{L_1,\ldots,L_n\}|}$ of upper bounds is initialized to $\langle \infty_m \rangle_{1 \leq m \leq |\{L_1,\ldots,L_n\}|}$ and then updated whenever a solution is found. For one, this permits to accomplish the simple approach to multi-criteria optimization, described at the beginning of this section, via lexicographic comparisons without scaling weights in view of priority levels. For another, dedicated multi-criteria optimization wrt a current priority level $prio$ merely requires to (temporarily) ignore upper bounds at less significant priority levels, thus providing easy means to strengthen the readily available optimization constraint by subtracting the value of $step$ from ub_{prio} (cf. Line 19 and 42 of Algorithm 1). To further facilitate such steps, *clasp* includes a single assumption $\overline{\alpha}$ in its optimization constraint and, for the most significant priority level $L = \max\{L_1, \ldots, L_n\}$, sets the weight $w@L$ of $\overline{\alpha}$ to $(\sum_{1 \leq i \leq n, L_i = L} w_i) + 1$. This makes sure that α belongs to every conflict constraint relying on the optimization constraint, so that these conflict constraints can be fixed (by discharging α) or withdrawn, respectively, immediately upon encountering either a solution or a conflict. To this end, *clasp* invokes the method `strengthenTagged()` when a solution is found and `removeTagged()` when a root-level conflict occurs, while keeping the assumption $\overline{\alpha}$ in place at the root level; applying either method turns $\overline{\alpha}$ into a fresh assumption without presuming any particular solver state, as otherwise required when performing constraint database simplifications.

The command-line parameters `--opt-hierarch` and `--opt-heuristic` allow for configuring (multi-criteria) optimization in *clasp*. If the value `0` is provided for the former, simple lexicographic optimization (without assumptions) is applied, while `1` and `2` switch to Algorithm 1 with the *leap* flag set to *false* and *true*, respectively. Furthermore, `--opt-heuristic` determines how `#minimize` statements are taken into account in *clasp*'s decision heuristics (Line 45 of Algorithm 1). While `0` falls back to the default heuristic, a static sign heuristic, preferably falsifying literals that occur in a `#minimize` statement, is applied for `1`. Value `2` switches to a dynamic heuristic that, after a solution has been found, falsifies its literals in a `#minimize` statement until a conflict is encountered. Finally, `3` combines `1` and `2`, thus falsifying literals subject to minimization if a respective variable is selected, while also picking such variables after a solution has been found (until hitting a conflict). The additional parameter `--restart-on-model` is a prerequisite for the values `2` and `3` to be effective; without it, they drop down to `0` and `1`, respectively.

5 Experiments

We developed the tool *aspcud*[4] applying our approach to multi-criteria optimization in ASP to Linux package configuration. At the start, *aspcud* translates a package configuration problem in Common Upgradability Description Format (CUDF; (Treinen and Zacchiroli 2009)) into ASP facts, described in Section 3. The translation involves mapping CUDF

[4] `http://www.cs.uni-potsdam.de/wv/aspcud`

package formulas to sets of packages (clauses) and tracing virtual packages that cannot directly be installed back to packages that implement them. Such flattening makes the problem encoding (cf. Figure 3) in ASP more convenient. Beyond syntactic simplifications, the translation by *aspcud* also exploits optimization criteria and package interdependencies to reduce the resulting ASP instance. For example, installing packages that are unrelated to an existing installation and install clauses merely degrades *paranoid* and *trendy* optimization criteria, so that such unrelated packages may be omitted in an ASP instance. Likewise, *paranoid* optimization criteria do not consider recommendations. They can thus be ignored if *paranoid* criteria are selected, but not in *trendy* optimization mode.

As ASP tools, *aspcud* (version 1.3.0) exploits *gringo* (version 3.0.3) for grounding and *clasp* (version 2.0.0-RC2) for solving. To illustrate the impact of the strategies and heuristics supported by *clasp*, our experiments consider several variants of it. Three settings are obtained by configuring `--opt-hierarch` with the values described above, indicated by a subscript:

- *clasp*$_0$: optimizing whole utility vectors (as described at the beginning of Section 4 and implemented also in *smodels* as well as *clasp* versions below 2.0.0),
- *clasp*$_1$: applying Algorithm 1 with the *leap* flag set to $false$, and
- *clasp*$_2$: applying Algorithm 1 with the *leap* flag set to $true$.

We further combine each *clasp*$_i$ ($i \in \{0,1,2\}$) with optimization-oriented heuristics, activated by setting `--opt-heuristic` to the value indicated by a superscript:

- *clasp*$_i^0$: applying no optimization-specific decision heuristic,
- *clasp*$_i^1$: applying the static sign heuristic to falsify literals of a `#minimize` statement,
- *clasp*$_i^2$: after a solution has been found, falsifying literals of a `#minimize` statement until a conflict is encountered, and
- *clasp*$_i^3$: combining the sign heuristic of *clasp*$_i^1$ with the dynamic approach of *clasp*$_i^2$.

We thus obtain twelve variants of *clasp*, each invoked with the (additional) command-line parameters `--sat-prepro`, `--heuristic=vsids`, `--restarts=128`, `--local-restarts`, and `--solution-recording`, which turned out to be helpful on large underconstrained optimization problems confronted in Linux package configuration. As mentioned above, *clasp*$_i^2$ and *clasp*$_i^3$ further require `--restart-on-model` to be effective, and we indicate the use of this parameter by writing *clasp*$_i^j$-r, where "-r" is mandatory for $j \in \{2,3\}$ and optional for $j \in \{0,1\}$. The reasonable combinations of the variable options amount to 18 variants of *clasp* to perform the optimization within *aspcud*.

For comparison, we also consider the package configuration tools *cudf2msu*[5] (version 1.0), *cudf2pbo*[6] (version 1.0), and *p2cudf*[7] (version 1.11). The PBO-based approaches of *cudf2pbo* and *p2cudf* are closely related to multi-criteria optimization in ASP via Algorithm 1, while the MaxSAT approach of *cudf2msu* utilizes unsatisfiable cores to iteratively refine lower bounds. The tools included for comparison belong to the leaders in a recent trial-run[1], called MISC-live, of the competition organized by mancoosi.

[5] `http://sat.inesc-id.pt/~mikolas/cudf2msu.html`
[6] `http://sat.inesc-id.pt/~mikolas/cudf2pbo.html`
[7] `http://wiki.eclipse.org/Equinox/p2/CUDFResolver`

Table 3 reports experimental results on package configuration problems used in the recent MISC-live run, divided by the tracks *paranoid*, *trendy*, and *user1–3*,[8] each applying different optimization criteria, where the criteria of the custom *user* tracks are as follows:

user1: Minimize first $|\mathcal{U}_U^P|$, second $|\mathcal{D}_O^P|$, and third $|\mathcal{C}_O^P|$.
user2: Minimize first $|\mathcal{C}_O^P|$, second $|\mathcal{D}_O^P|$, third $|\mathcal{R}_U^P|$, and fourth $|\mathcal{N}_O^P|$.
user3: Minimize first $|\mathcal{C}_O^P|$, second $|\mathcal{U}_U^P|$, third $|\mathcal{D}_O^P|$, and fourth $|\mathcal{N}_O^P|$.

We ran the five criteria combinations on 117 instances considered in the *paranoid* and *trendy* tracks of the MISC-live run (all instances except for the ones in the "debian-dudf" category, which were not available for download). For each track, the column headed by *S* provides the sums of solvers' scores according to the MISC-live ranking: a solver that returns a solution earns $b+1$ points, where b is the number of solvers that returned strictly better solutions; a solver that returns no solution earns $2 * s$ points, where s is the total number of participating solvers ($s = 21$ in our case); finally, a solver that crashes or returns a wrong solution (i.e., an invalid installation profile) is awarded $3*s$ points (for s as before). Note that a smaller score is better than a greater one, and solvers are ranked by their scores in ascending order. The columns headed by *T/O* report total runtimes per solver in seconds followed by the number of instances on which the solver was aborted, either before finding the optimum or while still attempting to prove it (or unsatisfiability, respectively). These statistics are used for tie-breaking wrt scores in MISC-live ranking, and they also yield valuable information regarding solvers' capabilities to prove optima: after 280 seconds of running, closeness of runtime exhaustion (300 seconds) is signaled to a solver, so that the remaining time can be used to output the best solution found so far. Accordingly, we count solutions returned after more than 280 seconds as aborts, which are not reflected in scores (columns *S*) if output solutions happen to be optimal without the proof being completed. We ran our experiments under MISC-live conditions on an Intel Quad-Core Xeon E5520 machine, possessing 2.27GHz processors and 48GB main memory, under Linux. The best scores and runtimes obtained among the variants of *clasp* as well as the best ones among its competitors are highlighted in bold face in Table 3.

Recall that two optimization criteria are applied in the *paranoid* track, three in the *user1* track, and four in the remaining tracks. One may expect solvers optimizing criteria in the order of significance (all but the variants of $clasp_0$) to have greater advantages the longer the sequence of criteria is. In fact, we observe that $clasp_0$, optimizing criteria in parallel, is competitive in the *paranoid* track; in particular, the static sign heuristic applied by the variants of $clasp_0^1$ helps them to achieve the smallest score. However, the gap to other solvers is not large, neither in terms of scores nor runtimes. Unlike this, the disadvantages of $clasp_0^0$ and $clasp_0^1$ variants are remarkable in the other four tracks; they are compensated to some extent by the optimization-oriented dynamic variable selection applied by $clasp_0^2$-r and $clasp_0^3$-r. Comparing the variants of $clasp_1$ and $clasp_2$, applying Algorithm 1, we note that they are less sensitive to heuristic aspects. Nonetheless, their relative performance varies over tracks, thus not suggesting any universal strategy to multi-criteria optimization. For

[8] The results of (a preliminary version of) *aspcud*, running $clasp_1^1$ in all five tracks, were scrambled in this trial-run due to scripting problems, which led to complete failure rather than a sub-optimal solution if an optimum could not be proven in time.

	paranoid		*trendy*		*user1*		*user2*		*user3*	
Solver	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*	*S*	*T/O*
$clasp_0^0$-r	431	2,287/6	1730	23,829/ 80	935	14,349/35	525	5,097/12	1031	14,184/37
$clasp_0^0$	416	2,294/6	2375	29,781/105	1727	21,897/73	1224	14,697/45	671	11,178/21
$clasp_0^1$-r	**410**	2,210/6	1560	22,660/ 73	898	13,466/30	502	4,654/ 9	980	13,682/35
$clasp_0^1$	**410**	2,326/6	2079	26,471/ 92	1723	21,525/72	922	10,767/31	658	10,675/23
$clasp_0^2$-r	427	2,135/6	712	16,867/ 51	527	5,891/11	426	2,981/ 5	587	7,628/20
$clasp_0^3$-r	429	**2,134**/6	740	17,079/ 52	507	5,863/12	425	3,044/ 6	576	7,769/21
$clasp_1^0$-r	425	2,428/6	579	16,713/ 50	550	5,819/14	434	3,000/ 6	710	8,958/25
$clasp_1^0$	417	2,418/6	549	16,544/ 50	**475**	5,318/12	**411**	2,538/ 5	502	6,279/16
$clasp_1^1$-r	429	2,405/6	622	17,304/ 50	518	5,908/13	438	2,976/ 6	676	8,938/23
$clasp_1^1$	427	2,372/6	613	16,946/ 49	490	5,478/12	416	2,562/ 5	496	6,144/16
$clasp_1^2$-r	427	2,352/6	571	16,646/ 50	518	5,358/13	418	2,582/ 5	471	6,356/16
$clasp_1^3$-r	429	2,346/6	**547**	**16,386**/ 50	499	**5,306**/12	413	2,498/ 5	497	6,255/16
$clasp_2^0$-r	425	2,392/6	806	16,598/ 50	523	5,583/13	421	2,677/ 6	479	5,548/12
$clasp_2^0$	417	2,364/7	748	17,132/ 50	487	5,823/14	422	2,583/ 5	482	5,592/15
$clasp_2^1$-r	416	2,378/6	752	17,269/ 52	492	5,663/12	414	**2,409**/ 5	451	**5,349**/11
$clasp_2^1$	425	2,365/6	864	17,128/ 51	517	6,151/15	412	2,681/ 5	463	5,972/14
$clasp_2^2$-r	445	2,402/6	706	16,551/ 50	528	5,788/13	419	2,700/ 5	**436**	5,519/13
$clasp_2^3$-r	434	2,345/6	748	16,982/ 51	518	5,850/14	415	2,559/ 5	457	5,360/13
cudf2msu	610	3,051/8	**669**	**5,318**/ 8	1270	8,709/18	548	**3,238**/ 7	**504**	4,750/ 9
cudf2pbo	465	**2,727**/7	1082	21,302/ 68	520	6,168/13	**462**	3,575/ 7	537	**3,487**/ 8
p2cudf	**463**	2,920/8	696	19,105/ 60	**516**	**3,947**/ 7	573	6,927/16	577	8,063/21

Table 3: *Results on package configuration problems used in a recent MISC-live run.*

instance, the variants of $clasp_1$, decreasing upper bounds linearly, are more successful than $clasp_2$ variants in the *trendy* track, where the large total runtimes and numbers of aborts indicate that many instances were hard to complete (proving optima failed in many cases). On the other hand, the exponential decrease scheme of $clasp_2$ enables some of its variants to achieve the smallest score and runtime in the *user3* track. Finally, comparing the variants of *clasp* with its three competitors, we observe that the ASP-based approach to Linux package configuration is highly competitive. In particular, its consistent performance is confirmed by scores, while each of the other tools achieved an impressive runtime (mainly by succeeding to prove optima) in some track: *cudf2msu* in *trendy*, *cudf2pbo* in *user3*, and *p2cudf* in *user1*. Unfortunately, *cudf2msu* produced non-optimal solutions and crashes in two tracks, *trendy* and *user1*, so that its ranking in these two tracks is not very conclusive.

6 Discussion

We presented an approach to dedicated Boolean multi-criteria optimization (Argelich et al. 2009) in modern ASP solvers (Gebser et al. 2007), admitting the imposition as well as the withdrawal of speculative upper bounds without solver relaunches. To this end, our approach exploits conflict-driven learning (cf. (Darwiche and Pipatsrisawat 2009; Marques-Silva et al. 2009)) and assumptions, as available in incremental SAT (Eén and Sörensson 2003). Albeit we modeled Linux package configuration in ASP, related PBO and MaxSAT modelings (Argelich et al. 2010) could be solved in the same fashion. However, while assumptions may be unnecessary (to compensate for unsatisfiability wrt infeasible bounds) in single-objective optimization, offered, e.g., by the PBO solvers *bsolo* (Manquinho and Marques-Silva 2004), *minisat+* (Eén and Sörensson 2006), and *pueblo* (Sheini and Sakallah 2006), pre-existing multi-criteria optimization techniques, like the ones presented in (Marques-Silva et al. 2011), lack tight solver integration. To our knowledge, among

the few systems implementing dedicated multi-criteria optimization, *sat4j* (Le Berre and Parrain 2010) and *wbo* (Manquinho et al. 2009) (when decreasing upper bounds rather than using unsatisfiable cores to refine lower bounds) rely on solver relaunches, i.e., the complete withdrawal of learned constraints, after an unsatisfiability proof showing optimality wrt the current optimization criterion. In contrast to them, the internalization of multi-criteria optimization functionalities in *clasp* enables maintaining valid constraints as well as guiding search by optimization-oriented heuristics. To this end, *clasp* offers several optimization-specific variations of its decision heuristics, which can be combined with linear or exponential upper bound decrease schemes. The additional integration of lower bound refinement techniques, proposed, e.g., in (Manquinho and Marques-Silva 2004) and (Manquinho et al. 2009), is an interesting subject to future work. In this process, competitions by mancoosi provide a venue for improving and sharing methods of optimization.

Acknowledgments. This work was partly funded by DFG grant SCHA 550/8-2. We are grateful to Daniel Le Berre for useful discussions on the subject of this work and to the mancoosi project team for organizing MISC(-live).

References

ARGELICH, J., LE BERRE, D., LYNCE, I., MARQUES-SILVA, J., AND RAPICAULT, P. 2010. Solving Linux upgradeability problems using Boolean optimization. In *Proceedings of the First International Workshop on Logics for Component Configuration (LoCoCo'10)*, I. Lynce and R. Treinen, Eds. Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 29. 11–22.

ARGELICH, J., LYNCE, I., AND MARQUES-SILVA, J. 2009. On solving Boolean multilevel optimization problems. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, C. Boutilier, Ed. AAAI Press/The MIT Press, 393–398.

BARAL, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

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

DARWICHE, A. AND PIPATSRISAWAT, K. 2009. Complete algorithms. See Biere et al. (2009), Chapter 3, 99–130.

DI COSMO, R., DURAK, B., LEROY, X., MANCINELLI, F., AND VOUILLON, J. 2006. Maintaining large software distributions: New challenges from the FOSS era. *EASST Newsletter 12*, 7–20.

EÉN, N. AND SÖRENSSON, N. 2003. An extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, E. Giunchiglia and A. Tacchella, Eds. Lecture Notes in Computer Science, vol. 2919. Springer-Verlag, 502–518.

EÉN, N. AND SÖRENSSON, N. 2006. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation 2*, 1–26.

FERRARIS, P. 2005. Answer sets for propositional theories. In *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. Lecture Notes in Artificial Intelligence, vol. 3662. Springer-Verlag, 119–131.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`.

GEBSER, M., KAMINSKI, R., KÖNIG, A., AND SCHAUB, T. 2011. Advances in *gringo* series 3. In *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic*

Reasoning (LPNMR'11), J. Delgrande and W. Faber, Eds. Lecture Notes in Artificial Intelligence, vol. 6645. Springer-Verlag, 345–351.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI Press/The MIT Press, 386–392.

LE BERRE, D. AND PARRAIN, A. 2010. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation 7*, 59–64.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

mancoosi. `http://www.mancoosi.org`.

MANQUINHO, V. AND MARQUES-SILVA, J. 2004. Satisfiability-based algorithms for Boolean optimization. *Annals of Mathematics and Artificial Intelligence 40,* 3-4, 353–372.

MANQUINHO, V., MARQUES-SILVA, J., AND PLANES, J. 2009. Algorithms for weighted Boolean optimization. In *Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, O. Kullmann, Ed. Lecture Notes in Computer Science, vol. 5584. Springer-Verlag, 495–508.

MARQUES-SILVA, J., ARGELICH, J., GRAÇA, A., AND LYNCE, I. 2011. Boolean lexicographic optimization: Algorithms & applications. *Annals of Mathematics and Artificial Intelligence,* to appear.

MARQUES-SILVA, J., LYNCE, I., AND MALIK, S. 2009. Conflict-driven clause learning SAT solvers. See Biere et al. (2009), Chapter 4, 131–153.

SHEINI, H. AND SAKALLAH, K. 2006. Pueblo: A hybrid pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation 2*, 165–189.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

SYRJÄNEN, T. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

SYRJÄNEN, T. 1999. A rule-based formal model for software configuration. Technical Report A55, Helsinki University of Technology. `http://www.tcs.hut.fi/Publications/bibdb/HUT-TCS-A55.ps`.

SYRJÄNEN, T. 2000. Including diagnostic information in configuration models. In *Proceedings of the First International Conference on Computational Logic (CL'00)*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. Stuckey, Eds. Lecture Notes in Computer Science, vol. 1861. Springer-Verlag, 837–851.

TREINEN, R. AND ZACCHIROLI, S. 2009. Common upgradability description format (CUDF) 2.0. Technical Report 003, mancoosi — managing software complexity. Available at (mancoosi).