Multi-Criteria Optimization in Answer Set Programming

Martin Gebser and Roland Kaminski and Benjamin Kaufmann and Torsten Schaub

Institut für Informatik, Universität Potsdam

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. To this end, we describe the Linux package configuration tool *aspcud* and compare its performance with systems pursuing alternative approaches.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Answer Set Programming, Multi-Criteria Optimization, Linux Package Configuration

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Solving multi-criteria optimization problems is of great interest in various application domains because it allows 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.

As detailed in the extended version of this paper (cf. [8]), we are interested in solving Linux package configuration problems by appeal to the multi-criteria optimization capacities of Answer Set Programming (ASP; [3]). To this end, we develop novel general-purpose strategies and heuristics in the context of modern (conflict-driven learning) ASP solving [10]. 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 [15, 16], using ASP for representing and solving configuration problems for the Debian GNU/Linux system. 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* [13], 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* [11] offers so-called weak constraints, serving the same purpose.

2 Background

The semantics of a (ground extended) logic program Π is given by particular models, called answer sets; see [13] for details. In addition to rules, Π can contain `#minimize` statements of the form

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

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 [9]. 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`$[\overline{\ell_1} = w_1@L_1, \dots, \overline{\ell_n} = w_n@L_n]$.

3 Multi-Criteria Optimization Algorithm

As detailed in [13], #`maximize` statements can be turned into #`minimize` statements, literals with negative weights be transformed such that weights become positive, 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* [13], 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 [2] 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. [1]), 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* [10], exploiting conflict-driven learning, 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 [6], 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. [5, 12]) with multi-criteria optimization is shown in Algorithm 1. The sequence $\langle \mathbf{L}_1, \dots, \mathbf{L}_{low} \rangle$ 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 $\langle ub_1, \dots, ub_{low} \rangle$ 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. Furthermore, if the current upper bound still needs to be recorded, a corresponding `#sum` constraint, as available in ASP input languages [14, 7], 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 $\langle ub_1, \dots, ub_{low} \rangle$ 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

Algorithm 1: CDNL-OPT

```
Input: A logic program $\Pi$, a statement $\#\texttt{minimize}[\ell_1 = w_1@L_1, \dots, \ell_n = w_n@L_n]$, and a flag
       $\mathit{leap} \in \{\mathit{true}, \mathit{false}\}$.
1  $\langle \mathbf{L}_1, \dots, \mathbf{L}_{low} \rangle \leftarrow \langle \max(\{L_1, \dots, L_n\} \setminus \{\mathbf{L}_1, \dots, \mathbf{L}_{m-1}\}) \rangle_{1 \leq m \leq |\{L_1, \dots, L_n\}|}$
2  $\mathit{assm} \leftarrow \mathit{prio} \leftarrow \mathit{step} \leftarrow 1$          // assumption, priority, and step counter
3  $\mathit{lb} \leftarrow 0$                                                 // lower bound
4  loop
5      PROPAGATE                                         // deterministically assign implied literals
6      if conflict then
7          if at root level then                        // unsatisfiability modulo optimization constraint
8              if $\mathit{assm} = 1$ then exit
9              ASSIGN $\alpha_{\mathit{assm}}$                      // deactivate old optimization constraint
10             $\mathit{lb} \leftarrow (\mathit{ub}_{\mathit{prio}} - \mathit{step}) + 1$
11             while $\mathit{prio} \leq \mathit{low}$ and $\mathit{ub}_{\mathit{prio}} = \mathit{lb}$ do
12                 if $\mathit{recd} = \mathit{true}$ then ADD $\#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{\mathit{prio}}]\mathit{lb}$
13                 $\mathit{lb} \leftarrow 0$
14                 $\mathit{recd} \leftarrow \mathit{true}$
15                 $\mathit{prio} \leftarrow \mathit{prio} + 1$
16             if $\mathit{prio} > \mathit{low}$ then exit
17             $\mathit{step} \leftarrow 1$
18             $\mathit{assm} \leftarrow \mathit{assm} + 1$
19             ADD $\big(\alpha_{\mathit{assm}} \vee \#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{\mathit{prio}}]\mathit{ub}_{\mathit{prio}} - \mathit{step}\big)$
20             ASSUME $\overline{\alpha_{\mathit{assm}}}$                   // activate new optimization constraint
21         else
22             ANALYZE                      // analyze conflict and add (violated) conflict constraint
23             BACKJUMP                     // unassign literals until conflict constraint is unviolated
24     else if solution then
25         if $\mathit{assm} = 1$ then $\mathit{recd} \leftarrow \mathit{true}$          // upper bound of witness yet unrecorded
26         else
27             ASSIGN $\overline{\alpha_{\mathit{assm}}}$                         // fix old optimization constraint
28             if $(\Sigma_{1 \leq i \leq n, L_i = \mathbf{L}_{\mathit{prio}}, \ell_i \text{assigned to true}}\, w_i) < \mathit{ub}_{\mathit{prio}} - \mathit{step}$ then $\mathit{recd} \leftarrow \mathit{true}$
29             else $\mathit{recd} \leftarrow \mathit{false}$
30         $\langle \mathit{ub}_1, \dots, \mathit{ub}_{\mathit{low}} \rangle \leftarrow \langle \Sigma_{1 \leq i \leq n, L_i = \mathbf{L}_m, \ell_i \text{assigned to true}}\, w_i \rangle_{1 \leq m \leq \mathit{low}}$
31         print answer set along with $\langle \mathit{ub}_1, \dots, \mathit{ub}_{\mathit{low}} \rangle$
32         $\mathit{prio}' \leftarrow \mathit{prio}$
33         while $\mathit{prio} \leq \mathit{low}$ and $\mathit{ub}_{\mathit{prio}} = \mathit{lb}$ do
34             if $\mathit{recd} = \mathit{true}$ then ADD $\#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{\mathit{prio}}]\mathit{lb}$
35             $\mathit{lb} \leftarrow 0$
36             $\mathit{recd} \leftarrow \mathit{true}$
37             $\mathit{prio} \leftarrow \mathit{prio} + 1$
38         if $\mathit{prio} > \mathit{low}$ then exit
39         if $\mathit{prio} = \mathit{prio}'$ and $\mathit{leap} = \mathit{true}$ then $\mathit{step} \leftarrow \min\{2 * \mathit{step}, \lceil (\mathit{ub}_{\mathit{prio}} - \mathit{lb})/2 \rceil\}$
40         else $\mathit{step} \leftarrow 1$
41         $\mathit{assm} \leftarrow \mathit{assm} + 1$
42         ADD $\big(\alpha_{\mathit{assm}} \vee \#\texttt{sum}[\ell_i = w_i \mid 1 \leq i \leq n, L_i = \mathbf{L}_{\mathit{prio}}]\mathit{ub}_{\mathit{prio}} - \mathit{step}\big)$
43         ASSUME $\overline{\alpha_{\mathit{assm}}}$                       // activate new optimization constraint
44         BACKJUMP                     // unassign literals until optimization constraint is unviolated
45     else DECIDE                                  // non-deterministically assign some literal
```

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 $\#$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. [5, 12]) 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.

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 $\#\texttt{minimize}[\ell_1 = w_1@L_1, \dots, \ell_n = w_n@L_n]$ in a single optimization constraint, using as data-structure a two-dimensional array of size $|\{L_1,\dots,L_n\}| * |\{\ell_1,\dots,\ell_n\}|$ with $w_1,\dots,w_n$ as its (non-zero) entries. Furthermore, the vector $\langle ub_m\rangle_{1\leq m\leq|\{L_1,\dots,L_n\}|}$ of upper bounds is initialized to $\langle\infty_m\rangle_{1\leq m\leq|\{L_1,\dots,L_n\}|}$ 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 $L = \max\{L_1,\dots,L_n\}$, 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.

4 Experiments

We developed the tool *aspcud*[1] 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; [17]) into ASP facts, described in the extended version of this paper [8]. The translation involves mapping CUDF 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. [8]) in ASP more convenient. Beyond syntactic simplifications, the translation by *aspcud* also exploits optimization criteria and package interdependencies to reduce the resulting ASP instance.

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 3 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:

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

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, $clasp^2_i$ and $clasp^3_i$ further require `--restart-on-model` to be effective, and we indicate the use of this parameter by writing $clasp^j_i$-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*.

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

For comparison, we also consider the package configuration tools *cudf2msu*[2] (version 1.0), *cudf2pbo*[3] (version 1.0), and *p2cudf*[4] (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[5], called MISC-live, of the competition organized by mancoosi.

Table 1 reports experimental results on package configuration problems used in the recent MISC-live run, divided by the tracks *paranoid*, *trendy*, and *user1–3*,[6] each applying a different combination of optimization criteria. Note that the number of lexicographically ordered utilities is two in the *paranoid* track, three in the *user1* track, and four in the *trendy* and *user2–3* tracks. 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 1.

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 instance,

2 `http://sat.inesc-id.pt/~mikolas/cudf2msu.html`
3 `http://sat.inesc-id.pt/~mikolas/cudf2pbo.html`
4 `http://wiki.eclipse.org/Equinox/p2/CUDFResolver`
5 `http://www.mancoosi.org/misc-live/20101126`
6 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 1 Results on package configuration problems used in a recent MISC-live run.

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.

5 Discussion

We presented an approach to dedicated multi-criteria optimization in ASP. In particular, we detailed the use of assumptions in modern (conflict-driven learning) Boolean constraint solvers, so that speculative upper bounds can be imposed temporarily and withdrawn after unsatisfiability proofs without relaunching the solver. In fact, our approach is readily applicable in related areas like PBO and MaxSAT. Albeit Linux package configuration tools based on these formalisms may already exploit similar techniques, we are unaware of precise specifications of them. In the future, regular comparisons in competitions by mancoosi could provide a fruitful platform for improving and sharing methods of optimization.

The interested reader is referred to the extended version of this paper [8] for a detailed description of solving Linux package configuration problems by appeal to the multi-criteria optimization capacities introduced in the previous sections.

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

1 J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva, and P. Rapicault. Solving Linux upgradeability problems using Boolean optimization. In I. Lynce and R. Treinen, editors, *Proceedings of the First International Workshop on Logics for Component Configuration (LoCoCo'10)*, pages 11–22. 2010.

2 J. Argelich, I. Lynce, and J. Marques-Silva. On solving Boolean multilevel optimization problems. In C. Boutilier, editor, *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 393–398. AAAI Press/The MIT Press, 2009.

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

4 A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

5 A. Darwiche and K. Pipatsrisawat. Complete algorithms. In Biere et al. [4], pages 99–130.

6 N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 502–518. Springer-Verlag, 2004.

7 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`.

8 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Multi-criteria optimization in ASP and its application to Linux package configuration. Available at `http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc11b.pdf`.

9 M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in *gringo* series 3. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 345–351. Springer-Verlag, 2011.

10 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.

11 N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

12 J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [4], pages 131–153.

13 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

14 T. Syrjänen. Lparse 1.0 user's manual. Available at `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

15 T. Syrjänen. A rule-based formal model for software configuration. Technical Report A55, Helsinki University of Technology, 1999.

16 T. Syrjänen. Including diagnostic information in configuration models. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. Stuckey, editors, *Proceedings*

of the First International Conference on Computational Logic (CL'00), pages 837–851. Springer-Verlag, 2000.

17 R. Treinen and S. Zacchiroli. Common upgradability description format (CUDF) 2.0. Technical Report 003, mancoosi — managing software complexity, 2009.