Progress in *clasp* series 3

M. Gebser[1,3], R. Kaminski[3], B. Kaufmann[3], J. Romero[3], and T. Schaub[2,3]*

[1]Aalto University, HIIT [2]INRIA Rennes [3]University of Potsdam

Abstract. We describe the novel functionalities comprised in *clasp*'s series 3. This includes parallel solving of disjunctive logic programs, parallel optimization with orthogonal strategies, declarative support for specifying domain heuristics, a portfolio of prefabricated expert configurations, and an application programming interface for library integration. This is complemented by experiments evaluating *clasp* 3's optimization capacities as well as the impact of domain heuristics.

1 Introduction

The success of Answer Set Programming (ASP; [1]) is largely due to the availability of effective solvers. Early ASP solvers *smodels* [2] and *dlv* [3] were followed by SAT-based ones, such as *assat* [4] and *cmodels* [5], before genuine conflict-driven ASP solvers like *clasp* [6] and *wasp* [7] emerged. In addition, there is a continued interest in mapping ASP onto solving technology in neighboring fields [8, 9].

In what follows, we provide a comprehensive description of *clasp*'s series 3 (along with some yet unpublished features already in *clasp* 2). Historically, *clasp* series 1 [6] constitutes the first genuine conflict-driven ASP solver, featuring conflict-driven learning and back-jumping. *clasp* series 2 [10] supports parallel search via shared memory multi-threading. *clasp* series 3 further extends its predecessors by integrating various advanced reasoning techniques in a uniform parallel setting. The salient features of *clasp* 3 include parallel solving of disjunctive logic programs, parallel optimization with orthogonal strategies, declarative support for specifying domain heuristics, a portfolio of prefabricated expert configurations, and an application programming interface for library integration. We detail these functionalities in Section 2 to 6 from a system- and user-oriented viewpoint. Section 7 is dedicated to an empirical study comparing the various optimization strategies of *clasp* 3. Also, we demonstrate the impact of domain heuristics and contrast their performance to using disjunctive logic programs when enumerating inclusion minimal answer sets.

We refer the interested reader to [11] for the formal foundations and basic algorithms underlying conflict-driven ASP solving as used in *clasp*. This includes basic concepts like completion and loop nogoods as well as algorithmic ones related to conflict-driven constraint learning (CDCL).

2 Disjunctive solving

Solving disjunctive logic programs leads to an elevated level of complexity [12] because unfounded-set checking becomes a co-NP-complete problem [13]. As a consequence,

* Affiliated with Simon Fraser University, Canada, and IIIS Griffith University, Australia.

corresponding systems combine a solver generating solution candidates with another testing them for unfounded-freeness. For instance, *dlv* [3] carries out the latter using a SAT solver [14], *claspD* [15] uses *clasp* for both purposes. Common to both is that search is driven by the generating solver and that the testing solver is merely re-invoked on demand. Such repeated invocations bear a great amount of redundancy, in particular, when using conflict-learning solvers because learned information is lost.

Unlike this, *clasp* 3 enables an equitable interplay between generating and testing solvers. Solver units are launched initially with their respective Boolean constraint problems, and they subsequently communicate in a bidirectional way. Constraints relevant for the unfounded set check are enabled using *clasp*'s interface for solving under assumptions (cf. Section 6). This allows both units to benefit from conflict-driven learning over whole runs. The theoretical foundations for this approach are laid in [16].[1]

clasp 3 decomposes the unfounded set problem based on the strongly connected components (SCCs) of the program's positive dependency graph. Head cycle components (SCCs sharing two head atoms of a rule) are associated with testing solvers responsible for complex unfounded set checks. Head cycle free components are checked using *clasp*'s tractable unfounded set checking procedure for normal programs. The generator combines propagation via completion nogoods with these unfounded set checks. Tractable unfounded set checks are performed once at each decision level. Because complex unfounded set checks are expensive, their frequency is limited by default (this is configurable using option `--partial-check`); only checks for total assignments are mandatory before a model is accepted. Finally, it is worth mentioning that *clasp* 3 propagates top-level assignments from generators to testers. That is, whenever a variable's truth value is determined by the generator, it is also fixed in the tester.

Building on *clasp*'s multi-threaded architecture, the assembly of generating and testing solvers is reproduced to obtain n threads running in parallel. This results in n generating and $k \times n$ testing solvers (given k head cycle components), all of which can be separately configured, for instance, by specifying portfolios of search strategies for model generation and/or unfounded set checking (cf. Section 5). Notably, different generators as well as testers solving the same unfounded set sub-problem share common data, rather than copying it n times. The testing solver can be configured via option `--tester`, accepting a string of *clasp* options. The individual performance of the $k \times n$ solvers and their respective problem statistics can be inspected by option `--stats=2`.

Another advance in *clasp* 3 is its extension of preprocessing to disjunctive ASP. Preprocessing starts with the identification of equivalences and resulting simplifications on the original program (controlled by option `--eq`). This extends the techniques from [17] to disjunctive logic programs. A subsequent dependency analysis of the program results in the aforementioned decomposition in head cycle components. This is followed by a translation of recursive weight constraints. In contrast to previous approaches, *clasp* 3 restricts this translation to weight constraints belonging to some head cycle component. Finally, representations for completion nogoods and unfounded set nogoods for each head cycle component are created (according to [16]). Notably, *clasp* 3's preprocessor can be decoupled with option `--pre`, providing a mapping between two disjunctive logic programs in *smodels* format. In this way, it can be used as a

[1] [16] also contains experiments done with an early and restricted prototype of *clasp* 3.

preprocessor for other ASP solvers for (disjunctive) logic programs relying on *smodels* format. For example, the program

```
 a ; b.                  c :- a.   a :- c.   d :- not c.
```

is translated by calling '`clasp --pre`' (and conversion to human-readable form) into

```
 a :- not b. b :- not a. c :- a.             d :- not a.
```

Here, *clasp* 3's preprocessor turns the disjunction '`a ; b`' into two normal rules due to a missing head cycle and identifies the equivalence between `a` and `c`.

3 Optimization

Lexicographic optimization of linear objective functions is an established component of ASP solvers, manifested by *#minimize* statements [2] and weak constraints [3]. Traditionally, optimization is implemented in ASP solvers via branch-and-bound search. As argued in [18], this constitutes a *model-guided* approach that aims at successively producing models of descending costs until an optimal model is found (by establishing the unsatisfiability of the problem with a lower cost). Since series 2, *clasp* features several corresponding strategies and heuristics [19], including strategies that allow for non-uniform descents during optimization. For instance, in multi-criteria optimization, this enables *clasp* to optimize criteria in the order of significance, rather than pursuing a rigid lexicographical descent. *clasp* 3 complements this with so-called *core-guided* optimization techniques originating in the area of MaxSAT [14]. Core-guided approaches rely on successively identifying and relaxing unsatisfiable cores until a model is obtained. The implementation in *clasp* 3 seamlessly integrates the core-guided optimization algorithms *oll*[2] [20] and *pmres* [21]. Both algorithms can be (optionally) combined with disjoint core preprocessing [22], which calculates an initial set of unsatisfiable cores to initialize the algorithms, and as a side effect provides an approximation of the optimal solution. Furthermore, whenever an algorithm relaxes an unsatisfiable core, constraints have to be added to the solver. These constraints can be represented using either equivalences or implications. The former offers a slightly stronger propagation at the expense of adding more constraints.

The specific optimization strategy is configured in *clasp* 3 via option `--opt-strategy`. While its first argument distinguishes between model- and core-guided optimization, the second one handles the aforementioned refinements.

Building on *clasp*'s multi-threaded architecture, model- and core-guided optimization techniques can be combined. As detailed in Section 5, *clasp* 3 supports optimization portfolios for running several threads in parallel with different approaches, strategies, and heuristics, exchanging lower and upper bounds of objective functions (in addition to conflict nogoods). This combination of model- and core-guided optimization makes the overall optimization process more robust, as we empirically show in Section 7.

Moreover, *clasp* 3 adds a new reasoning mode for enumerating optimal models via option `--opt-mode=optN`. As usual, the number of optimal answer sets can be restricted by adding an integer to the command line. Interestingly, this option can also be combined with the intersection and union of answer sets (cf. option `--enum-mode`),

[2] [20] contains experiments with an early prototype called *unclasp*.

respectively. This is of great practical relevance whenever it comes to identifying atoms being true or false in all optimal answer sets.

Finally, it is worth mentioning that *clasp* 3's optimization capacities can also be used for solving PB and (weighted/partial) MaxSAT problems. In fact, *clasp* 3 won the second place in the *Unweighted Max-SAT - Industrial* category by using its core-guided optimization in the Max-SAT Evaluation in 2014.

4 Heuristics

In many domains, general-purpose solving capacities can be boosted by domain-specific heuristics. To this end, *clasp* 3 provides a general declarative framework for incorporating such heuristics into ASP solving. The heuristic information is exploited by a dedicated `domain` heuristic in *clasp* when it comes to non-deterministically assigning a truth value to an atom. In fact, *clasp*'s decision heuristic is modifiable from within a logic program as well as from the command line. This allows for specifying context-dependent activation of heuristic biases interacting with a problem's encoding. This approach was formally introduced in [23][3] and extended in *clasp* 3 as described below. On the other hand, *clasp* 3's command line options allow us to directly refer to structural components of the program (optimization statements, strongly connected components, etc.) and do not require any additional grounding. The domain heuristic is enabled by setting option `--heuristic` to `domain`, which extends *clasp*'s activity-based `vsids` heuristic.

Heuristic information is represented by means of the predicate `_heuristic`. The ternary version of this predicate takes a reified atom, a heuristic modifier, and an integer to quantify a heuristic modification. There are four primitive heuristic modifiers, viz. `sign`, `level`, `init`, and `factor`. The modifier `sign` allows for controlling the truth value assigned to variables subject to a choice. For example, the program

```
{a}.  _heuristic(a,sign,1).
```

produces the answer set containing `a` first. The modifier `level` allows for using integers to rank atoms. Atoms at higher levels are decided before atoms with lower level. The default level for each atom is `0`. Atoms sharing the same level are decided by their `vsids` score. Modifiers `true` and `false` are defined in terms of `level` and `sign`.

```
_heuristic(X,level,Y) :- _heuristic(X,true,Y).
_heuristic(X,sign,1)  :- _heuristic(X,true,Y).
_heuristic(X,level,Y) :- _heuristic(X,false,Y).
_heuristic(X,sign,-1) :- _heuristic(X,false,Y).
```

The modifiers `init` and `factor` allow us to modify the scores of the underlying `vsids` heuristic. Unlike `level`, they only bias the search without establishing a strict ranking among atoms. The modifier `init` allows us to add a value to the initial heuristic score of an atom that decays as any `vsids` score, while `factor` allows us to multiply the `vsids` scores of atoms by a given value. For example, the following rule biases the solver to choosing `p(T-1)` whenever `p(T)` is true.

```
_heuristic(p(T-1),factor,2) :- p(T).
```

[3] [23] also contains experiments done with an early and restricted prototype, called *hclasp*.

clasp 3's structure-oriented heuristics are supplied via the command line. Apart from supplying `--heuristic=domain`, the heuristic modifications are specified by option `--dom-mod=`m`,`p, where m ranges from `0` to `5` and specifies the modifier:

m	Modifier	m	Modifier	m	Modifier
`0`	None	`1`	`level`	`2`	`sign` (positive)
`3`	`true`	`4`	`sign` (negative)	`5`	`false`

and p specifies bit-wisely the atoms to which the modification is applied:

`0`	Atoms only
`1`	Atoms that belong to strongly connected components
`2`	Atoms that belong to head cycle components
`4`	Atoms that appear in disjunctions
`8`	Atoms that appear in optimization statements
`16`	Atoms that are shown

Whenever m equals `1`, `3`, or `5`, the level of the selected atoms depends on p. For example, with option `--dom-mod=2,8`, we apply a positive `sign` to atoms appearing in optimization statements, and with option `--dom-mod=1,20`, we apply modifier `level` to both atoms appearing in disjunctions as well as shown atoms. In this case, atoms satisfying both conditions are assigned a higher level than those that are only shown, and these get a higher level than those only appearing in disjunctions.

Compared to programmed heuristics, the command line heuristics do not allow for applying modifiers `init` or `factor` and cannot represent dynamic heuristics. But they allow us to directly refer to structural components of the program and do not require any additional grounding. When both methods are combined, the choices modified by the `_heuristic` predicate are not affected by the command line heuristics. When launched with option `--stat`, *clasp* 3 prints the number of modified choices.

Apart from boosting solver performance, domain specific heuristics can be used for computing inclusion minimal answer sets [24, 25]. This can be achieved by ranking choices over shown atoms highest and setting their sign modifier to false. As an example, consider the following program

```
1 {a(1..3)}.  a(2) :- a(3).  a(3) :- a(2).  {b(1)}.  #show a/1.
```

Both the command line option '`--dom-mod=5,16`' as well as the addition of the heuristic fact '`_heuristic(a(1..3),false,1).`' guarantee that the first answer set produced is inclusion minimal wrt. the atoms of predicate `a/1`. Moreover, both allow for enumerating all inclusion minimal solutions in conjunction with option `--enum-mod=domRec`. In our example, we obtain the answer sets {`a(1)`} and {`a(2), a(3)`}. Note that this enumeration mode relies on solution recording and is thus prone to an exponential blow-up in space. However, this often turns out to be superior to enumerating inclusion minimal model via disjunctive logic programs, which is guaranteed to run in polynomial space. We underpin this empirically in Section 7.

Independent of the above domain-specific apparatus, *clasp* provides means for configuring the sign heuristics, fixing which truth values get assigned to which type of variables. In general, *clasp* selects signs based on a given sign heuristics. For instance, this can be progress saving (`--save-progress`; [26]) or an optimization-oriented heuristic (`--opt-heuristic`). Also, each decision heuristic in *clasp* implements a sign heuristic. For example, *clasp*'s `vsids` heuristic prefers the sign of a variable according to the frequency of the corresponding literal in learned nogoods. Whenever no sign heuristic applies, e.g. in case of ties, the setting of option `--sign-def` determines the sign; by default, it assigns atoms to false and bodies to true. Other options are `1` (assign true), `2` (assign false), `3` (assign randomly), and `4` (assign bodies and atoms in disjunctions true). Finally, the option `--sign-fix` permits to disable all sign heuristics and enforce the setting of `--sign-def`.

5 Configuration

Just as any modern conflict-driven ASP, PB, or SAT solver, *clasp* is sensitive to search configurations. In order to relieve users from extensive parameter tuning, *clasp* offers a variety of prefabricated configurations that have shown to be effective in different settings. A specific configuration is selected by means of option `--configuration`, taking one of the following arguments:

`frumpy`	Use conservative defaults similar to those used in earlier *clasp* versions
`jumpy`	Use more aggressive defaults (than `frumpy`)
`tweety`	Use defaults geared towards typical ASP problems
`trendy`	Use defaults geared towards industrial problems
`crafty`	Use defaults geared towards crafted problems
`handy`	Use defaults geared towards large problems
`<file>`	Use configuration file to configure solver(s)

The terms 'industrial' and 'crafted' refer to the respective categories at SAT competitions; 'aggressive defaults' restart search and erase learned nogoods quite frequently. Unlike previous *clasp* series relying on the default configuration `frumpy` aiming at highest robustness, the one of *clasp* 3, viz. `tweety`, was automatically identified by *piclasp*[4] (a configurator for *clasp* based on *smac* [27]) and manually smoothened afterwards. Such an automatic approach is unavoidable in view of *clasp* 3's huge space of 10^{60} configurations composed of more than 90 parameters.

Note that using a configuration file enables freely customizable solver portfolios in parallel solving. We rely on this for tackling optimization problems in Section 7 by running complementary optimization strategies in parallel. For an example of such a portfolio, call *clasp* with option `--print-portfolio`. The result constitutes a portfolio of complementary default configurations for parallel ASP solving. This also extends to the disjunctive case, where the configuration of testing solvers can be configured by option '`--tester=--configuration=<file>`' to apply the portfolio in `<file>` to the tester. Options given on the command-line are added to all configurations in a configuration file. If an option is given both on the command-line and in a configuration file, the one from the command-line takes precedence.

When solving in parallel, the configurations in the portfolio are assigned to threads in a round-robin fashion. That is, *clasp* runs with the configuration from the first line

[4] http://www.cs.uni-potsdam.de/piclasp

in thread `0`, with the one from the second line in thread `1`, etc., until all threads are (circularly) assigned configurations from the portfolio. The mapping of portfolios to threads is used for providing thread-specific solver statistics. That is, launching *clasp* 3 with `--stats=2` does not only provide statistics aggregated over all threads but also for each individual one. Moreover, the winning thread[5] is identified by this mapping (and printed after '`Winner:`').

Furthermore, *clasp*'s multi-threaded architecture was extended to handle more complex forms of nogood exchange. In addition to the global distribution scheme described in [10], *clasp* 3 implements a new thread-local scheme. The scheme can be configured by option `--dist-mode`. In the new scheme, each thread has a (lock-free) multi-producer/single-consumer queue. For distributing nogoods, threads push "interesting" nogoods onto the queues of their peers. For integrating nogoods, threads pop nogoods from their local queues. On the other hand, *clasp* 3 now supports topology-based nogood exchange. To this end, option `--integrate` allows for exchanging nogoods among `all` peers or those connected in the form of a `ring`, hyper `cube`, or extended hyper cube (`cubex`). With the global scheme, nogoods are distributed among all threads but only integrated by threads from their peers (via a peer check upon receive). With the thread-local scheme, threads distribute nogoods only to peers (via a peer check upon send). Hence, the thread-local scheme is more suited for a topology-based exchange.

6 Library

While *clasp* is a versatile stand-alone system, it can also be integrated as a library. To this end, *clasp* 3 provides various interfaces for starting and controlling operative solving processes. This includes interfaces for incremental solving, updating a logic program, managing solver configurations, and for (asynchronous as well as iterative) solving under assumptions. Furthermore, recorded nogoods, heuristic values, and other dynamic information can either be kept or removed after each solving step.

Figure 1 illustrates *clasp*'s C++ library. At its center is the **`ClaspFacade`** class, which provides a simplified interface to the various classes used for solving. The typical workflow for using the *clasp* library is as follows.

1. Construct a **`ClaspFacade`** and a **`ClaspConfig`** object.
2. Configure search, preprocessing, etc. options in the configuration object.
3. Obtain a **`LogicProgram`** object by calling `startAsp` with the respective configuration object.
4. Add (ground) rules to the logic program by calling `addRule`.
5. Call `prepare` for performing preprocessing and necessary initialization tasks, like creating and configuring solver objects.
6. Finally, call `solve` to start searching for models.

This workflow covers a single-shot solving process. For multi-shot solving, method `update` has to be called, which allows for continuing the above process at step 4.

[5] The winning thread either exhausts the search space or produces the last model if no complete search space traversal is necessary.

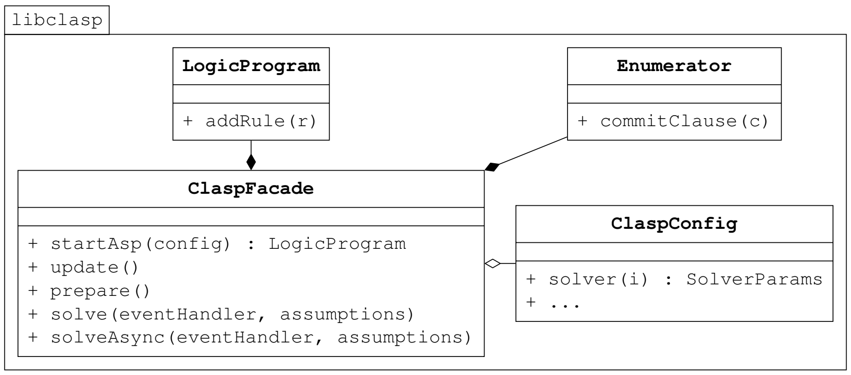
libclasp
LogicProgram
+ addRule(r)
Enumerator
+ commitClause(c)
ClaspFacade
+ startAsp(config) : LogicProgram
+ update()
+ prepare()
+ solve(eventHandler, assumptions)
+ solveAsync(eventHandler, assumptions)
ClaspConfig
+ solver(i) : SolverParams
+ ...

Fig. 1. Class Diagram for Excerpt of *clasp*'s C++ Library

This is especially interesting when combined with solving under assumptions (second parameter of `solve`). For example, planning problems typically require an a priori unknown horizon to find a solution. With the above workflow, the horizon can be extended at each step and assumptions can be used to check the goal situation at the current horizon. Also note that the configuration object can be updated at each step; the changes are propagated when calling the `update` method. For example, *clasp* 3 allows us to control the information kept between successive calls via attribute `solver(`i`).forgetSet` of **`ClaspConfig`**, which can be configured for each solver thread i individually. This includes heuristic scores, nogood activities, signs, and learnt nogoods.[6] For instance, reassigning previous truth values by keeping `heuristic scores` and `signs` usually makes the solver stay in similar areas of the search space.

Another interesting feature is asynchronous solving, using method `solveAsync`. This allows for starting a search in the background, which can be interrupted at any time. Use cases for this are applications that require to react to external events, as in assisted living or robotics scenarios.

Furthermore, the `solve` and `solveAsync` methods take an event handler as argument. This handler receives events at specific parts of the search, like the beginning and end of the search, as well as when a model is found. A model event is reported along with a reference to the underlying **`Enumerator`** object. At this point, it is possible to use the enumerator to add clauses over internal solver literals[7] to the current search. This is rather effective because it avoids program updates and preprocessing. And it is often sufficient for synthesizing a constraint, for instance, from the last obtained model.

Paired with corresponding interfaces of *gringo* 4, the extended low level interface of *clasp* 3 has led to *clingo* 4's higher level application programming interfaces (API) in *lua* and *python* [28].[8] Further applications using the *clasp* libray include the hybrid solvers *clingcon* [29] and *dlvhex* [30].

[6] In fact, these parameters are also controllable in *clingo* by option `--forget-on-step`.
[7] The **`LogicProgram`** class provides methods to map atoms to solver literals.
[8] The API reference can be found at http://potassco.sourceforge.net/gringo.html.

A final detail worth mentioning is that *clasp* 3 supports changing optimization statements between successive solving steps. This includes the extension and contraction of objective functions by adding or deleting weighted atoms from them. This is, for instance, relevant in planning domains whenever the horizon is extended.

7 Experiments

For studying the interplay of the various techniques discussed above, we conduct an empirical study on optimization problems. Despite their great practical relevance, only few such studies exist in ASP [19, 7]. Moreover, optimization problems are not only more complex than their underlying decision problems, but they also present quite an algorithmic challenge since solving them requires solving a multitude of SAT and UNSAT problems. More specifically, we carry out two series of experiments, one on sum-based optimization problems and another on inclusion minimality-based problems. In the first series, we investigate different optimization strategies, including core- and model-guided strategies as well as the impact of domain heuristics and multi-threading. The second series compares the use of domain heuristics with that of disjunctive logic programs for computing inclusion minimal stable models.

All experiments were run with *clasp* 3.1.2 on a Linux machine with two Intel Quad-Core Xeon E5520 2.27GHz processors, imposing a limit of 600 seconds wall-clock time and 6 GB of memory per run. A timeout is counted as 600 seconds. For capturing not only the successful solution of an optimization process but also its convergence, we regard the quality of solutions too. To be more precise, we extend the scoring used in the 2014 ASP competition by considering runtime whenever two solvers yield the same solution quality (see (iii) below). Let m be the number of participant systems, then the score s of a solver for an instance i in a domain p featuring n instances is computed as $s(p,i) = \frac{m_s(i) \cdot 100}{m \cdot n}$ where $m_s(i)$ is (i) 0, if s does neither provide a solution, nor report unsatisfiability, or (ii) the number of solvers that do not provide a strictly better result than s, where a confirmed optimum solution is considered strictly better than an unconfirmed one. Furthermore, (iii) for two equally good solutions, one is considered strictly better, if it is computed at least 30 seconds faster than the other one.

Accordingly, each entry in Table 1 gives average time, number of timeouts, and score wrt the considered set of instances (except for column *multi*). The benchmark classes are given in the first column, which also includes the number of instances and their source. Also, we indicate via superscripts mn and w, whether a class comprises a *m*ulti-objective optimization problem with n objectives and whether its functions are *w*eight-based. The body of Table 1 gives the results obtained by evaluating *clasp*'s optimization strategies on 636 benchmark instances from various sources.[9] The first three data columns give the results obtained for model-, core-, and heuristic-guided strategies relying on *clasp*'s default configuration `tweety`, viz. plain model-guided optimization (`--opt-strategy=bb`), core-guided optimization using the *oll* algorithm (`--opt-strategy=usc`), and model-guided optimization using heuristics preferring minimized atoms and assigning them to false (`--opt-strategy=bb`

[9] The benchmark set is available at http://www.cs.uni-potsdam.de/clasp/?page=experiments

Benchmark		*model*	*core*	*heuristic*	*model**	*core**	*heuristic**	*multi*
15-puzzle	(16)	260/ 5/ 90	45/ 0/ 100	425/ 9/ 62	266/ 5/ 83	21/ 0/ 100	249/ 5/ 88	9/ 0
Fastfood^{w}	(29)	9/ 0/ 100	290/ 13/ 55	30/ 0/ 100	22/ 0/ 100	290/ 14/ 67	10/ 0/ 100	7/ 0
Labyrinth	(29)	445/ 18/ 75	299/ 11/ 62	365/ 14/ 84	395/ 15/ 79	250/ 10/ 66	442/ 19/ 58	229/ 9
Sokoban	(28)	1/ 0/ 100	1/ 0/ 100	1/ 0/ 100	1/ 0/ 100	1/ 0/ 100	1/ 0/ 100	0/ 0
Tsp^{w}	(29)	600/ 29/ 57	600/ 29/ 0	600/ 29/ 100	600/ 29/ 70	600/ 29/ 32	600/ 29/ 73	600/ 29
Wbds	(29)	600/ 29/ 70	421/ 19/ 34	600/ 29/ 82	600/ 29/ 31	394/ 17/ 67	600/ 29/ 72	397/ 17
Abstract^{m2}	(30)	19/ 0/ 100	99/ 0/ 100	311/ 13/ 57	20/ 0/ 100	73/ 2/ 94	21/ 0/ 100	6/ 0
Connected	(26)	513/ 22/ 75	476/ 20/ 23	513/ 22/ 89	531/ 23/ 52	474/ 20/ 51	514/ 22/ 93	479/ 20
Crossing	(30)	372/ 16/ 78	177/ 5/ 83	451/ 20/ 66	381/ 17/ 61	174/ 6/ 88	367/ 16/ 86	162/ 5
MaxClique	(30)	593/ 29/ 20	50/ 0/ 100	528/ 23/ 61	370/ 13/ 75	23/ 0/ 100	313/ 8/ 91	21/ 0
Valves^{w}	(30)	508/ 24/ 79	543/ 27/ 10	561/ 28/ 7	515/ 25/ 87	561/ 28/ 55	513/ 25/ 92	518/ 25
Aspeed^{m2,w}	(30)	57/ 0/ 100	540/ 27/ 38	490/ 21/ 42	89/ 1/ 99	470/ 23/ 54	64/ 0/ 100	65/ 0
Expansion	(30)	103/ 3/ 92	1/ 0/ 100	40/ 0/ 100	63/ 2/ 96	1/ 0/ 100	30/ 0/ 100	0/ 0
Repair	(30)	113/ 1/ 97	0/ 0/ 100	10/ 0/ 100	32/ 0/ 100	1/ 0/ 100	44/ 0/ 100	1/ 0
Iscas85	(30)	129/ 4/ 96	0/ 0/ 100	158/ 7/ 88	134/ 4/ 92	0/ 0/ 100	306/ 13/ 71	0/ 0
Paranoid^{m2}	(30)	377/ 8/ 79	1/ 0/ 100	103/ 4/ 92	80/ 3/ 94	1/ 0/ 100	59/ 2/ 98	1/ 0
Trendy^{m4,w}	(30)	485/ 19/ 47	4/ 0/ 100	241/ 11/ 80	254/ 11/ 82	6/ 0/ 100	219/ 10/ 87	6/ 0
Metro^{w}	(30)	42/ 0/ 100	237/ 7/ 77	325/ 14/ 59	45/ 0/ 100	162/ 4/ 93	29/ 0/ 100	21/ 0
PartnerUnits	(30)	234/ 5/ 94	111/ 2/ 93	150/ 4/ 87	225/ 8/ 82	103/ 1/ 97	251/ 9/ 83	97/ 0
Ricochet	(30)	86/ 0/ 100	85/ 0/ 100	97/ 0/ 100	167/ 2/ 95	88/ 0/ 100	136/ 1/ 97	21/ 0
ShiftDesign^{m3}	(30)	600/ 30/ 19	23/ 0/ 100	105/ 5/ 86	436/ 16/ 67	44/ 1/ 99	351/ 13/ 80	29/ 0
Timetabling^{w}	(30)	407/ 17/ 63	8/ 0/ 100	205/ 10/ 84	208/ 10/ 84	31/ 1/ 97	280/ 11/ 73	4/ 0
SUM	(636)	6553/259/1731	4011/160/1676	6307/263/1724	5435/213/1829	3768/**156**/1859	5397/212/**1942**	*2674/105*
AVG		298/ 12/ 79	182/ 7/ 76	287/ 12/ 78	247/ 10/ 83	171/ 7/ 85	245/ 10/ 88	122/ 5

Table 1. Results for sum-based optimization

`--dom-mod=5,8`).[10] The starred columns reflect the best configurations obtained for each optimization strategy, viz. model-guided optimization with exponentially increasing steps (`--opt-strategy=bb,2`) using configuration `trendy`, core-guided optimization algorithm *oll*, disjoint core pre-processing, and problem relaxation (cf. Section 3; `--opt-strategy=usc,3`) using `crafty`, and hierarchic model-guided optimization with heuristics preferring to assign false to minimized atoms (cf. Section 4; `--opt-strategy=bb,1 --dom-mod=4,8`) using `trendy`. And the last column shows results obtained with multi-threading. Scores are only computed among single-threaded configurations.

First of all, we observe that core-guided optimization solves the highest number of optimization problems. The fact that *core** solves more problems than *core* is due to `crafty`'s slow restart strategy that is advantageous when solving UNSAT problems, which are numerous in core-guided optimization.[11] While the latter seems to have an edge over the model-guided strategy whenever the optimum can be established, it is vacuous when the optimum is out of reach since it lacks the anytime behavior of model-guided search. This is nicely reflected by the score of 0 obtained by *core* on *TSP*, where no model at all is outputted. The best anytime behavior is obtained by boosting model-guided search with *heuristic*s. Although no variant proves any optimum for *TSP*, the two *heuristic* strategies give the highest scores for *TSP*, reflecting the best solution quality. Interestingly, the less manipulative strategy of *heuristic** yields a better score. In fact, heuristic-guided search procedures show the best convergence to the optimum but often fall short in establishing its optimality. Otherwise, model-guided optimization appears

[10] Combining core-guided optimization with domain heuristics deteriorates results.

[11] In fact, using `trendy` with `crafty`'s restart strategy performs even slightly better.

to benefit from faster restart strategies, as comprised in `trendy`, since it involves solving several SAT problems. All in all, we observe that core-guided strategies dominate in terms of solved problems, while heuristic-guided ones yield the best solution quality. To have the cake and eat it, too, we can take advantage of *clasp*'s multi-threading capacities. To this end, we combined the three starred configurations with one running core-guided optimization with disjoint core pre-processing using `jumpy`.[12] The results are given in column *multi* and reflect a significant edge over each individual configuration. Interestingly, the number of timeouts is less than that of taking the ones of the respective best solver, viz. 106, and also surpasses this virtually best solver (taking 2785 seconds) as regards runtime.[13] In fact, the *multi* configuration performs at least as good as the other configurations on all but two benchmark classes. We trace this superior behavior back to exchanging bounds and constraints among the threads.

Our second series of experiments contrasts the usage of domain heuristics with that of disjunctive logic programs for computing inclusion-minimal stable models. All results are based upon *clasp*'s default configuration `tweety` and given in Table 2. The standard technique for encoding inclusion minimality in ASP is to use saturation-

Benchmark		*meta*	*heuristic*	*meta-heur.*	*meta*	*heuristic*	*meta-heuristic*	*meta-heur.-rec*
15-puzzle	(16)	25/ 0	14/ 0	23/ 0	321/ 7/ 91	408/ 9/ 75	354/ 7/ 69	444/ 9/ 38
Fastfood	(29)	1/ 0	0/ 0	0/ 0	356/ 14/ 59	210/ 9/ 100	348/ 14/ 65	268/ 10/ 71
Labyrinth	(29)	356/ 16	84/ 3	347/ 15	600/ 29/ 72	600/ 29/ 91	600/ 29/ 73	600/ 29/ 61
Sokoban	(28)	22/ 0	1/ 0	12/ 0	22/ 0/ 95	1/ 0/ 100	23/ 0/ 96	12/ 0/ 98
Tsp	(29)	7/ 0	0/ 0	7/ 0	600/ 29/ 48	600/ 29/ 100	600/ 29/ 58	600/ 29/ 44
Wbds	(29)	219/ 7	23/ 1	38/ 1	600/ 29/ 53	600/ 29/ 82	600/ 29/ 72	600/ 29/ 49
Connected	(26)	109/ 3	0/ 0	61/ 2	532/ 23/ 35	532/ 23/ 100	532/ 23/ 60	532/ 23/ 70
Crossing	(30)	98/ 1	14/ 0	14/ 0	600/ 30/ 32	600/ 30/ 99	600/ 30/ 42	600/ 30/ 76
MaxClique	(30)	189/ 3	0/ 0	3/ 0	600/ 30/ 25	600/ 30/ 100	600/ 30/ 50	600/ 30/ 75
Valves	(30)	600/ 30	560/ 28	600/ 30	600/ 30/ 98	560/ 28/ 100	600/ 30/ 98	600/ 30/ 98
Aspeed	(30)	600/ 30	4/ 0	581/ 29	600/ 30/ 73	600/ 30/ 100	600/ 30/ 74	600/ 30/ 75
Expansion	(30)	600/ 30	0/ 0	600/ 30	600/ 30/ 75	298/ 14/ 100	600/ 30/ 75	600/ 30/ 75
Repair	(30)	552/ 26	0/ 0	5/ 0	595/ 29/ 25	438/ 20/ 100	589/ 29/ 50	481/ 21/ 77
Iscas85	(30)	60/ 3	0/ 0	0/ 0	600/ 30/ 25	600/ 30/ 100	600/ 30/ 50	600/ 30/ 75
Paranoid	(30)	191/ 6	1/ 0	16/ 0	600/ 30/ 25	600/ 30/ 100	600/ 30/ 50	600/ 30/ 75
Trendy	(30)	411/ 18	3/ 0	133/ 0	581/ 29/ 27	580/ 29/ 100	581/ 29/ 51	581/ 29/ 75
Metro	(30)	126/ 5	54/ 1	33/ 1	571/ 27/ 42	576/ 28/ 70	581/ 28/ 65	573/ 27/ 78
PartnerUnits	(30)	600/ 30	168/ 4	507/ 9	600/ 30/ 42	168/ 4/ 98	596/ 29/ 61	501/ 9/ 78
Ricochet	(30)	405/ 16	57/ 0	266/ 10	388/ 14/ 46	56/ 0/ 100	285/ 11/ 77	264/ 10/ 83
Timetabling	(30)	600/ 30	16/ 0	85/ 1	600/ 30/ 27	283/ 14/ 98	600/ 30/ 51	336/ 15/ 82
SUM	(576)	5773/254	999/ **37**	3332/ 128	10568/500/1013	8908/**415**/**1913**	10490/497/1285	9991/450/1453
AVG		289/ 13	50/ 2	167/ 6	528/ 25/ 51	445/ 21/ 96	525/ 25/ 64	500/ 22/ 73

Table 2. Results for inclusion minimality-based optimization

based, disjunctive encodings (cf. [12]). We generate the resulting programs automatically from the respective benchmarks with the *metasp* system [31] and solve them with the disjunctive solving techniques described in Section 2. The results are given in the columns headed by *meta*. The first three data columns give average time and number of timeouts for computing one inclusion-minimal answer set. The column headed *heuristic* accomplishes this via the heuristic approach described in Section 4, viz. op-

[12] This provides us with an approximate solution and complements *core** due to fast restarts.

[13] Running the four best configurations from Table 1 yields 2668/108.

tion `--dom-mod=5,16`. We see that this is an order of magnitude better than the disjunctive approach in terms of both runtime and timeouts. Clearly, this is because the former deals with normal programs only, while the latter involves intractable unfounded set tests. Although the frequency of such tests can be reduced by guiding the generating solver by the same heuristics, it fails to catch up with a purely heuristic approach (see column *meta-heur.*). The picture changes slightly when it comes to enumerating inclusion-minimal answer sets. Here, *heuristic* faces an exponential space complexity, while *meta* runs in polynomial space.[14] The remaining columns summarize enumeration results, and add the score as an indicative measure by taking as objective value the number of enumerated models. Although the differences are smaller, *heuristic* still outperforms all variants of *meta*. Again, adding heuristic support improves the performance of *meta*. More surprisingly, the best *meta* configuration is obtained by abolishing polynomial space guarantees and using *clasp*'s solution recording for enumeration, viz. *meta-heur.-rec*. In fact, the added (negated) solutions further focus the search of the generator and thus lead to fewer unfounded set tests.

8 Discussion

We presented distinguishing features of the *clasp* 3 series. And we evaluated their interplay by an empirical analysis on optimization problems. Comparative studies contrasting *clasp* with other systems can be found in the ASP competition series. Many of *clasp*'s features can be found in one form or another in other ASP, SAT, or PB solvers. For instance, *dlv* features several dedicated interfaces, *wasp* [18] also implements core- and model-guided optimization, Rintanen uses heuristics in [32] to improve SAT planning, etc. However, the truly unique aspect of *clasp* 3 is its wide variety of features combined in a single framework. We demonstrated the resulting added value by the combinations in our experiments.

Acknowledgments. This work was funded by AoF (251170) and DFG (SCHA 550/8 and 550/9).

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
3. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM TOCL **7**(3) (2006) 499–562
4. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
5. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning **36**(4) (2006) 345–377
6. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence **187-188** (2012) 52–89

[14] Also, *meta* allows for query-answering, while *heuristic* requires a generate-and-test approach.

7. Alviano, M., Dodaro, C., Ricca, F.: Preliminary report on WASP 2.0. In Proceedings of NMR. (2014)
8. Janhunen, T., Niemelä, I., Sevalnev, M.: Computing stable models via reductions to difference logic. [33] 142–154
9. Liu, G., Janhunen, T., Niemelä, I.: Answer set programming via mixed integer programming. In Proceedings of KR, AAAI Press (2012) 32–42
10. Gebser, M., Kaufmann, B., Schaub, T.: Multi-threaded ASP solving with clasp. Theory and Practice of Logic Programming **12**(4-5) (2012) 525–545
11. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool Publishers (2012)
12. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Annals of Mathematics and Artificial Intelligence **15**(3-4) (1995) 289–323
13. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. Information and Computation **135**(2) (1997) 69–112
14. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS (2009)
15. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In Proc. of KR, AAAI Press (2008) 422–432
16. Gebser, M., Kaufmann, B., Schaub, T.: Advanced conflict-driven disjunctive answer set solving. In Proceedings of IJCAI, IJCAI/AAAI (2013) 912–918
17. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Advanced preprocessing for answer set solving. In Proceedings of ECAI, IOS (2008) 15–19
18. Alviano, M., Dodaro, C., Marques-Silva, J., Ricca, F.: On the implementation of weak constraints in wasp. In Proceedings of ASPOCP. (2014)
19. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-criteria optimization in answer set programming. In Technical Communications of ICLP. LIPIcs (2011) 1–10
20. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In Technical Communications of ICLP. LIPIcs (2012) 212–221
21. Narodytska, N., Bacchus, F.: Maximum satisfiability using core-guided maxsat resolution. In Proceedings AAAI, AAAI Press (2014) 2717–2723
22. Marques-Silva, J., Planes, J.: On using unsatisfiability for solving maximum satisfiability. CoRR **abs/0712.1097** (2007)
23. Gebser, M., Kaufmann, B., Otero, R., Romero, J., Schaub, T., Wanko, P.: Domain-specific heuristics in answer set programming. In Proceedings AAAI, AAAI Press (2013) 350–356
24. Castell, T., Cayrol, C., Cayrol, M., Le Berre, D.: Using the Davis and Putnam procedure for an efficient computation of preferred models. In Proceedings ECAI, Wiley (1996) 350–354
25. Di Rosa, E., Giunchiglia, E., Maratea, M.: Solving satisfiability problems with preferences. Constraints **15**(4) (2010) 485–515
26. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In Proceedings SAT. Springer (2007) 294–299
27. Hutter, F., Hoos, H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In Proceedings LION. Springer (2011) 507–523
28. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: *Clingo* = ASP + control: Preliminary report. In Technical Communications of ICLP. (2014)
29. Ostrowski, M., Schaub, T.: ASP modulo CSP: The clingcon system. Theory and Practice of Logic Programming **12**(4-5) (2012) 485–503
30. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: DLVHEX: A prover for semantic-web reasoning under the answer-set semantics. In Proceedings WI, IEEE (2006) 1073–1074
31. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theory and Practice of Logic Programming **11**(4-5) (2011) 821–839
32. Rintanen, J.: Planning as satisfiability: heuristics. Artificial Intelligence **193** (2012) 45–86
33. Erdem, E., Lin, F., Schaub, T., eds.: Proceedings of LPNMR. Springer (2009)