Unsatisfiability-based optimization in clasp*

Benjamin Andres, Benjamin Kaufmann, Oliver Matheis, and Torsten Schaub

University of Potsdam,
August-Bebel-Str. 89,
D-14482 Potsdam, Germany
{bandres,matheis,torsten}@cs.uni-potsdam.de

Abstract

Answer Set Programming (ASP) features effective optimization capacities based on branch-and-bound algorithms. Unlike this, in the area of Satisfiability Testing (SAT) the finding of minimum unsatisfiable cores was put forward as an alternative approach to solving Maximum Satisfiability (MaxSAT) problems. We explore this alternative approach to optimization in the context of ASP. To this end, we extended the ASP solver *clasp* with optimization components based upon the computation of minimal unsatisfiable cores. The resulting system, *unclasp*, is based on an extension of the well-known algorithms *msu1* and *msu3* and tailored to the characteristics of ASP. We evaluate our system on multi-criteria optimization problems stemming from realistic Linux package configuration problems. In fact, the ASP-based Linux configuration system *aspuncud* relies on *unclasp* and won four out of seven tracks at the 2011 MISC competition.

1998 ACM Subject Classification I.2.8 Problem Solving, Control Methods, and Search

Keywords and phrases answer-set-programming, solvers

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.212

1 Introduction

Answer Set Programming (ASP,[2]) utilizes effective and elaborate optimization techniques based on branch-an-bound algorithms [11]. While these technique have shown to be able solve many problems efficiently, an alternative to this approach emerged in the field of Satisfiability Testing (SAT) for solving Maximum Satisfiability (MaxSAT,[7]) problems. The approach of using unsatisfiable cores has already shown to be successful by the *Maximum Satisfiability with UNsatisfiable COREs* (*MSUnCore*,[10]) solver, being ranged as the best solver in the industrial category in the 2008 MaxSAT evaluation. A closer survey shows that the *MSUnCore* is able to solve some instances that are difficult for the ASP solver *clasp* [5] efficiently. To this end, we propose a new algorithm for solving optimization problems in ASP, combining the *MSUnCore* solving techniques of *msu1* and *msu3* with regard of the characteristics of ASP. The algorithm is extended with techniques for multi-criteria optimization and utilizes the algorithm from [9] for solving weighted optimization problems. The implemented algorithm forms a branch of the *clasp* solver, *unclasp*, specialized for solving unweighted multi-criteria optimization ASP problems. In fact, *aspuncud*, an ASP-based Linux configuration system based on *unclasp*, won four out of seven tracks at the 2011 Mancoosi International Solver Competition (MISC) competition [8]. The *unclasp* solver is available in the lab section of the Potsdam Answer Set Solving Collection [4].

* This work was partially funded by the German Science Foundation (DFG) under grant SCHA 550/8-2.

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 212–221

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The remainder of the paper is structured as follows. The fundamentals of the *MSUnCore* algorithms are introduced in section 2. Section 3 presents our adaptation of the unsatisfiable based *MSUnCore* algorithms for solving ASP optimization, and its implementation into *unclasp* Our experimental results, including instances from MISC and a number of (un)weighted problems, are discussed in section 4. Section 5 concludes the paper.

2 The *MSUnCore* Algorithm

The *MSUnCore* solver features several strategies for solving unweighted, partial MaxSAT problems. Partial MaxSAT is an extension of the SAT problem, in which the number of satisfied clauses of a given subset of the SAT problem is to be maximized, while all other clauses of the problem must still hold. The clauses of the subset are called *soft*, while all other are *hard* clauses. A partial MaxSAT is unweighted if no clause of the subset is favored, and weighted otherwise. To this end, all *MSUnCore* strategies utilize unsatisfiability based approaches. The basic idea behind unsatisfiability based optimization is trying to solve the given problem and to extract an unsatisfiable core if the problem is not satisfiable. An unsatisfiable core is a subset of clauses of the original problem whose conjunction is still unsatisfiable. All *soft* clauses of the extracted core are relaxed, allowing the solver to arbitrarily satisfy one of the clauses. Afterwards, the problem is tried to be solved again with the relaxed clauses. This procedure is iterated until the problem is satisfied or an unsatisfiable core with no *soft* clauses is identified, meaning that the problem is unsatisfiable. Each of the four strategies of MSUnCore, *msu1-4*, implements a different approach in utilizing the identified unsatisfiable core. In the following the *msu1* algorithm is explained as an introduction to unsatisfiability based optimization. Subsequently, the features of the other *MSUnCore* algorithms are presented.

Algorithm 1 presents the pseudo code of the *msu1* strategy. The algorithm consists of a main loop, identifying a solution with the maximum number of satisfied clauses. To this end, the loop iterates through the following steps. First, the problem is passed to a solver. If the solver returns satisfiability, an optimal solution for the problem is found and the loop ends. Otherwise, the reason for unsatisfiability, i.e. an unsatisfiable core, is taken from the solver. Next, all soft clauses of the identified core are relaxed, that is, a unique variable v is added to the clause. Since the newly added relaxation variable is unique, the solver is able to arbitrarily set the variable to true and thus to satisfy the clause. If the identified core does not contain any soft clauses, the core can not be relaxed and the loop ends returning the problem as unsatisfiable, since every solution to the problem must injure at least one clause within the core by definition. Finally, an *at-most-1* constraint is added to the problem to ensure that at most one of the newly added relaxation variables is set to true. If V is the set of newly added relaxation variables, then the *at-most-1* constraint for *msu1* consists of the following clauses:

the clause $\bigvee_{v \in V} v$, and one clause in the form of $\neg v_i \vee \neg v_j$ for any $v_i, v_j \in V$.

The first clause ensures that at least one of the relaxation variables is true, while the others prevent that more than one is true. Thus, $|V|$ variables and $1 + \binom{|V|}{2}$ clauses are added to the problem. Since in every iteration only one additional *soft* clause is satisfied through relaxation, the first valid solution found by the SAT solver is optimal. The number of additional variables and clauses needed is a major disadvantage for the *msu1* algorithm, especially for bigger problems. The *msu2* and the *msu3* algorithms offer two approaches for reducing the number of needed clauses and relaxation variables, respectively.

By introducing additional relaxation variables *msu2* is able to reduce the number of additional clauses to encode the *at-most-1* constraint from $\Theta(|V|^2)$ to $\Theta(|V|)$. Since ASP offers its own efficient encoding of the *at-most-1* constraint, this approach is not further examined. As stated above, *msu1* adds a new relaxation variable for every *soft* clause in an indentified unsatisfiable core. This leads to clauses with many relaxation variables in the case of intersection between cores. Thus, creating several possible combinations for satisfying one set of clauses, obfuscating the solving process. The *msu3* algorithm trades the ability to distinct between identified unsatisfiable cores for a reduced number of relaxation variables. At first, all identified unsatisfiable cores are removed from the problem without substitution, until the problem is satisfiable. While this is not compulsory, it allows to identify disjoint unsatisfiable cores efficiently. Afterwards, the identified cores are relaxed, but instead of adding a relaxation variable each time a clause occurs in an identified core as in *msu1*, *msu3* relaxes each clause only once. Thereby, potentially reducing the number of used relaxation variables. While *msu1-3* use true top-down approaches for finding optimal solutions, *msu4* combines the *msu3* approach with bottom-up search. Instead of allowing only one additional *soft* clause to be relaxed in each iteration, *msu4* states that each subsequent solution must satisfy at least one additional *soft* clause without being relaxed. Thus, approaching the optimal solution from the lower end. All identified unrelaxable cores are treated as in *msu4*.

Algorithm 1 Iterative UNSAT Core Elimination of *msu1*.

$T := \emptyset$ {set of all relaxation variables}
while SAT solver returns UNSATISFIABLE **do**
  LET UC be the UNSAT core provided by the SAT solver
  $S := \emptyset$ {set of new relaxation variables for UC}
  for all Clause $c \in UC$ **do**
    if c is relaxable **then**
      Allocate a new relaxation variable v
      $c := c \cup \{v\}$
      $S := S \cup \{v\}$
    end if
  end for
  if $S = \emptyset$ **then**
    return CNF UNSATISFIABLE
  else
    Add clauses enforcing the *one-hot* constraint for S to the SAT solver
    $T := T \cup S$
  end if
end while
$R := \{v | v \in T, v = 1\}; k := |R|$
return Satisfying Assignment, k, R.

3 Implementation of *unclasp*

The problem of ASP optimization is strongly related to partial MaxSAT. Both problems consist of an unrelaxable rule set and a linear optimization function. While the literals of an ASP optimization rule can be interpreted as distinct soft clauses of a partial MaxSAT problem, the relaxation variables used to solve a partial MaxSAT can be used to form an

optimization rule in ASP. We utilize this correlation in our approach for developing an unsatisfiability based algorithm for ASP optimization. This approach tries to combine the advantages of *msu1* with the practical improvements of *msu3* in regard of the characteristics of ASP. To this end, we create a branch of the ASP solver *clasp*, utilizing its sophisticated ASP solving technique. The resulting system, *unclasp*, is specialized for unweighted, hierarchic optimization problems. Although, able to solve weighted problems as well. The advantages gained by porting the *MSUnCore's* unsatisfiability based approach of solving MaxSAT, to an ASP solving strategy are presented in the following. Afterwards, an extension for solving weighted problems, is explained. Finally, the implementation of our approach into *unclasp* is presented.

When solving ASP optimization problems with the *msu3* approach, the one-literal clauses of an ASP optimization rule offer a distinct advantage over the longer clauses of MaxSAT. Since, in *msu3* each clause is limited to only one relaxation variable, one can interpret the negated literal as its own relaxation variable.

Take, for example, the one-literal clause $\{\neg l\}$. If extended by the relaxation variable v we get $\{\neg l \vee v\}$, which is equivalent $v \leftarrow l$. Since, an optimal solution minimizes the number of true relaxation variables, and v is unique in the problem description, $v \longleftrightarrow l$ follows. Thus, $\{l\}$ is the relaxed clause of $\{\neg l\}$.

With this, *msu3* can be processed for ASP problems without additional variables. The next strong point of ASP is its management of cardinality constraints. Cardinality constraints are satisfied if the number of satisfied literals of the constraint are within a specified range. The ASP solver *clasp* is able to handle the constraint as a single rule simply by counting the number of satisfied literals. *Clasp* only generates the specific clauses for the cardinality constraint when needed for resolution. This is done on demand. This does not only allow to encode *at-most-1* constraints efficiently, but also to formulate *at-most-n* constraints. Being constraints, that enforces the usage of at most n relaxation variables.

In respect to these two characteristics of ASP, our approach works as follows: The problem is given to the solver, and in case of unsatisfiability the core is extracted. All clauses from the optimization rule in the core are relaxed as described above, and an *at-most-1* constraint from the new relaxation variables is formulated as in *msu1*. In difference to *msu1*, the relaxed clauses are marked as *hard* for any successive solving pass. This prevents them to be relaxed a second time. Instead, the new *at-most-1* constraint becomes a soft rule. When a *at-most-n* constraint is encountered in an unsatisfiable core, it is relaxed by an *at-most-(n+1)* rule for the same variables. This algorithm iterates until the problem is solved or an unsatisfiable core without *soft* clauses is encountered.

For dealing with weighted problems the idea from [9] is added to the management of unsatisfiable cores. After the core is identified, each containing clause is split into two. This is done in such a way, that a maximum equal weighted core and a remainder is obtained. Now, the equal weighted core is interpreted as unweighted, while the clauses in the remainder are added to the problem. For example, the weighted core $(a = 3, b = 5, c = 4)$ is split into an equal weighted core $(a = 3, b = 3, c = 3)$ and its remainder $(b = 2, c = 1)$. The equal weighted core is now interpreted as unweighted (a, b, c) and the weighted clauses $b = 2$ and $c = 1$ are added to the problem.

ASP allows to formulate multiple optimization rules and to order them in a hierarchy. Meaning that an optimal solution to a hierarchic optimization problem, has an optimal value in the highest hierarchy level and in all lower levels in an optimal value in respect to the solutions possible for its predecessor levels. This is handled as a sequence of distinct optimizations with the identified optimal values from the higher levels as additional criterion.

The implementation of *unclasp* is based on the *clasp* solver by utilizing a modification of the internal *clasp* function *ClaspFacade::solveIncremental*. This function runs one loop, divided into four phases as shown in figure 1. The phases are surrounded by the control functions *initStep* and *nextStep*, initializing the next pass through the loop and deciding whether another pass is necessary respectively. Figure 2 shows the added interface for implementing our approach.

The loop differentiates in our implementation between the first and all successive run-throughs. In the first pass, the *initStep* function is skipped and the problem is read in the *read* step. During *preprocess* the rules of the problem are translated into an internal representation and the constant *minimizeconstraint_*, holding all literals to be optimized, is formulated by the build in *ProgramBuilder* function. Next, in the *getAssumptions* phase, *assumelevel* is called, extracting the literals of the current optimization level from *minimize-constraint_* and copying them into the *assumptions_* set. This is done to account for literals that influence more than one optimization level. Also, the weights of the minimization literals are copied into the *weightmap_*. In the *solve* step, a solution to the problem with respect to the *assumptions_* is searched for. This leads to three different outcomes determining the behavior of the *nextStep* function. If the problem is unsatisfiable under assumptions, that is, the extracted unsatisfiable core consists at least one literal from *assumptions_*, the loop starts another pass, beginning with *nextStep*. If no literal from *assumptions_* is in the identified core, the problem is unsatisfiable and the incremental solving process is terminated. Finally, in the case of satisfiability, *assumelevel* is called, checking whether another optimization level exists. If no further level in the optimization hierarchy exists, the solution found is optimal and the loop terminates. Otherwise, the *assumptions_* are transformed into facts by the *factifyassumptions* function and *assumelevel* extracts the new *assumptions_* from *minimizeconstraint_*. Then, the optimization continues with another pass through the loop.

In all successive passes the *initStep* function calls the *internal2program* function, which translates the unsatisfiable core from the previous pass back into program variables. Afterwards, *analysecore* is called for managing the translated core. First, the minimal weight of the core's literals is identified and the weight of these literals in the *weightmap* are reduced by this amount. Then an *at-most-1* constraint for the core's literals is formulated by the *addconstraint* function. A guard variable with the previous identified weight allows the relaxation of the variable, i.e. marking it as *soft* with the appropriate weight. The guard variable is saved in *addtoassumptions_* for later usage in the *getAssumptions* phase, when an internal representation for the guard is generated. If a literal of the core is already a guard variable of an *at-most-n* constraint, an appropriate *at-most-(n+1)* variable is added to *addconstraints*. All weights in the *weightmap* that are reduced to 0 are removed from *assumptions_*. The *read* phase is skipped after the first pass and the new constraints from *initStep* are added into the internal representation. In the *getAssumptions* phase the variables of *addtoassumptions_* are mapped to the internal literals with help of the buildin *SolverStrategies::SymbolTable*. The internal literals are then added to *assumptions_*. The *solve* phase and the *nextStep* function behave as described above.

4 Experiments

We evaluate our implementation on optimization instances taken from the MISC and the ASP problem collection asparagus [1]. In order to be able to compare our approach from Section 3 with the *MSUnCore* algorithms, we implemented an ASP optimizing variant of *msu1*, *msu3* and *msu4* in *unclasp*. Thus, *unclasp* is in no need to reduce the number of

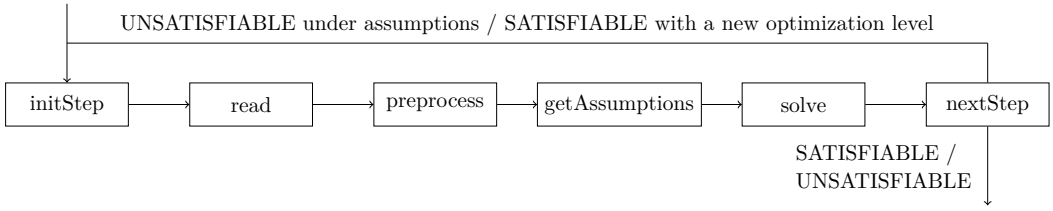

Figure 1 The incremental solving procedure.

Table 1 Runtime parameters of the optimizing strategies used.

new	`unclasp --opt-uncore=oll`
msu1	`unclasp --opt-uncore=msu1`
msu3	`unclasp --opt-uncore=msu3`
msu4	`unclasp --opt-uncore=msu4`
clasp	`clasp-2.0.0 --sat-prepro --restarts=128 --local-restarts --heuristic=VSIDS --solution-recording --opt-hierarch=1 --opt-heu=1`

clauses introduced by the *one-hot* condition followed by *msu2*. To compare the unsatisfiability based approach with the branch-and-bound approach we included the *clasp 2.0* solver into the benchmark. Table 1 presents the runtime parameters used for each strategy. The parameters for *clasp* are specialized for the MISC benchmark set. Below, we report the sequential runtimes on a Linux machine equipped with 3.4 GHz Intel Xeon CPUs and 32 GB RAM. Finally, a timeout was set after 300 seconds.

The MISC benchmark set consists of instances where, given a set of installed and available packages, a solution has to satisfy requests of package addition and removal, while minimizing the effect on the current installation. Packages may depend on or conflict each other, creating a combinational rich unweighted hierarchic optimization problem with a large number of widely independent optimization variables. The huge number of suboptimal solutions necessitate a sophisticated search heuristic and restart policy.

Figure 3 presents a solution cost distribution plot [6] of the runtime measured. The x-axis shows the runtime for solving one instance, while the y-axis labels the percentage of instances solved within the time. The plot shows that for smaller runtimes, up to 10 seconds, all approaches are able to solve a comparable number of instances. On larger runtimes the approaches start to differentiate from each other and a gap emerges. The *msu1* and our new approach clearly dominate the benchmark, able to solve all but two instances in less than 70 seconds, each. Please note, that these two instances were not solved by any approach within the time frame of 300 seconds, demonstrating their complexity. The next two best performing solvers are *msu3* and *msu4* with 10 and 12 unsolved instances, respectively. While being able to solve some of the instances faster than the unsatisfiability based approaches, *clasp* did not solve a higher percentage of MISC instances on any given runtime. In addition *clasp* could not solve 43 instances before reaching time-out. The performance of *msu4* and *clasp* indicates, that the bottom-up strategy used by them is not ideal. The benchmark shows further, that the reduction of relaxation variables pursued by *msu3* is not advantageous for the MISC problem class, overall.

To evaluate the performance of the unsatisfiability based strategies in general and our proposed algorithm in particular, we selected nine general optimization problems from the

```
class UncoreControl: public IncrementalControl {
public:
    virtual void initStep(ClaspFacade& f);
    void getAssumptions(ClaspFacade& f, LitVec& a);
    virtual bool nextStep(ClaspFacade& f);
protected:
    void initassumptions(ClaspFacade& f);
    void assumelevel(ClaspFacade& f);
    void removefromassumptions(const LitVec&);
    void factifyassumptions(ClaspFacade& f);
    virtual void analysecore(ClaspFacade& f, const LitVec&);
    Var addconstraint(ClaspFacade& f, const LitVec&,
        unsigned int bound);
    Literal internal2program(ClaspFacade& f, Literal v);

    const MinimizeConstraint* minimizeconstraint_;
    std::set<Literal> assumptions_;
    VarVec addtoassumptions_;
    Guardtable guardtable_;
    unsigned int level_;
    bool nextlevel_;
    std::vector<unsigned int>min_;
    SolveStats totalstats_;
    std::map<Literal,int> weightmap_;
};
```

Figure 2 Interface for implementing *unclasp* into *clasp*.

asparagus benchmark collection. Most of them were used in the ASP'09 competition [3]. Table 2 shows the selected optimization problems and the corresponding runtime of the optimizing technique. The upper six problems are unweighted, while the lower three are weighted. In the **15-puzzle** and **sokoban** instances the solver has to minimize the number of steps needed to solve the problem. Since all steps build upon each other, there are no combinatorics in the minimization function. Thus, the size of the identified core in every iteration of *unclasp* is one (i.e. the next step). Because of that, the unsatisfiability based algorithms behave the same. The branch-and-bound based algorithm of *clasp* also achieves similar runtime results. The **clique** problem describes the problem of finding the maximal clique in an undirected graph, given. Here, the bottom-up algorithms of *msu4* and *clasp* are clearly superior to the top-down algorithms, as the instances become larger. Of the top-down algorithms, our new approach is able to achieve the best runtime results, while *msu1* could not solve three of the five instances. The other three unweighted problems, **labyrinthpath**, **minimum postage stamp problem** (**mpsp**) and **weight bounded dominating set** (**wbds**) use a large number of optimization variables and also return large unsatisfiable cores. This highlights the different strategies of the unsatisfiability based approaches. In **labyrinthpath** *msu1's* ability to distinguish identified cores is advantageous over *msu3* reduced number of additional variables, as with the MISC benchmark set. In **mpsp** the opposite is the case, and *msu3* outperforms *msu1*, able to solve two more instances. Interestingly, *msu4* performs similar to *msu3*. Our approach achieves the best results on

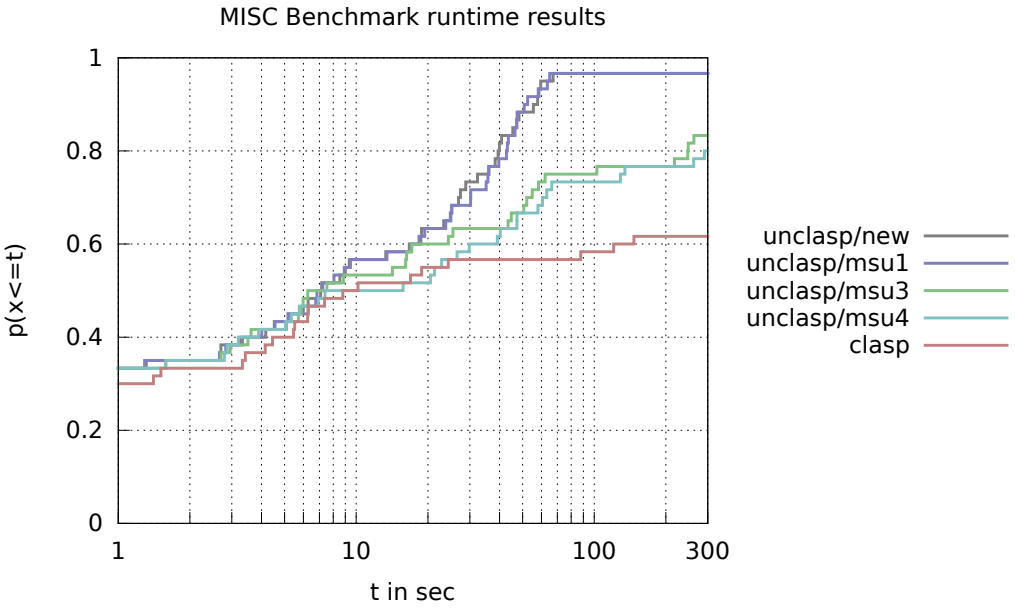

Figure 3 Solution cost distribution plot of the MISC instance runtime.

these three problems. Especially on the **mpsp** and **wbds** instances, where it is able to solve instances every other approach could not. *clasp* is not able to compete with the unsatisfiability based approach with the exception of *msu1* in the **mpsp** problem.

The bottom-up algorithms *msu4* and *clasp* perform well in the weighted problems, **companyctrl**, **opendoors** and **fastfood**, while *clasp* has a better runtime on **opendoors** and *msu4* on **fastfood**. The **companyctrl** problem demonstrates the benefit of the splitting algorithm. While the other approaches are able to solve the problem without much effort, the *msu3* algorithm, which is incompatible with the algorithm from [9], has much difficulty. The **fastfood** problem shows the limit of the top-down unsatisfiability based algorithms. While the problem is easy in general, the cores get too big after a few loop iterations. On this problem class the variable reduction of the *msu3* algorithm shows to be useful, allowing *msu3* to solve three of the five instances tested.

Overall, unsatisfiability based optimization has shown to be efficient in solving unweighted optimization problems. The benchmark demonstrates the effectiveness of our implementation, but also shows its limits on weighted optimization problems.

5 Discussion

We presented an approach to bring unsatisfiabilty based optimization to ASP. Our approach combines the *msu1* and *msu3* strategies of the *MSUnCore* MaxSAT solver with regard to the special characteristics of ASP. The resulting algorithm is specialized for solving unweighted hierarchical optimization problems. In fact, our implementation of the proposed algorithm into *unclasp* was able to to solve a number of problems faster than the traditional branch-and-bound approach utilized by *clasp*. This shows that the *unclasp* approach is a useful addition to the algorithms currently used by *clasp* for solving unweighted problems, expanding *clasp's* portfolio of optimization strategies.

Table 2 Runtime of selected optimization instances from asparagus.

	oll	msu1	msu3	msu4	clasp
15-puzzle					
init1	9,58	9,42	9,53	9,45	6,74
init1simple	0,93	0,93	0,93	0,94	2,02
init1simple2	0,42	0,42	0,42	0,42	1,48
init2	81,84	82,74	82,27	81,81	73,85
init3	13,38	13,28	13,44	13,43	16,81
sokoban					
dimitr_yo51s10	0,24	0,24	0,25	0,25	0,21
dimitr_yo51s14	1,65	1,65	1,66	1,66	1,06
dimitr_yo51s17	2,54	2,55	2,55	2,58	1,59
dimitr_yo52s10	1,67	1,67	1,67	1,67	0,65
dimitr_yo55s10	0,81	0,81	0,81	0,83	0,49
clique					
gen10_25	0,02	0,02	0,02	0,02	0,02
gen200_8000	4,77	300,00	21,35	0,73	0,77
gen300_20000	28,87	300,00	143,45	4,17	3,94
gen75_1000	0,06	8,19	0,12	0,07	0,05
gen100_2000	0,14	300,00	0,51	0,16	0,08
labyrinthpath					
l10_10_01	1,88	1,89	1,88	1,86	18,33
l11_11_01	2,41	1,88	2,39	2,37	300,00
l12_12_01	3,44	4,82	212,90	236,66	300,00
l13_13_01	300,00	300,00	300,00	300,00	300,00
l14_14_01	300,00	300,00	300,00	300,00	300,00
mpsp					
mpsp30-2	0,04	0,13	0,08	0,09	0,05
mpsp36-2	0,12	3,35	0,44	1,02	0,30
mpsp48-2	2,99	300,00	50,90	53,01	116,59
mpsp54-2	1,96	300,00	64,83	36,83	80,59
mpsp60-2	17,14	300,00	300,00	300,00	300,00
wbds					
r100_400_11_1	72,01	300,00	300,00	300,00	300,00
r100_400_11_13	3,40	300,00	300,00	300,00	300,00
r100_400_11_9	4,16	300,00	300,00	300,00	300,00
r150_600_11_17	300,00	300,00	300,00	300,00	300,00
r150_600_11_3	300,00	300,00	300,00	300,00	300,00
companyctrl					
02-company	0,85	0,88	285,46	0,93	0,74
12-company	3,96	4,02	4,52	4,08	3,91
22-company	1,46	1,52	300,00	1,52	1,13
32-company	0,96	0,94	300,00	0,92	0,85
42-company	1,54	1,51	1,76	1,71	1,52
opendoors					
level_00	0,09	0,10	0,09	0,11	0,11
level_05	0,19	0,26	0,19	0,22	0,20
level_10	0,37	0,37	0,37	0,62	0,28
level_17	24,73	6,76	24,17	71,98	4,92
level_28	300,00	300,00	300,00	300,00	18,64
fastfood					
a5.16.dl	300,00	300,00	32,36	12,15	13,02
a5.4.dl	300,00	300,00	300,00	3,84	1,15
fa8.17.dl	300,00	300,00	24,42	8,72	5,92
a8.8.dl	300,00	300,00	300,00	276,82	300,00
a9.11.dl	300,00	300,00	28,94	7,29	6,15

References

1 http://asparagus.cs.uni-potsdam.de.
2 C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
3 M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński. The second answer set programming competition. pages 637–654.
4 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):105–124, 2011.
5 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. pages 260–265.
6 H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.
7 D. Johnson. *Approximation algorithms for combinatorial problems*. Journal of Computer and System Sciences, Academic, 9, 1974.
8 mancoosi. http://www.mancoosi.org.
9 V. Manquinho, J. Marques-Silva, and J. Planes. Algorithms for weighted Boolean optimization. pages 495–508.
10 msuncore. http://www.csi.ucd.ie/staff/jpms/soft/soft.php.
11 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.