Answer Set Solving in Practice

Torsten Schaub
University of Potsdam
torsten@cs.uni-potsdam.de

Potassco

Potassco Slide Packages are licensed under a Creative Commons Attribution 3.0 Unported License.
1 Motivation
2 The asprin framework
3 Preliminaries
4 Language
5 Implementation
6 Summary
Preferences are pervasive

The identification of preferred, or optimal, solutions is often indispensable in real-world applications.

In many cases, this also involves the combination of various qualitative and quantitative preferences.

Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems.

Example

\[ \text{#minimize}\{40 : \text{sauna}, 70 : \text{dive}\} \]
Motivation

- Preferences are pervasive
- The identification of preferred, or optimal, solutions is often indispensable in real-world applications
- In many cases, this also involves the combination of various qualitative and quantitative preferences
- Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems
- Example: \#\text{minimize}\{40 : \text{sauna}, 70 : \text{dive}\}
Preferences are pervasive

The identification of preferred, or optimal, solutions is often indispensable in real-world applications. In many cases, this also involves the combination of various qualitative and quantitative preferences.

Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems.

Example: \( \#\text{minimize}\{40: \text{sauna}, 70: \text{dive}\} \)
Preferences are pervasive

The identification of preferred, or optimal, solutions is often indispensable in real-world applications

In many cases, this also involves the combination of various qualitative and quantitative preferences

Only optimization statements representing objective functions using sum or count aggregates are established components of ASP systems

Example: \#minimize\{40 : sauna, 70 : dive\}
The asprin framework

Outline

1 Motivation
2 The asprin framework
3 Preliminaries
4 Language
5 Implementation
6 Summary
asprin is a framework for handling preferences among the stable models of logic programs

- general because it captures numerous existing approaches to preference from the literature
- flexible because it allows for an easy implementation of new or extended existing approaches

asprin builds upon advanced control capacities for incremental and meta solving, allowing for

ASP solver
redundancies
via an implementation through ordinary ASP encodings

without any modifications to the
significantly reducing
asprin is a framework for handling preferences among the stable models of logic programs

- general because it captures numerous existing approaches to preference from the literature
- flexible because it allows for an easy implementation of new or extended existing approaches

asprin builds upon advanced control capacities for incremental and meta solving, allowing for

- search for specific preferred solutions without any modifications to the ASP solver
- continuous integrated solving process significantly reducing redundancies
- high customizability via an implementation through ordinary ASP encodings
asprin is a framework for handling preferences among the stable models of logic programs

- **general** because it captures numerous existing approaches to preference from the literature
- **flexible** because it allows for an easy implementation of new or extended existing approaches

asprin builds upon advanced control capacities for incremental and meta solving, allowing for

- search for specific preferred solutions without any modifications to the ASP solver
- continuous integrated solving process significantly reducing redundancies
- high customizability via an implementation through ordinary ASP encodings
asprin is a framework for handling preferences among the stable models of logic programs
- general because it captures numerous existing approaches to preference from the literature
- flexible because it allows for an easy implementation of new or extended existing approaches

asprin builds upon advanced control capacities for incremental and meta solving, allowing for
- search for specific preferred solutions without any modifications to the ASP solver
- continuous integrated solving process significantly reducing redundancies
- high customizability via an implementation through ordinary ASP encodings
Example

\#preference(costs, less(weight))\{40 : sauna, 70 : dive\}
\#preference(fun, superset)\{sauna, dive, hike, ∼bunji\}
\#preference(temps, aso)\{dive > sauna || hot, sauna > dive || ¬hot\}
\#preference(all, pareto)\{name(costs), name(fun), name(temps)\}
\#optimize(all)
Preliminaries

Outline

1 Motivation
2 The asprin framework
3 Preliminaries
4 Language
5 Implementation
6 Summary
Preliminaries

Preference

- A strict partial order $\succ$ on the stable models of a logic program. That is, $X \succ Y$ means that $X$ is preferred to $Y$.
- A stable model $X$ is $\succ$-preferred, if there is no other stable model $Y$ such that $Y \succ X$.
- A preference type is a (parametric) class of preference relations.
A strict partial order $\succ$ on the stable models of a logic program. That is, $X \succ Y$ means that $X$ is preferred to $Y$.

A stable model $X$ is $\succ$-preferred, if there is no other stable model $Y$ such that $Y \succ X$.

A preference type is a (parametric) class of preference relations.
A strict partial order \( \succ \) on the stable models of a logic program. That is, \( X \succ Y \) means that \( X \) is preferred to \( Y \).

A stable model \( X \) is \( \succ \)-preferred, if there is no other stable model \( Y \) such that \( Y \succ X \).

A preference type is a (parametric) class of preference relations.
Preliminaries

Preference

- A strict partial order $\succ$ on the stable models of a logic program. That is, $X \succ Y$ means that $X$ is preferred to $Y$.
- A stable model $X$ is $\succ$-preferred, if there is no other stable model $Y$ such that $Y \succ X$.
- A preference type is a (parametric) class of preference relations.
1 Motivation
2 The asprin framework
3 Preliminaries
4 Language
5 Implementation
6 Summary
■ weighted formula \( w_1, \ldots, w_l : \phi \)
where each \( w_i \) is a term and \( \phi \) is a Boolean formula

■ naming atom \( name(s) \)
where \( s \) is the name of a preference

■ preference element \( \Phi_1 > \cdots > \Phi_m \parallel \Phi \)
where each \( \Phi_r \) is a set of weighted formulas and \( \Phi \) is a non-weighted formula

■ preference statement \( \#preference(s, t)\{e_1, \ldots, e_n\} \)
where \( s \) and \( t \) represent the preference statement and its type and each \( e_j \) is a preference element

■ optimization directive \( \#optimize(s) \)
where \( s \) is the name of a preference

■ preference specification is a set \( S \) of preference statements and a directive
  \( \#optimize(s) \) such that \( S \) is an acyclic, closed, and \( s \in S \)
- weighted formula $w_1, \ldots, w_l : \phi$
  where each $w_i$ is a term and $\phi$ is a Boolean formula
- naming atom $\text{name}(s)$
  where $s$ is the name of a preference
- preference element $\Phi_1 > \cdots > \Phi_m \parallel \Phi$
  where each $\Phi_r$ is a set of weighted formulas and $\Phi$ is a non-weighted formula
- preference statement $\#\text{preference}(s, t)\{e_1, \ldots, e_n\}$
  where $s$ and $t$ represent the preference statement and its type
  and each $e_j$ is a preference element
- optimization directive $\#\text{optimize}(s)$
  where $s$ is the name of a preference
- preference specification is a set $S$ of preference statements and a directive
  $\#\text{optimize}(s)$ such that $S$ is an acyclic, closed, and $s \in S$
A weighted formula is defined as
\[ w_1, \ldots, w_l : \phi \]
where each \( w_i \) is a term and \( \phi \) is a Boolean formula.

A naming atom is defined as
\[ name(s) \]
where \( s \) is the name of a preference.

A preference element is defined as
\[ \Phi_1 > \cdots > \Phi_m \parallel \Phi \]
where each \( \Phi_r \) is a set of weighted formulas and \( \Phi \) is a non-weighted formula.

A preference statement is defined as
\[ \#preference(s, t)\{e_1, \ldots, e_n\} \]
where \( s \) and \( t \) represent the preference statement and its type, and each \( e_j \) is a preference element.

An optimization directive is defined as
\[ \#optimize(s) \]
where \( s \) is the name of a preference.

A preference specification is a set \( S \) of preference statements and a directive \( \#optimize(s) \) such that \( S \) is an acyclic, closed, and \( s \in S \)
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms.

The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

Example $\text{less(\text{cardinality})}$

$$(X, Y) \in \text{less(\text{cardinality})(E)}$$

if $|\{l \in E \mid X \models l\}| < |\{l \in E \mid Y \models l\}|$

$\text{dom(less(cardinality))} = \mathcal{P}(\{a, \neg a \mid a \in A\})$

(where $\mathcal{P}(X)$ denotes the power set of $X$)
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms.

The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

**Example** $\text{less}(\text{cardinality})$

$$(X, Y) \in \text{less}(\text{cardinality})(E)$$

if \[ |\{ l \in E \mid X \models l \}| < |\{ l \in E \mid Y \models l \}| \]

$\text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in A\})$

(\text{where } \mathcal{P}(X) \text{ denotes the power set of } X)
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms.

The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

Example $\text{less}(\text{cardinality})$

- $(X, Y) \in \text{less}(\text{cardinality})(E)$ if $|\{l \in E \mid X \models l\}| < |\{l \in E \mid Y \models l\}|$
- $\text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a \mid a \in A\})$
  (where $\mathcal{P}(X)$ denotes the power set of $X$)
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms.

The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

Example $\text{less}($cardinality$)$

- $(X, Y) \in \text{less}($cardinality$)(E)$ if $|\{ l \in E \mid X \models l \}| < |\{ l \in E \mid Y \models l \}|$

- $\text{dom}(\text{less}($cardinality$)) = \mathcal{P}(\{ a, \neg a \mid a \in A \})$
  (where $\mathcal{P}(X)$ denotes the power set of $X$)
A preference type $t$ is a function mapping a set of preference elements, $E$, to a (strict) preference relation, $t(E)$, on sets of atoms. The domain of $t$, $\text{dom}(t)$, fixes its admissible preference elements.

Example $\text{less}(\text{cardinality})$

- $(X, Y) \in \text{less}(\text{cardinality})(E)$ if $|\{l \in E | X \models l\}| < |\{l \in E | Y \models l\}|$
- $\text{dom}(\text{less}(\text{cardinality})) = \mathcal{P}(\{a, \neg a | a \in A\})$
  (where $\mathcal{P}(X)$ denotes the power set of $X$)
More examples

- **more(weight)** is defined as
  - $(X, Y) \in more(weight)(E)$ if $\sum_{(w:l) \in E, X \models l} w \geq \sum_{(w:l) \in E, Y \models l} w$
  - $\text{dom}(more(weight)) = \mathcal{P}\{(w : a, w : \neg a \mid w \in \mathbb{Z}, a \in A)\}$; and

- **subset** is defined as
  - $(X, Y) \in subset(E)$ if $\{l \in E \mid X \models l\} \subset \{l \in E \mid Y \models l\}$
  - $\text{dom}(less(cardinality)) = \mathcal{P}\{(a, \neg a \mid a \in A)\}$.

- **pareto** is defined as
  - $(X, Y) \in pareto(E)$ if $\bigwedge_{name(s) \in E} (X \succeq_s Y) \land \bigvee_{name(s) \in E} (X \succ_s Y)$
  - $\text{dom}(pareto) = \mathcal{P}\{n \mid n \in \mathbb{N}\}$;

- **lexico** is defined as
  - $(X, Y) \in lexico(E)$ if $\bigvee_{w:name(s) \in E} ((X \succ_s Y) \land \bigwedge_{v:name(s') \in E, v < w} (X =_{s'} Y))$
  - $\text{dom}(lexico) = \mathcal{P}\{(w : n \mid w \in \mathbb{Z}, n \in \mathbb{N})\}$. 

Torsten Schaub (KRR@UP)

Answer Set Solving in Practice

October 13, 2016

615 / 661

Potassco
A preference relation is obtained by applying a preference type to an admissible set of preference elements.

\[ \#\text{preference}(s, t)E \] declares preference relation \( t(E) \) denoted by \( \succ_s \)

Example: \( \#\text{preference}(1, \text{less}(\text{cardinality}))\{a, \neg b, c\} \) declares

\[ X \succ_1 Y \text{ as } |\{l \in \{a, \neg b, c\} | X \models l\}| < |\{l \in \{a, \neg b, c\} | Y \models l\}| \]

where \( \succ_1 \) stands for \( \text{less}(\text{cardinality})(\{a, \neg b, c\}) \)
A preference relation is obtained by applying a preference type to an admissible set of preference elements.

#preference(s, t) E declares preference relation t(E) denoted by $\succ_s$

Example #preference(1, less(cardinality)){$a, \neg b, c$}) declares $X \succ_1 Y$ as $|\{l \in \{a, \neg b, c\} \mid X \models l\}| < |\{l \in \{a, \neg b, c\} \mid Y \models l\}|$

where $\succ_1$ stands for less(cardinality)($\{a, \neg b, c\}$)
A preference relation is obtained by applying a preference type to an admissible set of preference elements.

\#preference(s, t) E declares preference relation \( t(E) \) denoted by \( \succ_s \).

Example \#preference(1, less(cardinality))\{a, \neg b, c\} declares

\[ X \succ_1 Y \text{ as } |\{l \in \{a, \neg b, c\} \mid X \models l\}| < |\{l \in \{a, \neg b, c\} \mid Y \models l\}| \]

where \( \succ_1 \) stands for less(cardinality)(\{a, \neg b, c\})
Preference program

- Reification \( H_X = \{\text{holds}(a) \mid a \in X\} \) and \( H'_X = \{\text{holds}'(a) \mid a \in X\} \)

- Preference program Let \( s \) be a preference statement declaring \( \succ_s \)

We define \( P_s \) as a preference program for \( s \), if for all sets \( X, Y \subseteq A \), we have

\[
X \succ_s Y \iff P_s \cup H_X \cup H'_Y \text{ is satisfiable}
\]

- Note \( P_s \) usually consists of an encoding \( E_{t_s} \) of \( t_s \), facts \( F_s \)
  representing the preference statement, and auxiliary rules \( A \)

- Note Dynamic versions of \( H_X \) and \( H_Y \) must be used for optimization
Preference program

- Reification $H_X = \{ holds(a) \mid a \in X \}$ and $H'_X = \{ holds'(a) \mid a \in X \}$

- Preference program Let $s$ be a preference statement declaring $\succ_s$ and let $P_s$ be a logic program.

We define $P_s$ as a preference program for $s$, if for all sets $X, Y \subseteq A$, we have

$$X \succ_s Y \iff P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note $P_s$ usually consists of an encoding $E_{ts}$ of $t_s$, facts $F_s$ representing the preference statement, and auxiliary rules $A$.

- Note Dynamic versions of $H_X$ and $H_Y$ must be used for optimization.
Preference program

- Reification $H_X = \{\text{holds}(a) \mid a \in X\}$ and $H'_X = \{\text{holds}'(a) \mid a \in X\}$

- Preference program Let $s$ be a preference statement declaring $\succ_s$ and let $P_s$ be a logic program.

We define $P_s$ as a preference program for $s$, if for all sets $X, Y \subseteq A$, we have

$$X \succ_s Y \text{ iff } P_s \cup H_X \cup H'_Y \text{ is satisfiable}$$

- Note $P_s$ usually consists of an encoding $E_{t_s}$ of $t_s$, facts $F_s$ representing the preference statement, and auxiliary rules $A$.

- Note Dynamic versions of $H_X$ and $H_Y$ must be used for optimization.
Preference program

- Reification \( H_X = \{ holds(a) \mid a \in X \} \) and \( H'_X = \{ holds'(a) \mid a \in X \} \)

- Preference program Let \( s \) be a preference statement declaring \( \succ_s \) and let \( P_s \) be a logic program. We define \( P_s \) as a preference program for \( s \), if for all sets \( X, Y \subseteq A \), we have

  \[
  X \succ_s Y \iff P_s \cup H_X \cup H'_Y \text{ is satisfiable}
  \]

- Note \( P_s \) usually consists of an encoding \( E_{t_s} \) of \( t_s \), facts \( F_s \) representing the preference statement, and auxiliary rules \( A \)

- Note Dynamic versions of \( H_X \) and \( H_Y \) must be used for optimization
We get a stable model containing \texttt{better(3)} indicating that
\{a, b\} ≻ 3 \{a\}, or \{a\} ⊂ \{a, ¬b\}
# `preference(3, subset){a, ¬b, c}`

\[ E_{\text{subset}} = \begin{cases} 
\text{better}(P) :- \text{preference}(P, \text{subset}), \\
\text{holds}'(X) : \text{preference}(P, _, _, \text{for}(X), _), \text{holds}(X); \\
1 \# \sum \{ 1, X : \text{not holds}(X), \text{holds}'(X), \\
\text{preference}(P, _, _, \text{for}(X), _) \}. 
\end{cases} \]

\[ F_3 = \begin{cases} 
\text{preference}(3, \text{subset}). \quad \text{preference}(3, 1, 1, \text{for}(a), ()). \\
\text{preference}(3, 2, 1, \text{for}(\neg(b)), ()). \\
\text{preference}(3, 3, 1, \text{for}(c), ()). 
\end{cases} \]

\[ A = \begin{cases} 
\text{holds}(\neg(A)) :- \text{not holds}(A), \text{preference}(_, _, _, \text{for}(\neg(A)), _). \\
\text{holds}'(\neg(A)) :- \text{not holds}'(A), \text{preference}(_, _, _, \text{for}(\neg(A)), _). 
\end{cases} \]

\[ H_{\{a, b\}} = \begin{cases} 
\text{holds}(a). \quad \text{holds}(b). 
\end{cases} \]

\[ H'_{\{a\}} = \begin{cases} 
\text{holds}'(a). 
\end{cases} \]

We get a stable model containing \text{better}(3) indicating that \{a, b\} \succ_3 \{a\}, or \{a\} \subset \{a, ¬b\}
Basic algorithm \textit{solveOpt}(P, s)

\begin{itemize}
\item \textbf{Input} : A program \( P \) over \( \mathcal{A} \) and preference statement \( s \)
\item \textbf{Output} : A \( \succ_s \)-preferred stable model of \( P \), if \( P \) is satisfiable, and \( \bot \) otherwise
\end{itemize}

\begin{verbatim}
Y ← solve(P)
if Y = ⊥ then return ⊥
repeat
  X ← Y
  Y ← solve(P ∪ E_t ∪ F_s ∪ R_A ∪ H'_X) ∩ A
until Y = ⊥
return X
\end{verbatim}

where \( R_X = \{ \text{holds}(a) \leftarrow a \mid a \in X \} \)
Sketched Python Implementation

```python
#script (python)

from gringo import *
holds = []

def getHolds():
    global holds
    return holds

def onModel(model):
    global holds
    holds = []
    for a in model.atoms():
        if (a.name() == "_holds"): holds.append(a.args()[0])

def main(prg):
    step = 1
    prg.ground([("base", [])])
    while True:
        if step > 1: prg.ground([("doholds", [step-1]),("preference", [0, step-1])])
        ret = prg.solve(on_model=onModel)
        if ret == SolveResult.UNSAT: break
        step = step+1

#end.

#program base.            #program doholds(m).
#show _holds(X,0) : _holds(X,0).    _holds(X,m) :- X = @getHolds().

#end.
```

Torsten Schaub (KRR@UP)
#script (python)

from gringo import *
holds = []

def getHolds():
    global holds
    return holds

def onModel(model):
    global holds
    holds = []
    for a in model.atoms():
        if (a.name() == "_holds"): holds.append(a.args()[0])

def main(prg):
    step = 1
    prg.ground(["base", []])
    while True:
        if step > 1: prg.ground(["doholds", [step-1]], ["preference", [0, step-1]])
        ret = prg.solve(on_model=onModel)
        if ret == SolveResult.UNSAT: break
        step = step+1

#end.

#program base.
#program doholds(m).
#show _holds(X,0) : _holds(X,0).  _holds(X,m) :- X = @getHolds().

#end.
Vanilla minimize statements

- Emulating the minimize statement

  \[
  \text{#minimize} \{ \text{C,X,Y} : \text{cycle(X,Y)}, \text{cost(X,Y,C)} \}.
  \]

  in \textit{asprin} amounts to

  \[
  \text{#preference(myminimize,less(weight))}
  \{ \text{C,(X,Y)} :: \text{cycle(X,Y)} : \text{cost(X,Y,C)} \}.
  \]

  \text{#optimize(myminimize)}.

- Note \textit{asprin} separates the declaration of preferences from the actual optimization directive
Vanilla minimize statements

- Emulating the `minimize` statement

```
#minimize { C,X,Y : cycle(X,Y), cost(X,Y,C) }.
```

in `asprin` amounts to

```
#preference(mymimimize,less(weight))
   { C,(X,Y) :: cycle(X,Y) : cost(X,Y,C) }.
#optimize(mymimimize).
```

- Note `asprin` separates the declaration of preferences from the actual optimization directive
Example
in asprin's input language

```prolog
#preference(costs,less(weight)){
  C :: sauna : cost(sauna,C);
  C :: dive : cost(dive,C)
}. 
#preference(fun,superset){ sauna; dive; hike; not bunji }. 
#preference(temps,aso){
  dive > sauna || hot;
  sauna > dive || not hot
}. 
#preference(all,pareto){name(costs); name(fun); name(temps)}. 

#optimize(all).
```
asprin’s library

- **Basic preference types**
  - subset and superset
  - less(cardinality) and more(cardinality)
  - less(weight) and more(weight)
  - aso (Answer Set Optimization)
  - poset (Qualitative Preferences)

- **Composite preference types**
  - neg
  - and
  - pareto
  - lexico

- See *Potassco Guide* on how to define further types
asprin’s library

- Basic preference types
  - subset and superset
  - less(cardinality) and more(cardinality)
  - less(weight) and more(weight)
  - aso (Answer Set Optimization)
  - poset (Qualitative Preferences)

- Composite preference types
  - neg
  - and
  - pareto
  - lexico

See Potassco Guide on how to define further types
asprin’s library

- Basic preference types
  - subset and superset
  - less(cardinality) and more(cardinality)
  - less(weight) and more(weight)
  - aso (Answer Set Optimization)
  - poset (Qualitative Preferences)

- Composite preference types
  - neg
  - and
  - pareto
  - lexico

- See Potassco Guide on how to define further types
Outline

1. Motivation
2. The asprin framework
3. Preliminaries
4. Language
5. Implementation
6. Summary
asprin stands for “ASP for Preference handling”
asprin is a general, flexible, and extendable framework for preference handling in ASP
asprin caters to
- off-the-shelf users using the preference relations in asprin’s library
- preference engineers customizing their own preference relations
asprin stands for “ASP for Preference handling”
asprin is a general, flexible, and extendable framework for preference handling in ASP
asprin caters to
off-the-shelf users using the preference relations in asprin’s library
preference engineers customizing their own preference relations
asprin stands for “ASP for Preference handling”

asprin is a general, flexible, and extendable framework for preference handling in ASP

asprin caters to

- off-the-shelf users using the preference relations in asprin’s library
- preference engineers customizing their own preference relations
Summary

Computing answer sets using program completion.  

Knowledge Representation, Reasoning and Declarative Problem Solving.  


Logic programming and knowledge representation.  

Towards an integration of answer set and constraint solving.

*Adaptive restart strategies for conflict driven SAT solvers.*  

*PicoSAT essentials.*  

*Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. 

Torsten Schaub (KRR@UP)  
Answer Set Solving in Practice  
October 13, 2016  
661 / 661
Answer set programming at a glance.

Answer set optimization.

Negation as failure.

*Handbook of Tableau Methods*.


Conflict-driven disjunctive answer set solving.

Heuristics in conflict resolution.

An extensible SAT-solver.
On the computational cost of disjunctive logic programming: Propositional case.

Answer Set Programming: A Primer.  

[22] F. Fages.  
Consistency of Clark’s completion and the existence of stable models.  

Answer sets for propositional theories.


*Mathematical foundations of answer set programming.*


*A Kripke-Kleene semantics for logic programs.*


*Abstract Gringo.*

*Potassco User Guide.*

*A user’s guide to gringo, clasp, clingo, and iclingo.*

*Engineering an incremental ASP solver.*

On the implementation of weight constraint rules in conflict-driven ASP solvers.
In Hill and Warren [49], pages 250–264.


Advanced preprocessing for answer set solving. 
In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, 
Proceedings of the Eighteenth European Conference on Artificial 

The conflict-driven answer set solver clasp: Progress report. 
In E. Erdem, F. Lin, and T. Schaub, editors, Proceedings of the 
Tenth International Conference on Logic Programming and 
Nonmonotonic Reasoning (LPNMR’09), volume 5753 of Lecture 

Solution enumeration for projected Boolean search problems. 
In W. van Hoeve and J. Hooker, editors, Proceedings of the Sixth 
International Conference on Integration of AI and OR Techniques in 
Constraint Programming for Combinatorial Optimization Problems

[38] M. Gebser, M. Ostrowski, and T. Schaub.  
*Constraint answer set solving.*  
In Hill and Warren [49], pages 235–249.

*Tableau calculi for answer set programming.*  

*Generic tableaux for answer set programming.*  
[41] M. Gelfond.
Answer sets.


Logic programming and knowledge representation — the A-Prolog perspective.

The stable model semantics for logic programming.

*Logic programs with classical negation.*

*Answer set programming based on propositional satisfiability.*

*Zum intuitionistischen Aussagenkalkül.*

Die formalen Regeln der intuitionistischen Logik.
In Sitzungsberichte der Preussischen Akademie der Wissenschaften, page 42–56. 1930.
Reprint in Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik, Akademie-Verlag, 1986.


The effect of restarts on the efficiency of clause learning.
In Veloso [74], pages 2318–2323.

Graphs and colorings for answer set programming.

A model-theoretic counterpart of loop formulas.
In L. Kaelbling and A. Saffiotti, editors, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI’05), pages 503–508. Professional Book Center, 2005.

The DLV system for knowledge representation and reasoning.

[54] V. Lifschitz.
Answer set programming and plan generation.

Introduction to answer set programming.

[56] V. Lifschitz and A. Razborov.
Why are there so many loop formulas?
[57] F. Lin and Y. Zhao.


[58] V. Marek and M. Truszczyński.

Nonmonotonic logic: context-dependent reasoning.


Stable models and an alternative logic programming paradigm.


Conflict-driven clause learning SAT solvers.
In Biere et al. [8], chapter 4, pages 131–153.

GRASP: A search algorithm for propositional satisfiability.

Integrating answer set reasoning with constraint solving techniques.

Integrating answer set programming and constraint logic programming.

A SAT solver primer.


Summary


Efficient algorithms for clause-learning SAT solvers.

Extending and implementing the stable model semantics.

[71] T. Son and E. Pontelli.
Planning with preferences using logic programming.


The well-founded semantics for general logic programs.  

[74] M. Veloso, editor.  

 Efficient conflict driven learning in a Boolean satisfiability solver.  