Logical reasoning and programming

Answer set programming

Karel Chvalovský

CIIRC CTU

These slides are mainly based on Eiter 2016 and Gebser et al. 2012.

Types of problem solving

We have seen two possible approaches how to solve a problem:

SAT

Input: a specification of the problem
Output: a *model* of the specification
Weakness: a limited language (no general rules like in Prolog)

Prolog

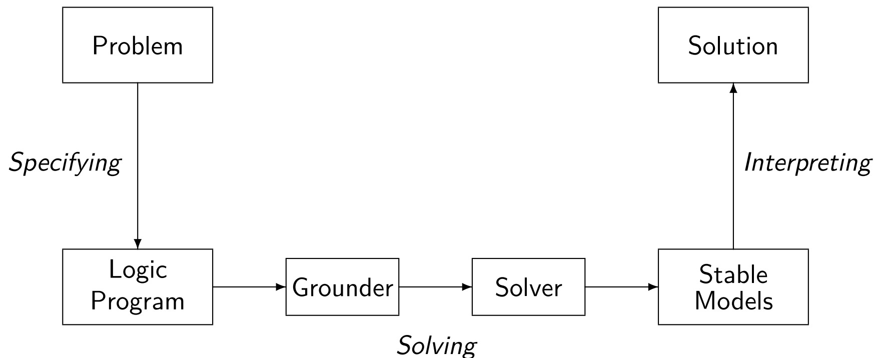
Input: a specification of the problem + query
Output: a *derivation* of the query from the specification
Weakness: not fully declarative (e.g. the order matters)

$$\text{ASP} = \text{LP} + \text{SAT} + \dots$$

Note that logic programming (LP) $\neq$ Prolog.

ASP based problem solving

We combine the expressive power of logic programming with solving similar to SAT.



source: Gebser et al. 2012

Notation

We shall write rules like

$$p(X) \leftarrow q(X), r(X). \tag{1}$$

that rightly resembles

```
p(X) :- q(X), r(X).
```
(2)

in Prolog. However, the order of elements in the body of (1) is not important, unlike in (2), because we want to be fully declarative now. Hence (1) faithfully means

$$q(X) \land r(X) \rightarrow p(X).$$

Non-monotonic reasoning

In classical logic the consequence relation is monotone that is by adding assumptions we can only add consequences not remove them. Sometimes non-monotonic reasoning is useful.

Let P_1 be

$$\begin{aligned}&flies(X) \leftarrow bird(X), \mathrm{not}\, penguin(X).\\&bird(tweety).\end{aligned}$$

where not is a non-standard negation, cf. NAF in Prolog. The meaning of the first rule is that if X is a bird and we do not know that X is a $penguin$, then X $flies$. We use the closed-world assumption (CWA)—if something is true, then we know that. Or in our case, if we do not know something, then it is false. We do not know $penguin(tweety)$ and hence $P_1 \models flies(tweety)$.

Let $P_2 = P_1 \cup \{penguin(tweety).\}$ then $P_2 \not\models flies(tweety)$.

Naïve grounding

We would like to use the language of logic programming (first-order), but also methods developed for SAT (propositional). We can do the naïve grounding—substitute all possible terms:

$$
\begin{array}{lcl}
giant(john). & & giant(john). \\
elf(bob). & & elf(bob). \\
tall(X) \leftarrow giant(X). & & tall(john) \leftarrow giant(john). \\
 & \Rightarrow & tall(bob) \leftarrow giant(bob). \\
small(X) \leftarrow elf(X). & & small(john) \leftarrow elf(john). \\
 & & small(bob) \leftarrow elf(bob).
\end{array}
$$

Both programs are equivalent and ground atoms behave like propositional atoms. From now on, if not otherwise specified, we assume that everything is grounded. Moreover, we assume that we have finite logic programs with no functional symbols (lead to infinite grounding).

Herbrand models

Let P be any logic program and P' be a finite logic program that contains only constants and no functional symbols. Recall that:

The Herbrand universe of P, $HU(P)$, is the set of ground terms in P. Hence $HU(P')$ is the finite set of all constants occurring in P'.

$$\{john, bob\}$$

The Herbrand base of P, $HB(P)$, is the set of ground atoms in P. Hence $HB(P')$ is also finite.

$$\{giant(john), giant(bob), elf(john), elf(bob), \\ tall(john), tall(bob), small(john), small(bob)\}$$

A Herbrand interpretation M of P, $M \subseteq HB(P)$, is a set of ground atoms in P that are considered true.

$$\{giant(john), elf(bob), tall(john), small(bob)\}$$

Positive logic programs

A *positive logic rule* r is of the form

$$h \leftarrow b_1, \ldots, b_n.$$

where h, b_1, ..., b_n are atoms, for $n \geq 0$. We define $H(r) = \{h\}$ and $B(r) = \{b_1, \ldots, b_n\}$.

A *positive logic program* is a finite set of positive logic rules.

Example

$P_1 = \{a \leftarrow b.\ b \leftarrow c.\ c.\}$ and $P_2 = \{a \leftarrow b.\ b \leftarrow a.\ c.\}$ are positive logic programs.

Model

A Herbrand interpretation M of P is a model of (positive) program P, if all rules in P are true in M that is if $r \in P$ and $B(r) \subseteq M$, then $H(r) \cap M \neq \emptyset$.

Example

$M_1 = \{a, b, c\}$ is a model of P_1 and P_2, but $M_2 = \{c\}$ is only a model of P_2.

Minimal model semantics

Logic programs usually have many models and we select the canonical one.

A model M of a program P is *minimal*, if there is no model $M' \subsetneq M$ of P.

Lemma

Every positive logic program P has exactly one minimal model $\min_{\subseteq}(P)$. Hence $\min_{\subseteq}(P) = \bigcap\{\, M \mid M \text{ is a model of } P \,\}$.

Example

For $P_1 = \{a \leftarrow b.\ b \leftarrow c.\ c.\}$ and $P_2 = \{a \leftarrow b.\ b \leftarrow a.\ c.\}$ we have $\min_{\subseteq}(P_1) = \{a, b, c\}$ and $\min_{\subseteq}(P_2) = \{c\}$.

Fixpoints

We can obtain $\min_{\subseteq}(P)$ by the consequence operator T_P which acts on interpretations. For an iterpretation M we define

$$T_P(M) = \{\, H(r) \mid r \in P \text{ and } B(r) \subseteq M \,\}.$$

Let $T_P^0 = \emptyset$ and $T_P^{i+1} = T_P(T_P^i)$, for $i \geq 0$.

Theorem

T_P has a least fixed point, denoted T_P^ω, and the sequence of T_P^i converges to T_P^ω.

Example

For $P_1 = \{a \leftarrow b.\ b \leftarrow c.\ c.\}$ we have $T_{P_1}^0 = \emptyset$, $T_{P_1}^1 = \{c\}$, $T_{P_1}^2 = \{c, b\}$, $T_{P_1}^3 = \{c, b, a\}$, $T_{P_1}^3 = T_{P_1}^4$, and hence $T_{P_1}^\omega = \{c, b, a\}$.

For $P_2 = \{a \leftarrow b.\ b \leftarrow a.\ c.\}$ we have $T_{P_2}^0 = \emptyset$, $T_{P_2}^1 = \{c\}$, $T_{P_2}^2 = T_{P_2}^3$, and hence $T_{P_2}^\omega = \{c\}$.

Negation

In Prolog we have negation as failure, which is different from negation in classical logic.

A *normal logic rule* r is of the form

$$h \leftarrow b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m.$$

where h, b_1, ..., b_n, c_1, ..., c_m are atoms, for $n, m \geq 0$. We call not "negation as failure", "default negation", or "weak negation".
We define $H(r) = \{h\}$, $B(r) = \{b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m\}$, $B(r)^+ = \{b_1, \ldots, b_n\}$, and $B(r)^- = \{c_1, \ldots, c_m\}$.
A *normal logic program* is a finite set of normal logic rules.

Model

A Herbrand interpretation M of P is a model of (normal) program P, if all rules in P are true in M that is if $r \in P$, $B(r)^+ \subseteq M$, and $B(r)^- \cap M = \emptyset$, then $H(r) \cap M \neq \emptyset$.

Semantics for negation

So called "wars of semantics" in logic programming.

$$
\begin{aligned}
P = \{ & man(dilbert). \\
& \quad single(dilbert) \leftarrow man(dilbert), \mathrm{not}\, husband(dilbert). \\
& \quad husband(dilbert) \leftarrow man(dilbert), \mathrm{not}\, single(dilbert). \}
\end{aligned}
$$

What is the correct model of P?

Two well-established approaches:

- single partial model (well-founded semantics)

  $man(dilbert)$ is true,
  $single(dilbert)$ and $husband(dilbert)$ are unknown
- alternative models (stable models, answer sets)

  $M_1 = \{man(dilbert), single(dilbert)\}$,
  $M_2 = \{man(dilbert), husband(dilbert)\}$

Answer sets

We apply the closed world assumption (CWA).

Reduct P^M

Given a program P and a set of atoms M, we define

$$P^M = \{\, H(r) \leftarrow B(r)^+ \mid r \in P \text{ and } B(r)^- \cap M = \emptyset \,\}.$$

In other words we obtain P^M from P by

- deleting every $r \in P$ s.t. $\mathrm{not}\, c_i$ is in its body and $c_i \in M$,
- deleting all negative literals in the bodies of remaining rules.

Note that P^M is a positive logic program and hence has a unique minimal model.

Answer set

A model M is an answer set (or stable model) of P, if M is the minimal model of P^M that is $M = \min_{\subseteq}(P^M)$.

One answer set

$P_1 = \{a \leftarrow a.\ b \leftarrow \mathrm{not}\, a.\}$ has one answer set $\{b\}$.

M	P_1^M	$\min_{\subseteq}(P_1^M)$
$\emptyset$	$\{a \leftarrow a, b \leftarrow\}$	$\{b\}$
$\{a\}$	$\{a \leftarrow a\}$	$\emptyset$
$\{b\}$	$\{a \leftarrow a, b \leftarrow\}$	$\{b\}$
$\{a, b\}$	$\{a \leftarrow a\}$	$\emptyset$

Many answer sets

$P_2 = \{a \leftarrow \text{not}\, b.\ b \leftarrow \text{not}\, a.\}$ has two answer sets $\{a\}$ and $\{b\}$.

M	P_2^M	$\min_{\subseteq}(P_2^M)$
$\emptyset$	$\{a \leftarrow, b \leftarrow\}$	$\{a, b\}$
$\{a\}$	$\{a \leftarrow\}$	$\{a\}$
$\{b\}$	$\{b \leftarrow\}$	$\{b\}$
$\{a, b\}$	$\emptyset$	$\emptyset$

No answer set

$P_3 = \{a \leftarrow \text{not } a.\}$ has no answer set.

M	P_3^M	$\min_{\subseteq}(P_3^M)$
$\emptyset$	$\{a \leftarrow\}$	$\{a\}$
$\{a\}$	$\emptyset$	$\emptyset$

Integrity constraints

Let P be a program and x be a fresh atom in P. A rule

$$x \leftarrow b_1, \ldots, b_n, \text{not } c_1, \ldots, \text{not } c_m, \text{not } x.$$

eliminates all answer sets of P that contain b_1,...,b_n and do not contain c_1,...,c_m. Because x is not important, we write

$$\leftarrow b_1, \ldots, b_n, \text{not } c_1, \ldots, \text{not } c_m.$$

and call such rules integrity constraints.

Complexity of normal logic programs

Theorem

Deciding whether a normal logic program P has some answer is

- NP*-complete if P is grounded (propositional),*
- $\mathrm{NEXPTIME}$*-complete if P is function-free.*

If P is grounded, then we can guess an answer set M of P (in NP). Computing P^M and testing whether $M = \min_{\subseteq}(P^M)$ is polynomial. Grounding can cause an exponential blow up in general and hence, roughly speaking, we are in $\mathrm{NEXPTIME}$.

Disjunctive rules

A *disjunctive logic rule* r is of the form

$$h_1 \mid \cdots \mid h_k \leftarrow b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m.$$

where h_1,...,h_k, b_1, ..., b_n, c_1, ..., c_m are atoms, for $k, n, m \geq 0$. We define $H(r) = \{h_1, \ldots, h_k\}$, $B(r) = \{b_1, \ldots, b_n, \mathrm{not}\, c_1, \ldots, \mathrm{not}\, c_m\}$, $B(r)^+ = \{b_1, \ldots, b_n\}$, and $B(r)^- = \{c_1, \ldots, c_m\}$.
A *disjunctive logic program* is a finite set of disjunctive logic rules.

Model

A Herbrand interpretation M of P is a model of (disjunctive) program P, if all rules in P are true in M that is if $r \in P$, $B(r)^+ \subseteq M$, and $B(r)^- \cap M = \emptyset$, then $H(r) \cap M \neq \emptyset$.

Note that it is no longer possible to compute stable models by a simple iteration as for normal programs.

Complexity of disjunctive logic programs

Theorem

Deciding whether a disjunctive logic program P has some answer is

- $\mathrm{NP}^{\mathrm{NP}}$*-complete if P is grounded (propositional),*
- $\mathrm{NEXPTIME}^{\mathrm{NP}}$*-complete if P is function-free.*

Computing P^M and testing whether $M = \min_{\subseteq}(P^M)$ is polynomial with an NP-oracle; ask the oracle whether $N \subsetneq M$ satisfies P^M.

ASP language

There exists a standard language for ASP called ASP-Core-2 and it is used in the ASP Competition (biannual).

Example

Our

$$p(X) \mid q(X) \leftarrow r(X), \text{not}\, s(X).$$

is

```
p(X) | q(X) :- r(X), not s(X).
```

Choice rules

A rule r

$$\{h_1; \dots; h_k\} \leftarrow b_1, \dots, b_n, \mathrm{not}\, c_1, \dots, \mathrm{not}\, c_m.$$

is called a choice rule. The meaning is that any subset of atoms in the head can be added to a stable model if the body is satisfied.

Example

$P_4 = \{a.\ \{b\} \leftarrow a.\}$ has two stable models $\{a\}$ and $\{a, b\}$.

A choice rule r can be replaced by normal rules

$$h' \leftarrow b_1, \dots, b_n, \mathrm{not}\, c_1, \dots, \mathrm{not}\, c_m.$$
$$h_i \leftarrow h', \mathrm{not}\, \overline{h_i}.$$
$$\overline{h_i} \leftarrow \mathrm{not}\, h_i.$$

for $1 \leq i \leq k$ if we introduce new atoms $h', \overline{h_1}, \dots, \overline{h_k}$. The resulting program has the same stable models if we ignore the newly introduced atoms.

Cardinality constraints

We also have special extended atoms

$$l\{b_1;\ldots;b_n;\mathrm{not}\,c_1;\ldots;\mathrm{not}\,c_m\}u$$

where $l,u \geq 0$. The meaning is that at least l and at most u atoms from $\{b_1,\ldots,b_n,\mathrm{not}\,c_1,\ldots,\mathrm{not}\,c_m\}$ are true in a stable model. We also allow that l or u are missing, meaning there is no corresponding bound. It is possible to use cardinality constraints both in heads and bodies.

Cardinality constraints can be expressed by normal rules, however, the translation is quadratic in space. Also a normal logic rule

$$h \leftarrow b_1,\ldots,b_n,\mathrm{not}\,c_1,\ldots,\mathrm{not}\,c_m.$$

is equivalent to

$$1\{h\} \leftarrow 1\{b_1\},\ldots,1\{b_n\},\{c_1\}0,\ldots,\{c_m\}0.$$

Weight constraints

An extension of cardinality constraints. We add weights to literals and the sum of weights of selected literals is bounded.

$$l\{w_1 : b_1; \ldots, w_n : b_n; w_{n+1} : \mathrm{not}\, c_1; \ldots; w_{n+m} : \mathrm{not}\, c_m\}u$$

where $w_1, \ldots, w_{n+m}$ and l, u are integers.

The possibility to have both positive and negative weights can affect the computational complexity of the problem (by one step in the polynomial time hierarchy).

Example

```
10 { 4:cost(tv) ; 6:ticket(phone) ; 8:cost(pc) } 14
```

Aggregate atoms

Similarly to weight constraints, we have aggregates. An aggregate element has form

$$t_1, \ldots, t_m : \ell_1, \ldots, \ell_n$$

where $t_1, \ldots, t_m$ are terms and $\ell_1, \ldots, \ell_n$ are literals (atoms a_i or $\mathrm{not}\, a_i$).

An aggregate atom has form

$$\#aggr\{e_1; \ldots; e_k\} \prec u$$

where e_i, for $1 \leq i \leq k$, are aggregate elements. $\#aggr$ can be for example $\#\,\mathrm{sum}$, $\#\,\mathrm{min}$, $\#\,\mathrm{max}$, $\#\,\mathrm{count}$ and $\prec$ is a relational symbol. We also allow $u \prec \#aggr\{e_1; \ldots; e_n\}$ and $u_1 \prec \#aggr\{e_1; \ldots; e_n\} \prec_2 u_2$.

Example

```
#sum{1:edge(1,2) ; 2:edge(1,3) ; 1:edge(1,4)} < 3
2 <= #max{ X, Y : p(X,Y) , q(Y,X) , r(X) } < 5
```

Conditional literals

A conditional literal

$$\ell : \ell_1, \ldots, \ell_n$$

is the set of all instances of literal ℓ such that literals ℓ_1,...,ℓ_n hold.

Example

If we have $\{vertex(0).\ vertex(1).\ vertex(2).\}$, then

$$\leftarrow edge(0, X) : vertex(X).$$

expands to

$$\leftarrow edge(0, 0), edge(0, 1), edge(0, 2).$$

However, the exact form of expansion is context dependent.

Weak constraints

We can prefer some models by adding weights and priority levels and ask for best models using a weak constraint

$$:\sim \ell_1, \ldots, \ell_n.\ [w@p, t_1, \ldots, t_m]$$

where w (weight) and p (priority level) are integers and $t_1, \ldots, t_m$ are terms. Priority levels make it possible to use the lexicographic ordering. We attempt to minimize.

Example

```
a | b | c.
:~ a. [1@1]
:~ b. [3@0]
:~ c. [2@1]
```

has the optimal model $\{$b$\}$.

Negations

Classical negation

We can express classical negation by introducing new symbols. Let a be atom, then for $\neg a$ we introduce a new symbol $\overline{a}$ and add

$$\leftarrow a, \overline{a}.$$

In ASP we express the classical negation $\neg a$ as $-a$.

Default negation in heads

By introducing fresh symbols we can also simulate default negations in heads. For example, let $\tilde{a}$ be fresh, then

$$\mathrm{not}\, a \leftarrow b, c.$$

can be expressed by

$$\leftarrow b, c, \mathrm{not}\, \tilde{a}.$$
$$\tilde{a} \leftarrow \mathrm{not}\, a.$$

Consequence relations for answer sets

We can define two natural types of consequence relations:

Brave — an atom a is a brave consequence of P, $P \models_B a$, if $M \models a$ for some stable model M of P,

Cautious — an atom a is a cautious consequence of P, $P \models_C a$, if $M \models a$ for every stable model M of P.

Both $\models_B$ and $\models_C$ are non-monotonic. Maybe surprisingly, for $\models_C$, unlike for $\models_B$, the following stronger result holds:

Lemma

There exists a program P such that $P \models_C a$ and $P \models_C b$, but $P \cup \{a.\} \not\models_C b$.

Take $P = \{a \leftarrow \mathrm{not}\, a.\ a \leftarrow b.\ b \leftarrow \mathrm{not}\, c.\ c \leftarrow \mathrm{not}\, b.\}$. The only stable model of P is $\{a, b\}$, but $P \cup \{a.\}$ has also $\{a, c\}$ as a stable model.

ASP solver

Different methods and approaches, but usually in two separate steps:

Intelligent grounding

Generate as small as possible finite grounding, which need not be a subset of the naïve grounding, that has the same models.

Solving — model search

Given a grounding generate candidate models and test them for stability.

Some solvers try to take advantage of combining these two steps, but it is not very common.

Extensions

ASP solvers usually have lot of extensions for example for external data access (DBs).

Efficient grounding

If we have a program

$$
\begin{aligned}
&p(a, b).\\
&p(b, c).\\
&p(X, Z) \leftarrow p(X, Y), p(Y, Z).
\end{aligned}
$$

then the naïve grounding says that we should substitute all possible combinations of $\{a, b, c\}$ for $\{X, Y, Z\}$. Hence the third rule has $3^3 = 27$ ground instances. However, such ground program is clearly equivalent to

$$
\begin{aligned}
&p(a, b).\\
&p(b, c).\\
&p(a, c) \leftarrow p(a, b), p(b, c).
\end{aligned}
$$

Grounding with functions

If we have a program (with function symbols)

$$q(f(a)).$$
$$p(X) \leftarrow q(X).$$

then the naïve grounding produces infinitely many ground instances, however, it is equivalent to

$$q(f(a)).$$
$$p(f(a)) \leftarrow q(f(a)).$$

On the other hand, if we have a program

$$q(f(a)).$$
$$p(X) \leftarrow \text{not}\, q(X).$$

then we can eliminate only one instance, because the only stable model $\{q(f(a)), p(a), p(f(f(a))), p(f(f(f(a)))), \ldots\}$ is infinite.

Grounding

- grounding is hard
  - in the worst case, grounding time can be exponential
  - selecting “right” rules is difficult
- optimizations
  - literal ordering
  - backjumping
  - magic sets
- arithmetic
  - `p(X+Y) :- q(X), r(Y), X<Y.`
- functional symbols
  - finitely-grounded programs
    - very expressive—correspond to terminating computations of Turing machines
    - but not recognizable
  - finite domain programs
    - all arguments must be finite domain

Solving

Given a ground program, we usually use techniques from modern SAT solvers — CDCL,...

Stable models as classical models

It is possible to express a normal logic program using completion in classical logic. Loosely speaking, we replace $\leftarrow$ by $\leftrightarrow$ (equivalence).

But we have to exclude unsupported loops from stable models by loop formulae. A loop formula forces all atoms in a loop to be false, unless they are externally supported. However, there can be exponentially many loop formulae so we add them on the fly. Moreover, we can learn conflicts separately on completions and loops.

Example

Let $P = \{p \leftarrow q.\ q \leftarrow p, \mathrm{not}\, r.\}$.
$CF(P) = \{p \leftrightarrow q, q \leftrightarrow p \wedge \neg r, r \leftrightarrow \bot\}$. $LF(P) = \{p \wedge q \rightarrow \bot\}$.
$CF(P) \cup LF(P) \equiv \neg p \wedge \neg q \wedge \neg r$.

ASP metodology — TSP data

You can play with the program here.

```
% Nodes
node(1..6).

% Edge Costs
cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).

% (Directed) Edges
edge(X,Y):-cost(X,Y,_).
```

ASP metodology — TSP solver

```
% Generate
{ cycle(X,Y) : edge(X,Y) } = 1 :- node(X).
{ cycle(X,Y) : edge(X,Y) } = 1 :- node(Y).
% Define
reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).
% Test
:- node(Y), not reached(Y).
% Display
#show cycle/2.

% Optimize
:~ cycle(X,Y), cost(X,Y,C). [C,X,Y]
```

TSP solution

```
clingo version 5.3.0
Reading from tsp.lp
Solving...
Answer: 1
cycle(1,4) cycle(4,2) cycle(3,1) cycle(2,6) cycle(6,5) cycle(5,3)
Optimization: 13
Answer: 2
cycle(1,4) cycle(4,2) cycle(3,1) cycle(2,5) cycle(6,3) cycle(5,6)
Optimization: 12
Answer: 3
cycle(1,2) cycle(4,1) cycle(3,4) cycle(2,5) cycle(6,3) cycle(5,6)
Optimization: 11
OPTIMUM FOUND

Models       : 3
  Optimum    : yes
Optimization : 11
Calls        : 1
Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat: 0.00
CPU Time     : 0.001s
```

Bibliography I

Eiter, Thomas (2016). *Answer Set Programming and Extensions*. VTSA Summer School 2016. URL: http://www.kr.tuwien.ac.at/staff/eiter/courses/vtsa16/.

Gebser, Martin et al. (2012). *Answer Set Solving in Practice*. Morgan & Claypool Publishers. ISBN: 1608459713, 9781608459711.