Lecture 3: Normal Programs

Outline

1. Negative conditions
2. Stable model semantics
3. Variables and domains
4. Programming tips
5. Problem solving

1. NEGATIVE CONDITIONS

- The semantics based on least models provides a logical foundation for rule-based reasoning: $P \models a$ iff $a \in \mathrm{LM}(P)$ for an atom a.
- In particular, atoms $a \in \mathrm{Hb}(P)$ that are not logical consequences of P, i.e., $P \not\models a$ holds, are false in $\mathrm{LM}(P)$ *by default*.
- In many applications, it is convenient/necessary to refer to complements of certain relations using negative conditions.
- The notion of answer sets based on *stable models* provides a declarative semantics for programs involving negative conditions.

Example. Consider the following definition of a *conscript*:

$$\text{Conscript}(X) \leftarrow \text{Person}(X), \sim\text{Female}(X).$$

Example

Consider the following set of rules involving negative conditions.

$$\text{Conscript}(x) \leftarrow \text{Person}(x), \sim\text{Female}(x).$$
$$\text{Female}(x) \leftarrow \text{Person}(x), \sim\text{Volunteer}(x), \sim\text{Conscript}(x).$$
$$\text{Person}(\text{joe}).$$

What would be the right answer for the query Conscript(joe)?

- The meaning of the rules depends on the order of application:

$$\text{Person}(\text{joe}), \sim\text{Female}(\text{joe}) \implies \text{Conscript}(\text{joe})$$
$$\text{Person}(\text{joe}), \sim\text{Volunteer}(\text{joe}), \sim\text{Conscript}(\text{joe}) \implies \text{Female}(\text{joe})$$

- Thus it seems non-trivial to combine recursive definitions with negation and, in particular, to obtain a declarative semantics.

2. STABLE MODEL SEMANTICS

- In 1988, Gelfond and Lifschitz proposed stable models in order to provide a declarative semantics for negative conditions in rules.
- The rules of *normal logic programs* are of the form

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m.$$

  where $\sim$ denotes *negation by default*.
- Stable models are based on the following two ideas:
  1. $M \models \sim c$ holds for a negative condition $\sim c \iff c \notin M$, and
  2. a model M is *stable* iff it is the least Herbrand model for the rules having their all negative conditions satisfied by M.

Example

Reconsider the program from the preceding example after grounding:

$$\text{Conscript(joe)} \leftarrow \text{Person(joe)}, \sim\text{Female(joe)}.$$
$$\text{Female(joe)} \leftarrow \text{Person(joe)}, \sim\text{Volunteer(joe)}, \sim\text{Conscript(joe)}.$$
$$\text{Person(joe)}.$$

- The model $M = \{\text{Person(joe)}, \text{Conscript(joe)}\}$ is stable.
- The negative conditions of the first and the last rule are true in M which is the least Herbrand model of the respective positive rules:

  $$\text{Conscript(joe)} \leftarrow \text{Person(joe)}. \quad \text{Person(joe)}.$$
- But $N = \{\text{Person(joe)}, \text{Female(joe)}\}$ is also stable (which suggests us to specify Joe's gender; or to revise the given rules somehow).

Definition of Stability

Definition. Let P be a normal logic program without variables and $M \subseteq \text{Hb}(P)$ an interpretation.

The Gelfond-Lifschitz *reduct* of P with respect to M is

$$P^M = \{a \leftarrow b_1, \ldots, b_n \mid a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \in P \text{ and } M \models \sim c_1, \ldots, \sim c_m\}.$$

Remark. Note that in the definition of P^M,

$$M \models \sim c_1, \ldots, \sim c_m \text{ iff } M \cap \{c_1, \ldots, c_m\} = \emptyset.$$

Definition. Let P be a normal logic program without variables.

An interpretation $M \subseteq \text{Hb}(P)$ is a *stable model* of P iff $M = \text{LM}(P^M)$.

Example

Consider a normal logic program P having the rules listed below:

$$a \leftarrow c, \sim b.$$
$$b \leftarrow \sim a.$$
$$c \leftarrow \sim d.$$
$$d \leftarrow \sim a.$$

1. The interpretation $M_1 = \{a, c\}$ is a stable model of P because $P^{M_1} = \{a \leftarrow c. \;\; c. \;\}$ and M_1 is the least model of P^{M_1}.
2. But $M_2 = \{a, d\}$ is not stable because $P^{M_2} = \{a \leftarrow c. \;\}$ for which the least model is $\emptyset$. Note that $M_2 \models P$ in the classical sense.
3. Finally, $M_3 = \{b, d\}$ is also a stable model of P.

The Γ_P Operator

Definition. Given a normal logic program P, define an operator $\Gamma_P : 2^{\text{Hb}(P)} \rightarrow 2^{\text{Hb}(P)}$ by setting

$$\Gamma_P(M) = \{a \mid a \in \text{Hb}(P) \text{ and } P^M \models a\} = \text{LM}(P^M).$$

Proposition. An interpretation $M \subseteq \text{Hb}(P)$ is a stable model of a normal program P iff $M = \Gamma_P(M)$.

The operator Γ_P is not monotonic but *antimonotonic*:

Proposition. For any normal program P and interpretations $M \subseteq N \subseteq \text{Hb}(P)$, $\Gamma_P(N) \subseteq \Gamma_P(M)$.

Proof. It is sufficient to note that $M \subseteq N$ implies $P^N \subseteq P^M$ and $\text{LM}(P^N) \subseteq \text{LM}(P^M)$ by the monotonicity of $\text{LM}(\cdot)$. □

Properties of Stable Models

- Unlike the least model of a positive program, stable models are not necessarily unique as demonstrated by programs given below:
  1. $P_0 = \{a \leftarrow \sim a. \}$ has no stable models.
  2. $P_1 = \{a \leftarrow \sim b. \}$ has one stable model $\{a\}$.
  3. $P_2 = \{a \leftarrow \sim b.\ \ b \leftarrow \sim a. \}$ has two stable models $\{a\}$ and $\{b\}$.

  ☞ We write $\mathrm{SM}(P)$ for the set of stable models of P.
- Stable models are *minimal* in the sense that if $M \in \mathrm{SM}(P)$ then there is no other $N \in \mathrm{SM}(P)$ such that $N \subset M$.
- A stable model $M \in \mathrm{SM}(P)$ is *strongly grounded* in the rules of P:

$$a \in M \text{ iff } P^M \models a.$$

Answer Set Programming

- A traditional PROLOG system answers a query Q either "yes" (with an answer substitution θ for the variables of Q) or "no".
- Stable models, or *answer sets*, are based on a novel interpretation of logic programs as sets of constraints on their models.
- Typically, an answer set—computed using a special search engine—captures a solution to the problem being solved.
- Rule-based languages are highly expressive:

  Many problems involving constraints can be reformulated as problems of finding a stable model for the respective set of rules.

3. VARIABLES AND DOMAINS

The *ground program* $\mathrm{Gnd}(P)$ is defined for normal logic programs P in the same way as for positive programs.

Definition. Let P be a normal logic program—potentially involving variables—and $\mathrm{Gnd}(P)$ the respective ground program.

A Herbrand interpretation $M \subseteq \mathrm{Hb}(P)$ is a stable model of P iff $M = \Gamma_{\mathrm{Gnd}(P)}(M) = \mathrm{LM}(\mathrm{Gnd}(P)^M)$.

Example. Let us consider $P = \{A(c,d).\ \ B(x) \leftarrow A(x,y), \sim B(y).\ \}$. The ground program $\mathrm{Gnd}(P)$ contains the following rules:

$$\begin{array}{lll} A(c,d). & B(c) \leftarrow A(c,c), \sim B(c). & B(c) \leftarrow A(c,d), \sim B(d). \\ & B(d) \leftarrow A(d,c), \sim B(c). & B(d) \leftarrow A(d,d), \sim B(d). \end{array}$$

The interpretation $M = \{A(c,d),\ B(c)\}$ is the only stable model of P.

Domain Predicates

- Ground programs $\mathrm{Gnd}(P)$ can become very large and they may contain many useless or redundant rules.
- A way to prune unnecessary rules is to introduce *domain predicates* which are relation symbols having a fixed interpretation.
- Even recursive definitions for domain predicates, like $G(\cdot,\cdot)$ below, can be tolerated unless recursion does not involve negation.

Example. Consider the following example:

$$\begin{array}{ll} D(a).\quad E(b). & F(x) \leftarrow D(x).\quad F(x) \leftarrow E(x). \\ G(x,y) \leftarrow D(x), E(y). & G(y,x) \leftarrow G(x,y), F(x), F(y). \\ R(x,y) \leftarrow G(x,y), \sim S(y,x). & S(y,x) \leftarrow G(x,y), \sim R(y,x). \end{array}$$

Here D, E, F, and G are domain predicates but R and S are not.

Example

Some observations about the preceding program, say P, follow:

- The Herbrand universe $\text{Hu}(P) = \{a, b\}$ is finite.
- The least Herbrand model for P' consisting of the first six rules of P is $\text{LM}(\text{Gnd}(P')) = \{D(a), E(b), F(a), F(b), G(a,b), G(b,a)\}$.
- The model $\text{LM}(\text{Gnd}(P'))$ can be represented as a set of facts.
- Only two ground instances of the last two rules each are needed:

$$R(b,a) \leftarrow G(a,b), \sim S(b,a). \qquad R(a,b) \leftarrow G(b,a), \sim S(a,b).$$
$$S(b,a) \leftarrow G(a,b), \sim R(b,a). \qquad S(a,b) \leftarrow G(b,a), \sim R(a,b).$$

- An intelligent grounder can simplify these rules further by dropping conditions $G(a,b)$ and $G(b,a)$ as they are satisfied for sure.

Restricting Domains of Variables

- The idea is to control the size of the resulting ground program by introducing domain predicates that fix the domain of each variable.

Definition. A normal program P is *strongly typed* or *strongly domain restricted* iff for each rule

$$R(\vec{t}) \leftarrow R_1(\vec{t_1}), \ldots, R_n(\vec{t_n}), \sim S_1(\vec{u_1}), \ldots, \sim S_m(\vec{u_m})$$

of P and for each variable x appearing in the rule, x appears in some of the positive conditions $R_i(\vec{t_i})$ where R_i is a domain predicate.

Example. Assuming that $D(\cdot)$ is the only domain predicate, the rule $R(x,y) \leftarrow D(x), D(y), \sim S(y,x)$ is strongly typed, but the rules $F(x,y) \leftarrow D(x), E(x)$ and $E(x) \leftarrow \sim D(x)$ are not.

4. PROGRAMMING TIPS

The logical connectives of propositional logic are available.

- The *conjunction* of conditions $c_1, \ldots, c_n$ is captured by a single (positive) rule $c \leftarrow c_1, \ldots, c_n$.
- Expressing the *disjunction* of conditions $d_1, \ldots, d_n$ requires the introduction of n rules $d \leftarrow d_1. \quad \ldots \quad d \leftarrow d_n$.
- A constraint $\leftarrow b_1, \ldots, b_n$ that formalizes the *negation* $\neg(b_1 \wedge \ldots \wedge b_n)$ is best expressed using a rule $f \leftarrow b_1, \ldots, b_n, \sim f$ where f is a new atom not appearing elsewhere in the program.

Example. One is supposed to have one or two delicacies out of three:

Some ← Cake. Some ← Bun. Some ← Cookie.

All ← Cake, Bun, Cookie. F ← All, ∼F. F ← ∼Some, ∼F.

Making Choices

- A choice between two atoms a and b can be expressed in terms of two normal rules $a \leftarrow \sim b$ and $b \leftarrow \sim a$.
- Such a choice can be generalized for any number of atoms and conditionalized by adding conditions in rule bodies.
- A typical approach in ASP is to express a number of choices and then exclude certain combinations using other rules or constraints.

Example. One is supposed to have coffee or tea—but not both—and also one of three delicacies in case tea is selected:

Coffee ← ∼Tea. Cake ← Tea, ∼Cookie, ∼Bun.

Tea ← ∼Coffee. Bun ← Tea, ∼Cookie, ∼Cake.

Cookie ← Tea, ∼Bun, ∼Cake.

Rules with Exceptions

- Normal programs enable context-dependent reasoning in which the applicability of rules depends dynamically on the context.
- In *common-sense reasoning*, it is typical to formalize the *normal* state of affairs including any *exceptions* to that.

Example. Birds do normally fly—unless we have an exceptional bird.

$$\text{Flies}(x) \leftarrow \text{Bird}(x), \sim\text{Abnormal}(x).$$

$$\text{Abnormal}(x) \leftarrow \text{Penguin}(x). \quad \text{Abnormal}(x) \leftarrow \text{Oily}(x). \quad \ldots$$

The stable models of this program, say P, behave as follows:

1. $\text{SM}(P \cup \{\text{Bird(tw)}.\ \}) = \{\{\text{Bird(tw)}, \text{Flies(tw)}\}\}$.
2. $\text{SM}(P \cup \{\text{Bird(tw)}.\ \text{Oily(tw)}.\ \}) = \{\{\text{Bird(tw)}, \text{Oily(tw)}, \text{Abnormal(tw)}\}\}$.

4. PROBLEM SOLVING

Checking the satisfiability of a propositional theory

A set of clauses S is translated into a normal program P_S as follows:

1. For each atom $a \in \text{Hb}(S)$, we introduce a new atom $\overline{a}$ and two rules $\overline{a} \leftarrow \sim a$ and $a \leftarrow \sim\overline{a}$.
2. Each clause $a_1 \vee \ldots \vee a_n \vee \neg b_1 \vee \ldots \vee \neg b_m$ from S is translated into
$$f \leftarrow \overline{a}_1, \ldots, \overline{a}_n, b_1, \ldots, b_m, \sim f$$
where $f \notin \text{Hb}(S)$ is a new atom.

$\Longrightarrow \text{Hb}(P_S) = \text{Hb}(S) \cup \{\overline{a} \mid a \in \text{Hb}(S)\} \cup \{f\}$.

Proposition. A set of clauses S has a model M, i.e., S is satisfiable, iff the program P_S has a stable model N such that $M = N \cap \text{Hb}(S)$.

Example

Consider the translation of $S = \{a \vee b,\ a \vee \neg b,\ \neg a \vee \neg b\}$ into a normal program. The translation P_S consists of the following rules:

$$a \leftarrow \sim\overline{a}. \qquad \overline{a} \leftarrow \sim a. \qquad b \leftarrow \sim\overline{b}. \qquad \overline{b} \leftarrow \sim b.$$

$$f \leftarrow \overline{a}, \overline{b}, \sim f. \qquad f \leftarrow \overline{a}, b, \sim f. \qquad f \leftarrow a, b, \sim f.$$

A number of observations can be made:

- Now, the set of clauses S has a model M iff the program P_S has a stable model N such that $M = N \cap \{a, b\}$.
- Because $N_1 = \{a, \overline{b}\}$ is a stable model of P_S, we know that $M_1 = \{a\}$ is a model of S.
- On the other hand, $N_2 = \{\overline{a}, \overline{b}\}$ is not a stable model of P_S.

Graph 3-Coloring

A graph G can be represented by facts of the form "$\text{Edge}(x,y)$." where x and y stand for nodes. The following normal program P_G^{3c} is a *uniform encoding* for the problem of coloring the nodes of G with three colors so that the endpoints of each edge have different colors.

$\text{Node}(x) \leftarrow \text{Edge}(x,y).\quad \text{Node}(y) \leftarrow \text{Edge}(x,y).$	(projection)
$\text{Black}(x) \leftarrow \text{Node}(x), \sim\text{White}(x), \sim\text{Grey}(x).$	(choices)
$\text{White}(x) \leftarrow \text{Node}(x), \sim\text{Black}(x), \sim\text{Grey}(x).$	
$\text{Grey}(x) \leftarrow \text{Node}(x), \sim\text{White}(x), \sim\text{Black}(x).$	
$\text{F} \leftarrow \text{Edge}(x,y), \text{Black}(x), \text{Black}(y), \sim\text{F}.$	(constraints)
$\text{F} \leftarrow \text{Edge}(x,y), \text{White}(x), \text{White}(y), \sim\text{F}.$	
$\text{F} \leftarrow \text{Edge}(x,y), \text{Grey}(x), \text{Grey}(y), \sim\text{F}.$	

Proposition. The graph G has a 3-coloring iff P_G^{3c} has a stable model.

Hamiltonian Cycles in Graphs

The problem is to check whether a given graph has a Hamiltonian cycle which visits all nodes of the graph exactly once. In addition to the edge relation, the following rules are introduced in program P_G^{H}.

1. The nodes of the graph are extracted from the edge relation:

   Node(x) ← Edge(x,y). Node(y) ← Edge(x,y). Same(x,x) ← Node(x).

2. Any cycle starts from a particular node chosen here.

   Start(x) ← Node(x), ∼Other(x).
   Other(x) ← Node(x), ∼Start(x).
   F ← Start(x), Start(y), ∼Same(x,y), Node(x), Node(y), ∼F.
   HasStart ← Start(x), Node(x).
   F ← ∼HasStart, ∼F.

3. Next the edges which are on the cycle are chosen.

   In($x1$,$x2$) ← Edge($x1$,$x2$), ∼Out($x1$,$x2$).
   Out($x1$,$x3$) ← In($x1$,$x2$), ∼Same($x2$,$x3$), Edge($x1$,$x2$), Edge($x1$,$x3$).
   Out($x3$,$x2$) ← In($x1$,$x2$), ∼Same($x2$,$x3$), Edge($x1$,$x2$), Edge($x3$,$x2$).

4. All nodes of the graph must be reachable via the cycle.

   Reached(x) ← Start(x).
   Reached(x) ← In(y,x), Reached(y), Edge(y,x).
   F ← Node(x), ∼Reached(x), ∼F.

Proposition. The program P_G^{H}—together with facts that describe the edge relation—has a stable model $\iff$ G has a Hamiltonian cycle.

OBJECTIVES

- You know what kind of problems arise when negative conditions are incorporated into recursive definitions.
- You are able to reproduce the definition of stable models and to prove simple properties about them.
- You can calculate stable models for simple normal logic programs (at least by exhaustive generation of model candidates).
- You are able to formalize simple constraint programming problems by describing their solutions in terms of rules.

TIME TO PONDER

As demonstrated above, a normal logic program can have several stable models, a unique stable model, or no stable models at all.

Problem. Design a propositional normal program P_n that has exactly $n \geq 0$ stable models.

How does the *length* of P_n, measured in the number of atoms and connectives, change as the function of n?