CSE 459/598, Spring 2010

8. Strong Negation and Planning

In ASP, a second kind of negation called *strong negation* turned out to be useful, in particular, for solving computational problems related to actions and planning.

Formulas with Strong Negation

Recall that propositional formulas are formed from atoms and the 0-place connective $\bot$ using the binary connectives $\land$, $\lor$ and $\to$ (Handout 1). In this section we assume that formulas may contain atoms of two kinds, *positive* and *negative*, and that each negative atom is an expression of the form $\sim\!A$, where A is a positive atom. The symbol $\sim$ is called *strong negation*.

Note that syntactically strong negation is not really a connective, according to this definition: it is allowed to occur in front of positive atoms only. For example, expressions $\sim\sim p$ and $\sim(p \land q)$ are not formulas.

A set of atoms is *coherent* if it does not contain "complementary" pairs of atoms A, $\sim\!A$.

Consider, for instance, the formula

$$(p \lor \neg p) \land q \land (\neg p \to \sim q). \tag{1}$$

It contains two positive atoms p, q and one negative atom $\sim\!q$. The answer sets of this formula are $\{p, q\}$ and $\{q, \sim q\}$. The first of them is coherent, and the second is not.

The coherent answer sets of (1) are identical to the answer sets of the formula

$$(p \lor \neg p) \land q \land (\neg p \to \sim q) \land \neg(q \land \sim q).$$

In logic programming notation, the last conjunctive term can be written as

$$\leftarrow q, \sim q. \tag{2}$$

In answer set programming with strong negation, the objects we would like to find are represented as the coherent answer sets of a program. In the input language of GRINGO, strong negation is written as `-`. The answer set solvers that accept input programs with strong negation, such as CLASP, generate coherent answer sets only, so that there is no need to include constraints (2) in their input.

8.1^e Use CLASP to compute the coherent answer set of (1).

Fluents and Actions

Some computational problems in artificial intelligence can be viewed as questions about transition systems of the kind familiar from the theory of finite automata. The states of such a system are characterized by the values of certain parameters, or *fluents*. We assume here that all these parameters are truth-valued. A change in the state of the system is caused by executing an *action*, or, more generally, by an *event*—a set of actions executed concurrently.

Consider, for instance, the system consisting of two devices d_1 and d_2, controlled by two switches a and b. The system is always in one of 4 possible states, characterized by the values of the fluents $on(d_1)$ and $on(d_2)$. The available actions are $toggle(a)$ and $toggle(b)$, so that 4 events are possible:

$$\emptyset,\ \{toggle(a)\},\ \{toggle(b)\},\ \{toggle(a), toggle(b)\}.$$

Assume that switch a controls device d_1, and switch b controls device d_2. Under this assumption, if each of the fluents $on(d_1)$, $on(d_2)$ is currently true (both devices are on) then the event $\{toggle(a)\}$ (toggling switch a) will make $on(d_1)$ false; the fluent $on(d_2)$ will remain true. Geometrically speaking, the directed graph corresponding to this transition system has an edge that

- starts in the state in which both $on(d_1)$ and $on(d_2)$ are true,
- is labeled $\{toggle(a)\}$, and
- leads to the state in which $on(d_1)$ is false and $on(d_2)$ is true.

The event $\emptyset$ ("doing nothing") does not change the value of either fluent. Geometrically speaking, the corresponding graph has 4 self-loops labeled $\emptyset$.

Some of the assertions about the effects of actions in this example depend on the implicit assumption that the values of fluents do not change without a cause, or, in other words, on the principle that in the absence of information to the contrary the values of fluents after an event are assumed to be the same as before.[1] This somewhat vague idea is called the *commonsense law of inertia*, and the problem of making it precise is known as the *frame problem*.

Describing Transition Systems by Logic Programs

The system of two devices and two switches described above can be represented by a logic program with strong negation using

[1] "Everything is presumed to remain in the state in which it is" (Leibniz's note in the margin of his *Introduction to a Secret Encyclopædia*, 1679).

- the atoms
$$on(x),\ \sim on(x)$$
($x \in \{d_1, d_2\}$), to characterize the initial value of the fluent $on(x)$;
- the atoms
$$toggle(s)$$
($s \in \{a, b\}$), to characterize the event under consideration;
- the atoms
$$on'(x),\ \sim on'(x)$$
($x \in \{d_1, d_2\}$), to characterize the value of the fluent $on(x)$ after the event;
- the atoms
$$controls(s, x)$$
($s \in \{a, b\}$, $x \in \{d_1, d_2\}$), to express that switch s controls device x.

We would like to write a program whose answer sets correspond to all transitions possible in this system, that is to say, to all edges of the corresponding directed graph. For instance, one of the answer sets of this program will consist of the atoms

$$\begin{gathered} controls(a, d_1),\ controls(b, d_2), \\ on(d_1),\ on(d_2), \\ toggle(a), \\ \sim on'(d_1),\ on'(d_2). \end{gathered}$$

The program consists of the following rules. The initial values of the fluents $on(x)$ can be chosen arbitrarily:

$$1 \leq \{on(x), \sim on(x)\}^c.$$

Whether or not to execute any of the actions $toggle(s)$ can be decided arbitrarily too:

$$\{toggle(s)\}.$$

Executing the action $toggle(s)$ reverses the value of the fluent $on(x)$, where x is the device controlled by switch s:

$$\begin{gathered} \sim on'(x) \leftarrow on(x), toggle(s), controls(s, x), \\ on'(x) \leftarrow \sim on(x), toggle(s), controls(s, x). \end{gathered}$$

In the absence of information to the contrary, the fluent $on(x)$ is assumed to remain true if it was true initially:

$$on'(x) \leftarrow on(x), not \sim on'(x).$$

Note the term $not \sim on'(x)$, which combines negation as failure (not) with strong negation ($\sim$). It translates the expression "in the absence of information to the contrary" into the language of logic programming: it says that it is impossible to use the rules of the program to establish $\sim on'(x)$, that is, to refute the formula in the head of the rule. Similarly, in the absence of information to the contrary, the fluent $on(x)$ is assumed to remain false if it was false initially:

$$\sim on'(x) \leftarrow \sim on(x), not\ on'(x).$$

These two rules show how to solve the frame problem in the language of answer set programming with strong negation. Finally, switch a controls d_1, and switch b controls d_2:

$$\begin{array}{l} controls(a, d_1), \\ controls(b, d_2). \end{array}$$

8.2^e How many answer sets do you think this program has? Use CLASP to verify your conjecture.

Prediction, Postdiction and Planning

We have seen how answer set programming with strong negation can be used to characterize the transitions $\langle s, e, s' \rangle$ of a transition system. Many questions that we may wish to ask about a transition system are not about individual transitions; they have to do with "histories" of the transition system, that is, paths

$$\langle s_0, e_0, s_1, e_1, \dots, e_{m-1}, s_m \rangle$$

in the corresponding directed graph. Here $s_0, \dots, s_m$ are successive states of the system, and e_i is the event leading from s_i to s_{i+1}. Transitions are histories of length $m = 1$.

For example, a *prediction* problem is a question about properties of the outcome s_m of a given sequence of events $e_0, \dots, e_m$ under some assumptions about the initial state s_0. A *postdiction* problem is a question about properties of the initial state s_0 under some assumptions about the outcome s_m

of a given sequence of events $e_0, \ldots, e_m$ that have led from s_0 to s_m. In a *planning* problem, we want to find a sequence $e_0, \ldots, e_m$ of events that would lead from a given initial state to a final state satisfying a given goal condition $G(s)$.

It is easy to generalize the program from the previous section to arbitrary values of m. We use the atoms $on(x,i)$ and $\sim on(x,i)$, where $x \in \{d_1, d_2\}$ and $0 \leq i \leq m$, to characterize the value of fluent $on(x)$ in state s_i, and the atoms $toggle(s,i)$, where $s \in \{a, b\}$ and $0 \leq i < m$, to express that event e_i includes action $toggle(s)$. The program consists of the following rules:

$$\begin{array}{l}
1 \leq \{on(x,0), \sim on(x,0)\}^c \\
\{toggle(s,i)\} \\
\sim on(x,i+1) \leftarrow on(x,i), toggle(s,i), controls(s,x) \\
on(x,i+1) \leftarrow \sim on(x,i), toggle(s,i), controls(s,x) \\
on(x,i+1) \leftarrow on(x,i), not\ \sim on(x,i+1) \\
\sim on(x,i+1) \leftarrow \sim on(x,i), not\, on(x,i+1) \\
controls(a, d_1) \\
controls(b, d_2)
\end{array}$$

($x \in \{d_1, d_2\}$, $s \in \{a, b\}$, $0 \leq i < m$).

8.3 (a) The devices d_1 and d_2 are currently on. If we toggle switch a and then toggle switches a and b simultaneously, what can you say about the resulting state? Instruct CLASP to answer this question. (b) We toggled switch a, and then toggled switches a and b simultaneously. In the resulting state the devices d_1 and d_2 are on. What can you say about the initial state? Instruct CLASP to answer this question.

8.4 The devices d_1 and d_2 are currently on, and we would like both of them to be off. This goal is to be achieved without performing more than one action at a time. Instruct CLASP to find all solutions that have the minimal possible length.

8.5 Consider the system consisting of a single individual who at any point in time can be either *alive* or not, and a single gun that at any point in time can be either *loaded* or not. Initially the person is alive and the gun is not loaded. Then the action *load* is executed, and, after a wait, the action *shoot*. Instruct CLASP to determine the outcome of this sequence of events.