Disjunction (1) — ‘shifting’

We have seen that the disjunction ‘A or B’ (A and B literals) can be approximated by means of a construct sometimes called ‘shifting’.

Example

\[
\begin{align*}
a &\leftarrow \neg b \\
b &\leftarrow \neg a \\
a &\leftarrow \neg b \\
b &\leftarrow \neg a \\
a &\leftarrow \neg b \\
b &\leftarrow \neg a
\end{align*}
\]

has answer sets \{a\}, \{b\}

Why ‘approximation’?

\[
\begin{align*}
a &\leftarrow \neg b \\
b &\leftarrow \neg a \\
a &\leftarrow \neg b \\
b &\leftarrow \neg a
\end{align*}
\]

has one answer set \{a\}

Fine. But consider:

\[
\begin{align*}
a &\leftarrow \neg b \\
b &\leftarrow \neg a \\
a &\leftarrow \neg b \\
b &\leftarrow \neg a \\
a &\leftarrow \neg b \\
b &\leftarrow a
\end{align*}
\]

has no answer sets!! (Check for yourself.)

‘Shifting’ is a good encoding of disjunction in certain circumstances — for instance, when the disjuncts are mutually exclusive.

Disjunction (2) — ‘Choice rules’

As we have seen (in the notes on ASP solvers) the disjunction ‘A or B’ (A and B literals) can also be expressed by means of a ‘choice rule’ (choose any subset) together with an integrity constraint (but not the empty subset), or using a cardinality constraint:

\[
1 \{A, B\} \leftarrow
\]

And as we have seen, the choice rule can be expressed as an extended logic program, as follows.

Example:

\[
\begin{align*}
a &\leftarrow \neg a' \\
a' &\leftarrow \neg a \\
b &\leftarrow \neg b' \\
b' &\leftarrow \neg b \\
f &\leftarrow \neg a, \neg b, \neg f
\end{align*}
\]

has answer sets \{a, b\}, \{a', b\}, \{a, b\}

\[
\begin{align*}
a &\leftarrow \neg a' \\
a' &\leftarrow \neg a \\
b &\leftarrow \neg b' \\
b' &\leftarrow \neg b \\
f &\leftarrow \neg a, \neg b, \neg f \\
a &\leftarrow b
\end{align*}
\]

has answer sets \{a, b\}, \{a\}

\[
\begin{align*}
a &\leftarrow \neg a' \\
a' &\leftarrow \neg a \\
b &\leftarrow \neg b' \\
b' &\leftarrow \neg b \\
f &\leftarrow \neg a, \neg b, \neg f \\
a &\leftarrow b \\
b &\leftarrow a
\end{align*}
\]

has answer sets \{a, b\}
Disjunctive logic programs

This part is for BACKGROUND INTEREST only. Details are not examinable. I have included it because you will see references to disjunctive logic programs, e.g. in the guide to clingo.

Clauses are as in extended logic programs or of the form

\[ A ; B \leftarrow L_1, \ldots, L_m \]

A and B are literals: i.e., either of the form \( p \) or \( \neg p \) where \( p \) is an atom. \( L_i \) are literals or nbf-literals as usual, i.e., of the form \( p \), \( \neg p \), not \( p \) or not \( \neg p \) where \( p \) is an atom.

\( A ; B \) is sometimes written \( A \mid B \). You can read it as the disjunction ‘A or B’ — except that disjunction in disjunctive logic programs has a particular special meaning that comes out because of its minimality semantics.

Answer sets are defined as usual, in terms of reducts. Given a disjunctive logic program \( P \) and a set of literals \( X \), the reduct \( P^X \) is defined in the usual way: delete from \( P \) any clause which has a literal not \( B \) in its body where \( B \in X \); delete all other literals not \( C \) in the remaining clauses (i.e., those conditions not \( C \) which are satisfied in \( X \), i.e., where \( C \notin X \)).

The reduct \( P^X \) contains no occurrences of negation-by-failure. We have the usual ‘stability requirement’: \( X \) is an answer set of \( P \) if \( X \) is an answer set of \( P^X \). The difference is that now there is the possibility of disjunction in clauses, the answer set of \( P^X \) might not be unique (unlike for extended logic programs, where the reduct is a set of definite clauses, and the answer set – the least Herbrand model – is unique.)

So what is the answer set of disjunctive logic program without negation-by-failure, i.e., of a set of definite clauses or clauses of the form

\[ A ; B \leftarrow L_1, \ldots, L_m \]

where \( A, B, L_1, \ldots, L_m \) are all literals (no negation-by-failure).

Let \( X \) be a set of literals. \( X \) satisfies a clause of the form (1) if whenever \( \{L_1, \ldots, L_m\} \subseteq X \) then either \( A \in X \) or \( B \in X \) (or both). Just as you would expect.

Now, by definition, \( X \) is an answer set of a disjunctive logic program \( P \) without negation-by-failure if \( X \) is a minimal set of literals that satisfies all the clauses of \( P \). (Or as we also say, when \( X \) is a minimal Herbrand model of \( P \).

All this seems very natural. But look at some simple examples. (Negation-by-failure works the same way—via reducts—as in extended logic programs.)

Simplest example:

\[
\begin{align*}
\{a ; b \leftarrow \} & \quad \text{models: } \{a\}, \{b\}, \{a, b\} \\
\{a ; b \leftarrow \} & \quad \text{answer sets (minimal): } \{a\}, \{b\} \\
\{a \leftarrow b \} & \quad \text{models: } \{a\}, \{a, b\} \\
\{a \leftarrow b \} & \quad \text{answer sets (minimal): } \{a\} \\
\{a ; b \leftarrow \} & \quad \text{models: } \{a\} \\
\{a \leftarrow b \} & \quad \text{models: } \{a, b\} \\
\{b \leftarrow a \} & \quad \text{models: } \{a, b\} \\
\end{align*}
\]

clingo, and many other ASP solvers, do not support disjunctive logic programs. (The computational complexity is much higher, because of the built-in minimality feature.)

And it is not clear how useful this reading of disjunction is for knowledge representation anyway. (It is useful for some algorithmic tasks.)