Choices

- In many applications, it becomes useful to choose or pick domain values, possibly depending on other values.
- For that, we can introduce a new “choice” construct or operator.
- So as with PCs, it will be possible to eliminate it (by defining it) if wanted (coming).
  But it may be useful to have it as given.
- A program $\Pi$ may have a “choice rule” with a “choice operator”:

\[
P(x, y, z) \leftarrow Q(\ldots, x, \ldots), S(\ldots, y, \ldots), R(\ldots, z, \ldots), \text{Choice}((x, y), z)
\]

For each combination of values $(x, y)$, non-deterministically choose a unique value for $z$ and put the trio $(x, y, z)$ into $P$ (while satisfying the other conditions in the body).
Example: Consider the relational table that does not satisfy the functional dependency \( R : AB \rightarrow C \)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>c</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

Repairing \( R \): (by tuple deletion)

\[
R'(x, y, z) \leftarrow R(x, y, z), \text{choice}((x, y), z).
\]

\( R(a, b, c) \). Etc.

Two stable models (two repairs):

\[
\{ R'(a, b, c), R'(a, c, e), R(a, b, c), \ldots \}
\]

\[
\{ R'(a, b, d), R'(a, c, e), R(a, b, c), \ldots \}
\]

Different choices for \( z \) appear in different stable models

Use of choice rules increases number of SMs

No negation in this example?

Or in the rule (*) above?
Negation, actually unstratified, is implicit in choice rules. It will reappear when we define the “choice operator” by means of regular rules.

So, no essential need for the *choice* operator (but nice, intuitive and useful having it!)

Replace choice rule (*) by:

\[
P(x, y, z) \leftarrow Q(\ldots, x, \ldots), S(\ldots, y, \ldots), R(\ldots, z, \ldots), \text{Chosen}(x, y, z)
\]

Next, define the *Chosen* predicate with two extra rules:

\[
\text{Chosen}(x, y, z) \leftarrow Q(\ldots, x, \ldots), S(\ldots, y, \ldots), R(\ldots, z, \ldots), \neg \text{DiffChoice}(x, y, z)
\]

\[
\text{DiffChoice}(x, y, z) \leftarrow \text{Chosen}(x, y, z'), z' \neq z
\]

Unstratified negation!

*choice* operator is an extra source of unstratified negation.
• The last two rules ensure that, for every pair of values \((x, y)\) that satisfies the body, the predicate \(Chosen(x, y, z)\) satisfies the functional dependency: \(xy \rightarrow z\)

• In (*) the choice operator can (or must) be replaced by the new predicate \(Chosen\) that forces the expected functional dependency

• Some systems do not support the choice operator, so \(Chosen\) (defined as above) has to be used instead

**Exercise:** In the example above, replace the \textit{choice} operator by its corresponding and concrete \textit{Chosen} predicate (adding its definition, of course); and compute the repairs of the DB with DLV

**Exercise:** Give a general program to solve the “Hamiltonian Cycle” problem
The Answer Set Programming Paradigm

- Normal programs with stable model semantics can be used to solve hard (and easy) combinatorial problems
- We use the more general notion of “Logic Programs with Answer-Set Semantics” (more below)
- Determining if there are stable models, and computing one (in the positive case) is good enough (brave semantics can be used)
- Answer-Set Programming is a new (logic) programming paradigm: Specify the problem’s conditions

The underlying “solver” will find a solution (if any), or report when none exists


- A form of declarative programming (as opposed to imperative)
• NPs (and more generally ASPS) are commonly used to implicitly specify by means of a general program $\Pi^G$ all the solutions of a general (usually combinatorial) problem $\mathcal{P}$

• A specific instance $I$ (input) for problem $\mathcal{P}$ is usually represented by means of a specific EDB $E$ for $\Pi^G$

• The SMs of the combined specification $\Pi^G \cup E$ become the solutions for $\mathcal{P}$ with instance $I$

  E.g. $\mathcal{P}$ could be 3-GC, and $I$ a concrete graph (c.f. page 23)

• Solutions are computed using a general underlying solver

  In many applications finding one model is good enough

• Commonly, the solver internally transforms the problem (so as SM computation) into SAT, the problem of determining the truth assignments that make a propositional formula true

  The latter a crucial problem in many areas of computer science
• SAT is a sufficiently and necessarily difficult problem; complete for the complexity class \( NP \)

• The computational data complexity of normal programs matches that of \( NP \)-complete problems (more below)

Then, normal programs can be used to solve any problem in the class \( NP \)

• The solutions can be computed using general implementations of the SM semantics, e.g. CLINGO or DLV

• ASPs are successfully used in different areas and problems
  1. Logic-based knowledge representation in general
     In particular, representation of commonsense knowledge and commonsense reasoning
  2. Solution of hard combinatorial problems
  3. Many applications in data management
     ASPs extend Datalog
In data management, ASP can be used to define complex views and express complex queries.

E.g. $R'$ on page 29

The ASP goes on top of a RDB

The stable models of $\text{EDB} + \text{ASP program}$ are extended with query atoms $\text{Ans}(a)$ and their specification by means of extra rules.
• Sometimes we need (or welcome) an extension of NPs
  To specify finitely many, but usually mutually exclusive, alternatives

• The SM semantics can be extended to disjunctive programs
Disjunctive Stable Model Semantics

• Now we admit rules of the form

\[ B_1 \lor \cdots \lor B_k \leftarrow A_1, \cdots, A_m, \text{not } A_{m+1}, \ldots, \text{not } A_n \]

with \( B_j, A_i \) atoms, and \textit{not} weak negation as before

E.g. \[ P(x) \lor T(x) \leftarrow R(x), Q(y), \text{not } T(y), \text{not } Q(x) \]

• Now we have the \textit{disjunctive stable model semantics}  
  (Gelfond & Lifschitz, 1991)

• The semantics is an extension of that for NPs  
  Based, as before, on testing candidates to SMs
Start with a candidate set of atoms $S \subseteq HB(\Pi)$

1. Pass from $\Pi$ to $\Pi_H$ (grounding)

2. Construct a new ground program $\Pi^S_H$:
   (a) Delete from $\Pi_H$ every rule that has a subgoal not $A$ in the body, with $A \in S$
   (b) From the remaining rules, delete the negative subgoals

3. We are left with a ground disjunctive positive program $\Pi^S_H$, without negation

   As a “disjunctive Datalog program” it may have several minimal (H-)models (a simple and natural extension of Datalog)

   $MinMod(\Pi^S_H)$ denotes the set of minimal models of $\Pi^S_H$

4. If $S \in MinMod(\Pi^S_H)$, we say that $S$ is a stable model of $\Pi$
Example: \[ p \lor q \leftarrow s, \, \text{not } p \]
\[ s \leftarrow \]

- For \( S_1 = \{s, q\} \), the residual program \( \Pi^{S_1} \) is:
  \[ p \lor q \leftarrow s \]
  \[ s \leftarrow \]

For this positive, disjunctive Datalog program:
\[ \text{MinMod}(\Pi^{S_1}) = \{ \{s, p\}, \, \{s, q\}\} \]

Two minimal models
\( S_1 \in \text{MinMod}(\Pi^{S_1}) \): is stable model

- For \( S_2 = \{s, p\} \), the residual program is:
  \( \Pi^{S_2} : s \leftarrow \)

\[ \text{MinMod}(\Pi^{S_2}) = \{ \{s\} \} \]

\( S_2 \notin \text{MinMod}(\Pi^{S_2}) \): not a stable model
• As before, every stable model of \( \Pi \) is a minimal \( H \)-model of \( \Pi \).

• A positive disjunctive program, i.e. without negation, may have several minimal models.
  They are also its stable models.

• Due to minimality, and unless forced by other rules, a disjunctive rule will pick only one of the disjuncts from the head to make it true.

• **Example:** (3-GC revisited) A disjunctive general program:

\[
\begin{align*}
\text{color}(x, \text{green}) \lor \text{color}(x, \text{blue}) \lor \text{color}(x, \text{red}) & \leftarrow \text{country}(x) \\
& \leftarrow \text{edge}(x, y), \text{color}(x, z), \text{color}(y, z)
\end{align*}
\]

Unstratified negation is hidden in the program constraint.

The disjunctive head assigns to each country a single color.

This program is equivalent to the one on page 23: they have the same SMs.
Only some disjunctive programs with negation can be rewritten as NPs, i.e. without disjunction

There is a syntactic class of disjunctive programs, that of the head-cycle free programs, that can be rewritten into equivalent non-disjunctive programs

By passing, in turns, the (positive) atoms in the head as negative literals in the body

This is the case of the program for 3-GC above (c.f. page 23)

Disjunctive logic programs with stable model semantics are very expressive

More than (non-disjunctive) NPs with stable model semantics

(under common complexity-theoretic assumptions; more coming ...)

So, some disjunctive programs cannot be expressed as equivalent NPs

Disjunctive programs can be used to solve harder combinatorial problems
Complexity Considerations

- For a fixed intentional program $\Pi$ (i.e. w/o EDB), and an input EDB $\mathcal{D}$ (i.e. data complexity), there are several decision problems:
  
  1. **Model Checking**: Given a set of ground atoms $\mathcal{M}$, is it a stable model (of $\Pi \cup \mathcal{D}$)?
  2. **Consistency**: Does the program have a stable model?
  3. Is a given ground atom a skeptical consequence?
  4. Is a given ground atom a brave consequence?

- They have a high time-complexity; at least $NP$-hard for the last three in the size of $\mathcal{D}$ (more below)

  For NPs, model checking can be done in PTIME

- The “functional problem” of **computing a stable model** is also hard
• Another difficult problem is Model Counting: How many stable models?
  Model Counting is crucial in many areas of Computer Science
• We recall several complexity measures: program complexity, query complexity, data complexity, combined complexity (combinations of the previous ones)
• They count the worst-case number of steps to reach the decision in terms of: $|\Pi|$ (leaving, $D$, $Q$ as fixed parameters), $|Q|$, $|D|$, $|\Pi \cup D|$, etc.
• In data management, the most relevant measure is data complexity
  In terms of the size of the underlying EDB, which can be large, while the “intentional part” of the program, short
• For example, the proper formulation of 3. above as a “data problem”:

\[ SQA(\Pi, A) := \{ D \mid \Pi \cup D \models_{skp} A \} \]

As a parameterized family of decision problems

The parameters are the intentional program \( \Pi \) and atom \( A \)

The decision problem is, for an instance \( D \), whether it belongs to \( SQA(\Pi, A) \) (do the same with the other problems!)

• Decision problems above, in data complexity, can be placed in the polynomial hierarchy:

1. Model checking is coNP-complete
   More precisely, there are \( \Pi, Q \) for which \( SQA(\Pi, Q) \) is coNP-complete
   For normal programs, i.e. no disjunction, it is always in PTIME

2. Certain QA becomes \( \Pi_2^P \)-complete
   For normal programs: coNP-complete

3. Brave QA becomes: \( \Sigma_2^P \)-complete
   For normal programs: NP-complete

\[
\begin{align*}
\Pi_0^P & := \Sigma_0^P := \Delta_0^P := P \\
\Delta_{i+1}^P & := P^{\Sigma_i^P} \\
\Sigma_{i+1}^P & := (NP)^{\Sigma_i^P} \\
\Pi_{i+1}^P & := (coNP)^{\Sigma_i^P}
\end{align*}
\]

E.g. \( \Sigma_1^P = NP^P = NP \)
\( \Pi_1^P = coNP^P = coNP \)
\( \Sigma_2^P = NP^{NP} \)
\( \Pi_2^P = coNP^{NP} \)
\( \ldots \)

- \( coNP \) is the class of decision problems whose complements are solvable in non-deterministic PTIME
- \( (coNP)^{NP} \) is the class of decision problems whose complements can be solved in non-deterministic PTIME with calls to an oracle that solves (in one step) problems in \( NP \)

etc.
Why “Answer Sets”? 

- As opposed to “stable models”
- ASPs may have also “classical negation” in heads and bodies (in addition to weak negation in bodies)
  
  \[ \neg A(x) \lor C(y) \leftarrow B(x, y), \neg S(y), \text{not } P(x), \text{not } \neg M(x, y) \]

- They describe, and have as models, worlds that are only partially represented as sets of classical literals

- For example, \( \mathcal{M} = \{ P(a), P(b), \neg P(c), Q(d), \neg Q(a) \} \) is an incomplete representation of a world
  
  - \( P(a), P(b), Q(d) \) are true
  - \( P(c), Q(a) \) are false
  - \( P(d), Q(b), Q(c) \) are uncertain

No application of the CWA

- ASPs have “answer sets” as models, which may be partial worlds
• We expect worlds and models to be “consistent”
  We cannot find something like this in them: \{\ldots, A, \neg A, \ldots\}

• ASPs extend disjunctive programs with only weak negation and SMs

• In ASPs, classical and weak negation may coexist

• CWA related to weak negation
  If we want, we may impose (specify) the CWA:
  \[ \neg A \leftarrow not \ A \]  
  (with \( A \) an atom)
  Making false what is not known to be true

• ASPs are also called “extended logic programs”