Chapter 10

Database Updates and Nonmonotonic Reasoning

10.1. Let $P$ be a Datalog program. Show that $\Gamma_{P[M]}^\omega(\emptyset)$ can be computed in time that is polynomial in the size of $P$’s Herbrand base.

**Answer:**

Let $\Gamma_{P[M]}(I)$ be defined as:

$$\Gamma_{P[N]}(I) = \{ h(r) \mid r \in ground(P), gp(r) \subseteq I, gn(r) \subseteq N \}$$

Then assume that we rename all the facts in $N$ with the special subscript $N$. Then every negated goal in $P$ can be transformed by removing the negation sign in from of each negated goal and adding the special subscript $N$ to its predicate name. Then, the computation of $\Gamma_{P[N]}(I)$ on the original program coincides with the computation of $T_P(I)$ on the transformed program.

Therefore, the desired conclusion follows from Exercise 8.16, provided that again we assume that the size of database dominates that of the rule base.

10.2. Use Example 10.23 and the results from Exercise 3.1 to show that deciding whether stable models exist for a given program is $\mathcal{NP}$-complete.

**Answer:**

Example 10.23 shows that deciding whether a given program has a stable model is $\mathcal{NP}$-hard. In fact, for any problem from the class of $\mathcal{NP}$-complete problems, we have a polynomial transformation into the Hamiltonian path problem. Thus, if a decision procedure on the existence of stable models for programs such as that of Example 10.23, would execute in time less than exponential, then we could easily derive a less than exponential decision procedure on all the remaining problems in the class.

To prove completeness we must give an exponential procedure that solves the problem. Our procedure is as follows:
for each $M \subseteq B_P$ check if $\Gamma_{P(M)}^\omega(\emptyset) = M$

Assuming that the computation of $\Gamma_{P(M)}^\omega$ is not worse than exponential, the resulting procedure is exponential in $|B_P|$. Moreover, as discussed in Exercise 10.1, the computation of $\Gamma_{P(M)}^\omega$ is polynomial when the size of the database dominates that of the rule base.

10.3. Consider the program

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

For this program, write all models, minimal models, fixpoints, minimal fixpoints, and stable models, if any. Also give $T_P^\omega$.

**Answer:**

We can enumerate all the Herbrand interpretations of this program and decide which are models. Thus:

- $\{a, b, c\}$ is a model ($B_P$ is always a model)
- $\{a, b\}$ is not a model (the first rule is not satisfied)
- $\{b, c\}$ is a model
- $\{a, c\}$ is a model
- $\{a\}$ is not model (the first rule is not satisfied)
- $\{b\}$ is a model
- $\{c\}$ is not a model (the second rule is not satisfied) The empty set is not model (the second rule is not satisfied).

Therefore, $\{a, c\}$ and $\{b\}$ are the only minimal models.

Since every fixpoint is also a model, we can test each model to see which is a fixpoint. Now,

\[
\begin{align*}
  T_P(\{b, c\}) & = \emptyset \\
  T_P(\{a, c\}) & = \{a\} \\
  T_P(\{b\}) & = \emptyset.
\end{align*}
\]

Thus this program has no fixpoint.

Therefore, we can use the theorem that every stable model is also a least fixpoint to conclude that this program has no stable model. Alternatively, we can apply the stability transformation to $P$, for each model of $P$. For instance, for $M = \{a, c\}$, $\text{ground}_M(P)$ reduces to the following fact:

\[
a.
\]

This transformed program has as least model $\{a\} \neq \emptyset$. 


for $M = \{b\}$, $\text{ground}_M(P)$ is as follows:

$$\text{c} \leftarrow \text{a}.$$  
$$\text{b} \leftarrow \text{a}.$$  

The least model of this transformed program is the empty set, rather than $M$.

For $M = \{b, c\}$, $\text{ground}_M(P)$ is the empty program, that has as least model the empty set. This confirms that $P$ has no stable model.

Let us now compute $T^\omega_P$:

$$T^1_P(\emptyset) = \emptyset$$  
$$T^1_P(\{a\}) = \emptyset$$  
$$T^2_P(\{a\}) = T_P(\{a\}) = \{a, b, c\}$$  
$$T^2_P(\emptyset) = T_P(\{a, b, c\}) = \emptyset$$

After that, the computation repeats itself with period 3 and never converges:

$$T^{J+3}_P(\emptyset) = \emptyset, \quad T^{J+3}_P(\{a\}) = \emptyset, \quad \text{and} \quad T^{J+3}_P(\{a, b, c\}) = \emptyset.$$  

But $T^\omega_P(\emptyset)$ is defined as the union of all these results. Thus, $T^\omega_P(\emptyset) = \{a, b, c\}$.

10.4. Prove that every stable model is a minimal model.

**Answer:**

See next problem.

10.5. Prove that every stable model for $P$ is a minimal fixpoint for $T_P$.

**Answer:**

The fact that $M$ is a model and a fixpoint follows directly from the definitions. We need to prove minimality. Let $T_P(N) = N$, with $N \subseteq M$. Then

$$N = T_P(N) = \Gamma_P(N) = \Gamma_{P(N)}(N)$$

But since our gamma function is also monotonic in its implicit argument shown as a subscript, and $N \supseteq M$, we have:

$$N = \Gamma_{P(N)}(N) \supseteq \Gamma_{P(M)}(N)$$

Since our gamma function is monotonic (in its explicit argument), we have

$$\Gamma_{P(M)}(N) \supseteq \Gamma_{P(M)}(\emptyset) = M$$

Thus, $N = M$.

If we assume that $N$ is a model, rather than a fixpoint, then $N \supseteq T_P(N)$ and the same reasoning holds.
10.6. Prove that the program in Example 10.5 is not locally stratified.

**Answer:**
Let $P$ be the program of Example 10.5. $P$’s Herbrand base is $B_P = \{ \text{even}(s^k(0)) \mid k \geq 0 \}$. By contradiction, assume that $P$ is locally stratified. Then, the bottom stratum in this local stratification must contain some atom even($s^j(0)$), with $j \geq 0$. Now, $\text{ground}(P)$ contains the following rule: even($s^j(0)$) $\rightarrow$ $-\text{even}(s^{j+1}(0))$. Thus, by the definition of local stratification, even($s^{j+1}(0)$) belongs to a stratum strictly lower than the bottom stratum—a contradiction.

10.7. Explain how the alternating fixpoint computes the stable model for a stratified program. Perform the computation on Example 8.7.

**Answer:**
Senior students who completed all the requirements for the cs major:

\[
\begin{align*}
\text{req}_\text{missing}(&\text{Name}) \leftarrow \text{student}(\text{Name}, \_ \text{senior}), \\
&\text{req}(\text{cs}, \text{Course}), \\
&-\text{hastaken}(\text{Name}, \text{Course}). \\
\text{hastaken}(\text{Name}, \text{Course}) &\leftarrow \text{took}(\text{Name}, \text{Course}, \text{Grade}). \\
\text{all}_\text{req}_\text{sat}(\text{Name}) &\leftarrow \text{student}(\text{Name}, \_ \text{senior}), \\
&-\text{req}_\text{missing}(\text{Name}).
\end{align*}
\]

Let us assume that the content of the \text{student} and \text{took} is that of Example 8.2, while that of \text{requirements} is as follows:

\[
\text{req}(\text{cs}, \text{cs101}).
\]

We will now outline the alternating fixpoint computation, using both the set of negative atoms $N$, or its complement $\bar{N}$.

(a) Let $D$ be the atoms in the database. Then, with $N = \emptyset$, $\Gamma_P^{\bar{N}}(\emptyset)$ consists of $D$ and the following \text{hastaken} atoms (the head of the only rule without negated goals):

\[
\begin{align*}
\text{hastaken}(\text{Joe Doe}', \text{cs123}). \\
\text{hastaken}(\text{Jim Jones}', \text{cs101}). \\
\text{hastaken}(\text{Jim Jones}', \text{cs143}). \\
\text{hastaken}(\text{Jim Black}', \text{cs143}). \\
\text{hastaken}(\text{Jim Black}', \text{cs101}).
\end{align*}
\]

Thus $N^1 = B_P - \bar{N}^1$ yields the first overestimate for negative facts—actually, only \text{req}_\text{missing} atoms are overestimated, there is no overestimation of \text{hastaken} atoms.

(b) Using the overestimate above, we find that the last rule is satisfied by all the senior students (as if every senior had no requirements missing.) Therefore, $\bar{N}^2$ contains \text{all}_\text{req}_\text{sat}(\text{Joe Doe}') and \text{req}_\text{missing}(\text{Joe Doe}') in addition to $\bar{N}^1$. Thus, $N^2 = A_P(\emptyset)$.
\[ A_1: \text{new\_student\_hist}(+, \text{Name}, \text{tba}, \text{tba}) \leftarrow \\
\quad \text{old\_took\_hist}(+, \text{Name}, _, _), \\
\quad -\text{old\_student\_snap}(\text{Name}, _, _). \]

\[
\text{new\_student\_snap}(\text{Name}, \text{Major}, \text{Level}) \leftarrow \\
\quad \text{old\_student\_snap}(\text{Name}, \text{Major}, \text{Level}), \\
\quad -\text{new\_student\_hist}(-, \text{Name}, \text{Major}, \text{Level}). \\
\text{new\_student\_snap}(\text{Name}, \text{Major}, \text{Level}) \leftarrow \\
\quad \text{new\_student\_hist}(+, \text{Name}, \text{Major}, \text{Level}).
\]

Therefore, we have a three-layer stratification, where at the bottom stratum we find the old-version of the predicates. At the next stratum we find \text{new\_student\_hist} and at the top stratum we have \text{new\_student\_snap}. Thus the active rules are computed first, and then new entries in the history relation are propagated to the snapshots.

In the synchronized version, \text{counter}(T) must be added to the tuples so generated; \( T \) is the argument in the \text{new} predicates, i.e., the new position in the history relation being filled by the active rules.

10.12. Given the relation \text{club\_member}((\text{Name}, \text{Sex}))\], write a nonrecursive Datalog program to determine whether there is exactly the same number of males and females in the club. Use choice but not function symbols. Answer:

\[
\begin{align*}
\text{male}(\text{Name}) & \leftarrow \text{club\_member}(\text{Name}, \text{m}). \\
\text{feml}(\text{Name}) & \leftarrow \text{club\_member}(\text{Name}, \text{f}). \\
\text{match}(\text{M}, \text{W}) & \leftarrow \text{male}(\text{M}), \text{feml}(\text{W}), \text{choice}((\text{M}), (\text{W})), \text{choice}((\text{W}), (\text{M})). \\
\text{unmatched} & \leftarrow \text{male}(\text{M}), \neg\text{match}(\text{M}, _). \\
\text{unmatched} & \leftarrow \text{feml}(\text{F}), \neg\text{match}(\text{_, F}). \\
\text{same\_number} & \leftarrow \text{match}(\text{_, }), \neg\text{unmatched}. \\
\text{same\_number} & \leftarrow \neg\text{male}(\text{_, }), \neg\text{feml}(\text{_, }).
\end{align*}
\]

10.13. Use choice to write a recursive program that takes a set of database facts \text{b}(X) and produces a relation \text{sb}(X, I), with \( I \) a unique sequence number attached to each \( X \).

Answer:

\[
\begin{align*}
\text{ssb}(\text{nil}, 0). \\
\text{ssb}(X, I + 1) & \leftarrow \text{ssb}(\_I, \text{b}(X)), \\
& \quad \text{choice}((X), (I)), \text{choice}((I), (X)). \\
\text{sb}(X, I) & \leftarrow \text{ssb}(X, I, \text{b}(X)).
\end{align*}
\]

Where \text{nil} is a new symbol.

10.14. Express the parity query using choice (do not assume a total order of the universe).
Answer:

\[
\text{next(nil, nil).}
\]

\[
\text{next(X, Y) ← next(X), br(Y), choice((X), (Y), choice((Y), (X))).}
\]

\[
\text{even(nil).}
\]

\[
\text{even(Y) ← odd(X), next(X, Y).}
\]

\[
\text{odd(Y) ← even(X), next(X, Y).}
\]

\[
\text{br_is_even ← even(X), ¬next(X, Y).}
\]

10.15. Write a program with choice and stratified negation that performs the depth-first traversal of a directed graph.

Answer:

We are given a directed graph whose arcs are stored by means of facts of the form \(g(x, y)\) and an initial node \(a\). The first set of rules orders the immediate successors of a node \(N\) into a chain:

\[
\text{next(N, start, start) ← g(N, _).}
\]

\[
\text{next(N, Y, Z) ← next(N, _Y), g(N, Z), choice((N, Y), (Z)), choice((N, Z), (Y)).}
\]

\[
\text{first(N, Y) ← next(N, X, Y), X = start, Y \neq start.}
\]

\[
\text{last(N, Z) ← next(N, Y, Z), ¬next(N, Z, _).}
\]

\[
\text{sterile(Y) ← g(N, Y), ¬g(Y, Z).}
\]

The second set of rules does the actual traversal, starting from a root node \(a\), and proceeding downward. The first three rules defining explored arcs perform the downward traversal beginning from \(a\). During this traversal, an arc is classified as a tree-arc, when its end-node is not the end-node of a previously classified tree arc—a constraint enforced by the choice goal in the fourth rule below. Thus, downward traversal develops through the visitation of the first child of the last-developed tree arc (second rule), and then the visitation of the next sibling of a solved arc (third rule).

The last three rules add an arc to the set of solved arcs, when (i) this is a sterile tree arc, (ii) its end-node has been previously reached (by another tree arc—the complementary condition of that used in the fourth rule) and (iii) the last child of the arc has been solved.

\[
\text{explored(a, Y) ← first(a, Y).}
\]

\[
\text{explored(Y, Z) ← tree(X, Y), first(Y, Z).}
\]

\[
\text{explored(Y, Z) ← solved(Y, X), next(Y, X, Z).}
\]

\[
\text{tree(X, Y) ← explored(X, Y), choice((Y), (X)).}
\]

\[
\text{solved(X, Y) ← tree(X, Y), sterile(Y).}
\]

\[
\text{solved(X, Y) ← explored(X, Y), tree(X1, Y), X \neq X1.}
\]

\[
\text{solved(W, Y) ← last(Y, X), solved(Y, X), tree(W, Y).}
\]

\[
\text{first(a, Y).}
\]

\[
\text{first(Y, Z).}
\]

\[
\text{solved(Y, X), next(Y, X, Z).}
\]

\[
\text{explored(X, Y), choice((Y), (X)).}
\]

\[
\text{tree(X, Y), sterile(Y).}
\]

\[
\text{explored(X, Y), tree(X1, Y), X \neq X1.}
\]

\[
\text{last(Y, X), solved(Y, X), tree(W, Y).}
\]
We next turn to the problem of detecting forward, back and cross arcs. The transitive closure predicate \( \text{treach}(y, x) \) denotes the fact \( y \) is reachable from \( x \), using tree-arcs. Thus, an arc \( \text{explored}(x, y) \) is (i) a forward arc if \( \text{treach}(x, y) \) is true but \( \text{tree}(x, y) \) is not, (ii) a back arc if \( \text{treach}(y, x) \) is true, and (iii) a cross arc if it is neither a tree arc, nor a forward arc, nor a back arc.

\[
\begin{align*}
\text{treach}(X, Y) & \leftarrow \text{tree}(X, Y) \\
\text{treach}(X, Z) & \leftarrow \text{tree}(X, Y), \text{treach}(Y, Z). \\
\text{forward}(X, Y) & \leftarrow \text{explored}(X, Y), \neg \text{tree}(X, Y), \text{treach}(X, Y). \\
\text{back}(X, Y) & \leftarrow \text{explored}(X, Y), \text{treach}(Y, X). \\
\text{cross}(X, Y) & \leftarrow \text{explored}(X, Y), \neg \text{tree}(X, Y), \neg \text{forward}(X, Y), \neg \text{back}(X, Y).
\end{align*}
\]

A list of the nodes of resulting tree in pre-order traversal is then easy to compute from \( \text{tree} \) nodes. This list can also be produced while \( \text{tree} \) is being computed [F. Giannotti, et al., “Programming with non-determinism in deductive databases,” \textit{AMAI}, Vol.19, No. 3-4, 1997].