5.1 Complexity of Logic

- The last lectures dealt with basics of first order logics
  - We showed how to write syntactically correct logical statements
  - We discussed interpretations and models
  - We showed how to deduce and to prove statements
  - We heard some stuff about history of logics

- So, we get closer to building a deductive DB
  - Essentially, a deductive DB will later check if a given statement can be followed given a set of facts and rules

- $\mathcal{W} \models \mathcal{W}$, i.e. $\mathcal{W} \cup \mathcal{W}^+$ satisfiable

- Now, it’s time to have a look at the computational complexity of logics
  - The check for validity and the check for satisfiability is especially important
  - A database is about performance
  - If it turns out that the anticipated complexity is prohibitive, we are in deep trouble

- Will some restrictions save the day?

5.1 Complexity of Logic

- First, let’s have a look on plain Boolean logic
  - i.e. no predicates, no quantifiers, universe is limited to \{true, false\}

- Like first order logic Boolean statements can also be valid, satisfiable, or unsatisfiable

- So, how do you test whether some Boolean statement $\mathcal{W}$ is satisfiable, valid, or unsatisfiable
  - This is commonly known as the SAT problem

- Unsatisfiable:
  - Check if $\mathcal{W}$ is satisfiable; if not, it is unsatisfiable
  - Valid:
    - Check if $\neg \mathcal{W}$ is unsatisfiable; if not, it is valid
  - Satisfiable:
    - Generate a substitution for all variables in $\mathcal{W}$
    - Evaluate the substituted expression
5.1 Complexity of Logic

- Unfortunately, SAT is in **NP**
  - Deterministic decidable algorithm
    - Generate all $2^n$ substitutions
    - Evaluate substituted expression for each substitution
    - In $O(n^2)$ each
    - Overall, in $O(n^2 2^n)$
  - Non-Deterministic semi-decidable algorithm
    - Guess any substitution
    - Evaluate substituted expression in $O(n^2)$
    - Continue until you find a working substitution
  - NP is a pretty bad property for an algorithm...

- **Example:** Is $W$ satisfiable?
  - $W_2 = (x \lor y \lor z) \land \neg x$
    - Yes, for $x = y = \text{false}$

- The default algorithm for solving the SAT problem is the **Davis-Putnam algorithm**
  - Solves the problem of satisfiability for a Boolean formula in conjunctive normal form
  - Complexity is somewhere around $O(1.8^n)$...
  - Basic idea: Build a pruned tree of possible substitutions

- Horn-SAT is very important as it is in **P-complete**
  - Example Horn-SAT problem:
    - $W = \left( \neg x_1 \land \neg x_2 \land \neg x_3 \lor x_1 \land \neg x_2 \land x_3 \right) \land \left( \neg x_2 \land \neg x_3 \land x_1 \right) \land \left( x_1 \lor x_2 \lor x_3 \right)$
  - This results in (implicative form):
    - Facts: $\text{true} \rightarrow x_1 \land x_2 \land x_3$
    - Defines: $(x_1 \land x_2 \land x_3) \land (\neg x_1 \land \neg x_2 \land x_3)$
    - Goals: $(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (x_1 \lor x_2 \lor \neg x_3)$
    - Whole set is satisfiable, if conjunction of implications is true
  - **Idea:** find all those variables which have to be true and look for any contradiction!
5.1 Complexity of Logic

- Find all those variables \( T \) which have to be true!
  - \( \text{Init: } T := \emptyset \) (i.e. all variables are false)
  - \( \text{Pick any unsatisfied implication } H_i \) (facts or defines)
    - \( H_i \equiv (x_3 \land \ldots \land x_n) \rightarrow \top \)
    - unsatisfied implication: all \( x_i \) are true, \( y \) is false
    - \( \text{Add } y \text{ to } T \) (thus \( H_i \) is satisfied now)
    - \( \text{Repeat until there are no unsatisfied implications} \)
- \( \mathcal{W} \) is satisfiable, iff \( T \) satisfies all clauses
  - Furthermore, \( T \) is the minimal set of variables which satisfies \( \mathcal{W} \), i.e. for each satisfying substitution \( T \) holds. TCT

---

5.1 Complexity of Logic

- So, lets switch to general first order logic. What changes with respect to to complexity?
  - \( \text{Universe of potentially unlimited size} \)
  - \( \text{Quantifiers} \)
    - A given sub-formula has to be true for all / some elements of the universe
  - \( \text{How does this affect our complexity?} \)
    - As an example, we will use the popular axiomatization of the number theory

---

5.1 Complexity of Logic

- So, how can we (naively) evaluate first order logic?
  - \( \forall x \left( x > 5 \right) \)
    - \( x = 0, \) Bang. Untrue.
  - \( \exists x \left( x > 5 \right) \)
    - \( x = 0, x = 1, x = 2, \ldots ; x = 6; \) Ok. True.
  - \( \forall x, y \left( 3 \times x + 3 > y \right) \right) \right) \)
    - \( x = 0, y = 0, x = 1, y = 0, x = 0, y = 1, x = 1, y = 1; x = 2, y = 0, \ldots \)
    - So, this seems to be true! Where do we stop?

---

5.1 Complexity of Logic

- Example Horn-SAT problem:
  - \( H_1 \equiv (\text{true} \rightarrow x_1), H_2 \equiv (\text{true} \rightarrow x_2), \)
  - \( H_3 \equiv (x_2 \land x_3) \rightarrow \text{false} \), \( H_4 \equiv (x_2 \land x_3 \rightarrow x_3) \)
- \( \text{Algorithm} \)
  - \( (\text{true} \rightarrow x_1) \Rightarrow T := \{ x_1 \} \)
  - \( (\text{true} \rightarrow x_2) \Rightarrow T := \{ x_2, x_3 \} \)
  - \( x_2 \land x_3 \rightarrow x_3 \Rightarrow T := \{ x_2, x_3, x_3 \} \)
  - \( \text{Does } T \text{ satisfy all clauses?} \)
    - It obviously satisfies \( H_1, H_2 \) and \( H_3 \)
    - \( H_4 \) also satisfies \( T \)
    - \( T \) satisfies \( \mathcal{W} \)
    - \( \text{If there was also an } H_1 \equiv (x_2 \land x_3 \rightarrow \text{false}), \mathcal{W} \text{ would be unsatisfiable} \)

---

5.1 Complexity of Logic

- \( \text{Number theory} \)
  - \( \mathcal{L}_\mathbb{N} = (\tau, 0, 1, +, \cdot, \leq, =, \neq, (x, y, z)) \)
  - \( \text{There is just the constant } 0 \)
  - \( \text{The } \tau \text{ function represents the successor function} \)
    - \( \text{i.e. } \tau(\tau(0))) = 3 \)
    - As using the successor function is very unhandy, we employ a shortcut notation for all natural numbers
      - \( \text{We may e.g. use } 3451 \text{ instead of } \tau(\tau(\tau(\tau(0)))) \)
  - \( \text{The functions } +, \cdot, \) and \( \leq \) represent addition, multiplication, and exponentiation
  - \( \text{The predicates } =, \) and \( \neq \) represent equality and the less-than-predicate

---

5.1 Complexity of Logic

- \( \text{Testing all substitutions for universes with unlimited size is kind of tricky} \)
- \( \text{Alternative Idea:} \)
  - \( \text{Use deductive systems to construct a proof from a set of valid axioms to the questionable statement} \)
  - \( \text{Number theory} \) has been axiomized several times on different styles
  - \( \text{Most popular: Peano arithmetic} \)
    - \( \text{Commonly } 15 \text{ axioms types inducing countable unlimited number of axioms} \)
    - \( \text{Introduced by Italian mathematician Giuseppe Peano in 1889} \)
5.1 Complexity of Logic

- But consider this:
  - \( \neg \exists n, x, y, z ((a^n + b^n = c^n) \land n > 2) \)
  - This is Fermat's Last Theorem
  - It took 357 years to show that this statement is provable for the natural numbers
    - 1637-1995
    - Proof did some really nasty tricks...
  - So, we seem to be in severe trouble here...

- Example: The Goodstein Theorem and the Paris-Kirby Theorem
  - Goodstein's Theorem: "Imagine Hercules fighting the Hydra, chopping off one of its heads after the other. But every time a head is chopped off, the Hydra regrows a finite number of heads (according to the Goodstein sequence). Still, Hercules will eventually defeat the Hydra as long as he does not give up."

- BUT: Paris-Kirby Theorem: "The Goodstein Theorem is not decidable."
  - i.e. there is no way to prove Goodstein within Peano arithmetic
  - Actually, there is no way to prove it at all using first order logic
  - Proof Sketch: "Show that the consistency of Peano arithmetic directly follows from the Goodstein theorem. If Goodstein was provable within Peano, the consistency of Peano was shown within itself. This is not possible according to Gödel's incompleteness theorem."
  - This is pretty bad. Obviously, we need some restrictions...
    - ...but first, we move to some algorithms

- So, where is the problem?
  - Remember Gödel's Incompleteness Theorem
    - "Any effectively generated theory capable of expressing elementary arithmetic cannot be both consistent and complete. In particular, for any consistent, effectively generated formal theory that proves certain basic arithmetic truths, there is an arithmetical statement that is true, but not provable in the theory."
    - Thus, for any non-trivial deductive system NT, there are statements which cannot be proven within that system
      - Unprooofable statements are called undecidable
        - NT ⊢ W
          - W is decidable
          - ¬W is undecidable
        - NT ⊢ W
          - W is decidable
          - ¬W is undecidable
      - Example: The Goodstein Theorem and the Paris-Kirby Theorem
        - Goodstein's Theorem: "Imagine Hercules fighting the Hydra, chopping off one of its heads after the other. But every time a head is chopped off, the Hydra regrows a finite number of heads (according to the Goodstein sequence). Still, Hercules will eventually defeat the Hydra as long as he does not give up."
        - ...the actual theorem is not important for us. But note that the theorem is indeed expressible and true within the Peano arithmetic
          - i.e. all Goodstein sequences are finite regardless of their start value.
        - BUT: Paris-Kirby Theorem: "The Goodstein Theorem is not decidable."
          - i.e. there is no way to prove Goodstein within Peano arithmetic
          - Actually, there is no way to prove it at all using first order logic
          - Proof Sketch: "Show that the consistency of Peano arithmetic directly follows from the Goodstein theorem. If Goodstein was provable within Peano, the consistency of Peano was shown within itself. This is not possible according to Gödel's incompleteness theorem."

- Jacques Herbrand
  - Born 1908 in Paris, finished his Doctorate degree 1929 at the Sorbonne in Paris
  - In early 1931, he got a fellowship at the university of Berlin, worked there with John von Neumann
  - Later moved to Göttingen to work with Emmy Noether
  - There, started his signature work "On the consistency of arithmetic"
    - "Before finishing, died during a hiking trip in the Alps in July 1931 at age 23"
5.1 Herbrand Theorem

- The Herbrand theorem (1928)
  - Informal: "A is a closed formula in universal prenex form. Then A is unsatisfiable if and only if there is a finite subset of its Herbrand expansion which is Boolean unsatisfiable"
    - Note that Herbrand himself messed the proof up, the flaw was discovered in 1960 by Dreben...
    - Today, proving is quite easy if the Compactness theorem is used
    - Universal prenex form: $A \equiv \forall y_1, \ldots, y_n \ F(y_1, \ldots, y_n)$ and $F$ being quantifier-free.
    - Very important: The problem of first order logics unsatisfiability is transformed to a Boolean unsatisfiability problem

- Some considerations
  - $TA$ can be checked for unsatisfiability in finite time
    - e.g. Davis-Putnam algorithm, etc
  - However, it is not known which $TA$ will show the unsatisfiability
    - There is potentially an unlimited number of $TA \subseteq E(A)$
  - If you did not find an unsatisfiable $TA$ yet, this either means
    - a) There are none and thus $A$ is satisfiable
    - b) You have not looked long enough
    - You cannot know which of both are true (reduces to the Halting problem...)  

- Thus, the Gilmore algorithm is semi-decidable
  - Answers only if $A$ is unsatisfiable, else is caught in endless loop

5.1 Complexity of Logic

- Herbrand Theorem (more formally)
  - Let be $A$ is in universal prenex form $A \equiv \forall y_1, \ldots, y_n \ F(y_1, \ldots, y_n)$. Then $A$ is unsatisfiable if and only if there is a finite set $TA$ of ground terms $t_i$ with $1 \leq i \leq k$ and $1 \leq i \leq n$ such that $TA := \{ F(t_{i_1}, \ldots, t_{i_n}) \mid 1 \leq i \leq k \}$ is unsatisfiable
    - $F(t_{i_1}, \ldots, t_{i_n})$ are called ground instances of $A$
    - Set of possible ground instances is potentially of unlimited size (e.g. Herbrand base)
    - The set of all possible ground instances is called Herbrand expansion $E(A)$
    - i.e. for a set of terms $TA$ holds $TA \subseteq E(A)$ and $TA$ finite
    - The Herbrand theorem can equivalently be stated for existentially quantified or mixed closed formulas
    - Transformation rules for $\forall$ and $\exists$ (Herbrandization)!

- This leads to a simple meta-algorithm for checking unsatisfiability (Gilmore algorithm,)
  - Preparation:
    - Take any first-order-logic formula $A$
    - Transform $A$ into universal prenex form $A' := \forall y_1, \ldots, y_n \ F(y_1, \ldots, y_n)$
      - i.e. pull all quantifiers to the front and transform to universal quantifiers
    - Be able to generate a the Herbrand expansion $E(A') = \{ A_1, A_2, \ldots \}$
  - Gilmore Algorithm
    - $k := 1$
    - While $A'_{k+1}$ is satisfiable (or "while not unsatisfiable")
      - $k++$
    - Return 'A is unsatisfiable'

- Restriction 1: Allow only a decidable subset of first order formulas
  - One such subset of first order logics are the so-called Schönfinkel-Bernays expressions:
    - Given a language without functional symbols and without the equality predicate
    - Given expressions in prenex form
      - $W \equiv \exists x_1, \ldots, x_n \, y_1, \ldots, y_n \, W_2$ with $W_2$ is quantifier free
    - Then it is decidable if $W$ has a model or not
      - $SB$-SAT problem

06.05.2009
• Easy proof:
  – Without functional symbols, the Herbrand base is finite
  – If the Herbrand base is finite, the Herbrand expansion is finite
  – If the Herbrand expansion is finite, you can generate all subsets of the expansion in finite time (and which are also finite)
  – Each check for unsatisfiability for a finite set of ground instances is in finite time

• However, there is a catch: \( SB\text{-SAT} \in \text{NEXP} \)
  – NEXP: The class of all non-deterministic exponential algorithms
  – What does that mean:
    • You can only guess the solution AND then you need an exponential amount of time to check if your guess was correct….
    • \( O(2^{p(n)}) \) using a non-deterministic Turing machine and unlimited space.
    • Or, you could unfold the problem to an deterministic machine which takes even longer….
    • This is obviously a very very bad complexity class….

• Additionally, Schönfinkel-Bernays severely restricts the expressiveness of logics
  – No functions!
    • This is even bad in the case where you actually can avoid functions as many predicates could be implemented more efficiently as functions

• So… how do we solve all this?
  – We need a subset of first order logic which has guaranteed finite Herbrand expansions
  – We should try to find subset which is in a better complexity class than NEXP
  – We should find a subset which does not limit the expressiveness too much
  – Approach: Restrict to first order logics allowing only for Horn clauses and non-recursive typed functions
    – Ground instances are thus Horn clauses
    – Check for unsatisfiability of finite subsets of Herbrand expansion is in \( P \)
    – Herbrand expansions is finite as the Herbrand universe is finite

• Relational databases distinguish between DDL and DML
  – DDL is for creating schemas or views
  – DML is for maintaining data and queries
    • Evaluation follows the (tuple) relational calculus

• In deductive databases both data and queries are specified by formulae
5.2 Datalog

• **A deductive database** consists of facts and rules
  - The set of facts is called *extensional database* (EDB)
    - If no functions are used in the facts, it can be stored as a simple relational database table
  - The set of rules is called *intentional database* (IDB)
    - The reflects the idea of views in relational databases, but allows for recursion

5.2 Datalog

• **Datalog** is a query and rule language specifically defined for deductive databases
  - Syntactically similar to Prolog
  - Introduced around 1978 for academic database research by Hervé Gallaire and Jack Minker
  - Used as the main foundation for expert systems theory during the 1980ies

5.2 Datalog Syntax

• **A database clause** (DB-clause) is defined as
  - \( A \leftarrow L_1, \ldots, L_n \) with an atomic formula \( A \in A_L \) and literals \( L_i \in L_L \)
    - \( A \) is referred to as head and \( L_1, \ldots, L_n \) as body (or body literals) of the DB-clause
    - often written as \( A :: L_1, \ldots, L_n \)
  - DB-clauses with \( n > 0 \) are called rules
  - DB-clauses with \( n = 0 \) and an atomic ground formula \( A \) are called facts
  - A DB-clause with only atomic body literals is called **definite**

5.2 Datalog Syntax

• **Example facts**
  - parent(John, Mary).
  - parent(John, Thomas).
  - parent(Thomas, George).
  - ...

• **Example rules**
  - grandmother(X,Y) \( \leftarrow \) parent(X,Z), parent(Z,Y), female(Y).
  - path(X,Y) \( \leftarrow \) edge(X,Y).
  - path(X,Y) \( \leftarrow \) edge(X,Z), path(Z,Y).
  - ...

5.2 Datalog Syntax

• The most important feature of Datalog is the possibility to use **recursion**
  - edge(3,2), edge(2,6), edge(2,5), edge(5,3), path(X,Y) \( \leftarrow \) edge(X,Y), path(X,Y) \( \leftarrow \) edge(X,Z), path(Z,Y).
  - Alternative ways for writing the last rule are:
    - path(X,Y) \( \leftarrow \) path(X,Z), edge(Z,Y).
    - or path(X,Y) \( \leftarrow \) path(X,Z), path(Z,Y).

5.2 Datalog Syntax

• **The definition** \( \text{def}(p) \) of a predicate symbol \( p \) is the set of facts/rules in the Datalog program, where \( p \) occurs in the **head**
  - grandmother(X,Y) \( \leftarrow \) parent(X,Z), parent(Z,Y), female(Y).
  - path(X,Y) \( \leftarrow \) edge(X,Y).
  - path(X,Y) \( \leftarrow \) edge(X,Z), path(Z,Y).

  - If a definition does not at all depend on some variable in a body literal, it is often written as \( x^* \) (don’t care)
    - \( p(X,Y) \leftarrow r(X,Z), q(Y,Z), ... \)
5.2 Datalog Syntax

- **Problem** of variables in heads of rules
  - Consider a rule \( p(X) \leftarrow r(Y) \).
  - What does it mean?
    - If there is a substitution for \( Y \) making \( r(Y) \) true, then \( p(X) \) is true for all possible substitutions for \( X \)?
    - If \( r(Y) \) is true for all possible substitutions of \( Y \), then \( p(X) \) is true?
    - Or only for \( Y=X \)?
- **Restriction:** In Datalog all variables used in a head predicate always have to occur in some body literal, too.
  - Similar problem arises, if a constant in the head would depend on variables in body literals: \( p(a) \leftarrow r(X) \).

5.2 Datalog Syntax

- A **database query** is defined as
  - \( ?L_1, \ldots, L_n \) with literals \( L_i \in L_\mathcal{L}, n>0 \)
    - Alternative notation \( \leftarrow L_1, \ldots, L_n \) or \( :: L_1, \ldots, L_n \).
  - A query with only atomic literals is called **definite**.
  - A definite query with \( n=1 \) is called a **Datalog query**.
- Why is this a query?
  - A set of DB-clauses \( \mathcal{W} \) and a query \( Q \equiv L_1, \ldots, L_n \) are unsatisfiable, iff \( \mathcal{W} \models \neg Q \) with \( \neg Q \equiv \exists (L_1 \land \ldots \land L_n) \).

5.2 Datalog Syntax

- **Example database**
  - parent(John, Mary). parent(John, Thomas). ...
  - female(Mary). female(Sonja). ...
  - grandmother(X,Y) \leftarrow parent(X,Z), parent(Z,Y), female(Y).
- **Example datalog query**
  - Who is John’s grandmother?
    - \(?grandmother(John,X)\).
    - grandmother(John, Sonja).
    - grandmother(John, Karen).

5.2 Datalog Syntax

- **Example database**
  - parent(John, Mary). parent(John, Thomas). ...
  - female(Mary). female(Sonja). ...
  - grandmother(X,Y) \leftarrow parent(X,Z), parent(Z,Y), female(Y).
- **Example datalog query**
  - Who is John’s grandmother?
    - \(?grandmother(John,X)\).
    - grandmother(John, Sonja).
    - grandmother(John, Karen).

5.3 Datalog Programs

- If predicate symbols defining facts never occur in the head of any rule, a set of DB-clauses is called a **Datalog\(^\text{neg}\)-program**
  - This name follows the idea of logic programming
  - There are different kinds of programs…

5.3 Datalog Language Classes

- Depending on the use of **functions** and **negation** several Datalog language classes can be distinguished
  - **Datalog\(^\text{neg}\)** programs do not contain function symbols
  - **Datalog\(^f\)** programs (or definite programs) do not contain negative literals
  - **Datalog** programs contain neither negative literals nor function symbols
5.3 Datalog Language Classes

- Expressiveness

Datalog\(^f\) \, \text{neg} \, \text{Datalog}^- \, \text{Datalog}^f

5.3 Program Classes

- Datalog programs can also be distinguished by their dependencies between predicates
  - We have seen already that negation in literals may sometimes lead to strange results...
    - Remember: closed world assumption
    - Idea: find out about the relation between different predicates by examining their respective definitions

5.3 Program Classes

- The program connection graph (PCG) of some program P consists of
  - Nodes for each predicate symbol p in P
  - Directed edges from node p to node q, if q is in the definition of p
  - An edge is negative, if q occurs in a negated literal, otherwise the edge is positive
- A recursive clique is a maximum subset of the predicates in P, such that between each two predicate symbols there is a path in the PCG

5.3 Program Classes

- A program is called hierarchic, if the PCG does not contain cycles
  - If there are cycles the program is called recursive
  - bachelor(X) ← male(X), ¬ married(X). is hierarchic

5.3 Program Classes

- A program is called stratified, if cycles in the PCG only consist of positive edges
  - goodpath(X, Y) ← path(X, Y), ¬ toll(X).
  - goodpath(X, Z) ← goodpath(X, Y), goodpath(Y, Z). is a stratified and recursive program

5.3 Program Classes

- path(X, Z) ← edge(X, Y), path(Y, Z). is recursive
- p(X, Y) ← q(Y, Z), s(Z).
  q(X, Y) ← r(Y), s(X).
  r(X) ← p(X, X). is also recursive
5.3 Program Classes

- even(X) ← number(X), ¬ odd(X).
- odd(X) ← number(X), ¬ even(X).

is a not stratified and recursive program

5.3 Stratification

- A stratification of some program \( P \) is a disjoint partitioning \( P = P_1 \cup ... \cup P_n \) of \( P \) into program parts (strata) such that
  - The definition of each predicate symbol is a subset of some stratum
  - The definition of a predicate symbol in a positive body literal of a DB-clause in \( P_i \) is part of a \( P_j \) with \( j \leq i \)
  - The definition of a predicate symbol in a negative body literal of a DB-clause in \( P_i \) is part of a \( P_j \) with \( j < i \)

5.3 Stratification

- Basic idea: layer the program such that definitions of negatively used predicates are always already given in previous layers
  - This effectively excludes the use of negation within recursion

- It can be proved that a program is stratified, if and only if it has a stratification

5.3 Stratification

- Stratification Algorithm
  - Takes a Datalog\textsuperscript{neg} program as input and outputs either the stratification or ‘not stratified’
  - Thus, the problem of stratification is syntactically decidable

- Initialization:
  - For each predicate symbol \( p \) do \( \text{stratum}[p] := 1 \)
  - \( \text{maxstratum} := 1 \)

- Output:
  - If \( \text{maxstratum} > \# \) predicates
    - then return ‘not stratified’
    - else for \( i := 1 \) to \( \text{maxstratum} \) do
      - for each \( \text{DB-clause} \) with head predicate \( p \) do
        - for each negative body literal with predicate \( q \) do
          - \( \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q] + 1) \)
        - for each positive body literal with predicate \( q \) do
          - \( \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q]) \)
      - \( \text{maxstratum} := \max(\text{maxstratum}, \# \text{predicates}) \)
      - until \( \text{maxstratum} > \# \) predicates
      - or the \( \text{stratum} \) function becomes stable

- For each \( \text{DB-clause} \) with head predicate \( p \) do
  - for each negative body literal with predicate \( q \) do
    - \( \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q] + 1) \)
  - for each positive body literal with predicate \( q \) do
    - \( \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q]) \)
  - \( \text{maxstratum} := \max(\text{maxstratum}, \# \text{predicates}) \)
  - until \( \text{maxstratum} > \# \) predicates
  - or the \( \text{stratum} \) function becomes stable

- For each \( \text{DB-clause} \) with head predicate \( p \) do
  - for each negative body literal with predicate \( q \) do
    - \( \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q] + 1) \)
  - for each positive body literal with predicate \( q \) do
    - \( \text{stratum}[p] := \max(\text{stratum}[p], \text{stratum}[q]) \)
  - \( \text{maxstratum} := \max(\text{maxstratum}, \# \text{predicates}) \)
  - until \( \text{maxstratum} > \# \) predicates
  - or the \( \text{stratum} \) function becomes stable
5.3 Stratification

- Example
  \[
  \text{goodpath}(X, Y) \leftarrow \text{path}(X, Y), \neg \text{toll}(X).
  \text{goodpath}(X, Z) \leftarrow \text{goodpath}(X, Y), \text{goodpath}(Y, Z).
  \]

- Initialization:
  \[
  \text{stratum[goodpath]} = \text{stratum[path]} = \text{stratum[toll]} := 1
  \]
  \[
  \text{maxstratum} := 1
  \]

- First loop (maxstratum = 1):
  first rule: \[
  \text{stratum[goodpath]} := \max(\text{stratum[goodpath]}, \text{stratum[path]} + 1) = 2
  \]
  second rule: \[
  \text{stratum[goodpath]} := \max(\text{stratum[goodpath]}, \text{stratum[toll]} + 1) = 2
  \]
  \[
  \text{maxstratum} := \text{stratum[goodpath]} = 2
  \]

- Second loop (maxstratum = 2):
  results in no more changes to the strata and the algorithm terminates with maxstratum < 3

- Hence the program is stratified and
  \[
  P1 := \{\text{def(path)}, \text{def(toll)}\}
  \]
  \[
  P2 := \{\text{def(goodpath)}\}
  \]

5.3 Stratification

- Example
  \[
  \text{goodpath}(X, Y) \leftarrow \text{path}(X, Y), \neg \text{toll}(X).
  \text{goodpath}(X, Z) \leftarrow \text{goodpath}(X, Y), \text{goodpath}(Y, Z).
  \]

- First loop (maxstratum = 1):
  first rule: \[
  \text{stratum[goodpath]} := \max(\text{stratum[goodpath]}, \text{stratum[path]} + 1) = 2
  \]
  second rule: \[
  \text{stratum[goodpath]} := \max(\text{stratum[goodpath]}, \text{stratum[toll]} + 1) = 2
  \]
  \[
  \text{maxstratum} := \text{stratum[goodpath]} = 2
  \]

- Second loop (maxstratum = 2):
  results in no more changes to the strata and the algorithm terminates with maxstratum < 3

- Hence the program is stratified and
  \[
  P1 := \{\text{def(path)}, \text{def(toll)}\}
  \]
  \[
  P2 := \{\text{def(goodpath)}\}
  \]

5.3 DES

- In this detour, we will show an example implementation of Datalog
  \[
  \text{DES: Datalog Educational System}
  \]
  \[
  \text{http://des.sourceforge.net}
  \]

5.3 DES

- Write some Datalog program in a text file editor

father(herrick, jenny).
mother(dorthy, jenny).
father(brant, herrick).
mother(angela, herrick).
father(louis, dorthy).
mother(sad, dorthy).
father(hubert, thekla).
mother(jenny, thekla).

parent(X, Y) : father(X, Y).
parent(X, Y) : mother(X, Y).
ancestor(X, Y, D) : parent(X, Y).
ancestor(X, Y, D) : ancestor(X, D1), ancestor(Z, Y, D2), D is D1 + D2.
5.3 DES

- Load (consult the program) the program
  - consult command
  - Filename relative to DES installation directory

- Review the program
  - listing command

- Run some queries
  - List all ancestors of Jenny with a distance of 2 (grandparents)
  - Is Thekla an ancestor of Jenny?

- Add some new rules from the shell
  - New rule with assert command

Next Lecture

- Semantics of Datalog
- Evaluation