Recursive Query Processing:
Naïve and semi-naïve query processing
Stratification for negation & recursion

Lecture 2-3

Lois Delcambre

(some slides from Dave Maier)

Naïve Evaluation of Recursive Datalog (without negation)

Naïve Evaluation of P

A “bottom-up” or “right-to-left” method.

Have a current set of facts f (a partial model). Use clauses in P in a right-to-left manner to derive additional facts that must also be in a minimum model for P.

Immediate Consequence Operator

If P is a Datalog program, the *immediate consequence operator* $I_P(f)$ computes all facts derivable from the set of facts f by a single application of the clauses in P.

Note: If f is a subset of the minimum model of P, then so is $I_P(f)$, $I_P(I_P(f))$, $I_P(I_P(I_P(f)))$, …

So start the process with f = ∅, because we know we have a subset of the minimum model.

Inflationary semantics of program P: Start with empty set of facts, apply I_P until no change.

Naïve Evaluation

Naïve(P, q)

$f_{old} = \emptyset$

$f = I_P(f_{old})$

while $f \neq f_{old}$ **do**

$f_{old} = f$

$f = I_P(f)$

return match(q, f)

Naïve always halts on programs that don't use computed predicates, with f as the minimum model.

Working through naïve evaluation (1)

f0 = ∅

```
Local(Id, Loc, 1) :- Local0(Id, Loc).
Local(Id, Loc, K) :- Op1(Id, In, Loc),
   Local(In, Loc, M), Sum(M, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   Local(In1, Loc, M), Local(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

Working through naïve evaluation (2)

```
Local(Id, Loc, 1) :- Local0(Id, Loc).
Local(Id, Loc, K) :- Op1(Id, In, Loc),
   Local(In, Loc, M), Sum(M, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   Local(In1, Loc, M), Local(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

f1 = I_P(f0) =

```
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

Working through naïve evaluation (3)

```
Local(Id, Loc, 1) :- Local0(Id, Loc).
Local(Id, Loc, K) :- Op1(Id, In, Loc),
   Local(In, Loc, M), Sum(M, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   Local(In1, Loc, M), Local(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

f2 = I_p(f1) =

```
Local(r, node1, 1).
Local(s, node1, 1).
Local(q, node1, 1).
Local(u, node2, 1).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

Working through naïve evaluation (4)

```
Local(Id, Loc, 1) :- Local0(Id, Loc).
Local(Id, Loc, K) :- Op1(Id, In, Loc),
   Local(In, Loc, M), Sum(M, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   Local(In1, Loc, M), Local(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

f3 = I_p(f2) =

```
Local(r, node1, 1).
Local(s, node1, 1).
Local(q, node1, 1).
Local(u, node2, 1).
Local(s1, node1, 2).
Local(u1, node1, 3).
Local(s2, node2, 2).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

Working through naïve evaluation (5)

```
Local(Id, Loc, 1) :- Local0(Id, Loc).
Local(Id, Loc, K) :- Op1(Id, In, Loc),
   Local(In, Loc, M), Sum(M, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   Local(In1, Loc, M), Local(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

f4 = I_p(f3) =

```
Local(r, node1, 1).
Local(s, node1, 1).
Local(q, node1, 1).
Local(u, node2, 1).
Local(s1, node1, 2).
Local(u1, node1, 3).
Local(j1, node1, 6).
Local(s2, node2, 2).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

Working through naïve evaluation (6)

```
Local(Id, Loc, 1) :- Local0(Id, Loc).
Local(Id, Loc, K) :- Op1(Id, In, Loc),
   Local(In, Loc, M), Sum(M, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   Local(In1, Loc, M), Local(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

f5 = I_p(f4) = f4

```
Local(r, node1, 1).
Local(s, node1, 1).
Local(q, node1, 1).
Local(u, node2, 1).
Local(s1, node1, 2).
Local(u1, node1, 3).
Local(j1, node1, 6).
Local(s2, node2, 2).
Local0(r, node1).
Local0(s, node1).
Local0(q, node1).
Local0(u, node2).
Op1(s1, r, node1).
Op1(s2, u, node2).
Op2(j1, s1, u1, node1).
Op2(u1, s, q, node1).
Op2(j2, j1, s2, node1).
```

Semi-Naïve Evaluation of Datalog

Semi-Naïve Evaluation

Naïve evaluation discovers the same fact over and over.

E.g., derive `local(r, node1, 1)` repeatedly

Idea: When evaluating a rule, make sure we use at least one newly discovered fact.

Otherwise, we derive a fact we've already seen.

Notice: after f1, only new facts are `local` facts. (Only need to work on recursive intensional predicates.)

Example: rediscovering facts vs. discovering new facts (using Naïve)

Consider the rule

```
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
  Local(In1, Loc, M), Local(In2, Loc, N),
  Sum(M, N, J), Sum(J, 1, K).
```

When computing f4 = I_P(f3).

If we use facts `Op2(u1, s, q, node1),`
`Local(s, node1, 1),Local(q, node1, 1)`, we derive **`Local(u1, node1, 3)`**, which we already had

If we use `Op2(j1, s1, u1, node1),`
`Local(s1, node1, 2),Local(u1, node1, 3)`, we get **`Local (j1, node2, 6)`**, which is new

Implementing Semi-Naïve:
use deltas (new facts from this round)

Suppose at each step in Naïve we determine

NewLocal(Id, Loc, K) if **Local(Id, Loc, K)** in $f - f_{old}$.

Replace this rule:

```
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
     Local(In1, Loc, M), Local(In2, Loc, N),
     Sum(M, N, J), Sum(J, 1, K).
```

With these rules:

```
Local(Id, Loc, K) :- Op1(Id, In, Loc),
   NewLocal(In, Loc, M), Sum(M, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   NewLocal(In1, Loc, M), Local(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local(Id, Loc, K) :- Op2(Id, In1, In2, Loc),
   Local(In1, Loc, M), NewLocal(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
Local(id, Loc, K) :- Op2(Id, In1, In2, Loc),
   NewLocal(In1, Loc, M), NewLocal(In2, Loc, N),
   Sum(M, N, J), Sum(J, 1, K).
```

One Other Modification for Semi-Naive

If P' is the modified program, then instead of

$f = I_P(f)$

we need

$f = I_{P'}(f) \cup f$

Note: Semi-Naïve doesn't guarantee we won't generate a fact at multiple steps, if there are distinct "proofs" of it, i.e., two ways to generate it.

```
MajReq(eng, wr310). MajReq(eng, wr408).
Prereq(wr309, wr310).
Prereq(wr401, wr408).
Prereq(wr309, wr401).
```

Dependency Graphs for Datalog Programs

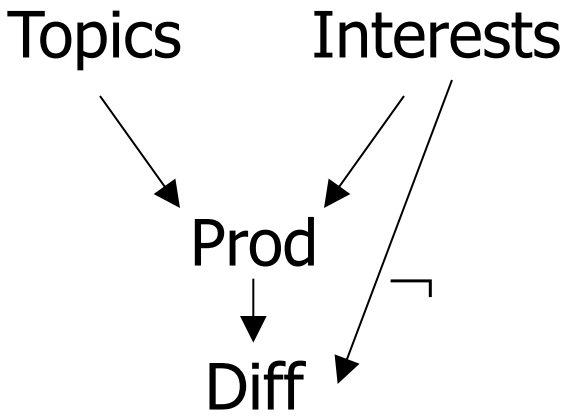
Datalog program no recursion with negation: dependency graph

Given a DB with two relations:
Topics(Topic) and Interests(Person,Topic)

What is this query computing?

Diff(a) :- Prod(a,b), ¬Interests(a,b).

Prod(a,b) :- Interests(a,c), Topics(b).



Create one node for each body predicate.

Draw an arrow from body predicate
to corresponding head predicate.

Acyclic dependency graph ≡ not recursive.

It's useful to label arrows if predicate is negative.

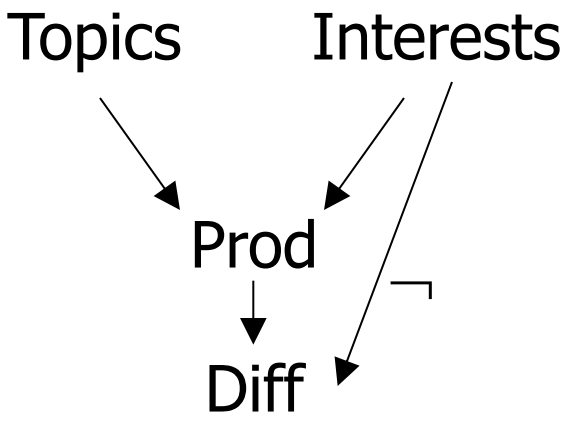
Datalog program no recursion with negation: dependency graph

Given a DB with two relations:
Topics(Topic) and Interests(Person,Topic)

What is this query computing?

Diff(a) :- Prod(a,b), ¬Interests(a,b).

Prod(a,b) :- Interests(a,c), Topics(b).



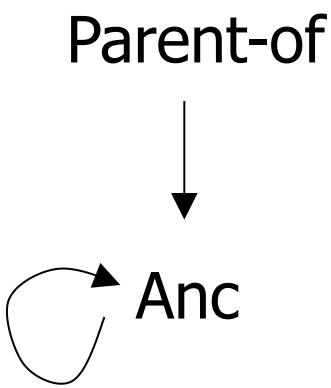
Evaluate predicates completely – according to the order of operations in the dependency graph.

Datalog with recursion without negation: dependency graph

Anc(x,y) :- Parent-of(x,y).

Anc(x,z) :- Anc(x,y), Parent-of(y,z).

Recursive Datalog program
has a cycle in the dependency graph.



Datalog with recursion without negation

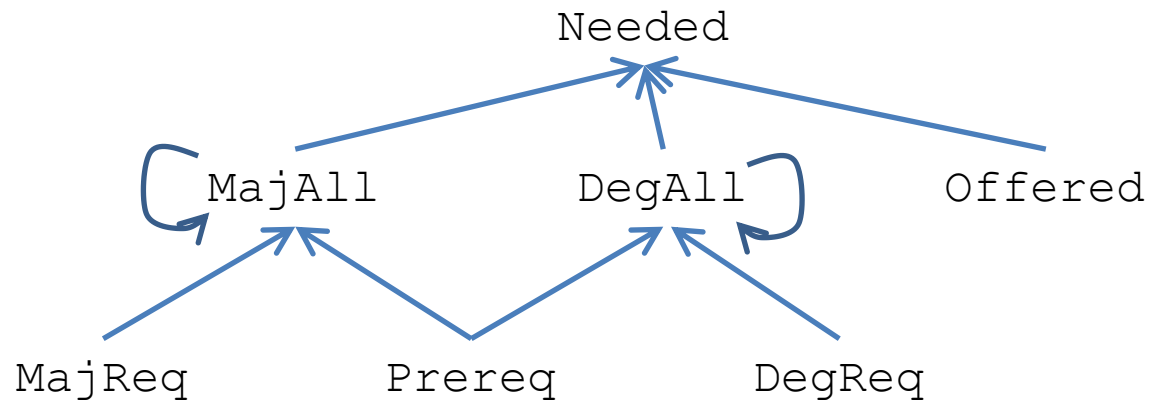
```
Needed(D, M, C) :- MajAll(M, C), Offered(D, M).
Needed(D, M, C) :- DegAll(D, C), Offered(D, M).

MajAll(M, C) :- MajReq(M, C).
MajAll(M, C) :- MajAll(M, C1), Prereq(C, C1).

DegAll(D, C) :- DegReq(D, C).
DegAll(D, C) :- DegAll(D, C1), Prereq(C, C1).
```

Here too … you can evaluate in order of dependency graph

- Compute `DegAll` from `DegReq` and `Prereq` and `DegAll`
- Compute `MajorAll` from `MajReq` and `Prereq`
- Compute `Needed` from `MajAll`, `DegAll`, `Offered`



Evaluating Datalog with Recursion and Negation

Now consider Datalog with recursion & negation

Person(Dan). (one tuple in base relation)

Student(x) :- Person(x), ¬Employee(x).

Employee(x) :- Person(x), ¬Student(x).

What is the query answer?

Either Student(Dan) or Employee(Dan), depending on which rule fires first.

Two different query answer!

We have a problem – for any data.

Student(x) :- Person(x), ¬Employee(x).

Employee(x) :- Person(x), ¬Student(x).

Imagine we have some number of persons in the database.

They will ALL end up at Students or as Employees, depending on which rule fires first.

Two different query answers!

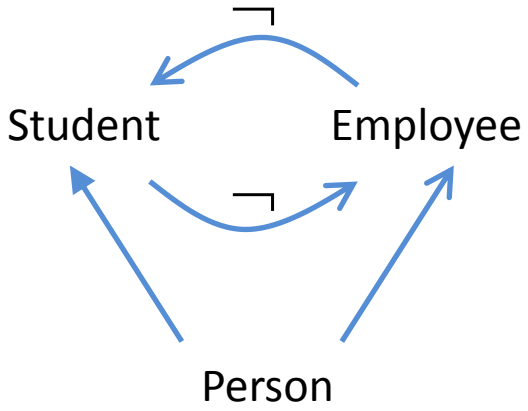
What does the dependency graph look like?

Person(Dan). (one tuple in base relation)

Student(x) :- Person(x), ¬Employee(x).

Employee(x) :- Person(x), ¬Student(x).

This is a recursive program.



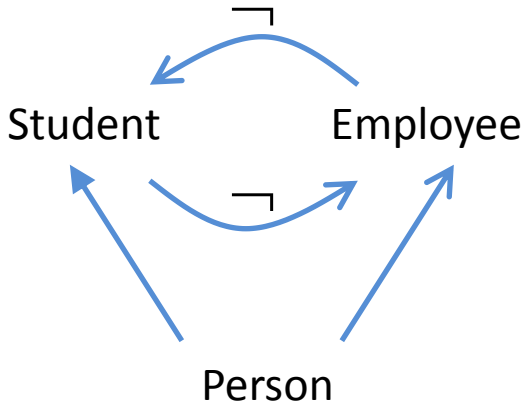
Now consider recursion & negation

Person(Dan). (one tuple in base relation)

Student(x) :- Person(x), ¬Employee(x).

Employee(x) :- Person(x), ¬Student(x).

We have a problem when a cycle in a dependency graph includes (at least) one negative edge.



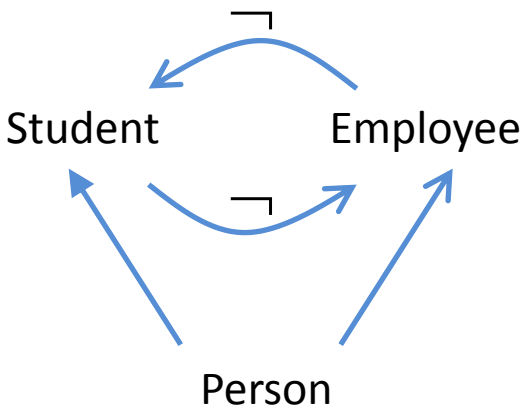
Cycle with one (or more) negative edges

Person(Dan). (one tuple in base relation)
Student(x) :- Person(x), ¬Employee(x).
Employee(x) :- Person(x), ¬Student(x).

Recursion (for Student, say) is attempting to add tuples to answer.

But it is subtracting Employee tuples (to find the Students).

And … Employees are being computed recursively at the same time.



Another example with negation & recursion

Node(a).
Node(b).
Node(c).
Node(d).
Arc(a,b).
Arc(c,d).
Start(a).

These are facts from the database.

Draw the dependency graph; label the negative edges.

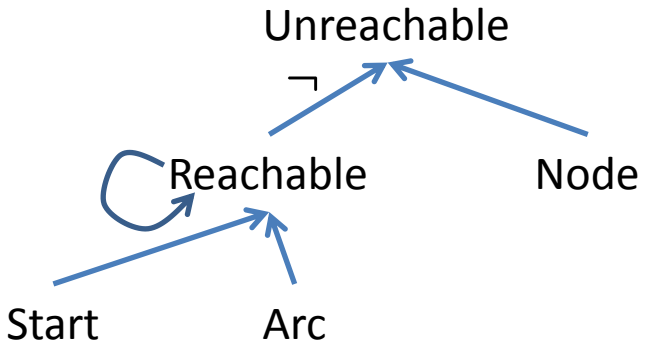
Reachable(z) :- Start(z).

Reachable(y) :- Reachable(x), Arc(x,y).

Unreachable(x) :- Node(x), ¬Reachable(x).

Dependency graph for this program; notice the negative edge

Node(a).
Node(b).
Node(c).
Node(d).
Arc(a,b).
Arc(c,d).
Start(a).



Reachable(z) :- Start(z).
Reachable(y) :- Reachable(x), Arc(x,y).
Unreachable(x) :- Node(x), ¬Reachable(x).

Stratification – for Datalog with negation and recursion

A stratification of a Datalog program P
is a partition of P into strata or layers where, for every rule, H :- A_1, …, A_m, $\neg B_1$, …, B_q

- The rules that define the predicate H are all in the same strata.
- For all positive predicates in this rule, their strata is less than or equal to the strata of this rule.
  $Strata(A_i) \leq Strata(H)$.
- For all negative predicates in this rule, their strata is strictly less than the strata of this rule.
  $Strata(B_i) < Strata(H)$.

Intuition behind stratification

- In any strata, compute a predicate – in its entirety.
- If you EVER use a predicate negatively, then make sure it is ALREADY, COMPLETELY computed.
- So … stratification prevents you from using recursion at the same time you're computing a predicate that is used negatively.

Exercise: Define a stratification for this datalog program.

Node(a).

Node(b).

Node(c).

Node(d).

Arc(a,b).

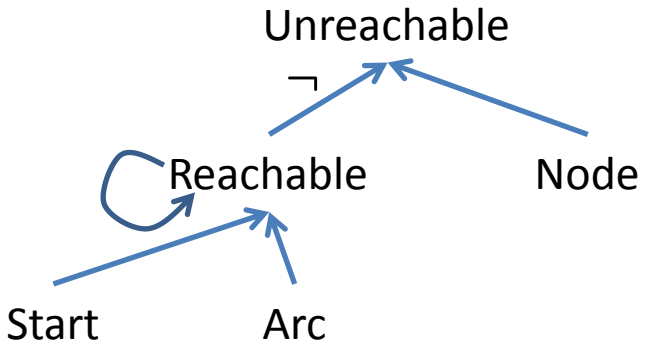
Arc(c,d).

Start(a).

Reachable(z) :- Start(a).

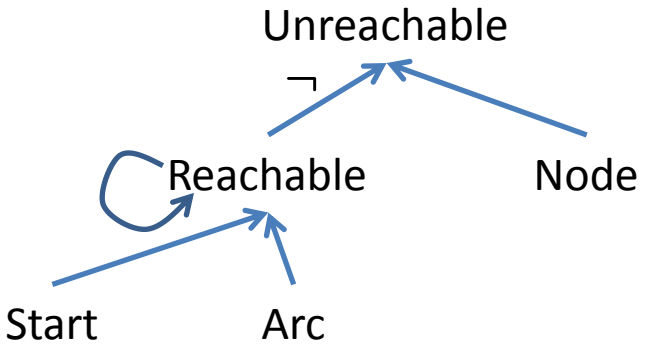
Reachable(y) :- Reachable(x), Arc(x,y).

Unreachable(x) :- Node(x), ¬Reachable(x).



Exercise: Define a stratification for this datalog program.

Node(a).
Node(b).
Node(c).
Node(d).
Arc(a,b).
Arc(c,d).
Start(a).



Compute all Reachable facts (positively) in Strata 1. Then, use Reachable negatively in Strata 2.

Reachable(z) :- Start(a).
Reachable(y) :- Reachable(x), Arc(x,y). Strata 1

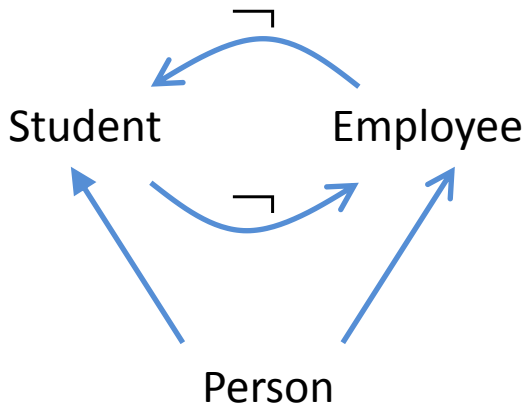
Unreachable(x) :- Node(x), ¬Reachable(x). Strata 2

Can you define a stratification for this program?

Person(Dan). (one tuple in base relation)

Student(x) :- Person(x), ¬Employee(x).

Employee(x) :- Person(x), ¬Student(x).

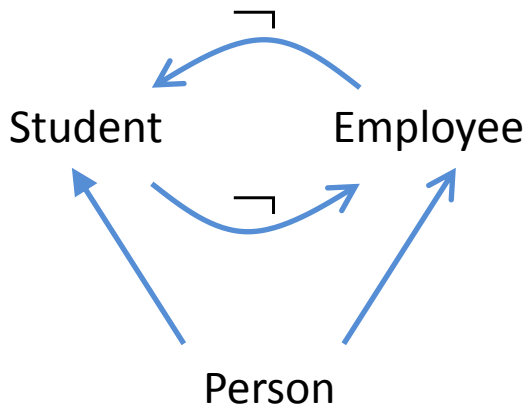


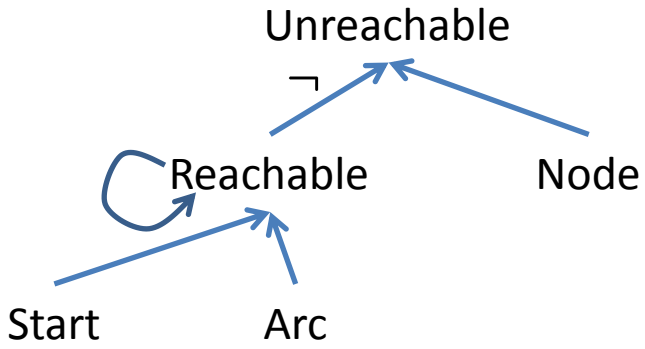
Compare these two dependency graphs

Person(Dan). (one tuple in base relation)

Student(x) :- Person(x), ¬Employee(x).

Employee(x) :- Person(x), ¬Student(x).





The one on left has a cycle with negation; the one on right doesn't. The on left can't be stratified.

Comments

- Stratified Datalog programs always have a unique answer.
- Note that Stratification forces the choice of one of various possible answers (models).
- Not all Datalog programs can be stratified.
- Some programs can be stratified in more than one way. If so, then all stratifications will produce the result.