Recursion vs Aggregation, Negation

- Negation before or after recursion: OK

(1) E(X,Y) :- A(X,Y), not B(X,Y). // $E := A \setminus B$
(2) TC(X,Y) :- E(X,Y).
(3) TC(X,Y) :- E(X,Z), TC(Z,Y). $\}$ $TC := E^+$
(4) nTC(X,Y) :- node(X), node(Y), not TC(X,Y).

node(x) ← E(x,_).
node(x) ← E(_,x)
(0)

"complement of TC ($=E^+$) ⇝ one cannot reach Y from X via $E^{(+)}$"

"Wrong schedule":
(1)
(4)
{(2), (3)} until no change

"correct schedule":
(0)
(1)
(2)
Repeat (3) until no change
(4)

Stratified Datalog:

when evaluating a rule with negation, e.g

A(..) :- B(..).., ¬C(..), ..

all rules defining C must have been applied before applying the rule with ¬C(..) in the body.

⇒ in Rule-Goal graph there must not be cycles with a negative edge.

This is not allowed:

W(X) ← m(X,Y), ¬ W(Y)

Stratified Datalog

Definition 5.1 (Stratification, S-Datalog) Let P be a Datalog$^{\neg}$ program and r, r' rules of P of the form

$$\begin{array}{lll} r: & \cdots & \leftarrow \cdots\ p(\ldots)\ \cdots \\ r': & p(\ldots) & \leftarrow \cdots \end{array}$$

Then r is said to *depend positively* on r', denoted $r' \rightarrow r$. If instead p occurs negated in the body of r, then r *depends negatively* on r', denoted $r' \stackrel{\neg}{\rightarrow} r$. The *dependency graph* $\mathcal{G}_P$ of P consists of the positive and negative dependencies $r' \stackrel{(\neg)}{\rightarrow} r$. We write $r' \rightsquigarrow r$, if there is a path from r' to r in $\mathcal{G}_P$, and $r' \stackrel{\neg}{\rightsquigarrow} r$, if the path involves at least one negative dependency. P is called *stratified* if there exists a partition $P = P_1 \dot\cup \ldots \dot\cup P_n$ such that for all $i, j = 1, \ldots, n$ and all $r' \in P_i$, $r \in P_j$:

- if $r' \rightsquigarrow r$ then $i \leq j$, and
- if $r' \stackrel{\neg}{\rightsquigarrow} r$ then $i < j$.

The sequence $P_1, \ldots, P_n$ is called a *stratification* of P with the *strata* P_i. By *S-Datalog* we denote the class of stratified Datalog$^{\neg}$ programs. □

Computing the Stratified Model

In S-Datalog programs, a rule r depends negatively only on rules from strictly lower strata, whereas it may depend positively also on rules from the same stratum. One can show that P is stratified iff $\mathcal{G}_P$ contains no *negative cycle* $r \stackrel{\neg}{\rightsquigarrow} r$.[2] The strata P_i are given by the *strongly connected components* (*scc*) of $\mathcal{G}_P$.[3] Clearly, no scc contains a negative edge (otherwise P would not be stratified). Therefore, a topological sort of the scc's yields the desired stratification $P_1, \ldots, P_n$.

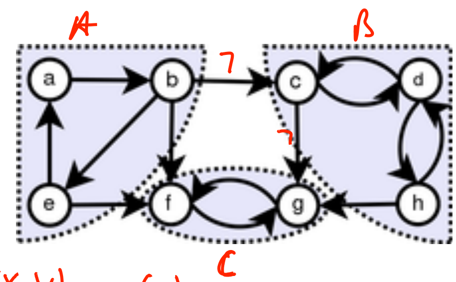
Let P be a S-Datalog program with n strata. For every database $\mathcal{D}$, $P \cup \mathcal{D}$ has a canonical model $\mathcal{S}_{P\cup\mathcal{D}}$ called the *stratified model* of $P \cup \mathcal{D}$ which is obtained by successively evaluating the fixpoints $T^{\omega}_{P_i}$ of the strata P_i as follows:

$$\begin{array}{lll} \mathcal{I}_0 & := & \mathcal{D}, \\ \mathcal{I}_i & := & \mathcal{I}_{i-1} \cup T^{\omega}_{P_i}(\mathcal{I}_{i-1}) \text{ for all } i = 1, \ldots, n, \\ \mathcal{S}_{P\cup\mathcal{D}} & := & \mathcal{I}_n. \end{array}$$

SCCs: Strongly Connected Components

- In the mathematical theory of directed graphs, a graph is said to be ***strongly connected*** if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph form a ***partition*** into ***subgraphs*** that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.

stratified



total order of SCCs: A > B > C

w(X) :- m(X,Y), ¬w(Y).

EXCURSION: Dealing with unstratifed Negation ➔ Well-founded Datalog

The intuition of the well-founded semantics can be explained by the following well-known game example.

Example 5.2 (Win-Move Game) The game is given by a set of positions and a set of moves between them (cf. Figure 5.1). There are two players moving alternately on the given move-graph. A player who cannot move loses. Hence, a position x is won, if there is a move to some position y which is lost (since then the opponent has to move). Clearly, positions without outgoing moves are immediately lost. Games of this general type are described by the following non-stratified program in an intuitive and declarative way:

P_{game}: win(X) ← move(X,Y), ¬win(Y).

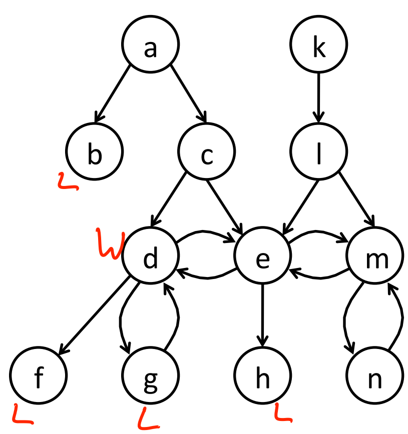
The rule states that win(X) holds if there is a move to Y such that ¬win(Y) holds. Consider, for example, the move-graph given by the database

$$\mathcal{D} = \{\text{move(a,b), move(b,a), move(b,c), move(c,d)}\}$$

Recursion vs Aggregation, Negation

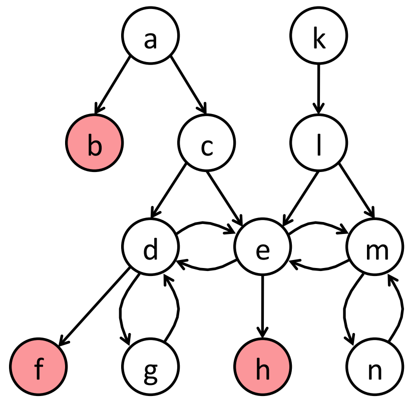
- Rule-goal graph has **no negative** cycles ➔
  - Can be "stratified" into layers (strata)
  - Evaluate lower strata, then move to higher ones
  - All recursion/loops are monotone
- But recursion "through negation" (or "through aggregation") is problematic!
  - Rule-goal graph has **negative cycles**
  - p(X) :- q(X), not p(X) … madness …
    - What does this rule even mean? If p(X) isn't true, then it is true?
  - win(X) :- move(X,Y), not win(Y) … (sanity:)
    - On the other hand: this rule makes some sense! Computes whether X is won (or lost/drawn) in a game defined by move(X,Y)

A Game

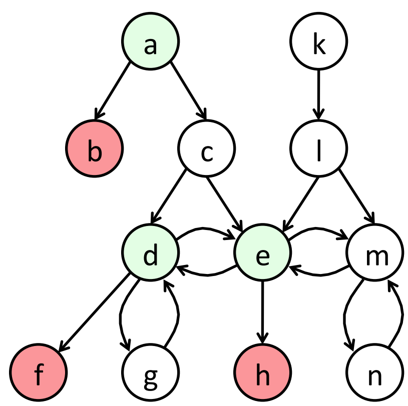


All successors **won** ➔ position **lost**
Some successor **lost** ➔ position **won**

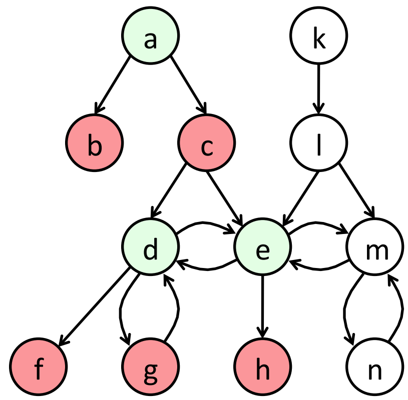
Solving the Game



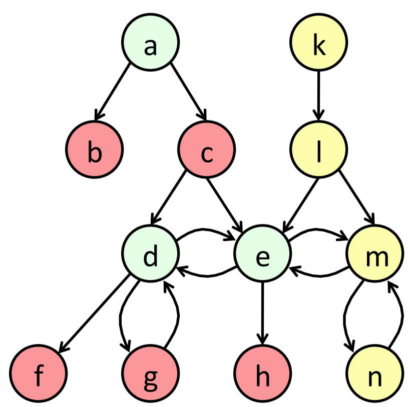
Solving the Game



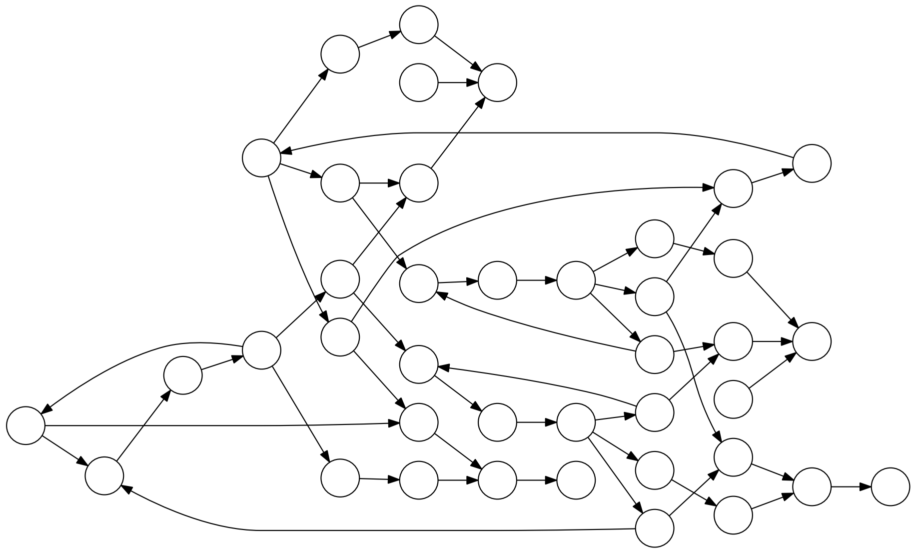
Solving the Game



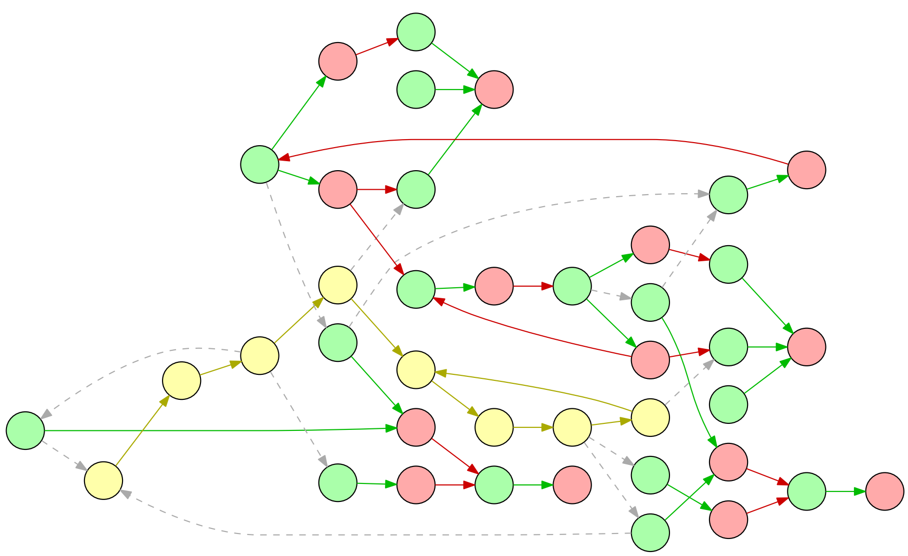
Solving the Game



Unsolved Game



Solved Game



Datalog Semantics & Evaluation

- Model-theoretic
  - View program P as a set of logic formulas F
  - Consider the (minimal!) models of F
  - Not covered in this class
- Proof-theoretic
  - Look at the rules as axioms, then find proof trees that give you the desired answers
  - Similar to Prolog's backtracking (SLD-NF resolution)
  - Not covered in this class
- Fixpoint semantics
  - Evaluate the rules "bottom-up"
  - Let's look at this!

Back to SQL … (now with recursion)

- Recall sub-queries:

```
SELECT *
FROM Wine
WHERE Year + 2 >=
    (SELECT avg(Year) from Wine)
AND year - 2 <=
    (SELECT avg(Year) from Wine)
```

- Named subqueries to the rescue!

WITH <*subqname*> (<*list-of-columns*>) **AS**
(<*query-expression*>)
<*QUERY*>

… back to SQL

- Named sub-queries to the rescue!

WITH *subqname* (<*list-of-columns*>) AS
(<*query-expression*>)
QUERY

```
WITH Age(Average) AS
    (SELECT avg(Year) FROM Wine)
SELECT *
FROM Wine, Age
WHERE Year -2 <= Average
AND Year + 2 >= Average
```


This comes handy for SQL with Recursion!

- SQL-99 WITH Statement:

WITH R1 AS (*query*),
R2 AS (*query*),
…,
Rn AS(*query*)
<*Query involving* R1, …, Rn *and other rels*>

- Idea:
  - use above named query + RECURSIVE keyword

PostgreSQL + Recursion: Summary

- PostgreSQL like SQL99 standard supports linear recursion
- Useful in many applications (e.g. graph queries, bill-of-materials, etc)
  - Alternatives aren't pretty (embedded SQL, and/or triggers, etc)
- Termination can be an issue!
  - UNION vs UNION ALL
  - Limit k no duplicate elim. can lead to non-termination
    - doesn't always work, e.g. today's discussion; another variant: db-class.org video

ECS-165B 19