CDCL
Part 2: Implementation Details

CIS700 — Fall 2023

Kris Micinski

Last week, looked at CDCL, zChaff.

Studied CDCL basics: like DPLL, but includes clause learning.

Today: how do we **justify** clause learning conceptually?
Also: study *implementation details* of modern CDCL solvers.

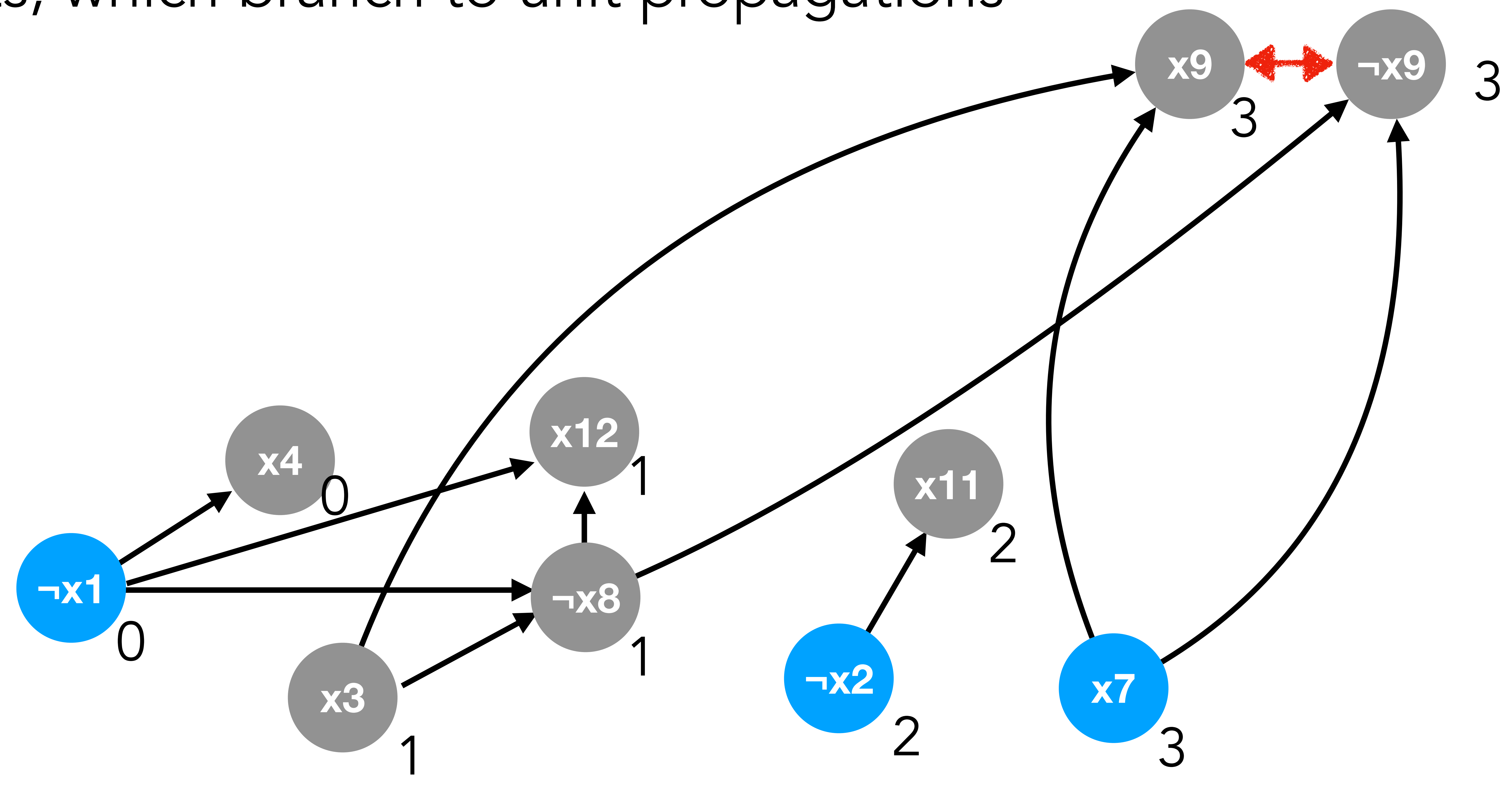
Algorithm 1 Typical CDCL algorithm

```
CDCL(φ, ν)
if (UnitPropagation(φ, ν) == CONFLICT)
    then return UNSAT
dl ← 0                                          ▷ Decision level
while (not AllVariablesAssigned(φ, ν))
      do (x, v) = PickBranchingVariable(φ, ν)                  ▷ Decide stage
         dl ← dl + 1                            ▷ Increment decision level due to new decision
         ν ← ν ∪ {(x, v)}
         if (UnitPropagation(φ, ν) == CONFLICT)                ▷ Deduce stage
           then β = ConflictAnalysis(φ, ν)                     ▷ Diagnose stage
                if (β < 0)
                  then return UNSAT
                  else Backtrack(φ, ν, β)
                       dl ← β                   ▷ Decrement decision level due to backtracking
return SAT
```

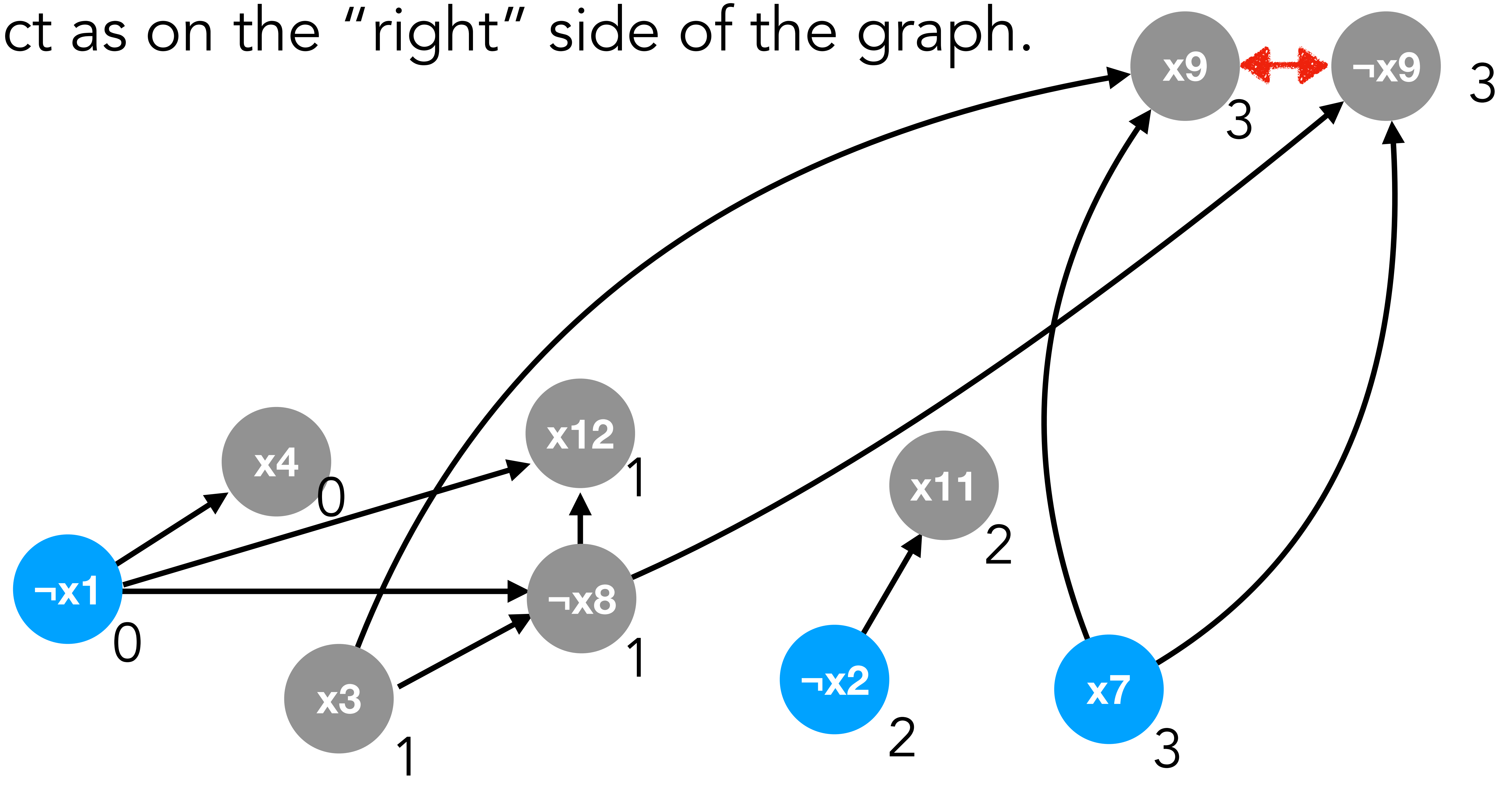
[Joao Marques-Silva, Ines Lynce and Sharad Malik From the "Handbook of Satisfiability]

Recall the implication graph, where vertices are literals

Decisions are roots, which branch to unit propagations

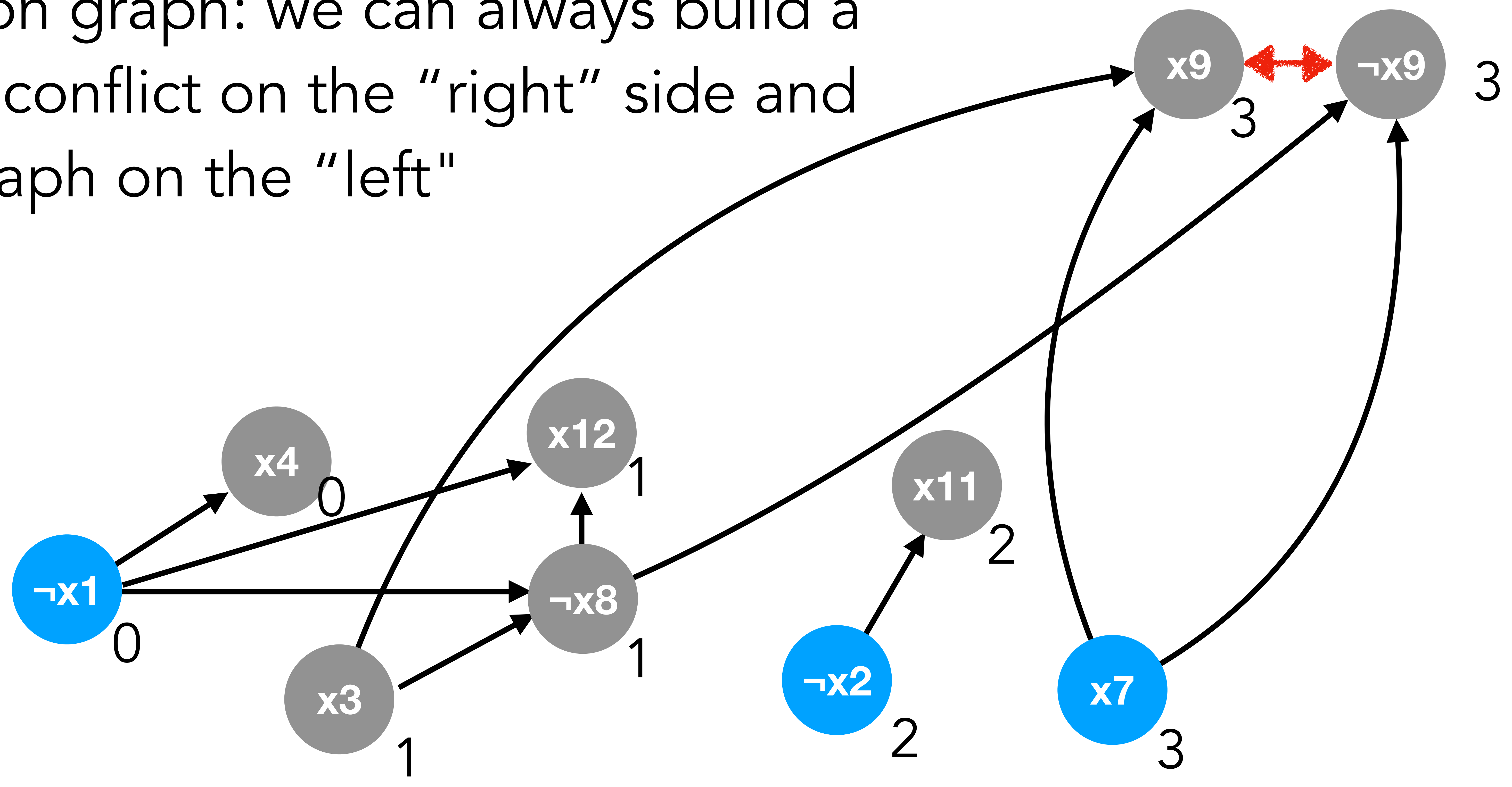


The conflict is always the "most recent" thing that happened in the graph. So WLOG, we can always visualize the conflict as on the "right" side of the graph.

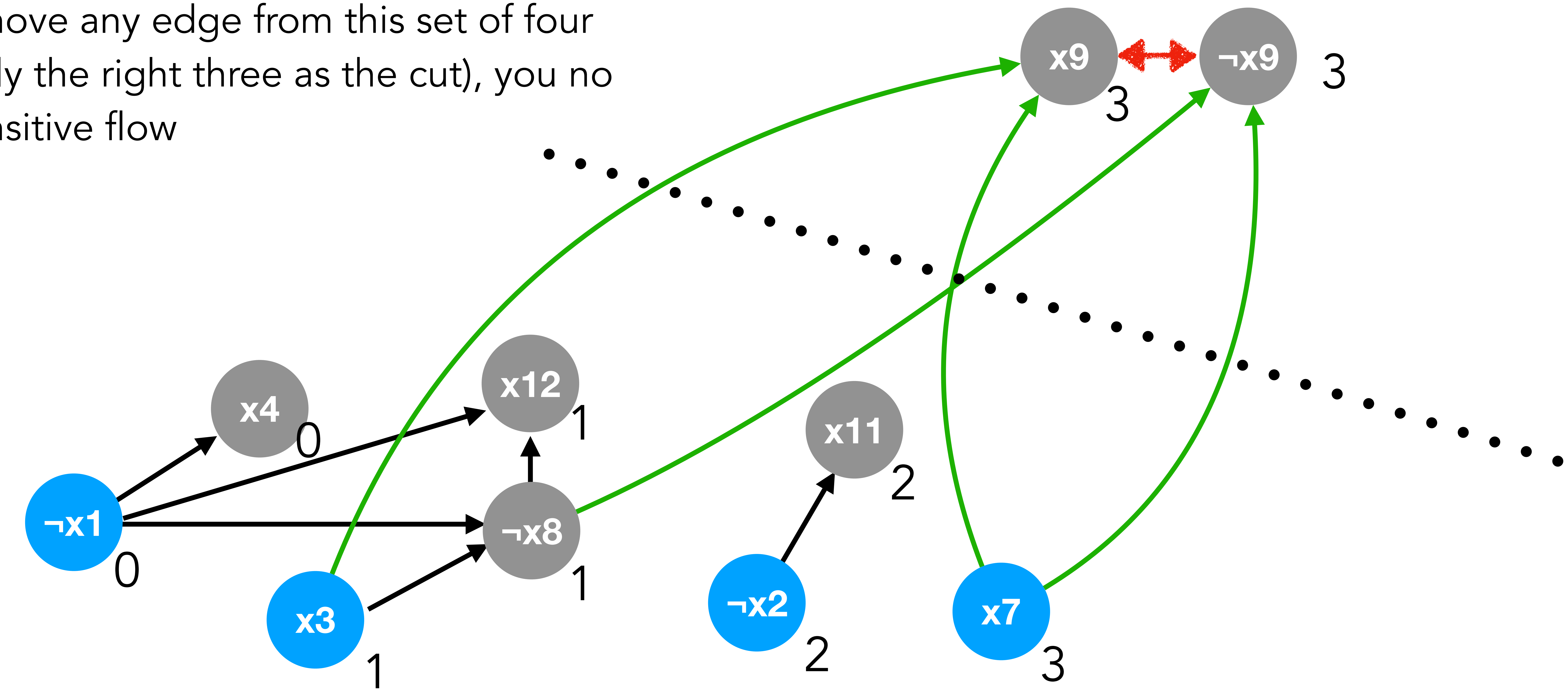


A **cut** is a set of edges which—when removed—break reachable flows.

For the implication graph: we can always build a cut that puts the conflict on the “right” side and the rest of the graph on the “left"

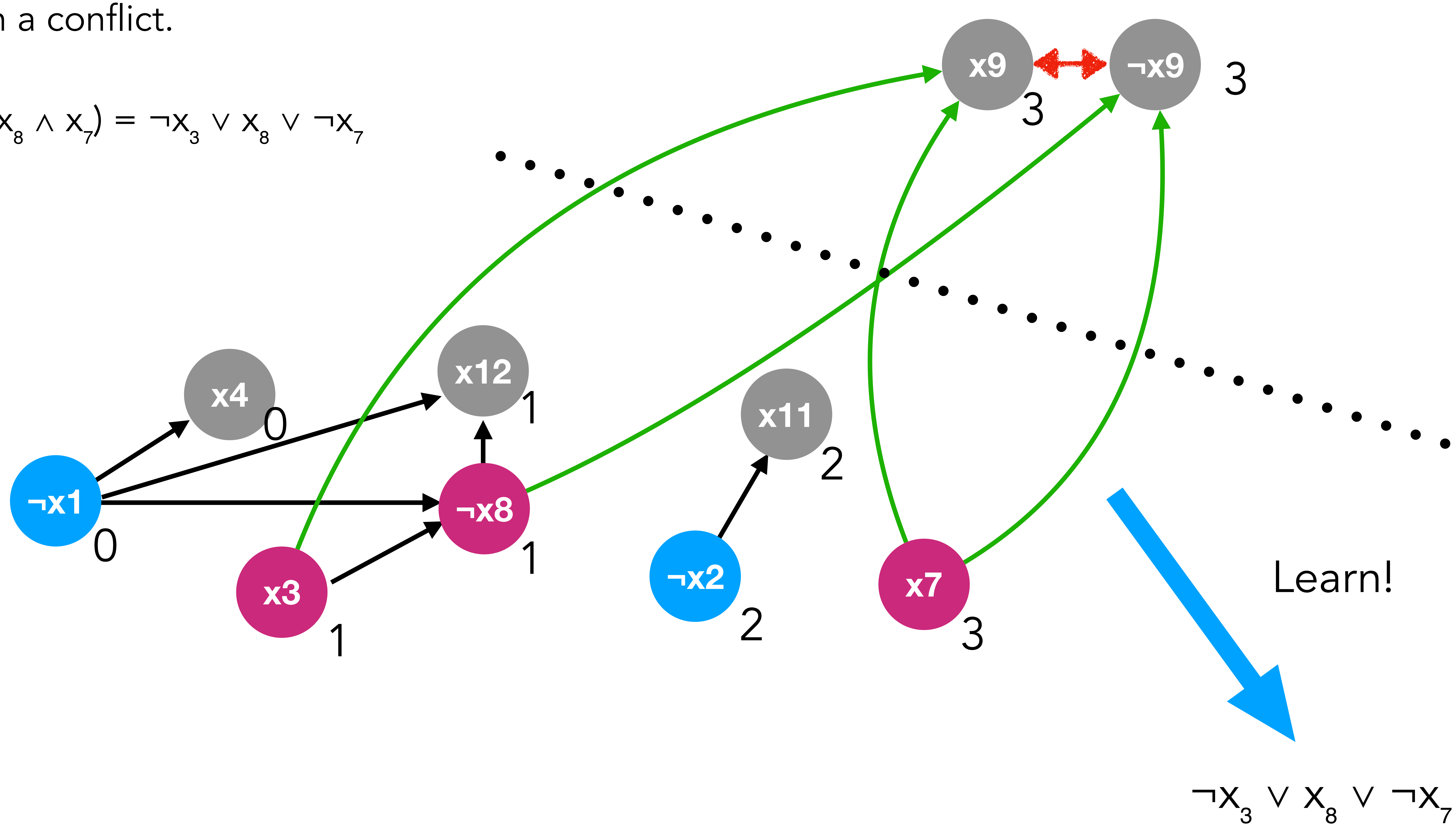


For example, these green edges form a cut in the implication graph.
Notice that if you remove any edge from this set of four (for example, take only the right three as the cut), you no longer cut off the transitive flow



Intuitively, we can look back at these roots and say: whenever each of these things is true, we know we're destined to end up with a conflict.

Thus, we ***learn*** $\neg(x_3 \wedge \neg x_8 \wedge x_7) = \neg x_3 \vee x_8 \vee \neg x_7$



UIP — Unique Implication Point

A Unique Implication Point in an implication graph is a dominator for a conflict. I.e., a node x such that all paths eventually reaching the conflict must go through x

Most modern CDCL-based solvers use UIP cuts

UIP — Unique Implication Point

A Unique Implication Point in an implication graph is a dominator for a conflict. I.e., a node x such that all paths eventually reaching the conflict must go through x

Most modern CDCL-based solvers use UIP cuts because they produce unit clauses after backtracking

Non Chronological Backtracking

Let's say we're at decision level 10, and we learn the following clause:

$\neg x_7@3 \lor x_3@5 \lor \neg x_5@10$

We add this clause to our database. Then, instead of backtracking to try $x_5@10$ (as DPLL would), we instead backtrack to decision level 5! Now, by construction, the clause is ***unit*** and we can propagate $\neg x_5$

In general the trick is this: take your learned clause, backtrack to the second-highest decision level. When you do that, you know all of the variables (except the one from the most-recent decision level) must be false (you wouldn't have kept propagating if one had been true!)

This strategy produces ***asserting literals***. Upon backtracking, there is guaranteed to be a unit clause which will allow unit propagation to occur. This lowers the burden of *guessing* on the search.

Visualization of CDCL

I will defer to these excellent notes with visualizations of UIPs, UIP cuts, and their associated learned clauses

https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html

SAT Solver Internals

UIP cuts in the implication graph are a beautiful theory for understanding how to justify the correctness of learned clauses.

Unfortunately, they don't really tell us how to implement things. Modern solvers do not literally materialize an implication graph—calculating dominators on-the-fly is quite laborious and compute-intensive.

There are some key tricks the solver exploits.

Watchlist Tricks

MiniSAT and others use representational tricks, i.e., vectors to represent clauses with watch literals in a canonical location

We will discuss these tricks on Thursday's class.

Analyzing the Trail

As the solver does its work, it builds up a **decision trail**, which could be (in principle) extended to an implication graph.

Surprisingly, it is possible to inspect ***only*** the trail to construct the learned clause, without materializing the implication graph or calculating UIPs via dominators.

Algorithm for Learning via UIPs

"In a breadth-first manner, continue to trace literals of the current decision level, until there is just one left."

```
Input: confl – the conflicting clause,
       reason – mapping from vars to clauses

Outputs: out_clause, the output clause; out_btlevel, the bt level
seen-vars = {}
counter = 0
lit p = ⊥
do
  // initially when p is ⊥, reason returns each lit
  p_reason = confl.reason(p) // returns the "reason" vector
  // For each literal in the reason vector…
  for (int j = 0; j < p_reason.size(); j++) {
    lit q = p_reason[j];
    if (var(q) ∉ seen-vars):
      Add var(q) to seen-vars
      If decision level of q is current decision level:
        counter++
      Else if (decision level of q is > 0):
        Push ¬q onto the learned clause
  // Select next literal
  do
    p = trail.last
    confl = reason[var(p)]
    undoOne(); // Pop one decision from the trail
  while (p ∉ seen-vars)
  counter -= 1;
while (counter > 0)
out_clause[0] = ¬p
```

- Initially, p = ⊥, which sets p_reason to the conflict clause
- E.g., if the conflict is $\neg x_3 \vee x_5 \vee \neg x_8$
- p_reason is $x_3 \vee \neg x_5 \vee x_8$
- For each literal q in p_reason:
  - Mark q as seen
  - If q comes from the current level, bump count & exclude it from learned clause
  - If q was at DL > 0, add ¬q to output
  - out_level is max of old level and q's DL
- Throw away decisions until you hit one of the "seen" ones, call that p
- At the end of everything there is **one** (asserting) literal at the current decision level: p
  - So set out_clause[0] = ¬ p

```
Input: confl — the conflicting clause,
       reason — mapping from vars to clauses

Outputs: out_clause, the output clause; out_btlevel, the bt level
seen-vars = {}
counter = 0
lit p = ⊥
do
  // initially when p is ⊥, reason returns each lit
  p_reason = confl.reason(p) // returns the "reason" vector
  // For each literal in the reason vector…
  for (int j = 0; j < p_reason.size(); j++) {
    lit q = p_reason[j];
    if (var(q) ∉ seen-vars):
      Add var(q) to seen-vars
      If decision level of q is current decision level:
        counter++
      Else if (decision level of q is > 0):
        Push ¬q onto the learned clause
  // Select next literal
  do
    p = trail.last
    confl = reason[var(p)]
    undoOne(); // Pop one decision from the trail
  while (p ∉ seen-vars)
  counter -= 1;
while (counter > 0)
out_clause[0] = ¬p
```

Horn Clauses and Datalog

CIS700 — Fall 2023

Kris Micinski

Today, we'll talk about how to operationalize the rules from last class as a specific programming paradigm: logic programming

Review: Resolution

The resolution rule tells us how to infer new knowledge from preexisting knowledge

$$\frac{P \vee \ldots \vee Q \qquad \neg Q \vee R \ldots \vee}{P \vee \ldots R \vee \ldots}$$

If we derive ⊥, we know the original formula is tantamount to ⊥

We can view resolution as giving us the **transitive closure of our current knowledge base to explicate latent implications**

The Problem with Resolution

Resolution may or may not be helpful—it may produce new clauses which are not useful

Churning on unproductive work can be very costly

$$\frac{\overbrace{P \lor \ldots}^{n} \lor Q \qquad \neg Q \lor \overbrace{R \ldots \lor}^{n}}{\underbrace{P \lor \ldots R \lor \ldots}_{2n}}$$

Resolution-based solvers must judiciously select how to apply resolution

In future lectures, we'll see how DPLL, CDCL, and related systems apply resolution intelligently (but heuristically) to scale SAT solving to formulas with tens of thousands of variables and millions of clauses.

Today, we'll focus on a simpler logic programming language based on a restricted form of clauses

Horn Clauses

A horn clause is a clause with at most one positive (i.e., not negated) literal

$\neg B_0 \vee \ldots \vee \neg B_n \vee H$ or, equivalently… $H \leftarrow B_0 \wedge \ldots B_n$

H is the "head" of the clause, and the B_ns are the "body"
"If everything in the body is true, the head must be true"

Datalog

Horn clauses allow **chain forward** reasoning: if the body is true, then the head must be true

The language **Datalog** implements chain forward Horn clauses over a universe of atoms; in this lecture we'll look at Datalog, its foundations and applications, and its implementation

Datalog

- **Declarative** language used to implement analytics queries over large amounts of data
- Extends **SQL** with the ability to deduce “facts”
- For example, starting with an initial database of **edges**, transitively compute a **path** relation

```
.decl edge(x:number, y:number)
.input edge

.decl path(x:number, y:number)
.output path

path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

Example: Transitive Closure in Soufflé

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

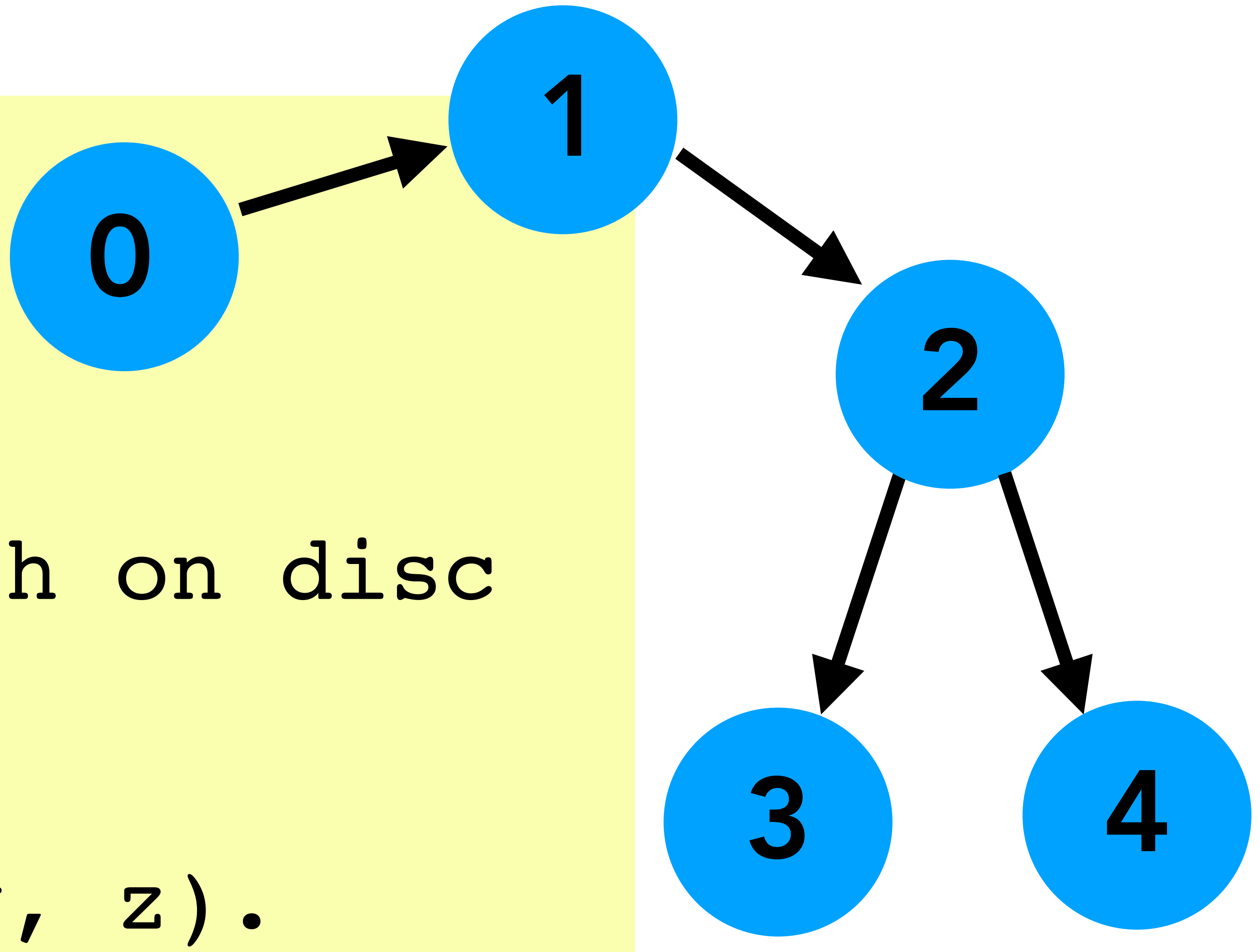
path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).
```

Input: Extensional DataBase (EDB)

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).

// Extensional DataBase (EDB)
edge(0,1). edge(1,2). edge(2,3). edge(2,4).
```

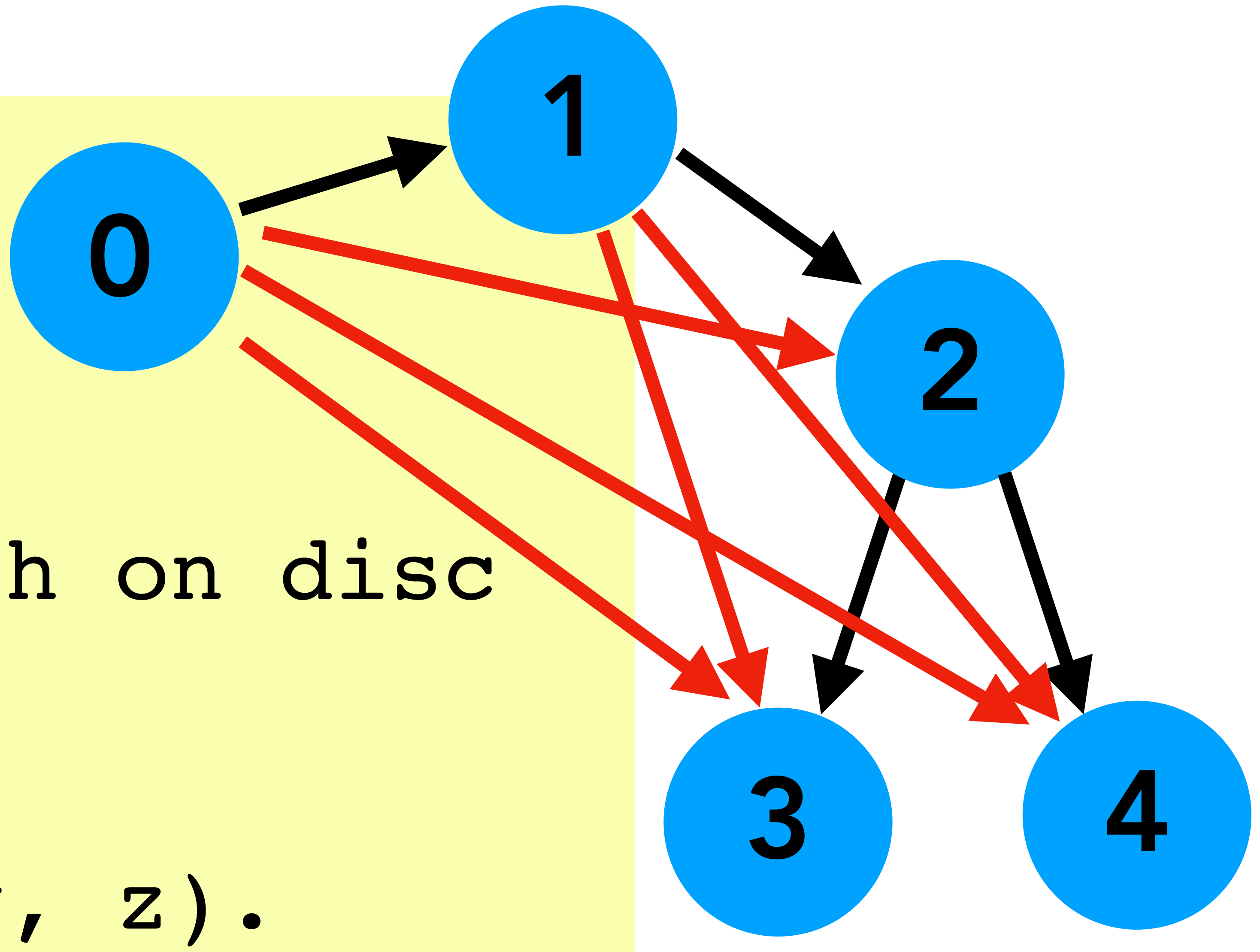


Computation **materializes** the result

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

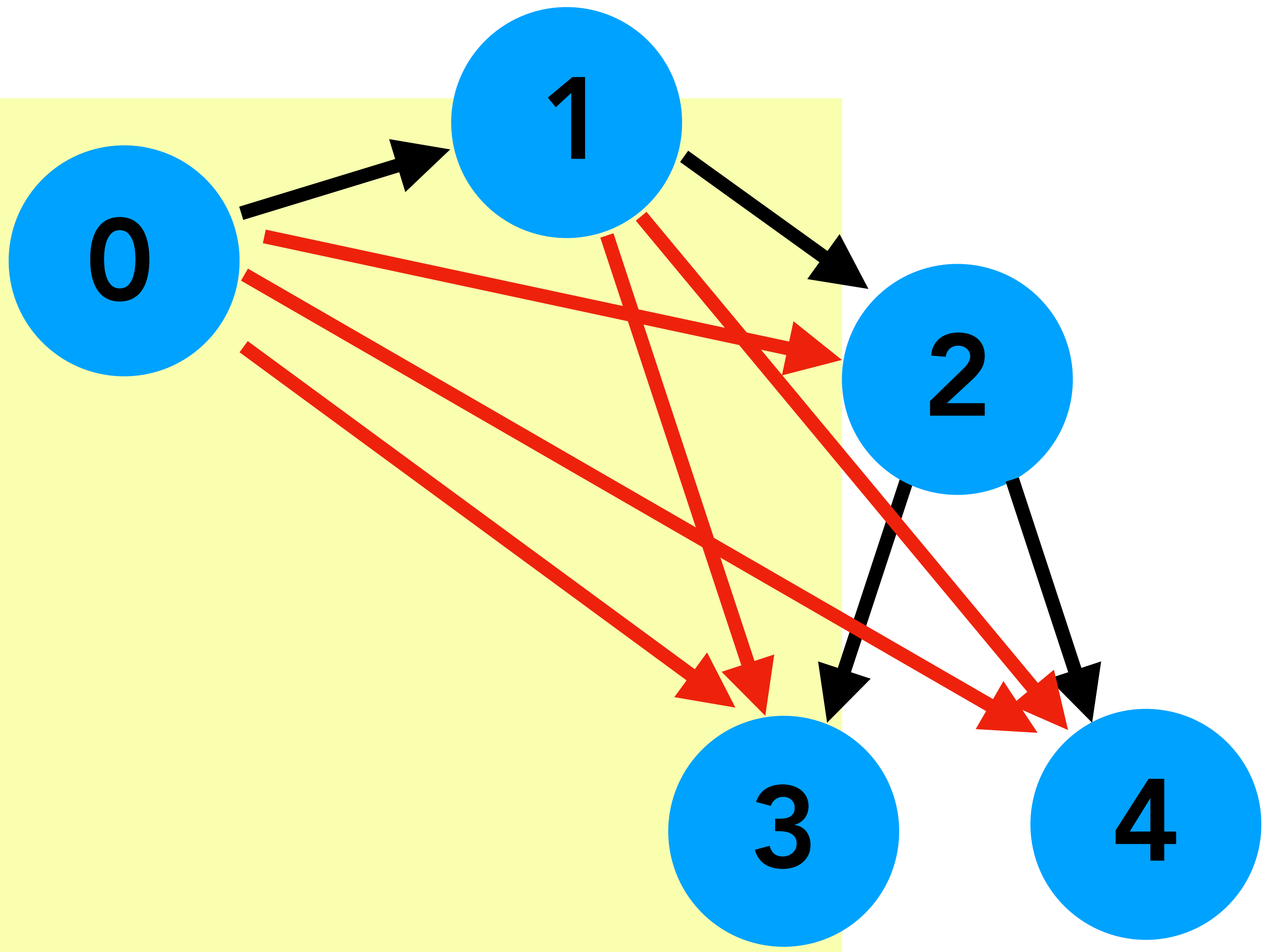
path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).

// Extensional DataBase (EDB)
edge(0,1). edge(1,2). edge(2,3). edge(2,4).
```



Let’s run it and see

```
kmicinski % souffle tc.dl
kmicinski % cat path.csv
0  1
0  2
0  3
0  4
1  2
1  3
1  4
2  3
2  4
```



Challenge: Triangle Counting, etc…

Write a Soufflé program which takes an input edge of the same form as before. You should output triples

How does this generalize to k-clique (k > 3)?

What is (worst-case) runtime complexity of k-clique, increasing with k? (Hint: k-clique is NP complete!)

Conjunction in the rule heads

A conjunction in the head is technically disallowed:

$$H_0 \wedge H_1 \leftarrow B_0 \wedge \ldots B_n$$

But this is only superficial: we can simply refactor this into two rules

$$H_0 \leftarrow B_0 \wedge \ldots B_n$$

$$H_1 \leftarrow B_0 \wedge \ldots B_n$$

Disjunction in rule heads

Horn clauses allow **chain forward** reasoning: if the body is true, then the head must be true

Notice that this rules out (a) negation in the body and (b) disjunction in the head; consider the alternative:

$$H_0 \vee H_1 \leftarrow B_0 \wedge \ldots B_n \qquad H_0 \vee H_1 \vee \neg B_0 \vee \ldots \vee \neg B_n$$

Here, when we know the body is true, we know that either $H_0 \vee H_1$ is true—this means we need to consider *both* possibilities

Extending Datalog to include disjunction in the head is called ***disjunctive Datalog*** and is much more complex

Datalog Programs

- Consist of **facts** and **rules**
- Facts stipulate extensionally-known data
  - Form “input” database, real impls. don’t generally have many facts (instead loaded via CSV)
  - Formal Datalog: facts must be “flat,” i.e., relation arguments must be atoms
- Rules: if everything in the body is true, then head is true

Rules

- Must be Horn-clauses
  - $P(x, \ldots) \leftarrow Q(y, \ldots), R(z, \ldots)$
- Head implied by conjunction (and) of body clauses
- Variables in head must be **ground** (appear in body)
- Negation is **not** allowed, *except* when stratified
  - Stratified negation easy to add metatheoretically: run stratified stuff first; then treat it as an EDB

Datalog Applications — Graph Mining

- **k-Clique** computation (e.g., big social network graphs)

```
two_clique(x, y) :- edge(x,y), edge(y,x).
three_clique(x,y,z) :- two_clique(x,y), two_clique(x,z), two_clique(y,z)
four_clique(a,b,c,d) :- three_clique(a,b,c), two_clique(a,d), …
…
```

- **Pagerank, SSSP,** and **Connected Components** can be calculated if we also add *recursive aggregation*
  - Yihao will discuss this in several weeks.
- Datalog a popular implementation target for social-media mining and graph mining broadly.

Datalog Applications — Program Analysis

- Datalog's rough expressive power is reachability-based analyses over graphs, where the graph structure is dynamic
  - Most scalable points-to (and related) analysis to date (DOOP, cclyzer, ddisasm) use Soufflé — fast single-node compilation
    - Scales to hundreds of thousands of lines in hours, variety of experimental context sensitivities

```
void a(Foo *x) {
  x.f(0);
}
void b(Foo *x) {
  x.f(1);
}
int main() {
  Baz *baz = new Baz();
  Bar *bar = new Bar();
  a(baz);
  b(bar);
}
```

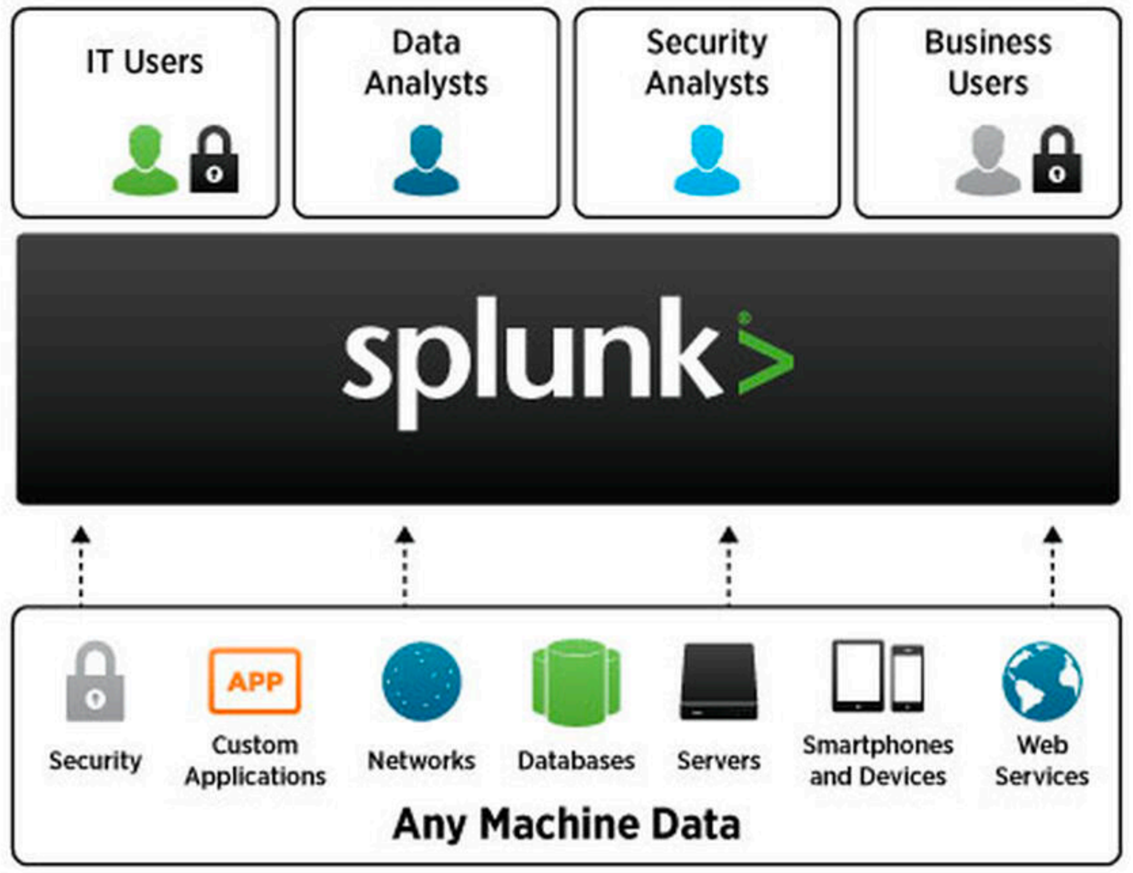
```
class Foo {
  virtual void f(int x) = 0;
}
class Bar : Foo {
  virtual void f(int x) { return 1 / x; }
}
class Baz : Foo {
  virtual void f(int x) { return 1 + x; }
}
```

Datalog Applications — Business Analytics/Databases

- Datalog is roughly the backend structure of many business analytics platforms.

- Lots of industry applications consisting of ad-hoc implementations that scale to things like customer logs, etc…

Datomic: high-speed in-memory (memcached) database via Datalog



Datomic
Data, meet Simple

Semantics of Datalog

- Typically given via an extensional (model-theoretic) and intensional (iteration to a fixed-point up a lattice) semantics
- **Model-theoretic semantics** gives ground truth, but does not immediately lend itself to efficient calculation
  - Can argue about universal properties of models, etc…
- **Fixed-point semantics** gives operational semantics
  - Efficient implementation, semi-naive evaluation, etc…

Model-Theoretic Semantics

- A program P consists of a set of **Rules** and a set of **Facts**. There is a set of **Predicates** whose arguments are variables or ground **Terms**

```
// Facts
edge(0,1). edge(1,2). edge(1,3). edge(2,4). edge(3,4).

// Rules — A <- B /\ C /\ …
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

Preds = {path, edge}
Terms = {0,1,2,3,4}

Model-Theoretic Semantics

- A program P consists of a set of **Rules** and a set of **Facts**. There is a set of **Predicates** whose arguments are variables or ground **Terms**
- The **Herbrand base** is the set of all ground instances of predicates from terms in the program

```
// Facts
edge(0,1). edge(1,2). edge(1,3). edge(2,4). edge(3,4).

// Rules – A <- B /\ C /\ …
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

Herbrand base is…

{edge(0,0), …, edge(4,4), path(0,0), …, path(4,4)}

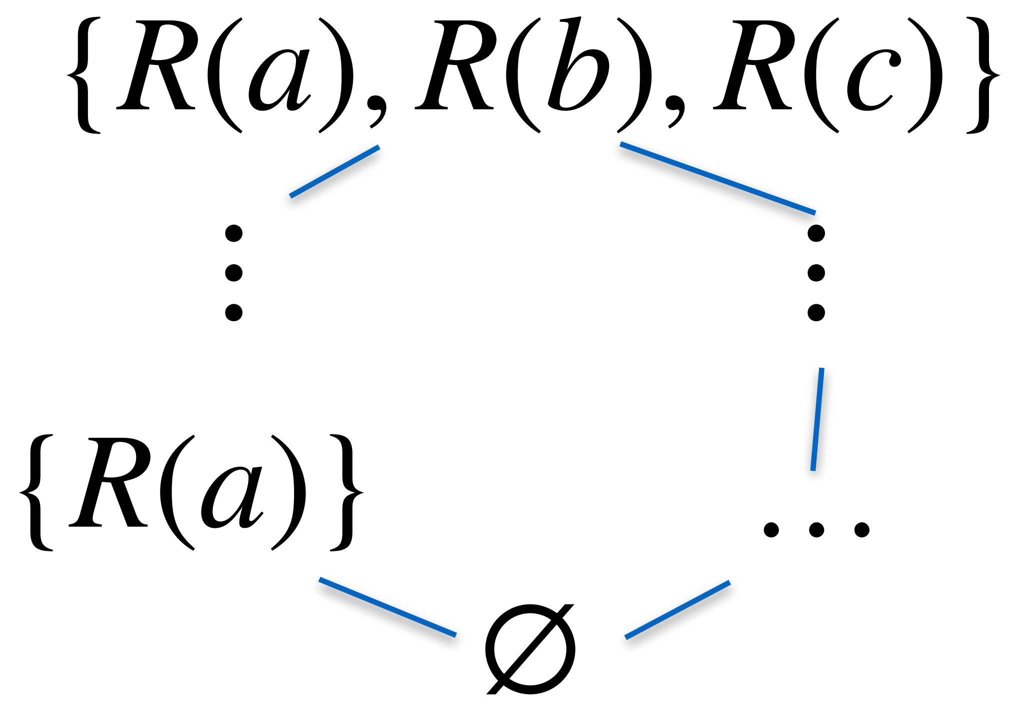
Model-Theoretic Semantics

- A program P consists of a set of **Rules** and a set of **Facts.** There is a set of **Predicates** whose arguments are variables or ground **Terms**

- The **Herbrand base** is the set of all ground instances of predicates from terms in the program

- The Herbrand base forms a *lattice* (it is a set!)—join is ∪, meet is ∩, ordering is via inclusion

- The Herbrand base is **finite**



Herbrand Interpretations

- Any subset of the Herbrand base forms an **interpretation**: a classification of ground atoms as either “true” or “false.”

- Interpretations do not have to be consistent with the program

Herbrand Interpretations

- Any subset of the Herbrand base forms an **interpretation**: a classification of ground atoms as either “true” or “false.”

- Interpretations do not have to be consistent with the program

```
q(1)
p(x) :- q(x)
```

Four possible Herbrand interpretations

{} {p(1)} {q(1)} {p(1),q(1)}

Herbrand *Models*

- An interpretation is a **model** when every rule in the program is satisfied by the model

```
q(1)
p(x) :- q(x)
```

Four possible Herbrand interpretations

{} {p(1)} {q(1)} {p(1),q(1)}

This one is a model

Least Herbrand Models

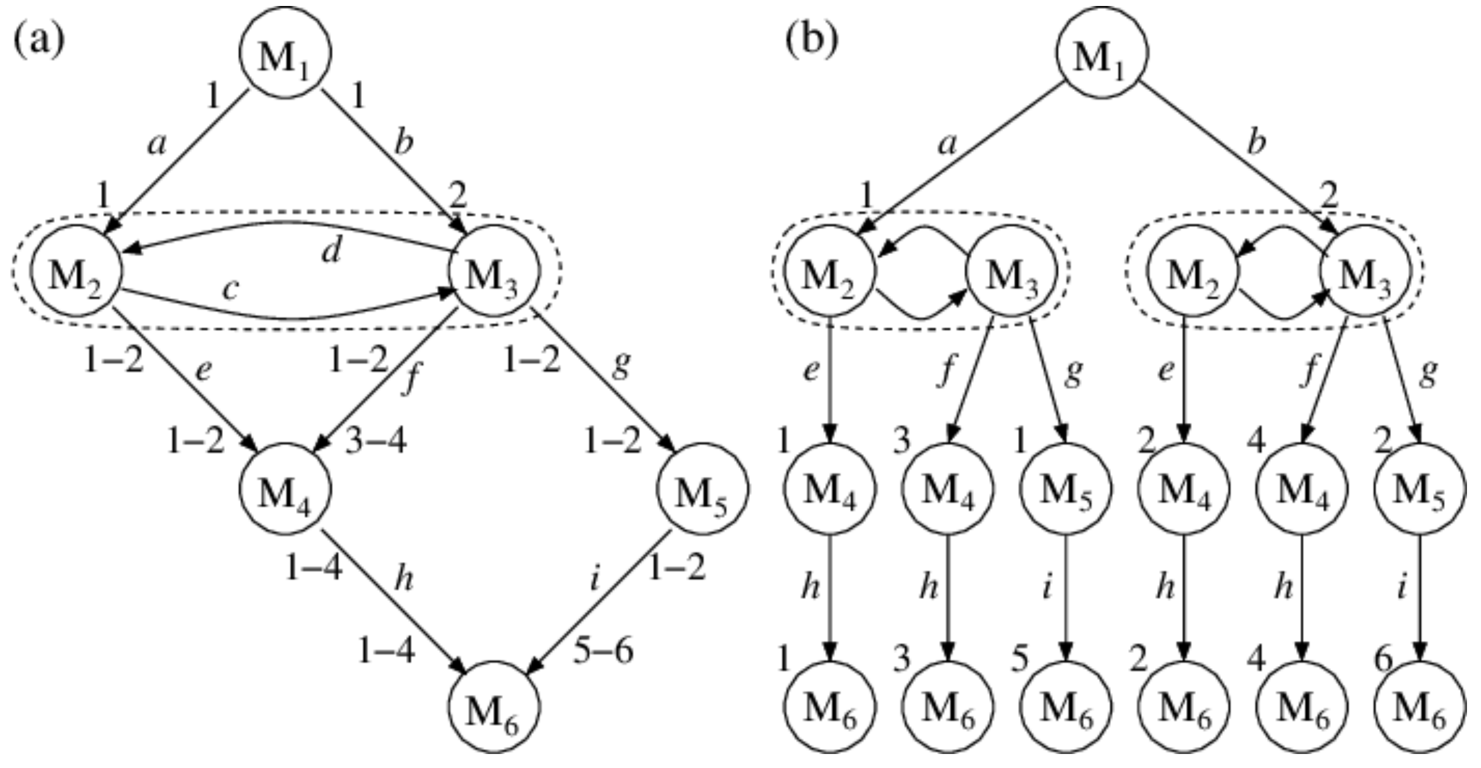
- Model theory—semantics of P is its **least** Herbrand model
- Many (but not all) larger interpretations may also be models…

```
q(1)
p(x) :- q(x)
r(2)
```

{q(1),r(2),p(1),q(2),p(2)}	Another larger (not least) model
{r(2),q(1),p(1),q(2)}	Not a model (requires p(2))
{q(1),p(1),r(2)}	**Least** Herbrand model for P
{}	Not a model (too small)

Implementing Datalog

- You need a **tuple representation strategy** and a **computation strategy**
- Early 2000s: bddbddb, Whaley and Lam scale inclusion-based alias analysis to Java-sized systems via *Binary Decision Diagrams (BDDs)*
  - Variable ordering posed a significant problem
  - Modern implementations use **relational algebra** w/ **explicit representation** (tries)



bddbddb
OVER ONE QUATTUORDECILLION RELATIONS SERVED

Translation to Relational Algebra

- Datalog ~= superficial syntax on top of relational algebra
  - Projection, Selection, Renaming, Joins, etc…
- Relational algebra is “just a bunch of for loops”
  - We built modern processors to be good at dense loops over good trie-like data structures

```
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

```
for(x in path):
  for(z in path):
    for all y such that edge(z,y):
      insert path(x,y)
```

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program
- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics
- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

$\{\}$

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program
- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics
- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

{}   {edge(0,1), edge(1,2)}

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program
- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics
- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

{…,path(0,1),path(1,2)}

{} {edge(0,1), edge(1,2)}

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program
- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics
- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

{…,path(0,1),path(1,2)}

{} {edge(0,1), edge(1,2)}

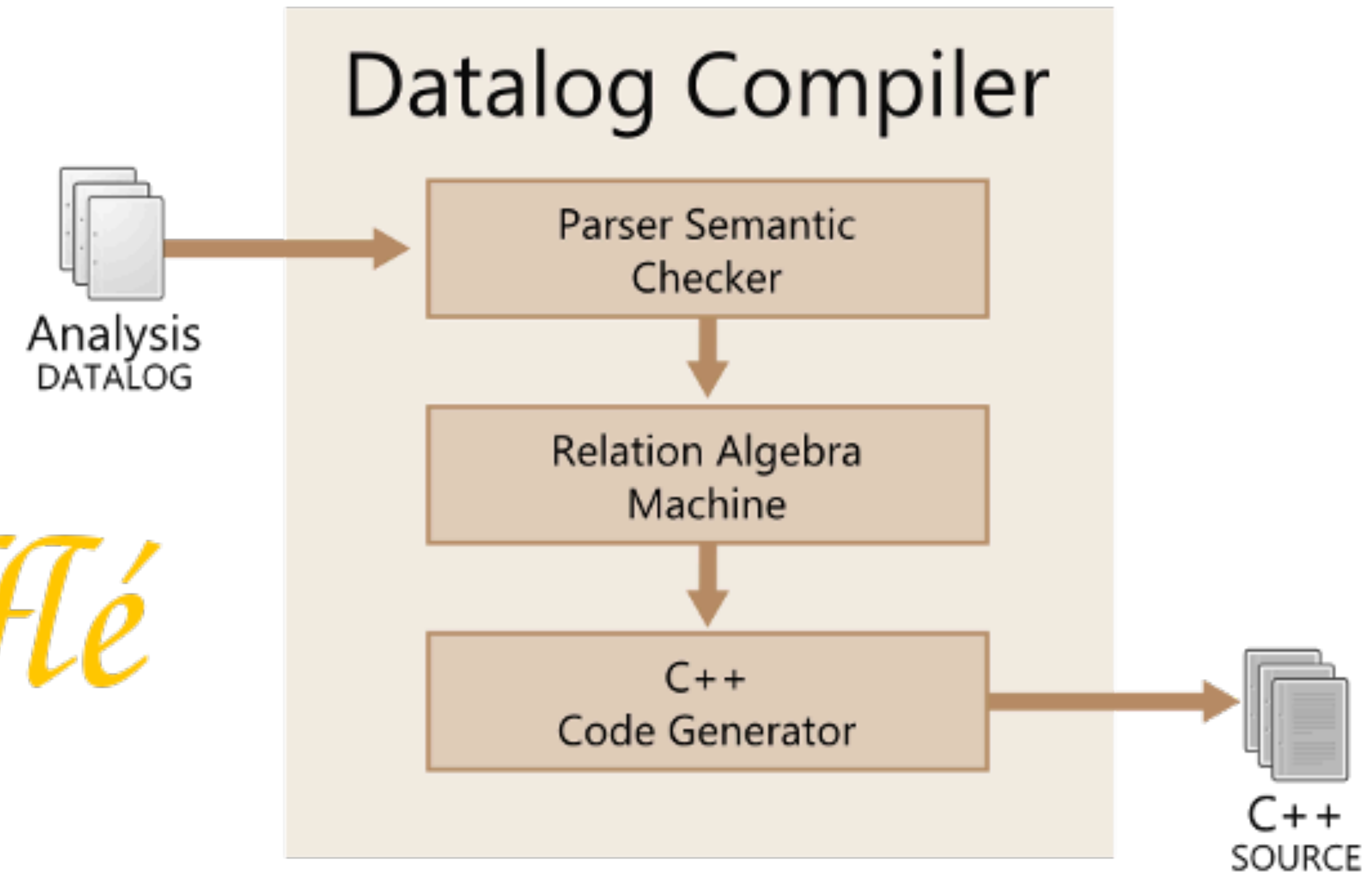
{…,path(0,2)}

Fixed-Point = Least Herbrand Model

- Easy theorem to prove: define an *immediate consequence* operator that derives immediate consequences of facts in program
  - Immediate consequence of fact is itself
  - Head is immediately consequence of ground bodies
- By induction, iterating immediate consequence operator starting with {} gives us a least-Herbrand interpretation

Modern Datalog Compilation

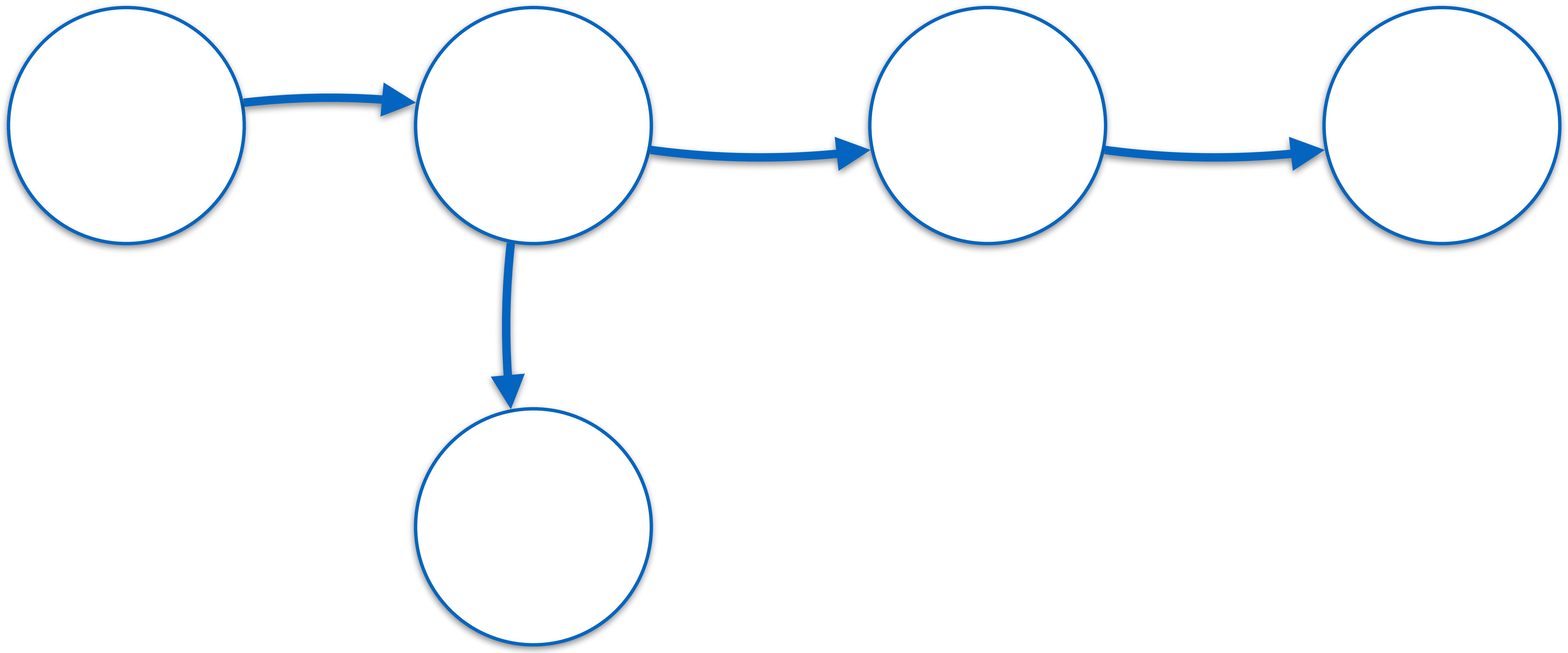
- Continued resurgences in Datalog: semi-naive evaluation, BDDs (Whaley et al.), compilation to relational algebra
- Modern engines work by generating compiled relational algebra kernels (for loops), pushes stress onto high-performance tuple representation
  - Also join planning / RA compilation



Semi-Naive Evaluation

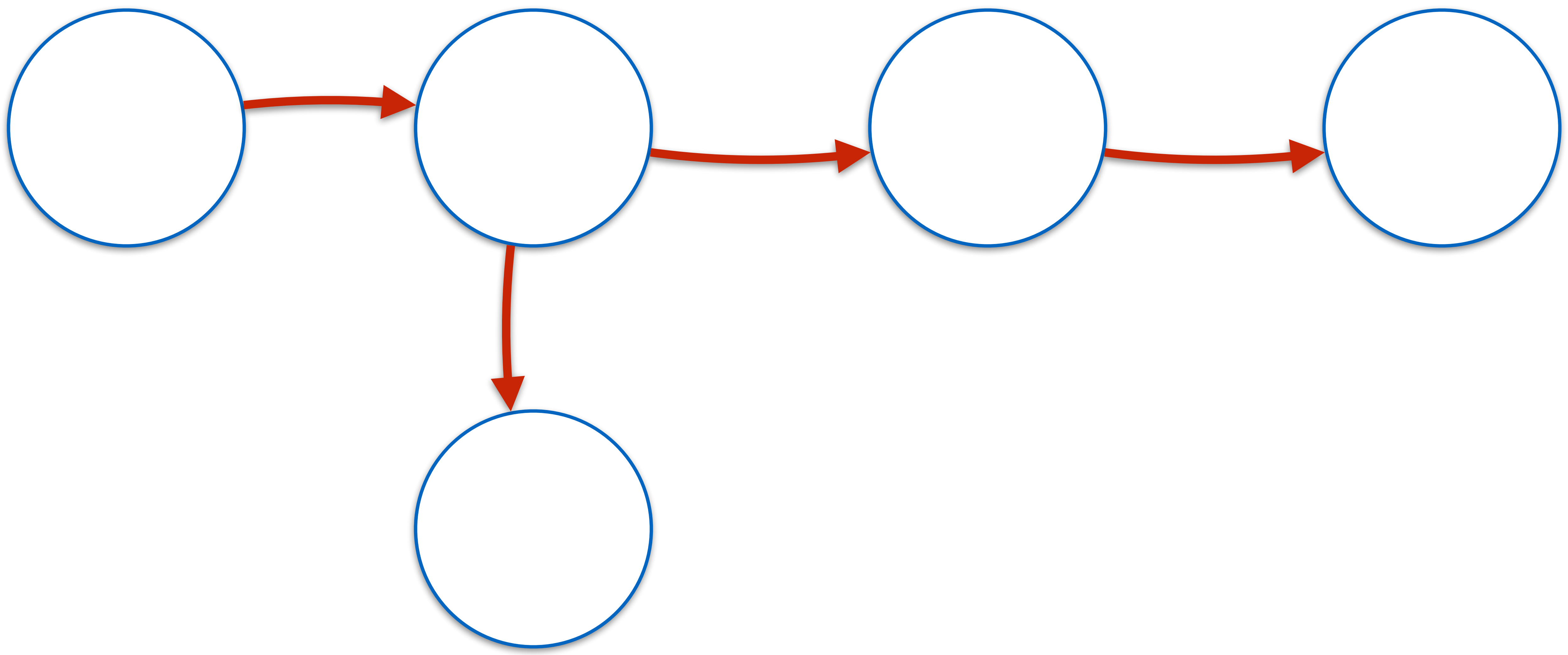
- Each iteration we **reexamine** lots of tuples
- Datalog is *monotonic*: each iteration strictly grows result
- Here: result is monotonically-increasing set of tuples
  - “Sets of tuples” is the **only** lattice DL supports!
- Thus, no need to look at old tuples; only need to consider new tuples that may cause rules to “fire”

Semi-Naive Evaluation



Semi-Naive Evaluation

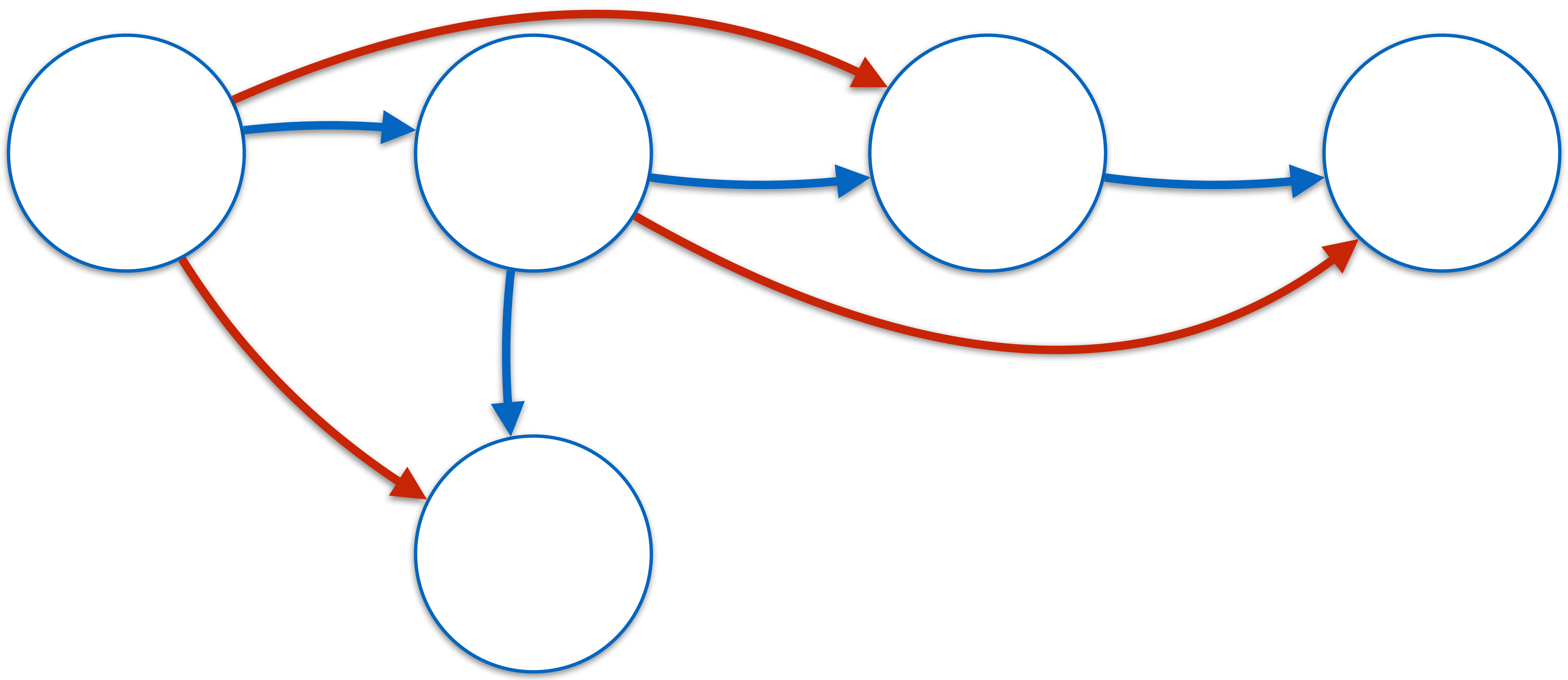
First, discover all edges in `path`



Semi-Naive Evaluation

First, discover all edges in `path`

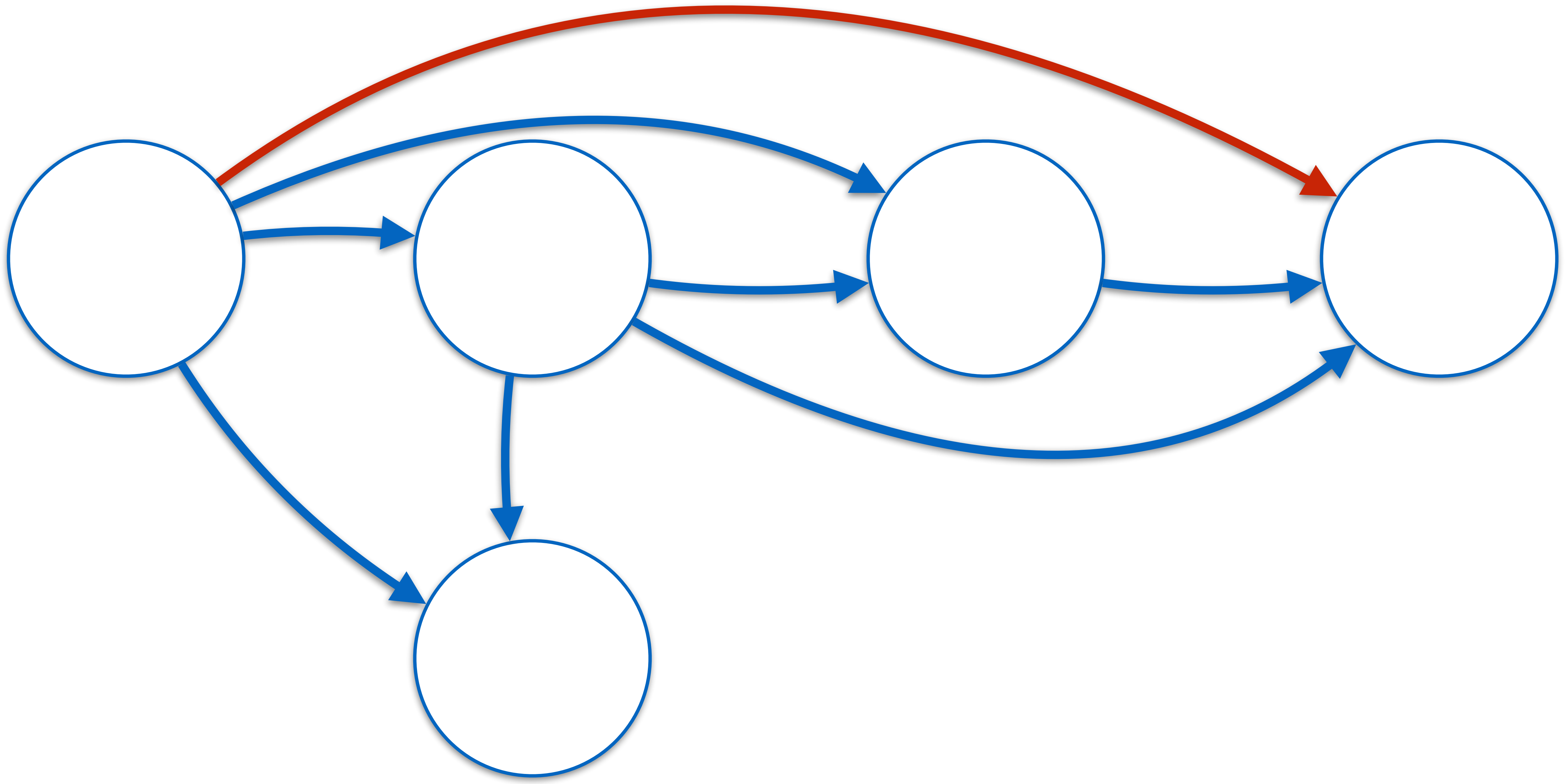
Now, find next iteration…



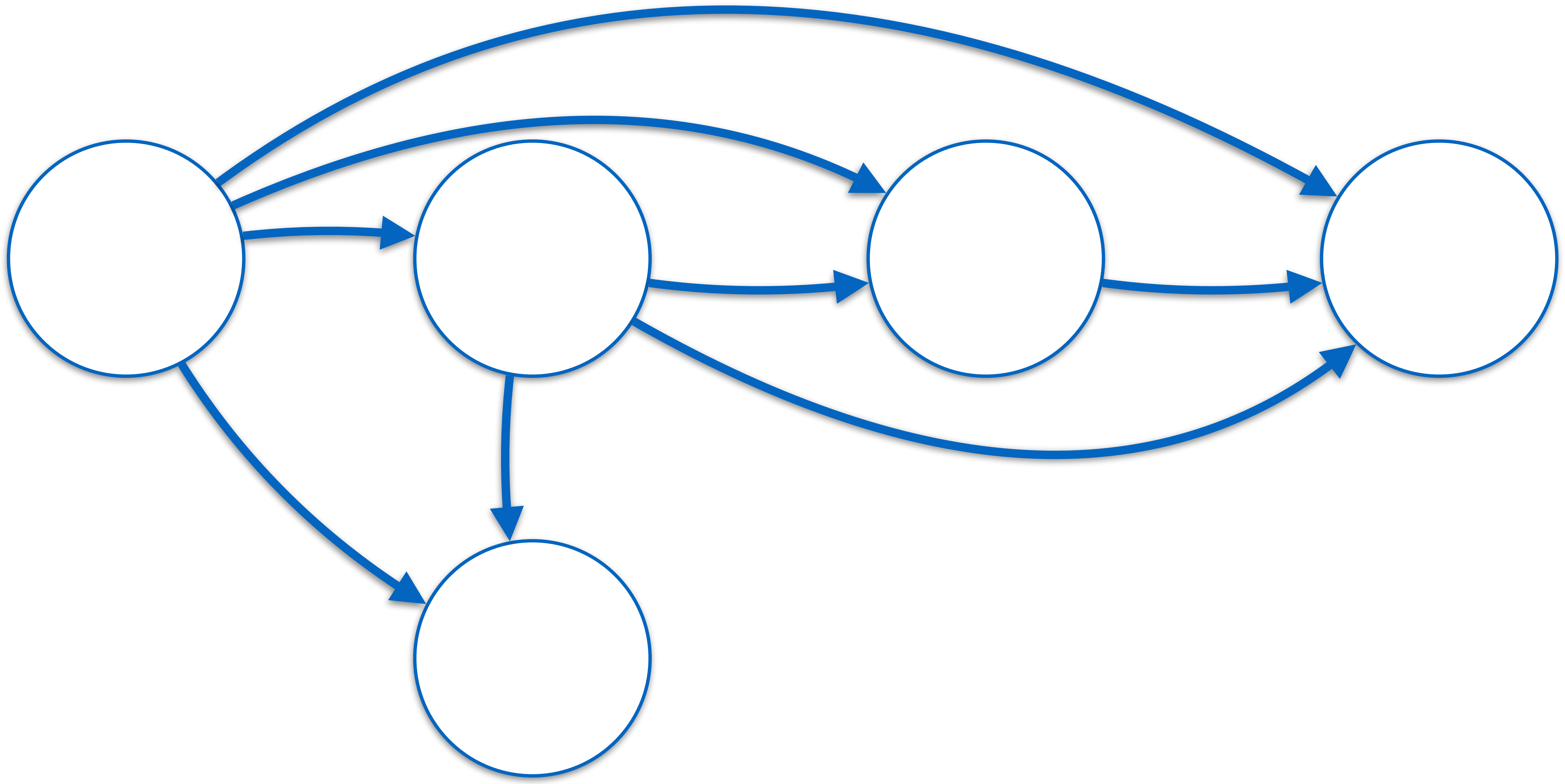
Those all go into Δ, then move into full as a new iteration enters Δ

Semi-Naive Evaluation

Eventually get to a point where nothing new can be discovered…



Semi-Naive Evaluation



At which point `full` contains the result set

Semi-Naive Evaluation

- Compiler adds delta versions (in below rule: join Δ with full, joining Δ with Δ doesn’t work—would force facts to be discovered at same iteration).
- Heads implicitly add to “fresh” version, which becomes delta at end of each iteration
- After each iteration, delta merged into free; free becomes the “fresh” tuples

```
p(x) :- q(x), s(x)
p(x) :- q_delta(x), s_full(x)
```

Partitioning

- We are forced to do one of the following:
  - `[(p x y z) <- (r x y) (int_rel x y z)]`
    `[(int_rel x y z) <- (q x z) (g y z)]`
  - `[(p x y z) <- (int_rel x y z) (g y z)]`
    `[(int_rel x y z) <- (r x y) (q x z)]`

Partitioning contd...

- `[(p x y z) <— (r x y) (q x z) (g y z)]`
  - `[(p x y z) <- (r x y) (int_rel x y z)]`
    `[(int_rel x y z) <- (q x z) (g y z)]`
    - Good if we expect a small number of zs shared between q and g
  - `[(p x y z) <- (int_rel x y z) (g y z)]`
    `[(int_rel x y z) <- (r x y) (q x z)]`
    - Good if we expect a small number of xs shared between r and q

Partitioning is akin to let*

- ```
  [(p x y z) <-
     (a x y)            ;; and last this
     —
     (b x z)            ;; then this
     —
     (c x y) (d y z)] ;; first compute this join
  ```

- Partitioning adds **sequential** cost—tuples in intermediate relation must propagate to outer joins
  ```

Tree Partitioning

- Other partitioning strategies exist, e.g., could partition into a tree, which may expose parallelism in sub-binary-joins that can be done in parallel