DATA516/CSED516 Scalable Data Systems and Algorithms

Lecture 7

Datalog

Announcements

- HW4 is posted: 3 mini-homeworks
- Project Milestone due on Nov. 26
- Last three lectures:
  - Nov. 23 – last regular lecture
  - Nov. 30 – meetings to discuss your project
  - Dec. 07 – project presentations

Outline

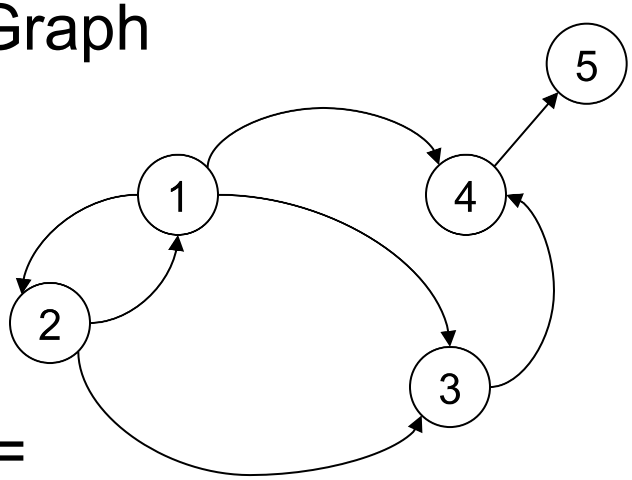
- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions

Datalog program

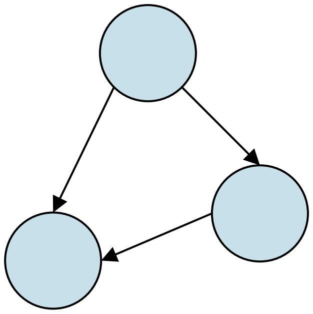
- A datalog program = several rules
- Rules may be recursive
- Set semantics only

Processing Graphs in Datalog

Graph



Pattern Matching

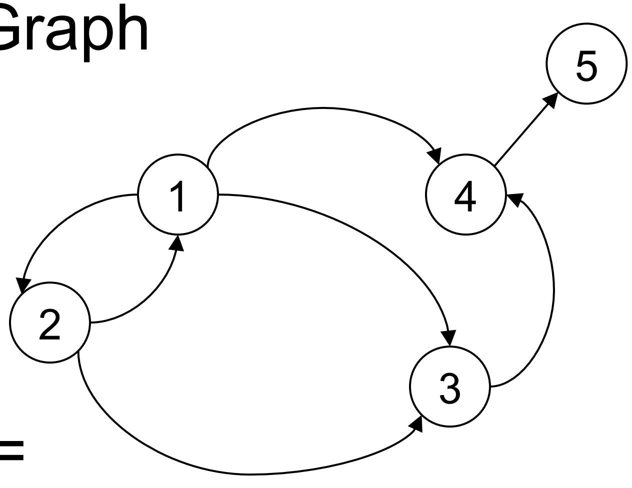


R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

Processing Graphs in Datalog

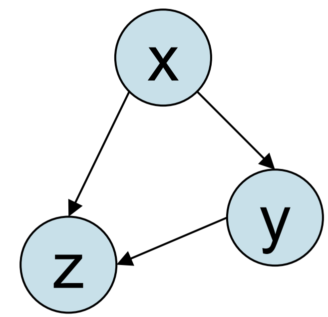
Graph



R=

src	dst
1	2
2	1
2	3
1	4
3	4
4	5
1	3

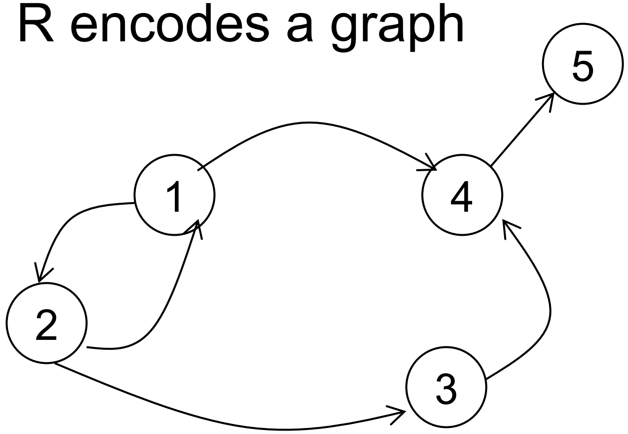
Pattern Matching



Answer(x,y,z) :- R(x,y), R(x,z), R(y,z)

Example

R encodes a graph



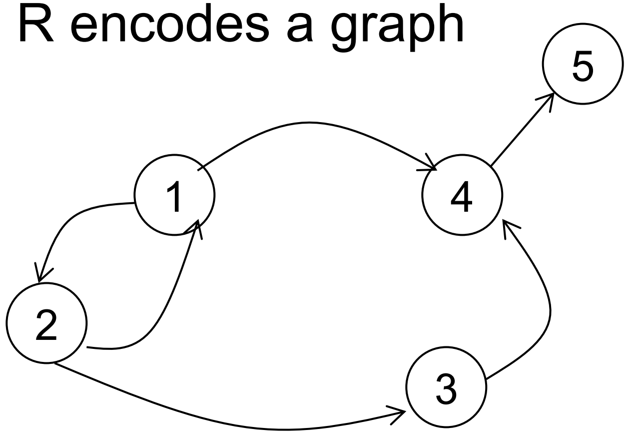
Descendants of node 2

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph



R=

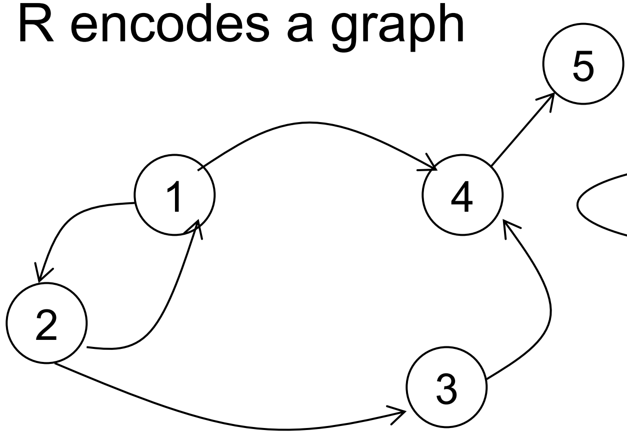
1	2
2	1
2	3
1	4
3	4
4	5

Descendants of node 2

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)

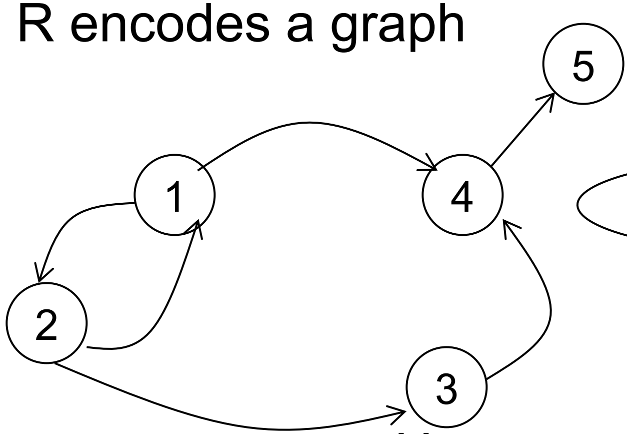
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)

D(y) :- D(x), R(x,y)

How recursion works in datalog:

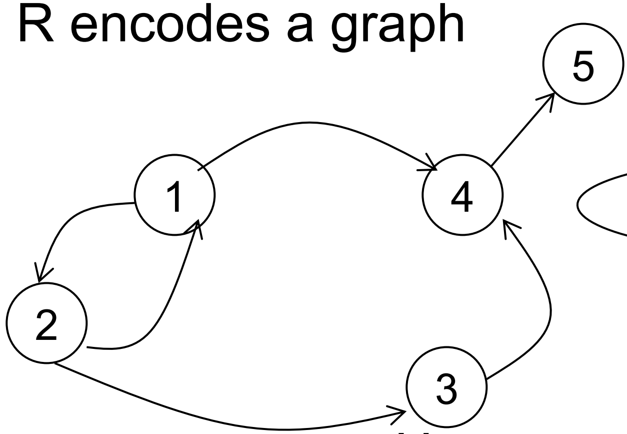
Initially D = empty

R=

1	2
2	1
2	3
1	4
3	4
4	5

Example

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

How recursion works in datalog:

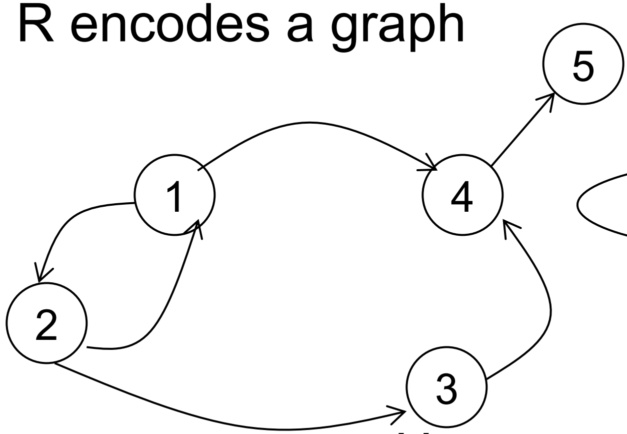
Initially D = empty

- Compute both rules:

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

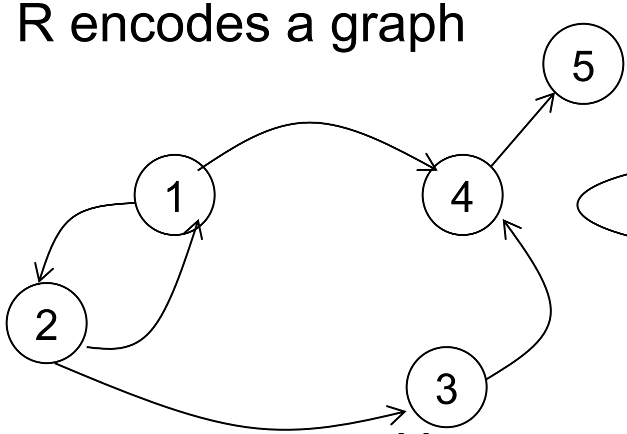
Initially D = empty

- Compute both rules:
  …now D = {1,3}

{1,3}
D(x) :- R(2, x)
{ }
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:

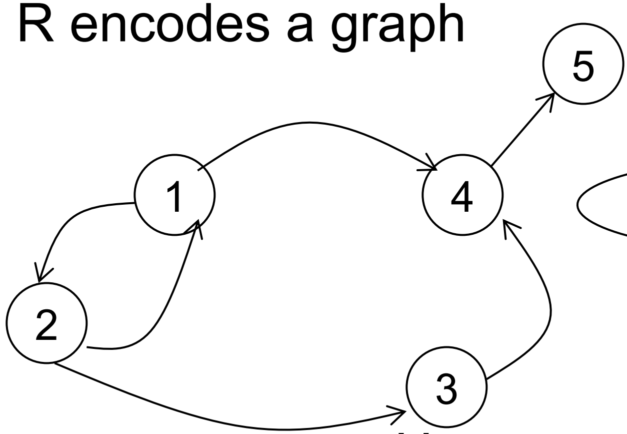
{1,3}
D(x) :- R(2, x)

{ }
D(y) :- D(x), R(x,y)

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

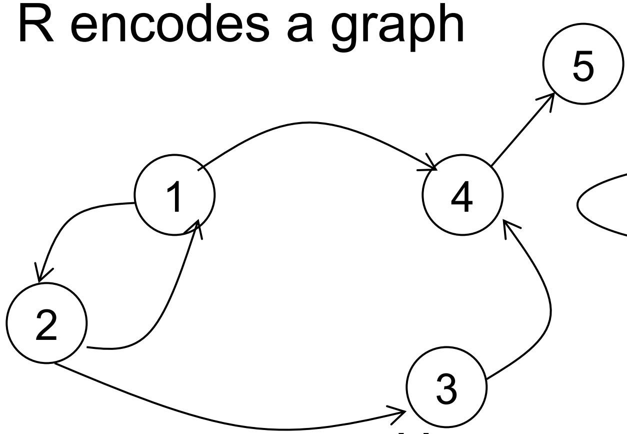
- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}

{1,3} D(x) :- R(2, x)
{ } D(y) :- D(x), R(x,y)

{1,3} D(x) :- R(2, x)
{2,4 } D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:

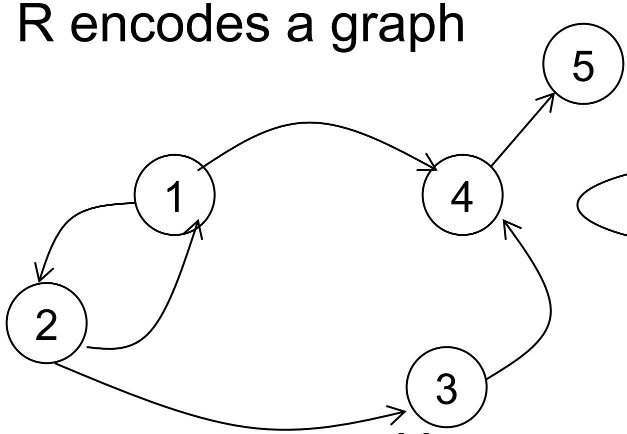
{1,3} D(x) :- R(2, x)
{ } D(y) :- D(x), R(x,y)

{1,3} D(x) :- R(2, x)
{2,4 } D(y) :- D(x), R(x,y)

D(x) :- R(2, x)
D(y) :- D(x), R(x,y)

Example

R encodes a graph



Descendants of node 2

Recursive rule

$D(x)$:- $R(2, x)$
$D(y)$:- $D(x), R(x,y)$

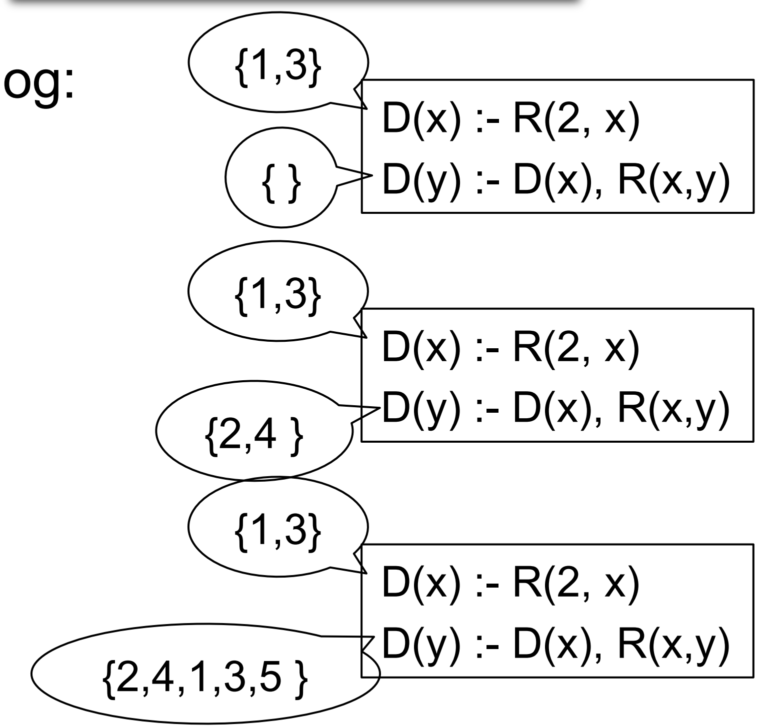
R=

1	2
2	1
2	3
1	4
3	4
4	5

How recursion works in datalog:

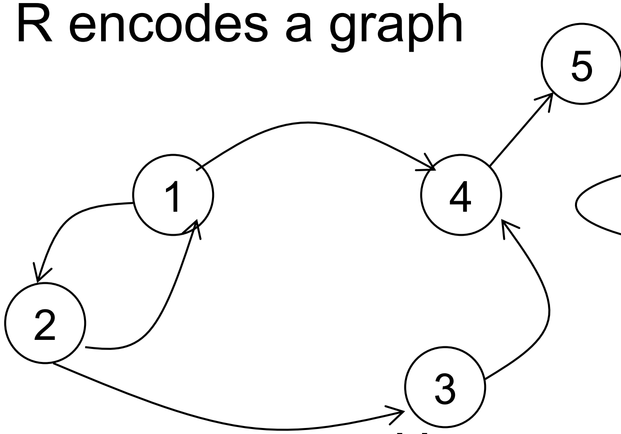
Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:
  …now D = {1,3,2,4,5}



Example

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

Descendants of node 2

Recursive rule

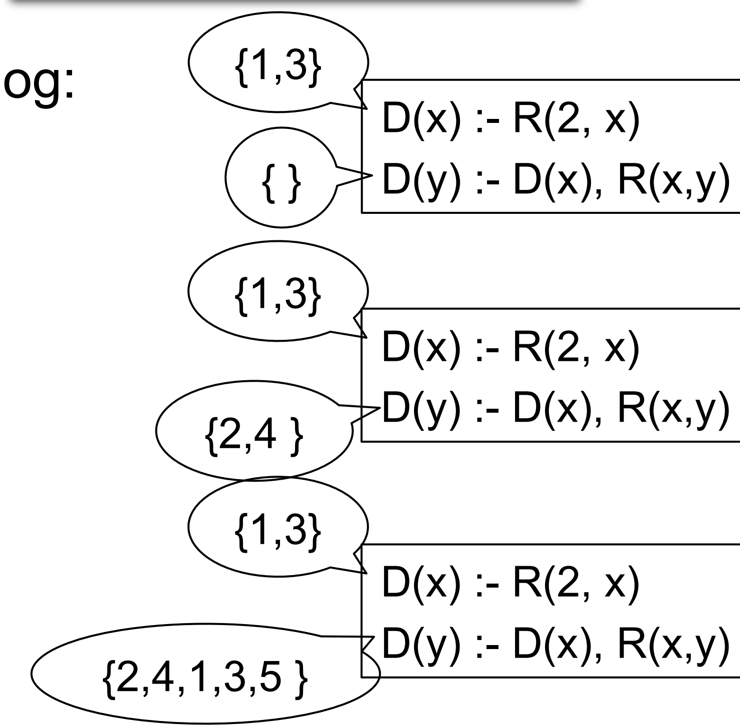
D(x) :- R(2, x)

D(y) :- D(x), R(x,y)

How recursion works in datalog:

Initially D = empty

- Compute both rules:
  …now D = {1,3}
- Compute both rules:
  …now D = {1,3,2,4}
- Compute both rules:
  …now D = {1,3,2,4,5}
- Compute both rules:
  …nothing new. STOP



Outline

- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions

Naïve Evaluation Algorithm

- Every rule → SPJ* query

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule → SPJ* query

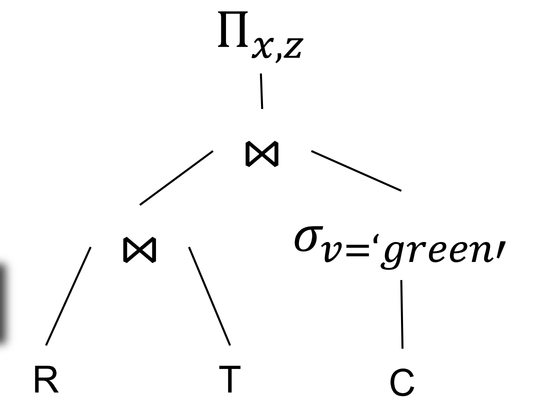
```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

- Every rule → SPJ* query

T(x,z) :- R(x,y), T(y,z), C(y,'green')

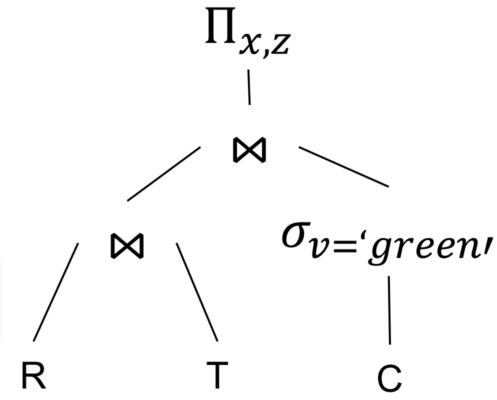


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

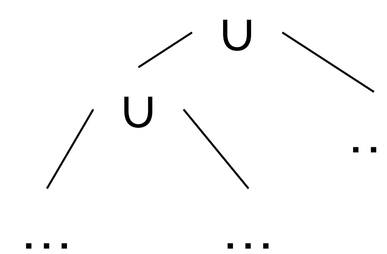
- Every rule → SPJ* query

```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```



- Multiple rules same head → USPJ+

```
T(x,y) :-  …
T(x,y) :-  …
      …
```

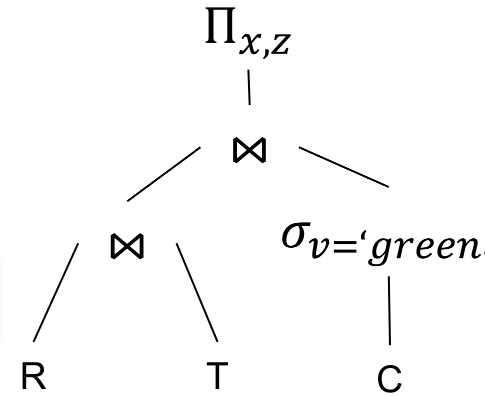


*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

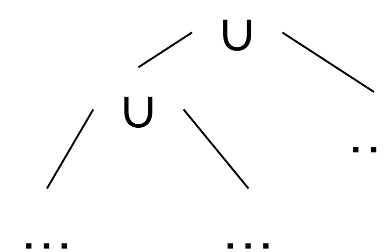
- Every rule → SPJ* query

```
T(x,z) :- R(x,y), T(y,z), C(y,'green')
```



- Multiple rules same head → USPJ+

```
T(x,y) :-  …
T(x,y) :-  …
   …
```



- Naïve Algorithm:

$IDBs := \emptyset$
repeat $IDBs := USPJs$
until no more change

*SPJ = select-project-join
+USPJ = union-select-project-join

Naïve Evaluation Algorithm

```
D(x) :- R(2,x)
D(y) :- D(x),R(x,y)
```

Naïve Evaluation Algorithm

D(x) :- R(2,x)
D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R))$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)
D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R);$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)

D(y) :- D(x),R(x,y)

$$\Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R);$$

Naïve Evaluation Algorithm

D(x) :- R(2,x)
D(y) :- D(x),R(x,y)

$D := \emptyset$;
repeat
$D := \Pi_{R.dst}(\sigma_{R.src=2}(R)) \cup \Pi_{R.dst}(D \bowtie_{D.node=R.src} R)$;
until [no more change]

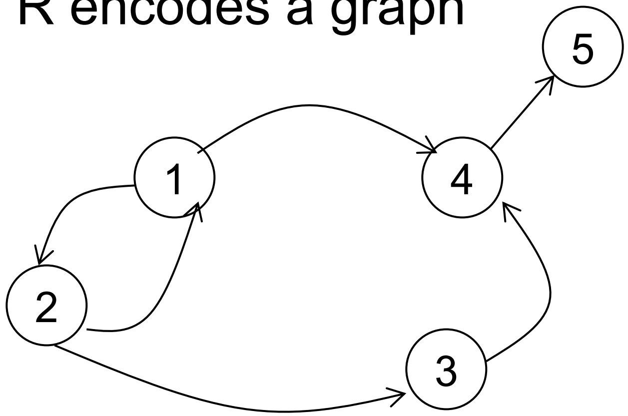
Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database

Example

R encodes a graph



R=

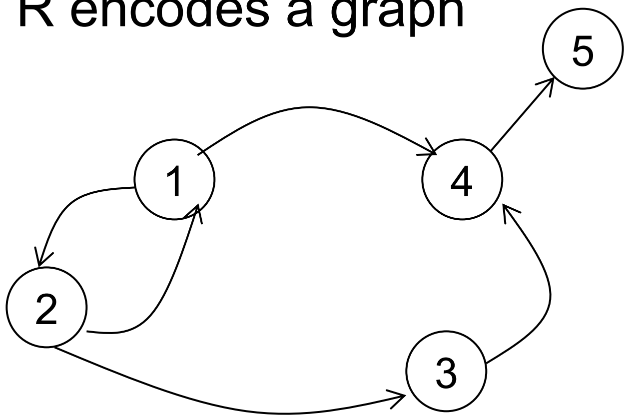
1	2
2	1
2	3
1	4
3	4
4	5

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

Example

R encodes a graph



T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

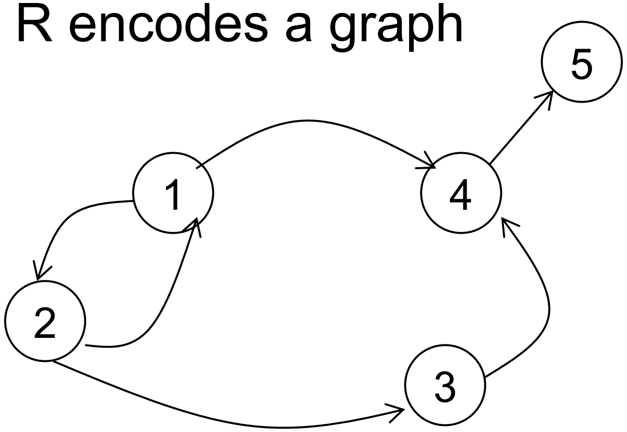
Example

$T(x,y)$:- $R(x,y)$

$T(x,y)$:- $R(x,z), T(z,y)$

What does it compute?

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:

T is empty.

First iteration:

T =

1	2
2	1
2	3
1	4
3	4
4	5

First rule generates this

Second rule generates nothing (because T is empty)

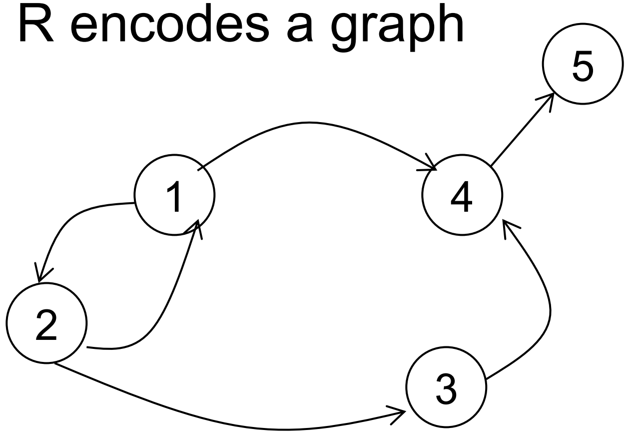
Example

$T(x,y)$:- $R(x,y)$

$T(x,y)$:- $R(x,z), T(z,y)$

What does it compute?

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:

T is empty.

First iteration:

T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:

T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

First rule generates this (rows 1 2 through 4 5)

Second rule generates this (rows 1 1 through 3 5)

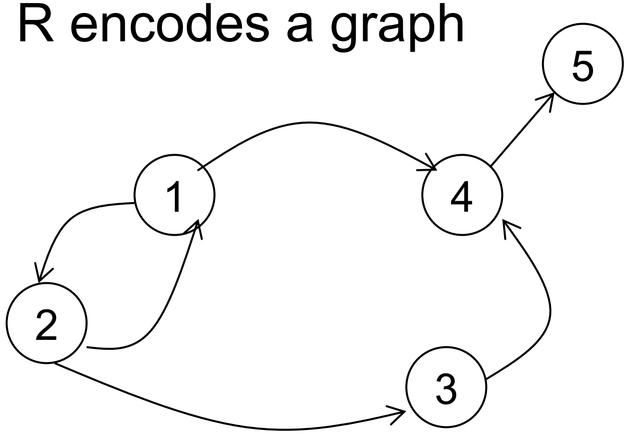
New facts

Example

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

What does it compute?

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

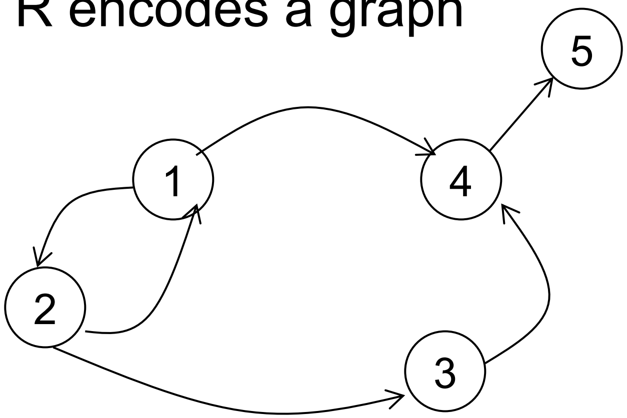
Third iteration:
T =

1	2	Both rules
2	1	Both rules
2	3	First rule
1	4	First rule
3	4	First rule
4	5	First rule
1	1	
2	2	Second rule
1	3	Second rule
2	4	Second rule
1	5	Second rule
3	5	Second rule
2	5	Second rule

New fact (2, 5)

Example

R encodes a graph



$$T(x,y) \text{ :- } R(x,y)$$
$$T(x,y) \text{ :- } R(x,z),\ T(z,y)$$

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:

T is empty.

First iteration:

T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:

T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

Third iteration:

T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

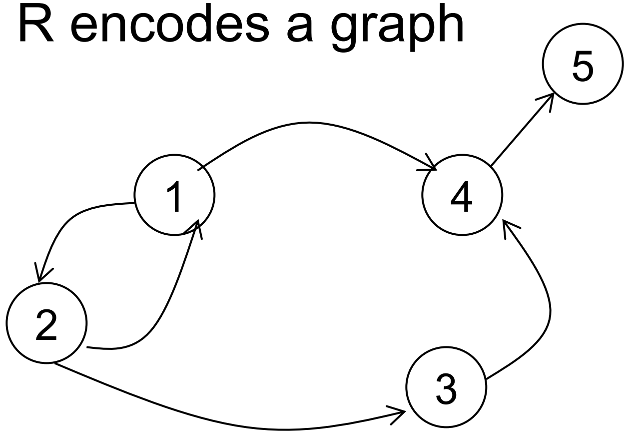
Fourth iteration

T = (same)

No new facts. DONE

Example

R encodes a graph



$T(x,y) \text{ :- } R(x,y)$

$T(x,y) \text{ :- } R(x,z), T(z,y)$

What does it compute?

R=

1	2
2	1
2	3
1	4
3	4
4	5

Initially:
T is empty.

First iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5

Second iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5

Third iteration:
T =

1	2
2	1
2	3
1	4
3	4
4	5
1	1
2	2
1	3
2	4
1	5
3	5
2	5

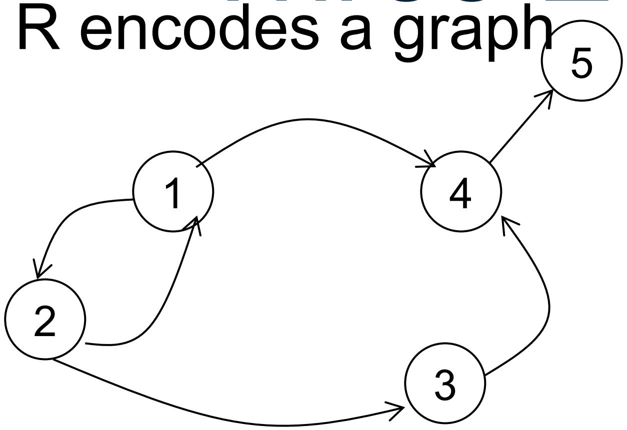
Fourth iteration
T =
(same)

No new facts. DONE

Iteration k computes pairs (x,y) connected by path of length ≤ k

Three Equivalent Programs

R encodes a graph



R=

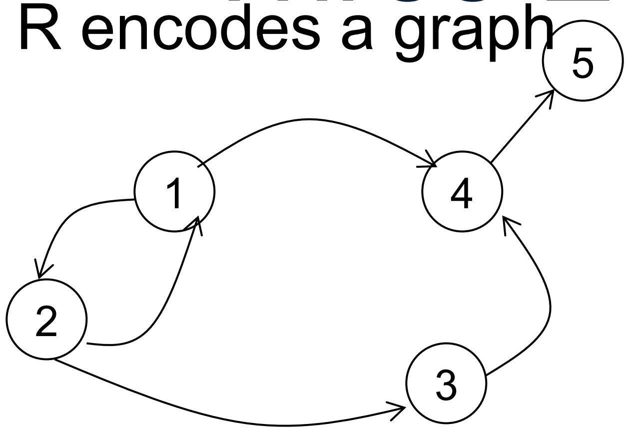
1	2
2	1
2	3
1	4
3	4
4	5

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

Right linear

Three Equivalent Programs

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

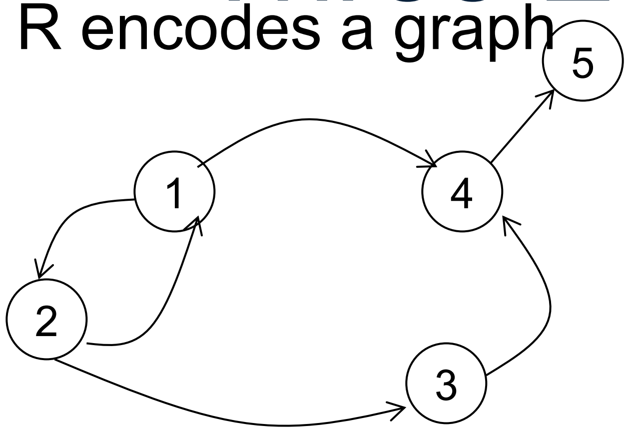
Right linear

```
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)
```

Left linear

Three Equivalent Programs

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

Right linear

T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)

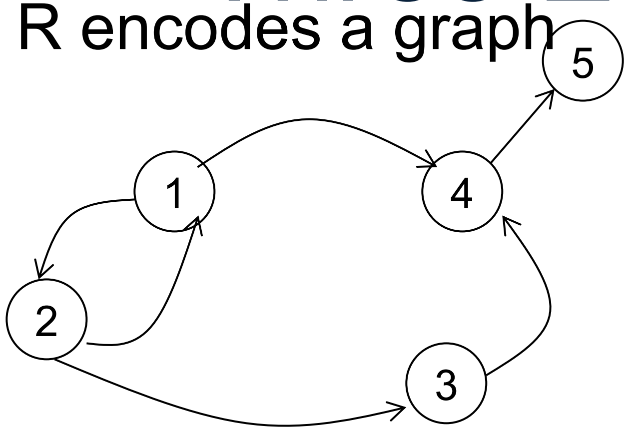
Left linear

T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

Non-linear

Three Equivalent Programs

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

Right linear

T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)

Left linear

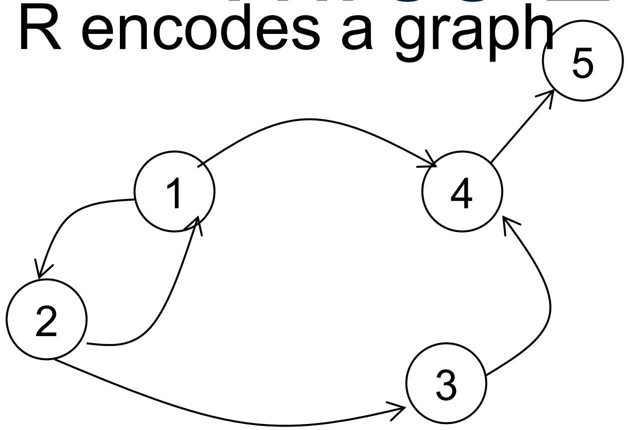
T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

Non-linear

Question: how many iterations does each require?

Three Equivalent Programs

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)

Right linear

T(x,y) :- R(x,y)
T(x,y) :- T(x,z), R(z,y)

Left linear

#iterations = diameter

T(x,y) :- R(x,y)
T(x,y) :- T(x,z), T(z,y)

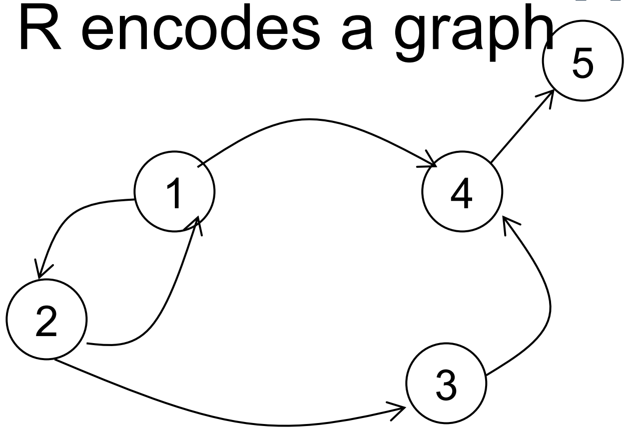
Non-linear

#iterations = log(diameter)

Question: how many iterations does each require?

Multiple IDBs

R encodes a graph



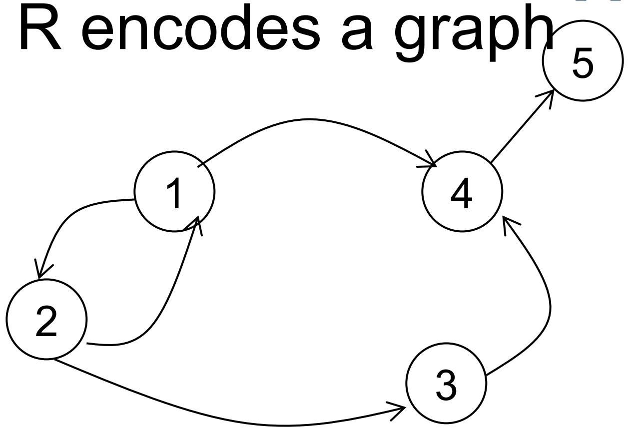
Find pairs of nodes (x,y) connected by a path of <u>*even*</u> length

R=

1	2
2	1
2	3
1	4
3	4
4	5

Multiple IDBs

R encodes a graph



Find pairs of nodes (x,y) connected by a path of *even* length

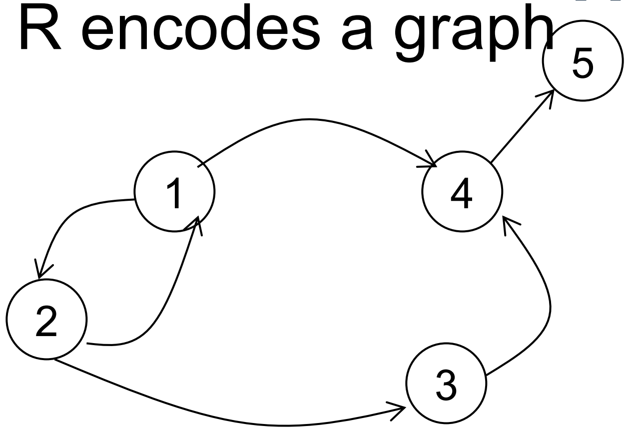
R=

1	2
2	1
2	3
1	4
3	4
4	5

Odd(x,y) :- R(x,y)

Multiple IDBs

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

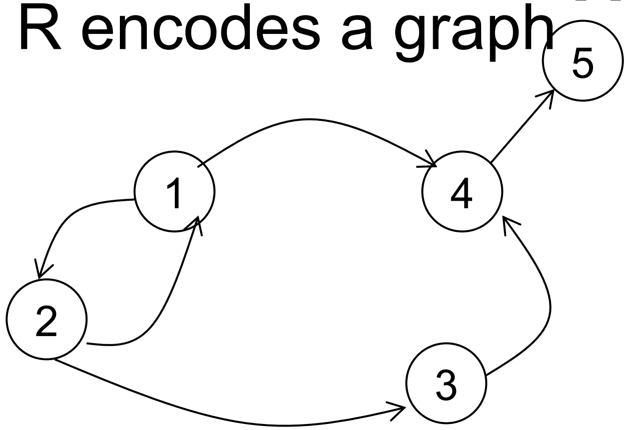
Find pairs of nodes (x,y) connected by a path of *even* length

Odd(x,y) :- R(x,y)

Even(x,y) :- Odd(x,z), R(z,y)

Multiple IDBs

R encodes a graph



Find pairs of nodes (x,y) connected by a path of *even* length

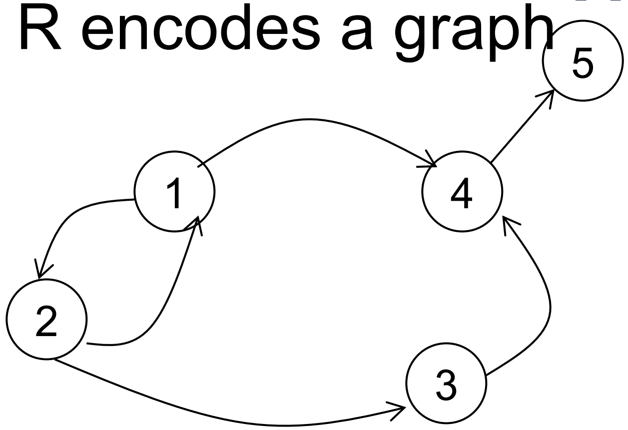
R=

1	2
2	1
2	3
1	4
3	4
4	5

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z), R(z,y)
Odd(x,y) :- Even(x,z), R(z,y)
```

Multiple IDBs

R encodes a graph



R=

1	2
2	1
2	3
1	4
3	4
4	5

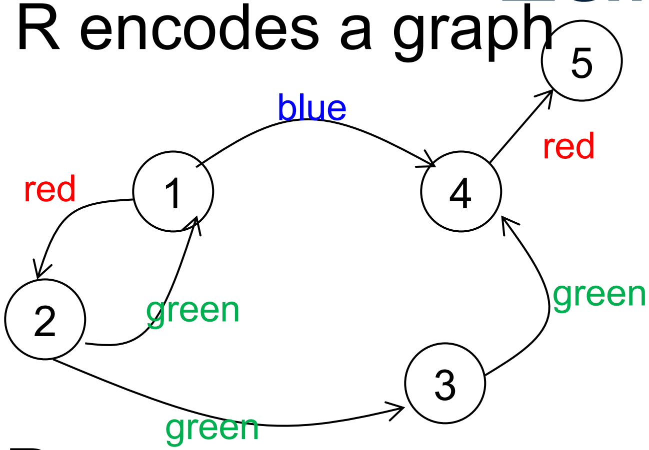
Find pairs of nodes (x,y) connected by a path of *even* length

Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z), R(z,y)
Odd(x,y) :- Even(x,z), R(z,y)

Two IDBs: Odd(x,y) and Even(x,y)

Labeled Graphs

R encodes a graph



Find pairs of nodes (x,y) connected by a ***green*** path

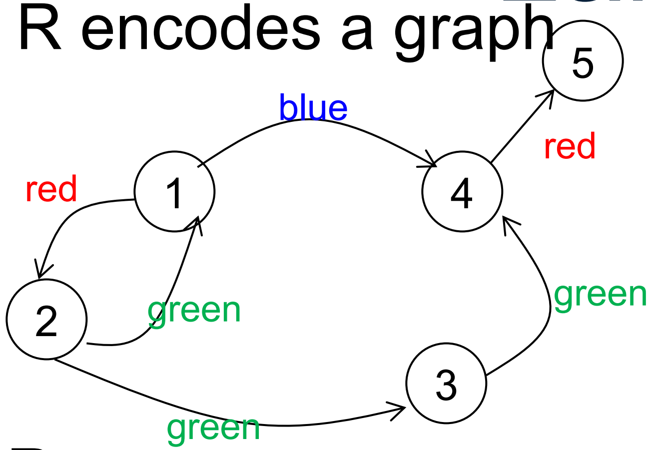
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

GreenP(x,y) :-

Labeled Graphs

R encodes a graph



Find pairs of nodes (x,y) connected by a ***green*** path

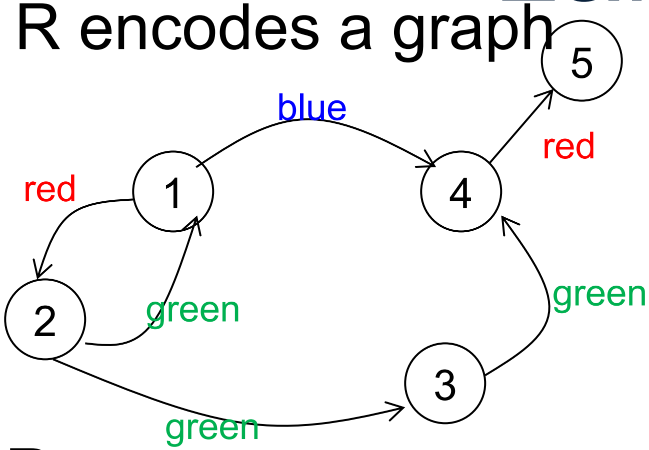
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

GreenP(x,y) :- R(x,y,'green')

Labeled Graphs

R encodes a graph



Find pairs of nodes (x,y) connected by a *green* path

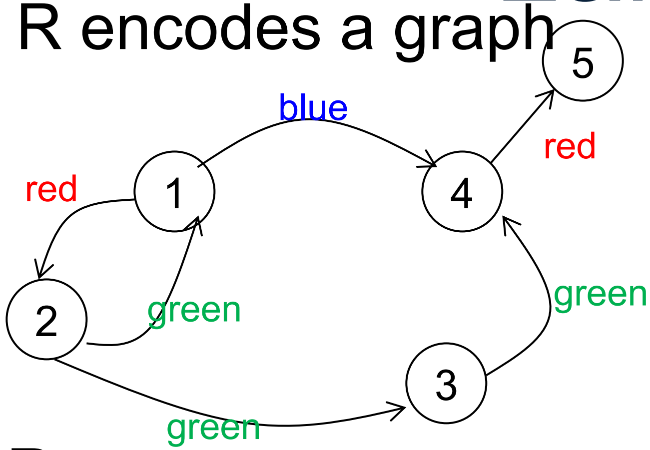
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

GreenP(x,y) :- R(x,y,'green')
GreenP(x,y) :- R(x,z,'green'),GreenP(z,y)

Labeled Graphs

R encodes a graph



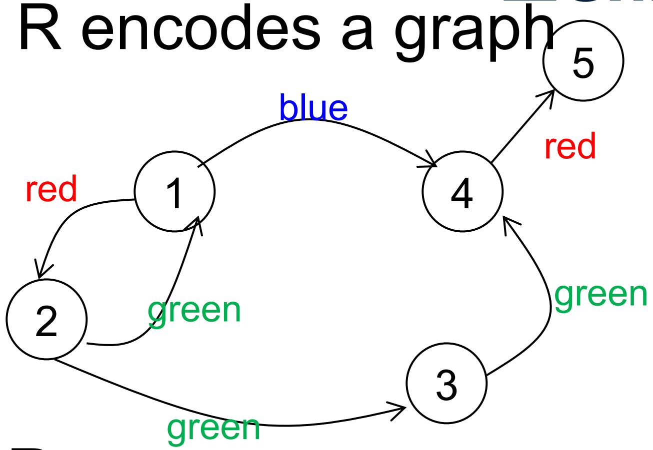
Find pairs of nodes (x,y) connected by a *monochromatic* path

R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Labeled Graphs

R encodes a graph



Find pairs of nodes (x,y) connected by a *monochromatic* path

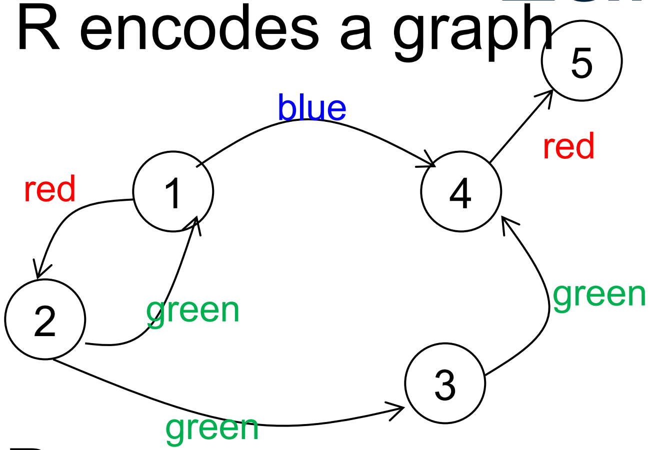
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

P(x,y,c) :- R(x,y,c)

Labeled Graphs

R encodes a graph



Find pairs of nodes (x,y) connected by a *monochromatic* path

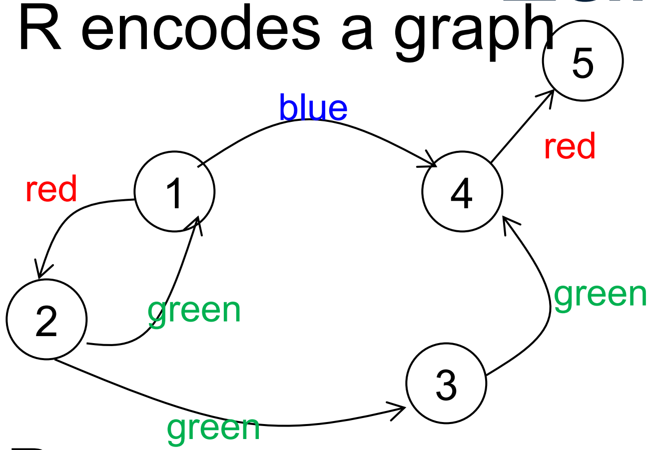
R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

P(x,y,c) :- R(x,y,c)
P(x,y,c) :- R(x,z,c),P(z,y,c)

Labeled Graphs

R encodes a graph



R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

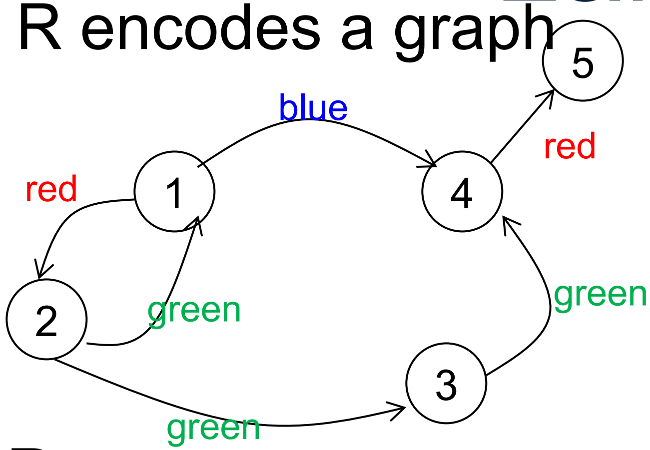
Find pairs of nodes (x,y) connected by a *monochromatic* path

We join on both the node z, and the color c

P(x,y,c) :- R(x,y,c)
P(x,y,c) :- R(x,z,c),P(z,y,c)

Labeled Graphs

R encodes a graph



Find pairs of nodes (x,y) connected by a *monochromatic* path

R=

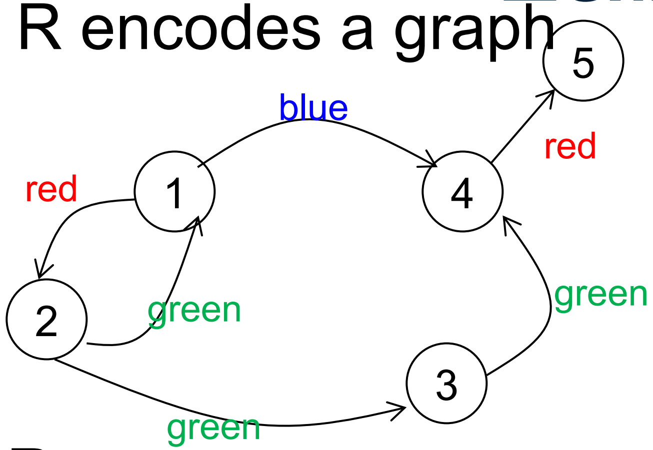
1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

We join on both the node z, and the color c

P(x,y,c) :- R(x,y,c)
P(x,y,c) :- R(x,z,c),P(z,y,c)
Answer(x,y) :- P(x,y,c) – why needed?

Labeled Graphs

R encodes a graph



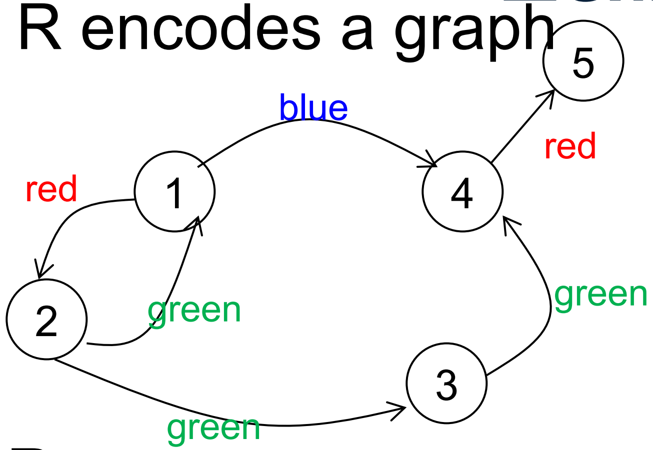
Find all nodes reachable from node 2 by a path containing exactly one red edge.

R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Labeled Graphs

R encodes a graph

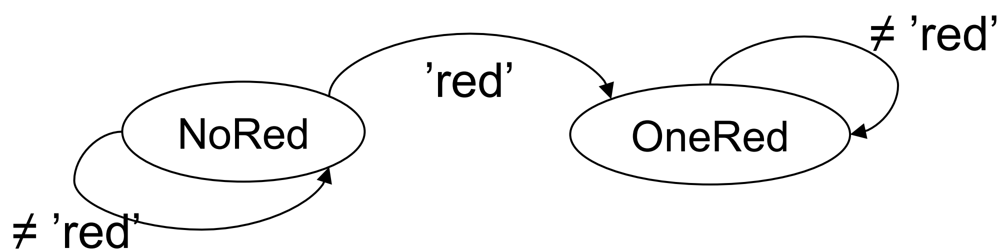


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

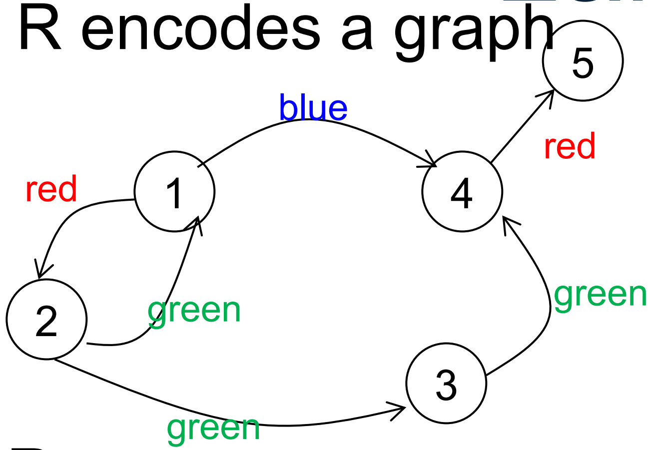
Find all nodes reachable from node 2 by a path containing exactly one red edge.

Automaton:



Labeled Graphs

R encodes a graph

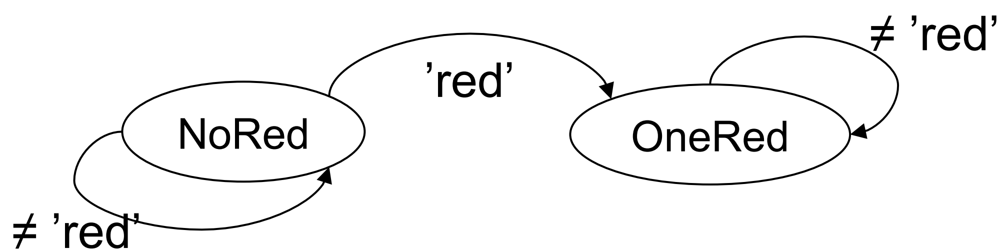


Find all nodes reachable from node 2 by a path containing exactly one red edge.

R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

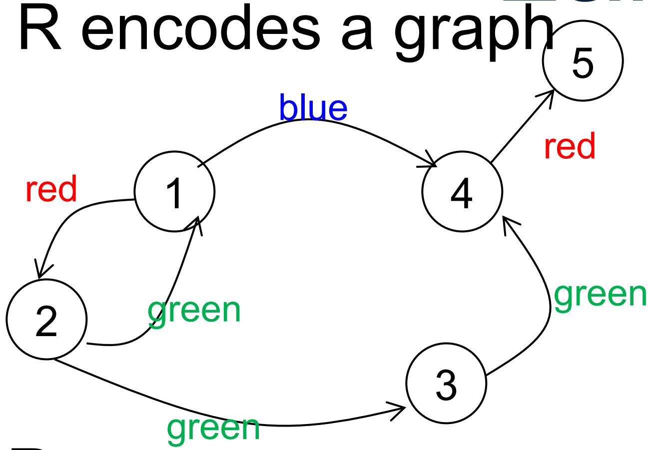
Automaton:



NoRed(2). :- .

Labeled Graphs

R encodes a graph

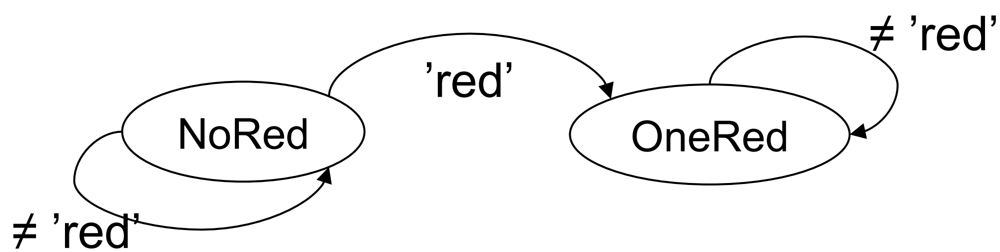


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Find all nodes reachable from node 2 by a path containing exactly one red edge.

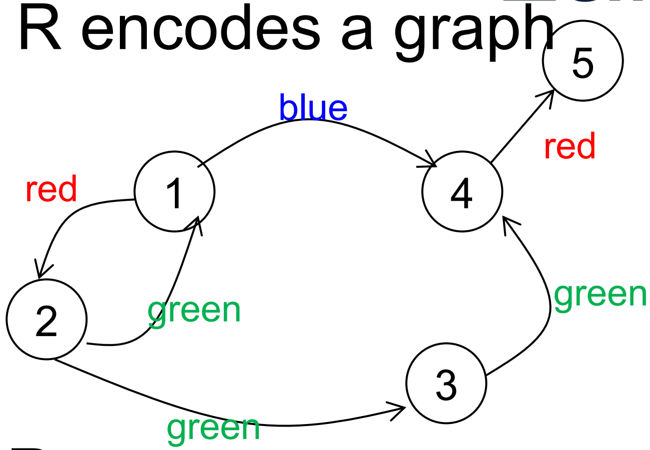
Automaton:



NoRed(2). :- .
NoRed(y) :- NoRed(x), R(x,y,c), c!='red'.

Labeled Graphs

R encodes a graph

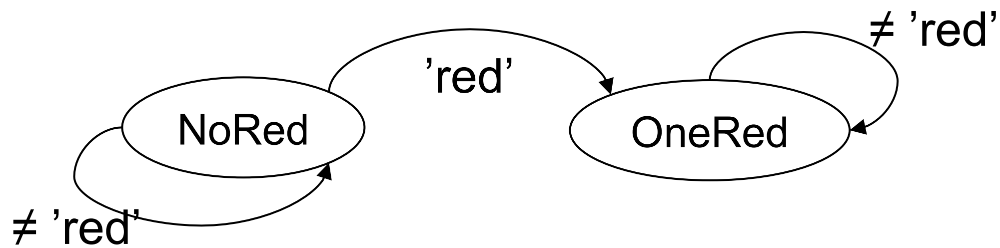


R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Find all nodes reachable from node 2 by a path containing exactly one red edge.

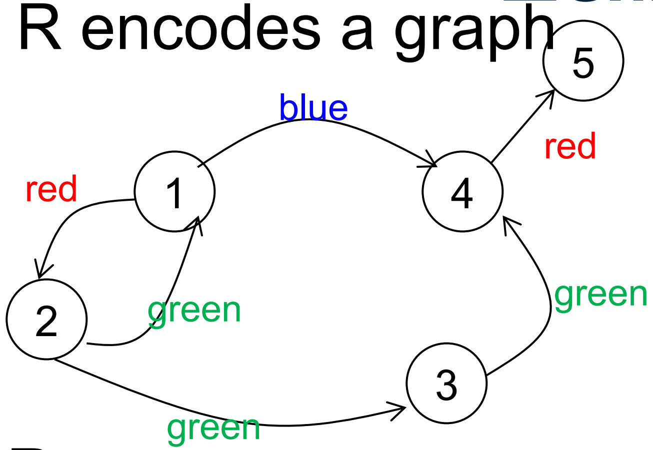
Automaton:




```
NoRed(2).   :- .
NoRed(y)    :- NoRed(x), R(x,y,c), c!='red'.
OneRed(y) :- NoRed(x), R(x,y,'red').
```

Labeled Graphs

R encodes a graph

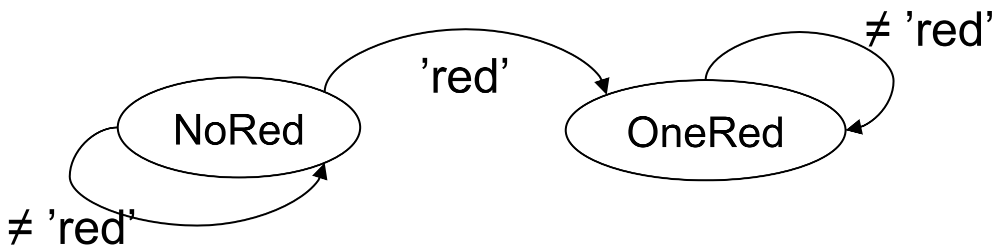


Find all nodes reachable from node 2 by a path containing exactly one red edge.

R=

1	2	red
2	1	green
2	3	green
1	4	blue
3	4	green
4	5	red

Automaton:




```
NoRed(2).   :- .
NoRed(y)    :- NoRed(x), R(x,y,c), c!='red'.
OneRed(y) :- NoRed(x), R(x,y,'red').
OneRed(y) :- OneRed(x), R(x,y,c), c!='red'.
```

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

- Single IDB
  - Called: Common Table Expression, CTE
  - Cannot write Odd/Even, Red/NoRed, etc

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

- Single IDB
  - Called: Common Table Expression, CTE
  - Cannot write Odd/Even, Red/NoRed, etc
- Linear query only
  - Cannot write T(x,y) :- T(x,z), T(z,y)

Discussion: Recursion in SQL

SQL has limited form of recursion, BUT:

- Single IDB
  - Called: Common Table Expression, CTE
  - Cannot write Odd/Even, Red/NoRed, etc
- Linear query only
  - Cannot write T(x,y) :- T(x,z), T(z,y)
- Has bag semantics (really???)
  - May not terminate!

Discussion: Recursion in SQL

Relation T is called a Common Table Expression CTE

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z), T(z,y)
```

```
with recursive T as(
    select * from R
    union
    select distinct R.x, T.y
    from R, T
    where R.y=T.x
)
select * from T;
```

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *together*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *together*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

$\text{Odd} \coloneqq \emptyset; \text{Even} \coloneqq \emptyset;$

repeat

$\text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R);$

$\text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R);$

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *together*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

$\text{Odd} \coloneqq \emptyset$; $\text{Even} \coloneqq \emptyset$;

repeat

$\text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R)$;

$\text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R)$;

$\text{Odd} := \text{Odd}_{\text{new}}$

$\text{Even} := \text{Even}_{\text{new}}$

Naïve Evaluation Algorithm

- When multiple IDBs: need to compute their new values *together*:

```
Odd(x,y) :- R(x,y)
Even(x,y) :- Odd(x,z),R(z,y)
Odd(x,y) :- Even(x,z),R(z,y)
```

$\text{Odd} \coloneqq \emptyset;\ \text{Even} \coloneqq \emptyset;$

repeat

$\text{Even}_{\text{new}} := \Pi_{x,y}(\text{Odd} \bowtie R);$

$\text{Odd}_{\text{new}} := R \cup \Pi_{x,y}(\text{Even} \bowtie R);$

if $\text{Odd}=\text{Odd}_{\text{new}} \wedge \text{Even}=\text{Even}_{\text{new}}$

then break

$\text{Odd}:=\text{Odd}_{\text{new}}$

$\text{Even}:=\text{Even}_{\text{new}}$

Naïve Evaluation Algorithm

The Naïve Evaluation Algorithm:

- Always terminates
- Always terminates in a number of steps that is polynomial in the size of the database

Before we show this, a digression: **monotone queries**

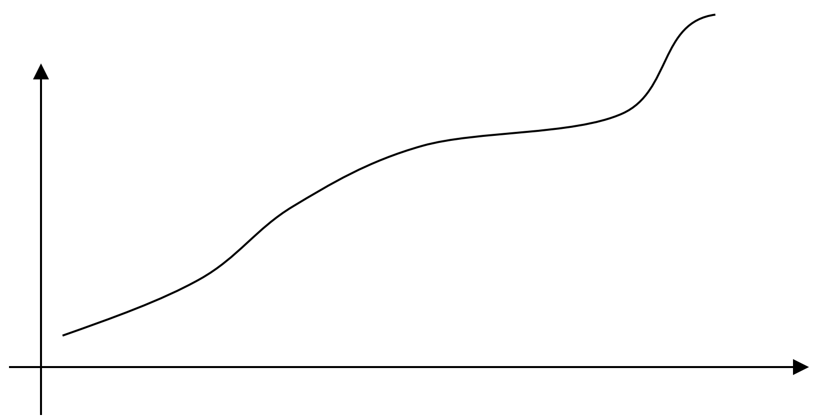
Outline

- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions
- Semi-naïve Evaluation Algorithm

Review: Montone Functions

- A function f(x) is called *monotonically increasing*, or just *monotone* if:

$$\textbf{If} \quad x \leq y \quad \textbf{then} \quad f(x) \leq f(y)$$



Monotone Queries

- A query with input relations R, S, T, … is called *monotone* if, whenever we increase a relation, the query answer also increases (or stays the same)
- *Increase* here means *larger set*

Monotone Queries

- A query with input relations R, S, T, … is called *monotone* if, whenever we increase a relation, the query answer also increases (or stays the same)
- *Increase* here means *larger set*
- Mathematically

$$\textbf{If } R \subseteq R', S \subseteq S', \ldots \textbf{ then } Q(R, S, \ldots) \subseteq Q(R', S', \ldots)$$

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

NON-MONOTONE

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

Supplier(sno,sname,scity,sstate)

Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

NON-MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

NON-MONOTONE

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno != 2 )
```

Supplier(sno,sname,scity,sstate)
Supply(sno,pno,price)

Which Queries are Monotone?

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno = 2
```

MONOTONE

```
SELECT x.city, count(*)
FROM Supplier x
GROUP BY x.city
```

```
SELECT DISTINCT x.sno, x.name
FROM Supplier x, Supply y
WHERE x.sno = y.sno and y.pno != 2
```

MONOTONE

NON-MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno IN (SELECT y.sno
                FROM Supply y
                WHERE y.pno = 2 )
```

MONOTONE

```
SELECT x.sno, x.sname FROM Supplier x
WHERE x.sno NOT IN (SELECT y.sno
                    FROM Supply y
                    WHERE y.pno != 2 )
```

NON-MONOTONE

Which Ops are Monotone?

- Selection: σ_{pred}
- Projection: $\Pi_{A,B,\ldots}$
- Join: $\bowtie$
- Union: $\cup$
- Difference: $-$
- Group-by-sum: $\gamma_{A,B,sum(C)}$

Which Ops are Monotone?

- Selection: σ_{pred} **MONOTONE**
- Projection: $\Pi_{A,B,\ldots}$ **MONOTONE**
- Join: $\bowtie$ **MONOTONE**
- Union: $\cup$ **MONOTONE**
- Difference: $-$ **NON-MONOTONE**
- Group-by-sum: $\gamma_{A,B,sum(C)}$ **NON-MONOTONE**

Fun Fact

- A SELECT-FROM-WHERE query (without aggregates or subqueries) is monotone

```
SELECT [DISTINCT] …
FROM R1 x1, R2 x2, …
WHERE …
```

Fun Fact

- A SELECT-FROM-WHERE query (without aggregates or subqueries) is monotone

```
SELECT [DISTINCT] …
FROM R1 x1, R2 x2, …
WHERE …
```

- Proof: the nested loop semantics! When we add tuples to one relation, we cannot lose answers:

```
for x1 in R1 do:
  for x2 in R2 do:
    …
```

Tips for Writing SQL Queries

- If the English formulation of a query is non-monotone, then you *<u>need</u>* to use a subquery OR aggregate in SQL

Return SUPPLIERS who supply some product with price > $10000

Return SUPPLIERS who supply only products with price > $10000

Back to Datalog

Naïve Algorithm:

- Always terminates
- Terminates in a number of steps that is polynomial in the size of the database
- This is cool!
  Compare with java, python, etc

Assumptions:

- Set semantics only
- Monotone rules only
- No "value invention"

Will show this next

$IDB_0 \coloneqq \emptyset; \quad t \coloneqq 0$
repeat $IDB_{t+1} \coloneqq USPJ(IDB_t); \; t \coloneqq t+1$
until no more change

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

$IDB_0 := \emptyset; \quad t := 0$
repeat $IDB_{t+1} := USPJ(IDB_t); \; t := t + 1$
until no more change

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction

$IDB_0 := \emptyset; \quad t := 0$
repeat $IDB_{t+1} := USPJ(IDB_t);\ t := t + 1$
until no more change

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

$IDB_0 := \emptyset; \quad t := 0$
repeat $IDB_{t+1} := USPJ(IDB_t); \; t := t + 1$
until no more change

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0 (= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:

$$\text{USPJ}(IDB_t) \subseteq \text{USPJ}(IDB_{t+1})$$

$IDB_0 := \emptyset;\quad t := 0$
repeat $IDB_{t+1} := USPJ(IDB_t);\ t := t + 1$
until no more change

Naïve Evaluation Algorithm

Fact: every USPJ query is monotone
Proof: uses only $\sigma, \Pi, \bowtie, \cup$

Fact: the IDBs increase: $IDB_t \subseteq IDB_{t+1}$

Proof: by induction $IDB_0(= \emptyset) \subseteq IDB_1$

Assuming $IDB_t \subseteq IDB_{t+1}$ we have:
$IDB_{t+1} = \text{USPJ}(IDB_t) \subseteq \text{USPJ}(IDB_{t+1}) = IDB_{t+2}$

Naïve Evaluation Algorithm

Consequence: The naïve algorithm *terminates*, in $O(n^k)$ steps, where:

- n = number of distinct values in the DB
- k = arity of widest IDB relation

Proof: IDBs increases to ≤ $O(n^k)$ facts

Recap

Naïve Algorithm:

- Always terminates
- Terminates in a number of steps that is polynomial in the size of the database
- This is cool! Compare with java, python, etc

Assumptions:

- Set semantics only
- Monotone rules only
- No “value invention”

Will show this next

Outline

- Recap: Datalog basics
- Naïve Evaluation Algorithm
- Monotone Queries
- Non-monotone Extensions

Non-monotone Extensions

- Aggregates
- Grouping
- Negation

No standard syntax

We will follow Souffle

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Aggregates

Q(m) :- m = min x : { Actor(x, y, _), y = 'John' }

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Aggregates

Q(m) :- m = min x : { Actor(x, y, _), y = 'John' }

Meaning (in SQL)

```
SELECT min(id) as m
FROM Actor as a
WHERE a.name = 'John'
```

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Aggregates

Q(m) :- m = min x : { Actor(x, y, _), y = ‘John’ }

Meaning (in SQL)

```
SELECT min(id) as m
FROM Actor as a
WHERE a.name = ‘John’
```

Aggregates in Souffle:

- count
- min
- max
- sum

Grouping

Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }

Grouping

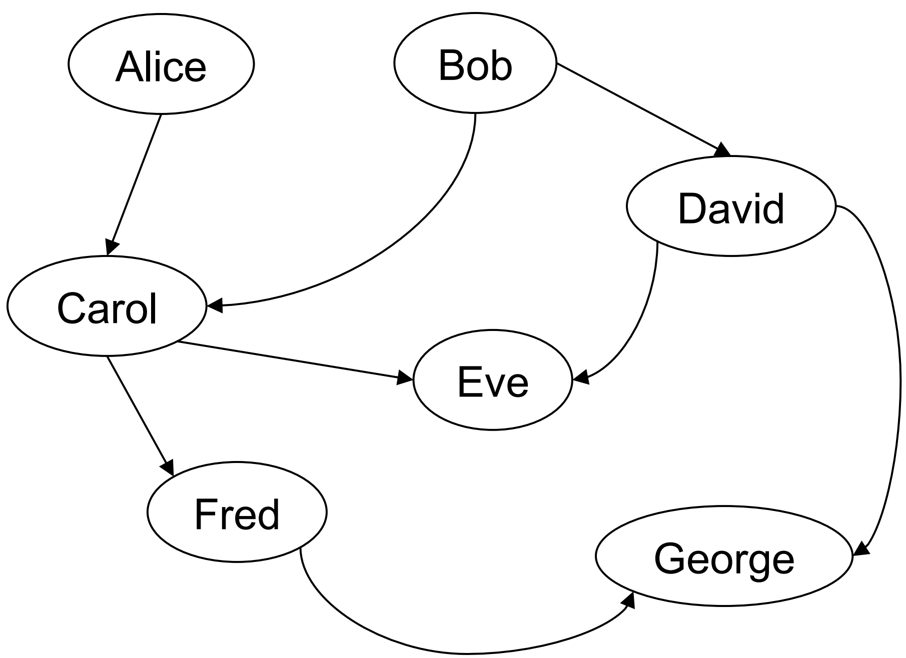
Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }

Meaning (in SQL)

```
SELECT m.year, count(*)
FROM Movie as m
GROUP BY m.year
```

Examples

A genealogy database (parent/child)



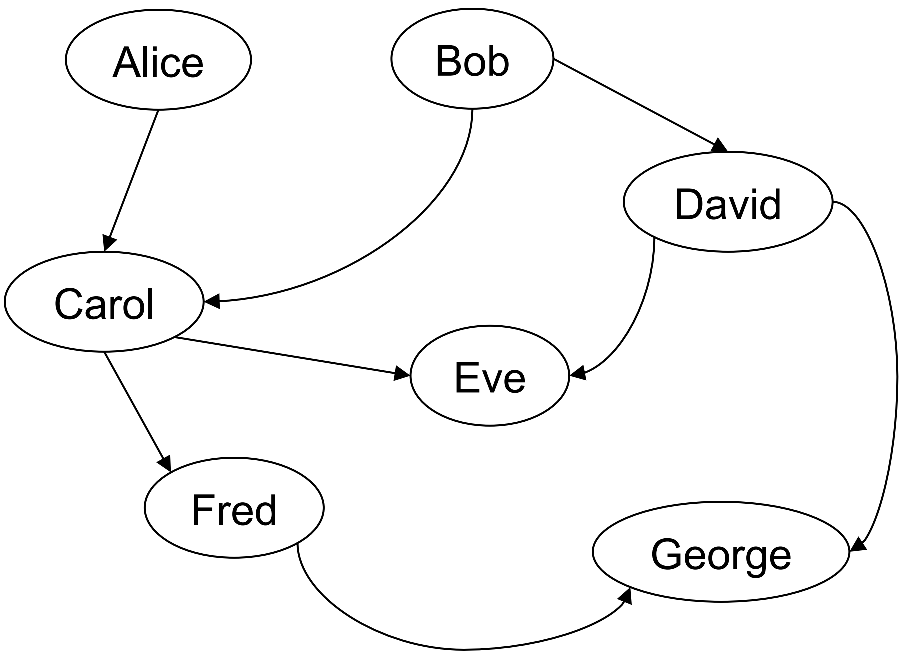
ParentChild

p	c
Alice	Carol
Bob	Carol
Bob	David
Carol	Eve
…	

ParentChild(p,c)

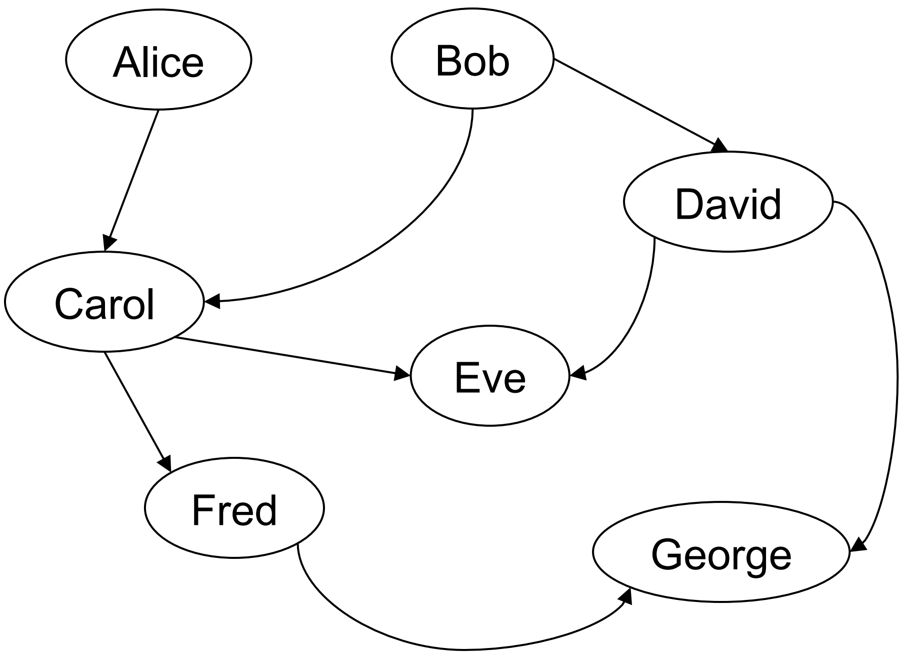
Count Descendants

For each person, count his/her descendants



Count Descendants

For each person, count his/her descendants

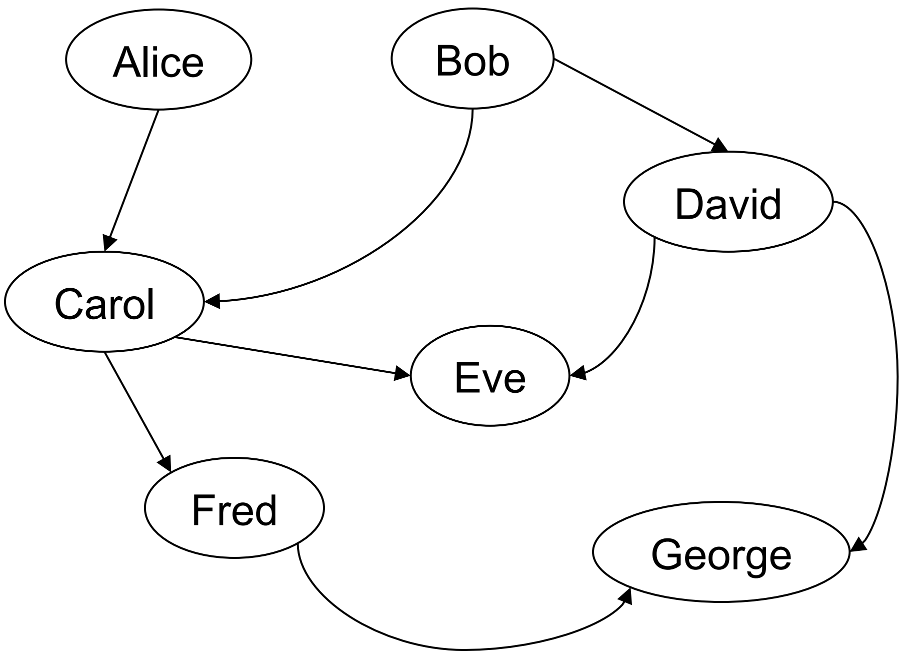


Answer

p	cnt
Alice	4
Bob	5
Carol	3
David	2
Fred	1

Count Descendants

For each person, count his/her descendants



Answer

p	cnt
Alice	4
Bob	5
Carol	3
David	2
Fred	1

Note: Eve and George do not appear in the answer (why?)

Count Descendants

Compute transitive closure of ParentChild

```
// for each person, compute his/her descendants
```

Count Descendants

Compute transitive closure of ParentChild

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Count Descendants

For each person, compute the total number of descendants

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Count Descendants

For each person, compute the total number of descendants

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.
```

Count Descendants

How many descendants does Alice have?

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.
```

Count Descendants

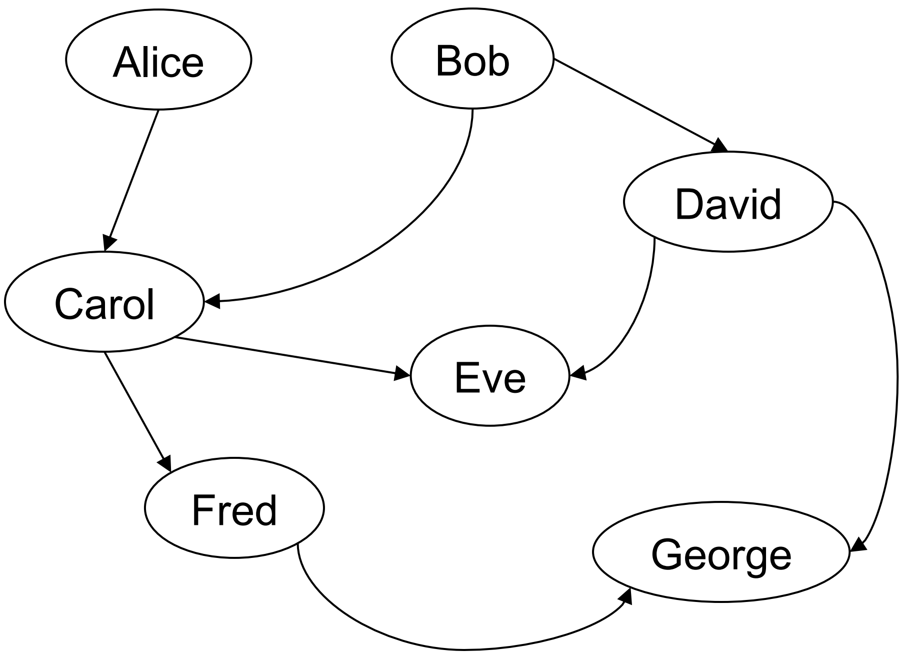
How many descendants does Alice have?

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,_) }.

// Find the number of descendants of Alice
Q(d) :- T(p,d), p = "Alice".
```

ParentChild(p,c)

Negation: use "!"

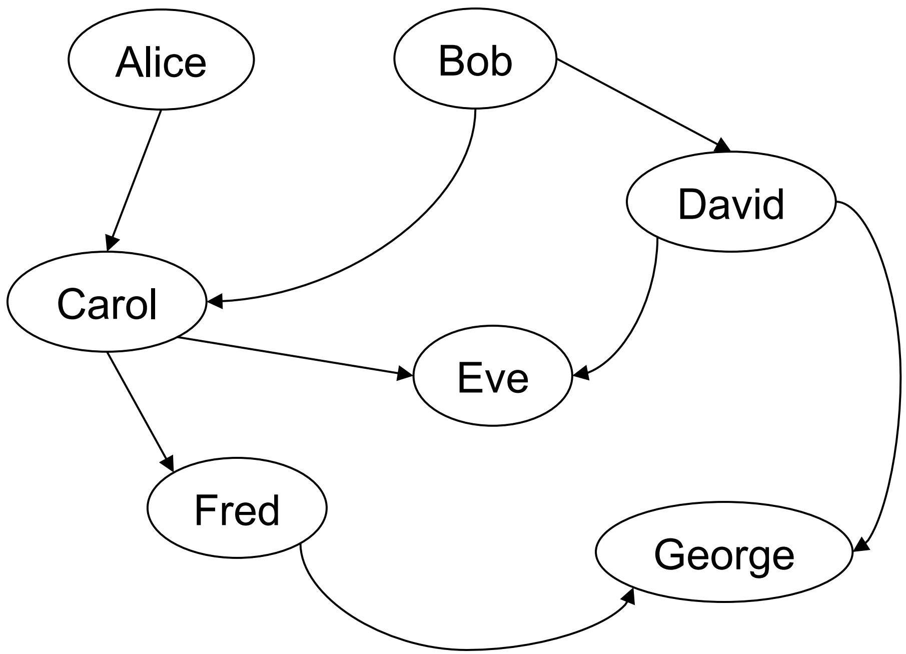
Find all descendants of Bob that are not descendants of Alice



ParentChild(p,c)

Negation: use “!”

Find all descendants of Bob that are not descendants of Alice



Answer

x
David

Negation: use "!"

Find all descendants of Bob that are not descendants of Alice

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Negation: use “!”

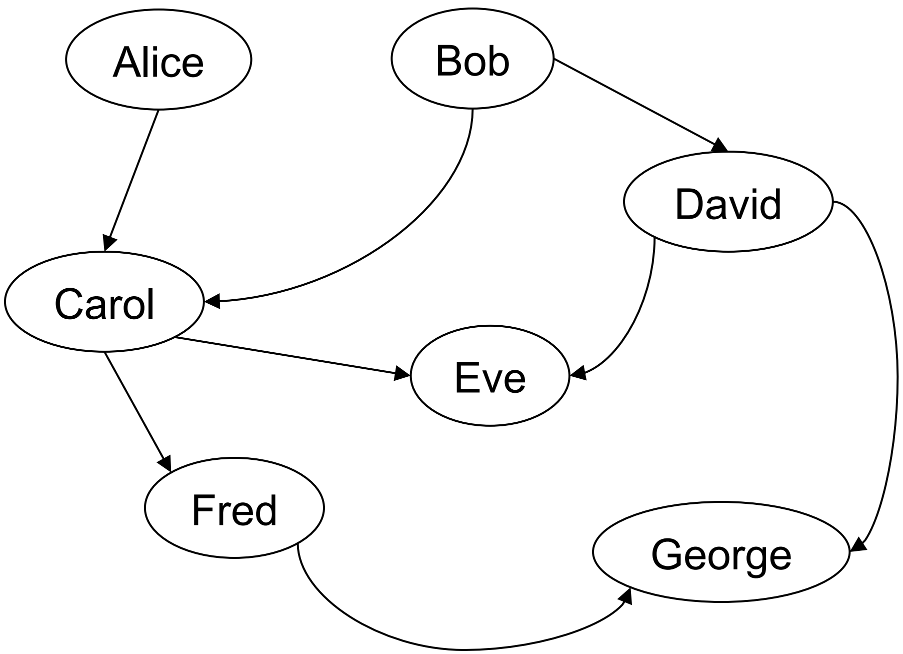
Find all descendants of Bob that are not descendants of Alice

```
// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// Compute the answer: notice the negation
Q(x) :- D(“Bob”,x), !D(“Alice”,x).
```

ParentChild(p,c)

Same Generation

Two people are in the *same generation* if they are descendants at the same generation of some common ancestor



SG

p1	p2
Carol	David
Eve	George
Fred	George
Fred	Eve

Same Generation

Compute pairs of people at the same generation

```
// common parent
```

ParentChild(p,c)

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)
```

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)


// parents at the same generation
```

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
```

ParentChild(p,c)

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
```

Problem: this includes answers like SG(Carol, Carol)

And also SG(Eve, George), SG(George, Eve)

How to fix?

ParentChild(p,c)

Same Generation

Compute pairs of people at the same generation

```
// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y), x < y


// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y),
               SG(p,q),  x < y
```

Stratified Datalog

Recursion conflicts with non-monotone queries

- Example: what does this mean?

```
Happy(Bob):- !Happy(Alice).
Happy(Alice) :- !Happy(Bob).
```

- A program is ***stratified*** if it can be partitioned into *strata,* such that every IDB predicate in a non-monotone position has been defined in an earlier stratum

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
```

Stratum 1

```
T(p,c) :- D(p,_), c = count : { D(p,_) }.
Q(d) :- T(p,d), p = "Alice".
```

Stratum 2

May use D in an agg since it was defined in previous stratum

Stratified Datalog

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
---------------------------------------------
T(p,c) :- D(p,_), c = count : { D(p,_) }.
Q(d) :- T(p,d), p = “Alice”.
```

Stratum 1

Stratum 2

May use D in an agg since it was defined in previous stratum

```
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
---------------------------------------------
Q(x) :- D(“Alice”,x), !D(“Bob”,x).
```

Stratum 1

Stratum 2

May use !D

```
Happy(Bob):- !Happy(Alice).
Happy(Alice) :- !Happy(Bob).
```

Non-stratified

Some Datalog Optimizations

- Every USPJ optimized traditionally
- Semi-naïve evaluation
- Magic sets
- Asynchronous execution

Summary

- Datalog = light-weight syntax, recursion
- Data independence, optimizations
- Limitations:
  - Monotone queries work great
  - Non-monotone queries: various restrictions