Win-Move is Coordination-Free (Sometimes)

Daniel Zinn[1] TJ Green[1,2] Bertram Ludäscher[2]

[1] LogicBlox, Inc.
[2] University of California, Davis

March 13th 2012

Intro

Shared Vision

Develop Datalog-inspired language(s) for writing *correct* and *scalable* distributed programs.

This talk: What can be computed without coordination?

Conjecture: CALM (Consistency And Logical Monotonicity) [Hel10]

A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

Theorem: CALM [ANVdB11]

Q can be distributedly computed by a transducer that is coordination-free if and only if Q is monotone.

Intro

Shared Vision

Develop Datalog-inspired language(s) for writing *correct* and *scalable* distributed programs.

This talk: What can be computed without coordination?

Conjecture: CALM (Consistency And Logical Monotonicity) [Hel10]

A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

Theorem: CALM [ANVdB11]

Q can be distributedly computed by a transducer that is coordination-free if and only if Q is monotone.

Intro

Shared Vision

Develop Datalog-inspired language(s) for writing *correct* and *scalable* distributed programs.

This talk: What can be computed without coordination?

Conjecture: CALM (Consistency And Logical Monotonicity) [Hel10]

A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

Theorem: CALM [ANVdB11]

Q can be distributedly computed by a transducer that is coordination-free if and only if Q is monotone.

Intro

Shared Vision

Develop Datalog-inspired language(s) for writing *correct* and *scalable* distributed programs.

This talk: What can be computed without coordination?

Conjecture: CALM (Consistency And Logical Monotonicity) [Hel10]

A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

Theorem: CALM [ANVdB11]

Q can be distributedly computed by a transducer that is coordination-free if and only if Q is monotone.

Contributions

Distributed Algorithm for Win-Move

- No global barriers

Beyond Win-Move: Semi-Monotone Datalog$^{\neg}$

- Well-suited for distributed evaluation
- Different levels of safety restrictions imply different level of coordination-freeness

Formal Investigation of Coordination-Free Query Classes

- Make precise a notion of coordination-freeness, parameterized by knowledge about input distribution
- Demonstrate that parameters give rise to strict hierarchy of coordination-free query classes:

$$\mathcal{M} = \mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$$

Distributed Reasoning

Centralized

I

e(a,b). e(b,c).

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

I

e(a,b). e(b,c).

Find rule & satisfying variable assignment; insert head into I

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

I

e(a,b). e(b,c).

Find rule & satisfying variable assignment

r_1 : X $\mapsto$ b, Y $\mapsto$ c

Program P

```
tc(X,Y) ← e(X,Y).
tc(X,Z) ← e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

I

e(a,b). e(b,c).
tc(b,c).

Find rule & satisfying variable assignment

r_1 : X $\mapsto$ b, Y $\mapsto$ c

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Centralized

I

e(a,b). e(b,c).
tc(b,c). tc(a,c).

Find rule & satisfying variable assignment

r_2: X↦a, Y↦b, Z↦c

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Centralized

I

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment

r_1: X $\mapsto$ a, Y $\mapsto$ b

Program P

```
tc(X,Y) ← e(X,Y).
tc(X,Z) ← e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

I

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

I

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed

Distributed Reasoning

Centralized

I

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

Distribute relations eg, via Hash h.

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed

Distribute relations eg, via Hash h. Partition e according to second, tc first attribute. Let h: h(a)=1. h(b)=2. h(c)=3.

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

I_2

e(a,b).

I_3

e(b,c).

Distribute relations eg, via Hash h. Partition e according to second, tc first attribute. Let h: h(a)=1. h(b)=2. h(c)=3.

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

I_2

e(a,b).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

I_2

e(a,b).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.

h_3, r_1 : X $\mapsto$ b, Y $\mapsto$ c **yields** tc(b,c).

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

I_2

e(a,b).
tc(b,c).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.

h_3, r_1: X $\mapsto$ b, Y $\mapsto$ c **yields** tc(b,c).

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed

I_1

I_2

e(a,b).
tc(b,c).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.
h_2, r_2: X↦ a, Y ↦ b, Z ↦ c
yields tc(a,c).

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into *I*

Program *P*

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed

I_1

tc(a,c).

I_2

e(a,b).
tc(b,c).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.
h_2, r_2: X↦ a, Y ↦ b, Z ↦ c
yields tc(a,c).

Program *P*

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

tc(a,c).

I_2

e(a,b).
tc(b,c).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.

h_2, r_1: X $\mapsto$ a, Y $\mapsto$ b **yields** tc(a,b).

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

tc(a,c).
tc(a,b).

I_2

e(a,b).
tc(b,c).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.

h_2, r_1: X $\mapsto$ a, Y $\mapsto$ b **yields** tc(a,b).

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

tc(a,c).
tc(a,b).

I_2

e(a,b).
tc(b,c).

I_3

e(b,c).

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed

I_1

tc(a,c).
tc(a,b).

I_2

e(a,b).
tc(b,c).

I_3

e(b,c).

Encode location of tuple into tuple itself, via *location specifier*

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into *I*

Program *P*

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed

I_1

tc(1,a,c).
tc(1,a,b).

I_2

e(2,a,b).
tc(2,b,c).

I_3

e(3,b,c).

Encode location of tuple into tuple itself, via *location specifier*

Program *P*

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Find rule & satisfying variable assignment; insert head into I

Distributed

I_1

tc(1,a,c).
tc(1,a,b).

I_2

e(2,a,b).
tc(2,b,c).

I_3

e(3,b,c).

Encode location of tuple into tuple itself, via *location specifier*

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

Program P'

```
tc(h(X),X,Y)←e(h(Y),X,Y).
tc(h(X),X,Z)←
e(h(Y),X,Y),tc(h(Y),Y,Z).
```

Distributed Reasoning

Centralized

$I = I_1 \cup I_2 \cup I_3$

e(a,b). e(b,c).
tc(b,c). tc(a,c). tc(a,b).

Distributed

I_1

tc(1,a,c).
tc(1,a,b).

I_2

e(2,a,b).
tc(2,b,c).

I_3

e(3,b,c).

Find rule & satisfying variable assignment; insert head into I

Find host & rule & valuation where local database satisfies body; send derived head-fact to destination.

Program P

```
tc(X,Y)←e(X,Y).
tc(X,Z)←e(X,Y),tc(Y,Z).
```

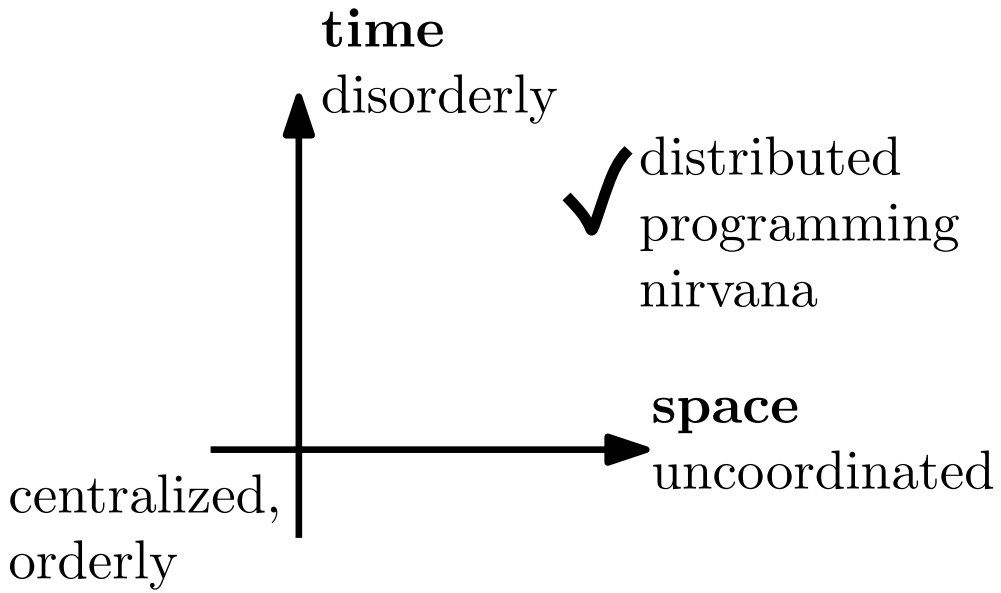
Program P'

```
tc(h(X),X,Y)←e(h(Y),X,Y).
tc(h(X),X,Z)←
e(h(Y),X,Y),tc(h(Y),Y,Z).
```

Goal: Generalize Beyond Positive Datalog

When is Distributed Reasoning Safe?

Separate concerns:



Time-Safety

Is it safe to execute rules out-of order?

Space-Safety

Are local inferences valid globally?

Can all global inferences be done with only local knowledge?

Time-Safety Violation: Negation

```
b ← c.
a ← ¬b.
```

Input $\{c\}$

Rule 1 first.

Output

c. b.

Rule 2 first.

Output

c. a. b.

Disorderly evaluation in general:

- non-deterministic
- not desired semantics

Time-Safety Violation: Negation

```
b ← c.
a ← ¬b.
```

Input $\{c\}$

Rule 1 first.

Output

c. b.

Rule 2 first.

Output

c. a. b.

Disorderly evaluation in general:

- non-deterministic
- not desired semantics

Deploy Coordination [MAC+11, ACHM11]

Strategy: Mimic Stratified Execution

- decompose into multiple semi-positive programs,
- allow disorderly evaluation within one stratum, and
- use global barriers in between strata

Well-founded Semantics / Locally Stratified Programs

- perform alternating fixpoint evaluation,
- barriers between alternating rounds

Number of Barriers (cf. CRON-Conjecture [Hel10])

- for stratified program is its maximum stratum number
- for well-founded programs depends on input instance

Can we do better?

Deploy Coordination [MAC+11, ACHM11]

Strategy: Mimic Stratified Execution

- decompose into multiple semi-positive programs,
- allow disorderly evaluation within one stratum, and
- use global barriers in between strata

Well-founded Semantics / Locally Stratified Programs

- perform alternating fixpoint evaluation,
- barriers between alternating rounds

Number of Barriers (cf. CRON-Conjecture [Hel10])

- for stratified program is its maximum stratum number
- for well-founded programs depends on input instance

Can we do better?

Deploy Coordination [MAC+11, ACHM11]

Strategy: Mimic Stratified Execution

- decompose into multiple semi-positive programs,
- allow disorderly evaluation within one stratum, and
- use global barriers in between strata

Well-founded Semantics / Locally Stratified Programs

- perform alternating fixpoint evaluation,
- barriers between alternating rounds

Number of Barriers (cf. CRON-Conjecture [Hel10])

- for stratified program is its maximum stratum number
- for well-founded programs depends on input instance

Can we do better?

Deploy Coordination [MAC+11, ACHM11]

Strategy: Mimic Stratified Execution

- decompose into multiple semi-positive programs,
- allow disorderly evaluation within one stratum, and
- use global barriers in between strata

Well-founded Semantics / Locally Stratified Programs

- perform alternating fixpoint evaluation,
- barriers between alternating rounds

Number of Barriers (cf. CRON-Conjecture [Hel10])

- for stratified program is its maximum stratum number
- for well-founded programs depends on input instance

Can we do better?

Outline

Outline

The Win-Move Game

```
win(X) ← move(X,Y), ¬win(Y).
```

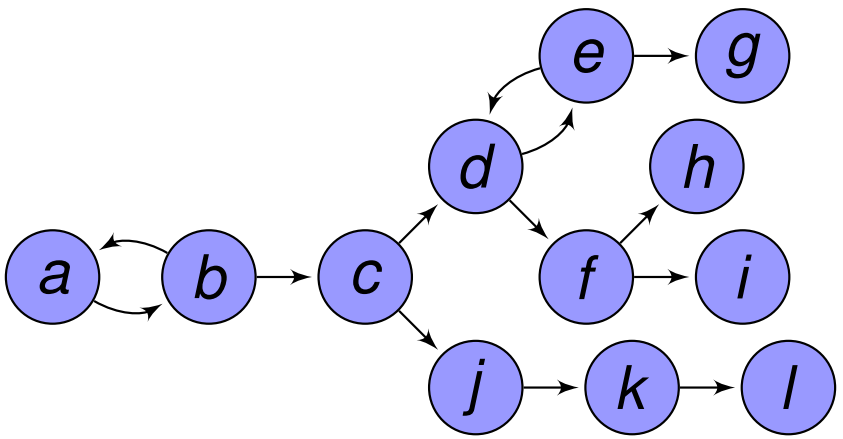
- Negation through recursion
- Canonical example of a non-stratifiable, non-monotone program
- Well-founded model is the solution of a two-player game...

Win-Move Rules / Example

Rules

- Two players in turn move a pebble on the `move` graph
- `move` is a graph encoding legal moves
- You lose if you cannot move

Examples

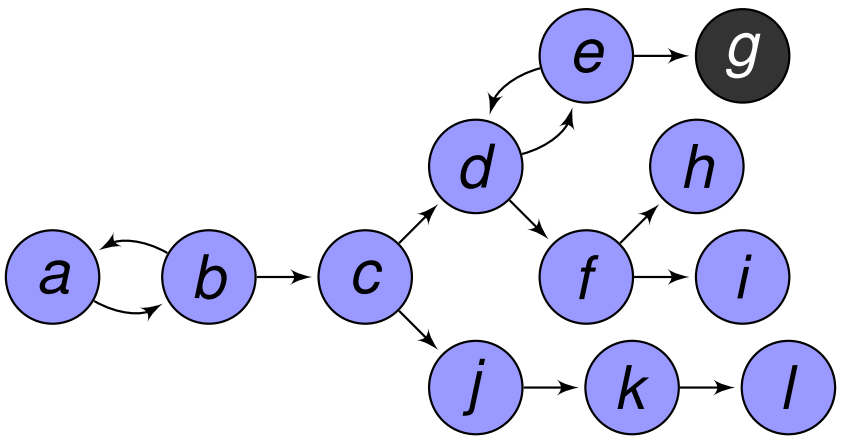


Win-Move Rules / Example

Rules

- Two players in turn move a pebble on the `move` graph
- `move` is a graph encoding legal moves
- You lose if you cannot move

Examples

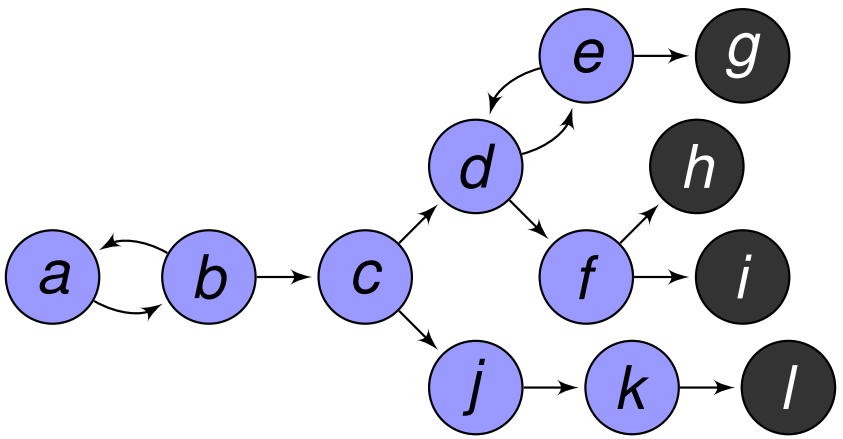


Win-Move Rules / Example

Rules

- Two players in turn move a pebble on the `move` graph
- `move` is a graph encoding legal moves
- You lose if you cannot move

Examples

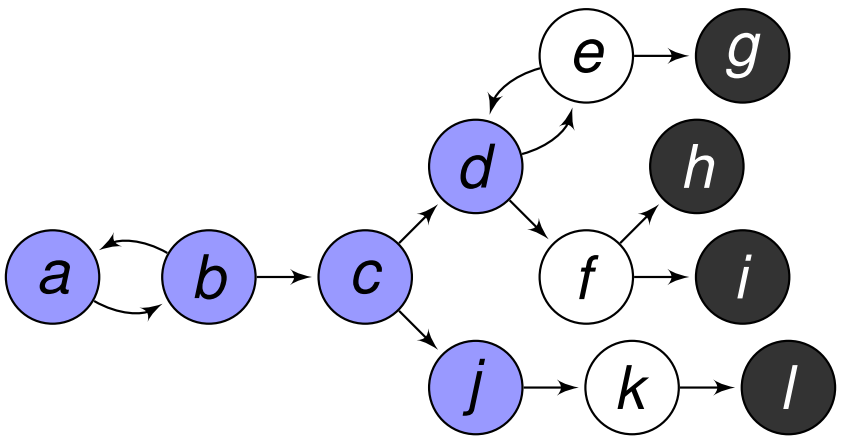


Win-Move Rules / Example

Rules

- Two players in turn move a pebble on the `move` graph
- `move` is a graph encoding legal moves
- You lose if you cannot move

Examples

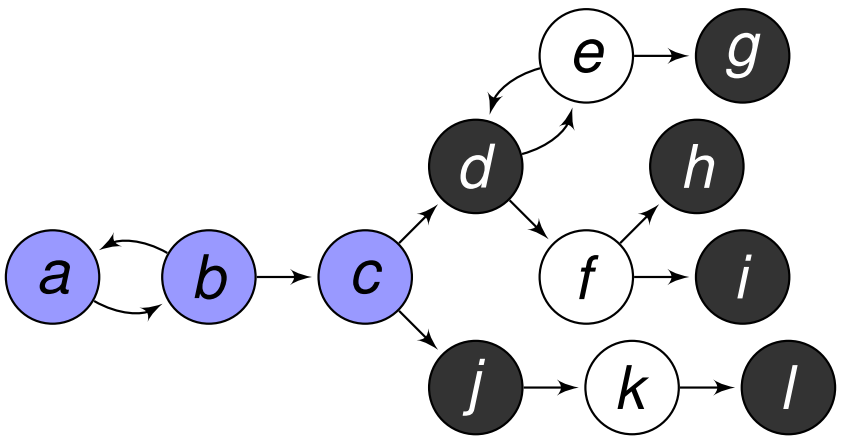


Win-Move Rules / Example

Rules

- Two players in turn move a pebble on the `move` graph
- `move` is a graph encoding legal moves
- You lose if you cannot move

Examples

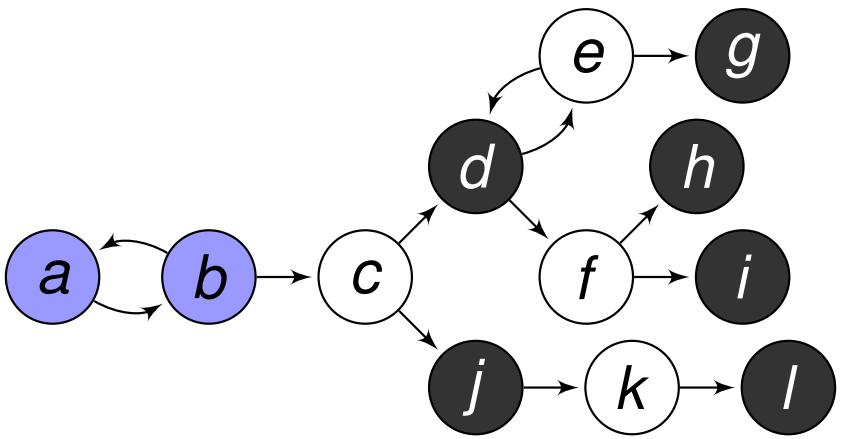


Win-Move Rules / Example

Rules

- Two players in turn move a pebble on the `move` graph
- `move` is a graph encoding legal moves
- You lose if you cannot move

Examples

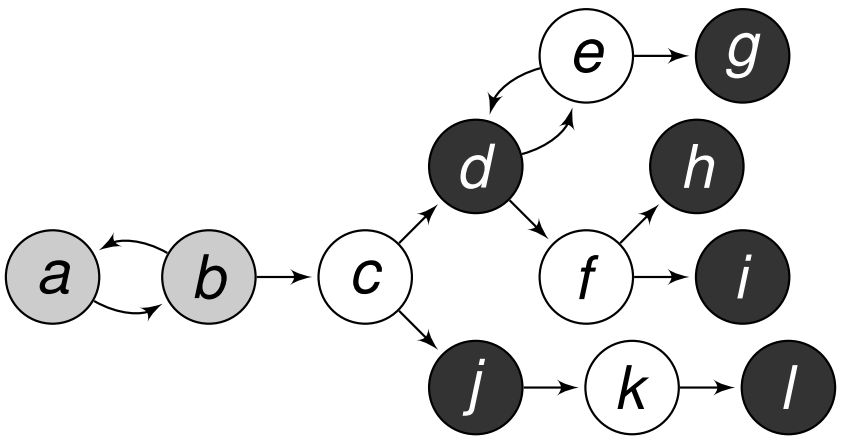


Win-Move Rules / Example

Rules

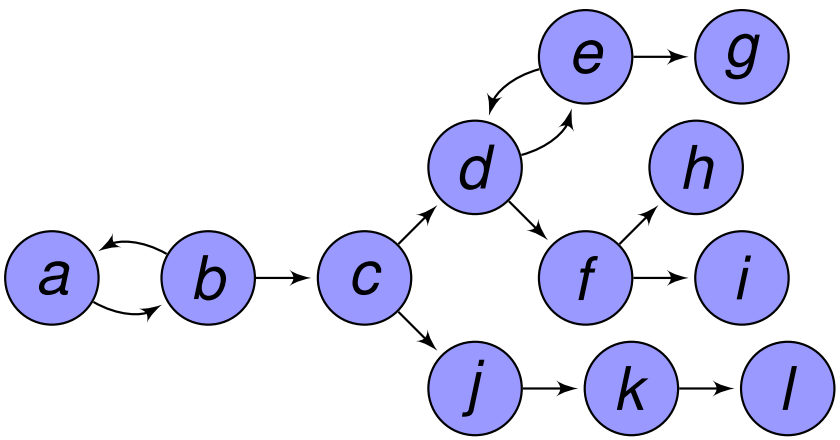
- Two players in turn move a pebble on the `move` graph
- `move` is a graph encoding legal moves
- You lose if you cannot move

Examples



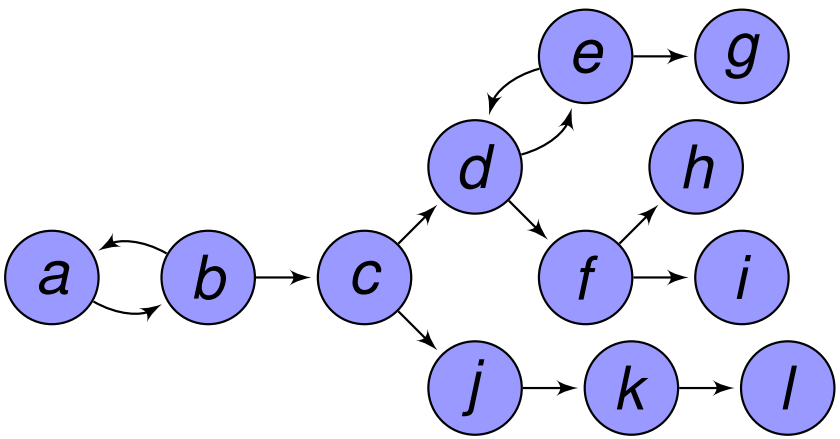
Eample: Doubled Program for Win-Move

P_o: $\texttt{win}^o\texttt{(X)} \leftarrow \texttt{move(X,Y)},\ \neg\texttt{win}^u\texttt{(X)}.$
P_u: $\texttt{win}^u\texttt{(X)} \leftarrow \texttt{move(X,Y)},\ \neg\texttt{win}^o\texttt{(X)}.$



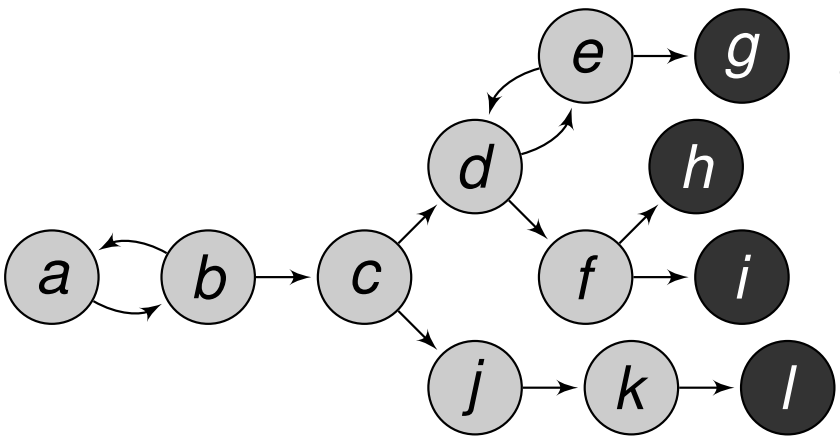
Eample: Doubled Program for Win-Move

P_o: `may_win(X)` $\leftarrow$ `move(X,Y),` $\neg$`won(X).`
P_u: `won(X)` $\leftarrow$ `move(X,Y),` $\neg$`may_win(X).`



Eample: Doubled Program for Win-Move

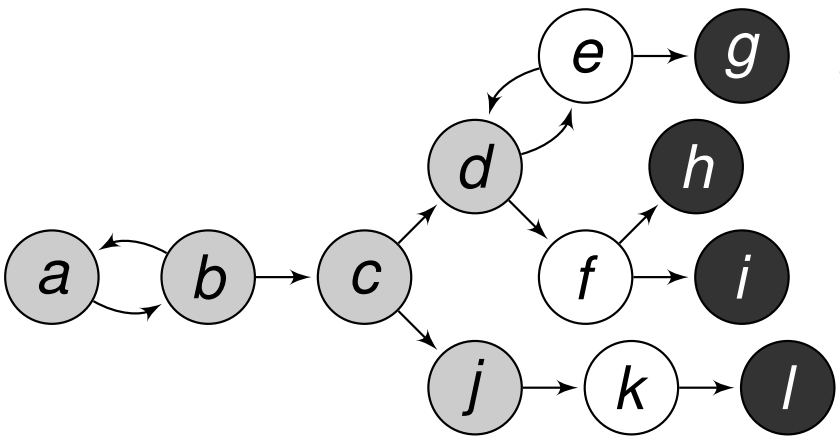
P_o: `may_win(X)` $\leftarrow$ `move(X,Y),` $\neg$`won(X).`
P_u: `won(X)` $\leftarrow$ `move(X,Y),` $\neg$`may_win(X).`



	estimates
V_0	`may_win` $= \{a, b, c, d, e, f, j, k\}$

Eample: Doubled Program for Win-Move

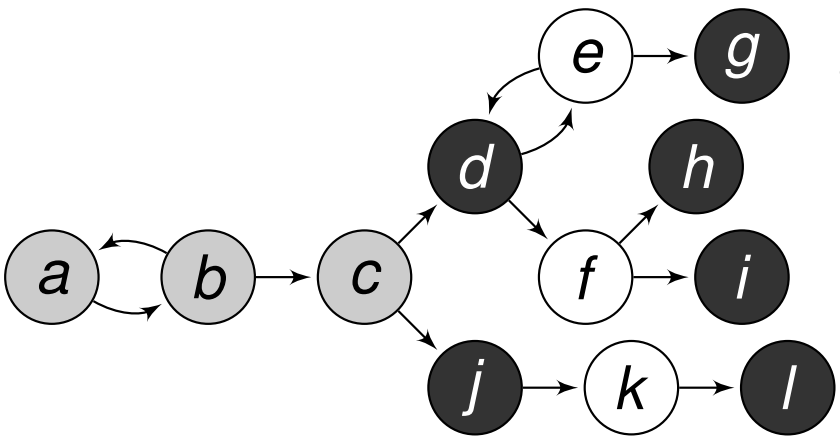
P_o: `may_win(X)` $\leftarrow$ `move(X,Y),` $\neg$`won(X).`
P_u: `won(X)` $\leftarrow$ `move(X,Y),` $\neg$`may_win(X).`



	estimates
V_0	`may_win` $= \{a, b, c, d, e, f, j, k\}$
U_1	`won` $= \{e, f, k\}$

Eample: Doubled Program for Win-Move

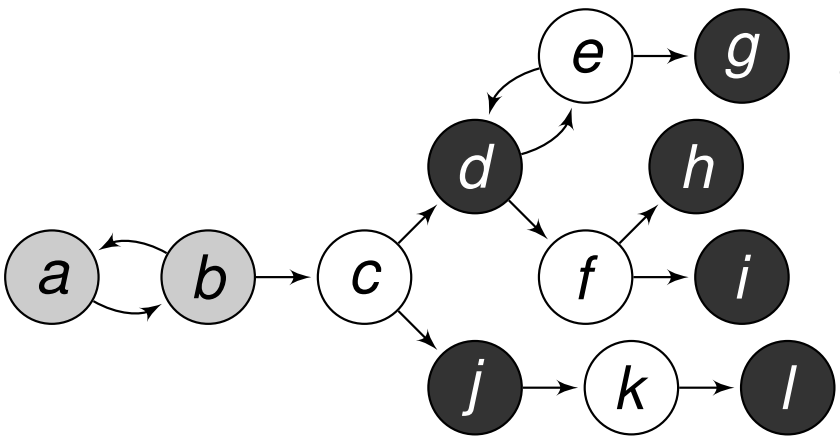
P_o: `may_win(X)` $\leftarrow$ `move(X,Y),` $\neg$`won(X).`
P_u: `won(X)` $\leftarrow$ `move(X,Y),` $\neg$`may_win(X).`



	estimates
V_0	`may_win` $= \{a, b, c, d, e, f, j, k\}$
U_1	`won` $= \{e, f, k\}$
V_1	`may_win` $= \{a, b, c, \cancel{d}, e, f, \cancel{j}, k\}$

Eample: Doubled Program for Win-Move

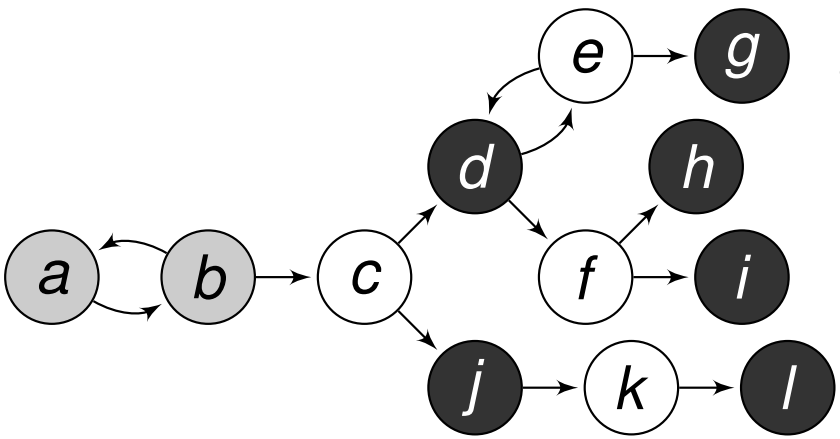
P_o: `may_win(X)` $\leftarrow$ `move(X,Y),` $\neg$`won(X).`
P_u: `won(X)` $\leftarrow$ `move(X,Y),` $\neg$`may_win(X).`



	estimates
V_0	`may_win` $= \{a, b, c, d, e, f, j, k\}$
U_1	`won` $= \{e, f, k\}$
V_1	`may_win` $= \{a, b, c, e, f, k\}$
U_2	`won` $= \{c, e, f, k\}$

Eample: Doubled Program for Win-Move

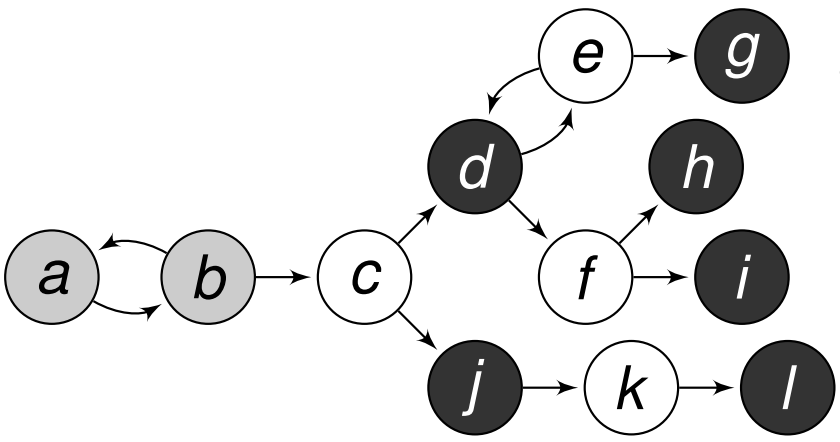
P_o: `may_win(X)` $\leftarrow$ `move(X,Y),` $\neg$`won(X).`
P_u: `won(X)` $\leftarrow$ `move(X,Y),` $\neg$`may_win(X).`



	estimates
V_0	`may_win` $= \{a, b, c, d, e, f, j, k\}$
U_1	`won` $= \{e, f, k\}$
V_1	`may_win` $= \{a, b, c, e, f, k\}$
U_2	`won` $= \{c, e, f, k\}$
V_2	`may_win` $= \{a, b, c, e, f, k\}$

Eample: Doubled Program for Win-Move

P_o: `may_win(X)` $\leftarrow$ `move(X,Y),` $\neg$`won(X).`
P_u: `won(X)` $\leftarrow$ `move(X,Y),` $\neg$`may_win(X).`



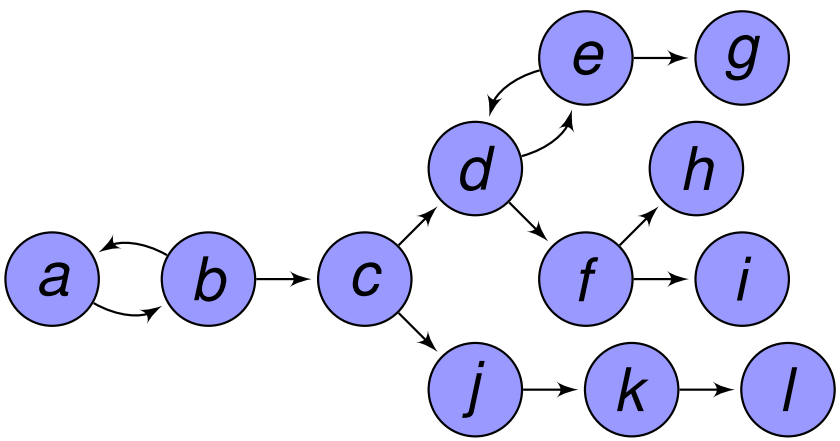
	estimates
V_0	`may_win` $= \{a, b, c, d, e, f, j, k\}$
U_1	`won` $= \{e, f, k\}$
V_1	`may_win` $= \{a, b, c, e, f, k\}$
U_2	`won` $= \{c, e, f, k\}$
V_2	`may_win` $= \{a, b, c, e, f, k\}$

- Strict waves from perimeter to inside of graph
- Global barrier when moving inside (unboundedly many)

Disorderly Win-Move

Key Idea:

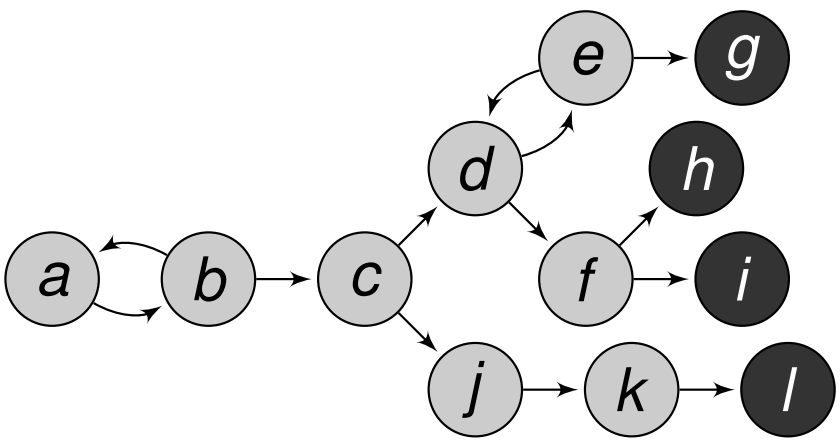
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

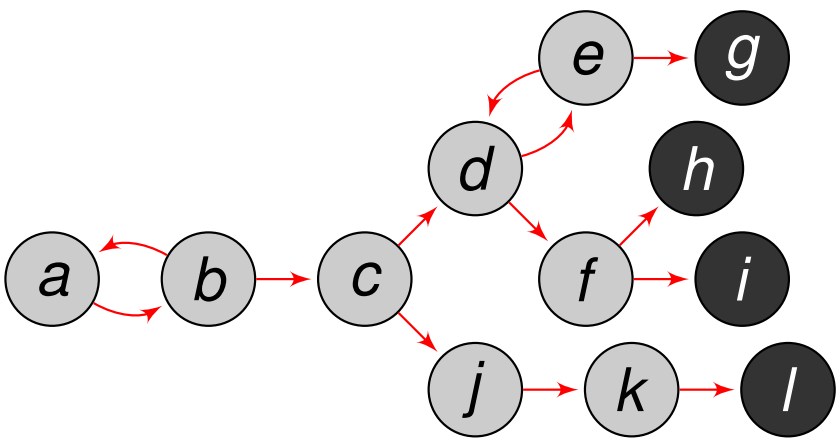
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

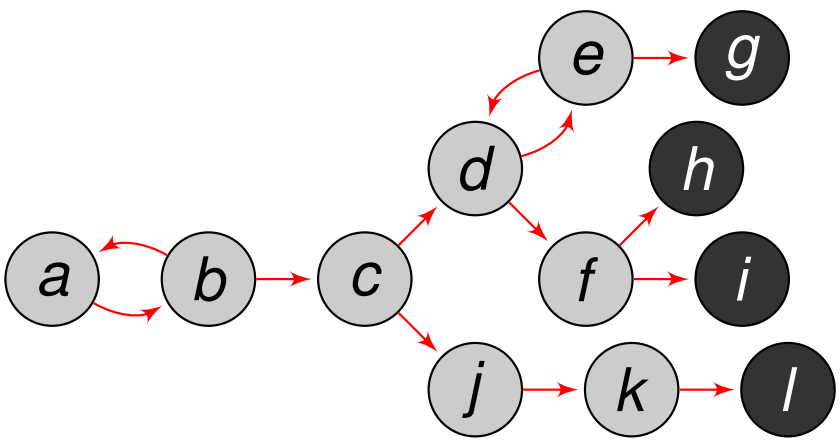
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

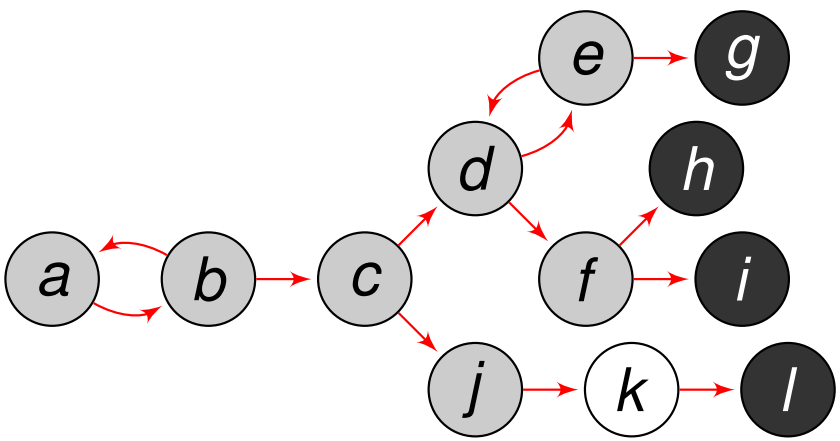
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

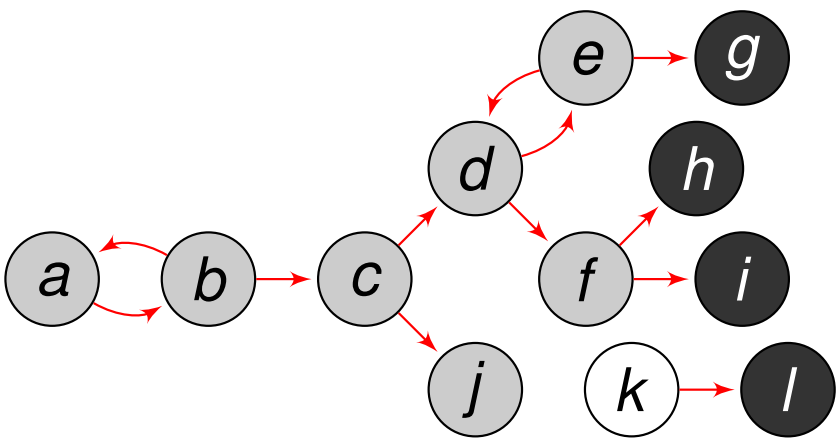
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

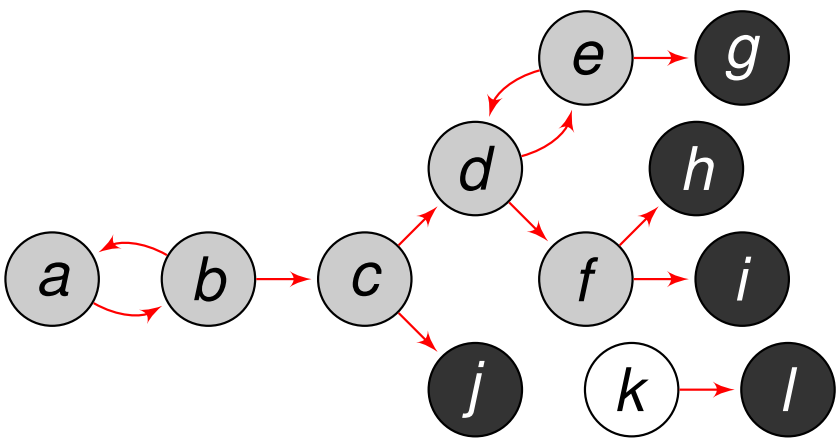
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

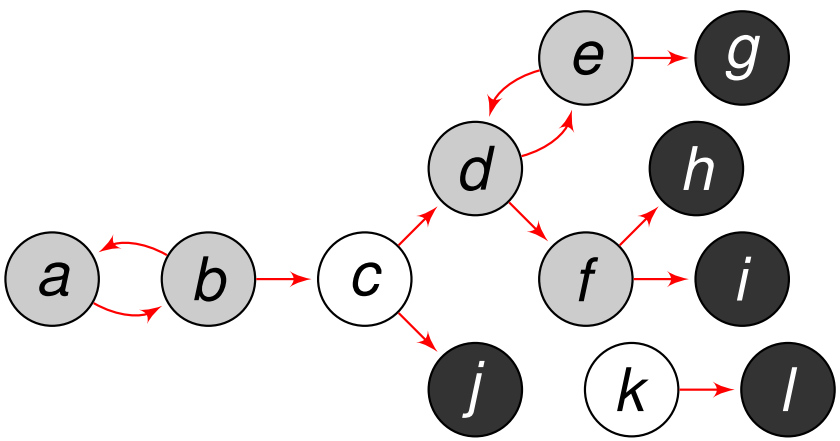
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

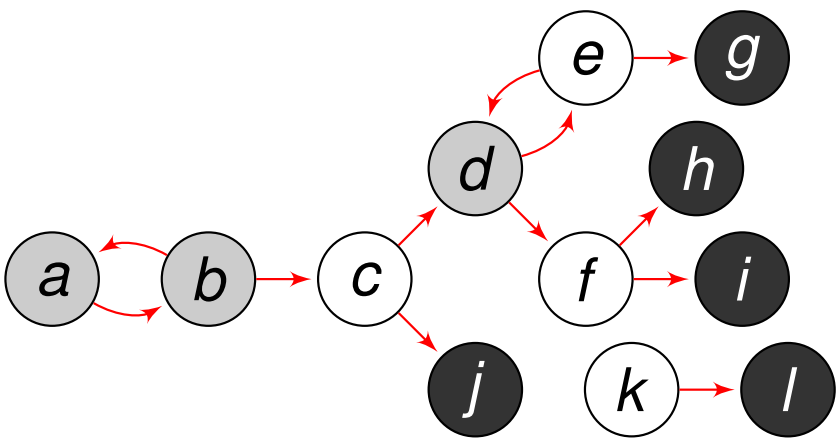
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

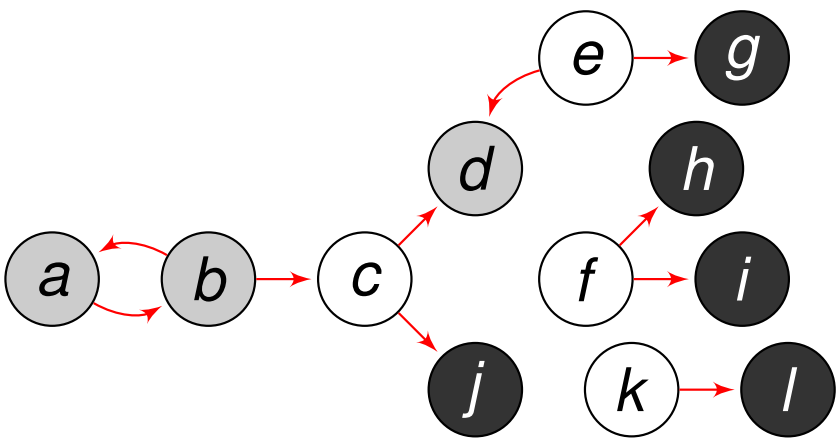
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

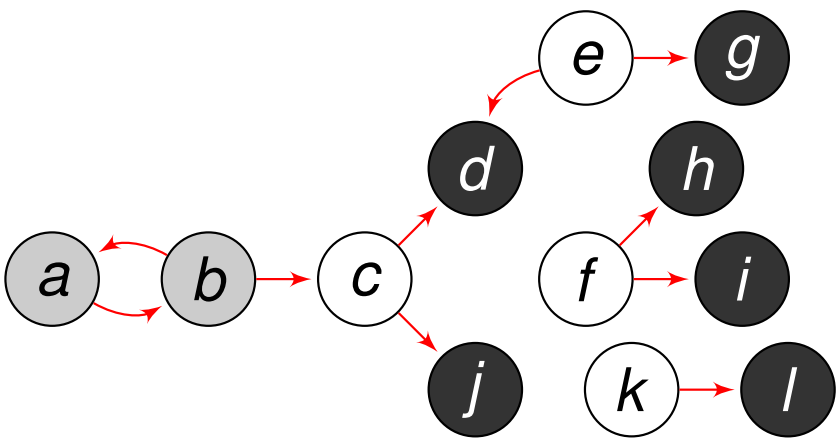
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

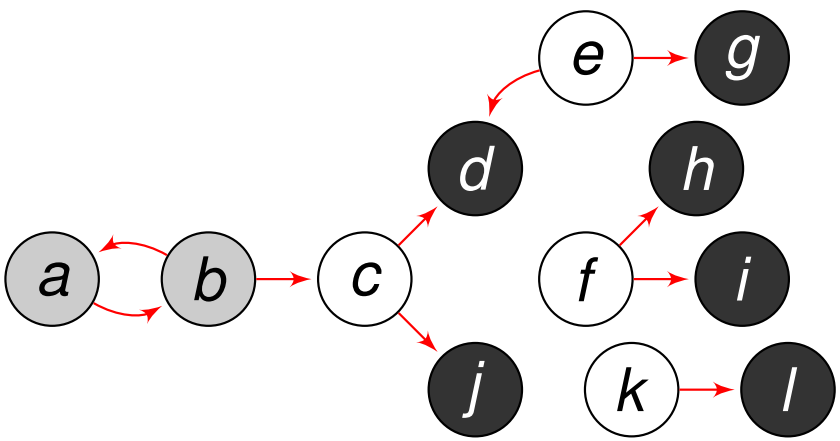
- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



Disorderly Win-Move

Key Idea:

- For each position, keep track of *good moves*
- A node is won if a move leads to a lost position
- Good moves to won positions should be removed
- A node is lost if there is no good move anymore



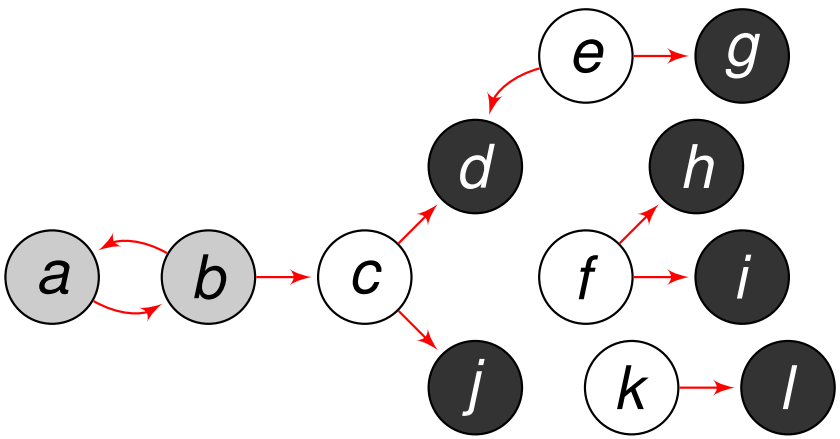
same result as alternation fixpoint

Disorderly Win-Move

Initialization:

```
may_win(X) ← move(X,Y).
good_move(X,Y) ← move(X,Y).
```

(P_{init})

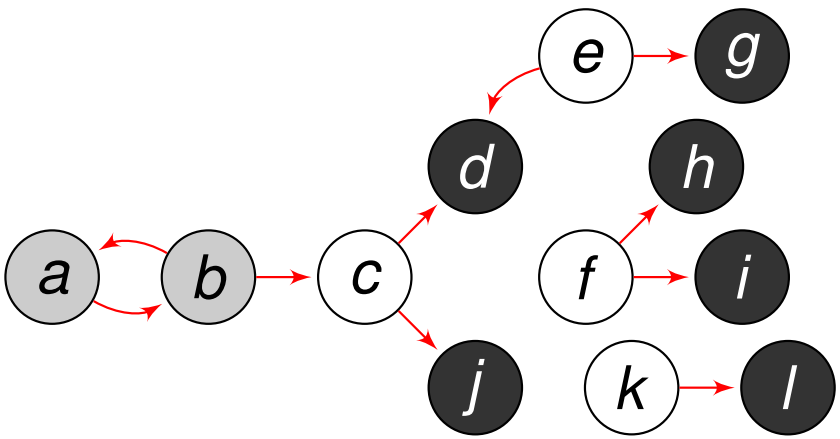


Disorderly Win-Move

Evaluation:

```
won(X) ← move(X,Y), ¬may_win(Y).
¬good_move(X,Y) ← won(Y), move(X,Y).
¬may_win(X) ← move(X,_), ∀Y ¬good_move(X,Y).
```

(P_{wm})



How to make these notions precise?

How to generalize beyond win-move?

Outline

Strategy

- Define non-deterministic semantics that models distributed evaluation
- Restrict to programs whose results are, ultimately, deterministic (but admit disorderly evaluation nevertheless)

Disorderly Semantics (II)

Syntax (Datalog$_{\forall}^{\neg}$)

- Negation in body and head
- Optional $\forall$-quantified variables in negated body atoms

Schema partitioned into

- *edb*(P) read-only input relations
- *eidb*(P) pre-initialized, updateable relations
- *idb*(P) initially empty, updateable relations (for output)

Semantics

- Interpret rules as *production rules*; are "fired" until fixpoint
- Non-deterministically choose rule instantiations
- Defer updates non-deterministically
  to model tuple sending across network (non-local rules).

Disorderly Semantics (II)

Syntax (Datalog$^{\neg}_{\forall}$)

- Negation in body and head
- Optional $\forall$-quantified variables in negated body atoms

Schema partitioned into

- *edb*(*P*) read-only input relations
- *eidb*(*P*) pre-initialized, updateable relations
- *idb*(*P*) initially empty, updateable relations (for output)

Semantics

- Interpret rules as *production rules*; are "fired" until fixpoint
- Non-deterministically choose rule instantiations
- Defer updates non-deterministically
  to model tuple sending across network (non-local rules).

Disorderly Semantics (II)

Syntax (Datalog$^{\neg}_{\forall}$)

- Negation in body and head
- Optional $\forall$-quantified variables in negated body atoms

Schema partitioned into

- *edb*(*P*) read-only input relations
- *eidb*(*P*) pre-initialized, updateable relations
- *idb*(*P*) initially empty, updateable relations (for output)

Semantics

- Interpret rules as *production rules*; are "fired" until fixpoint
- Non-deterministically choose rule instantiations
- Defer updates non-deterministically
  to model tuple sending across network (non-local rules).

Disorderly Semantics (II)

Syntax (Datalog$^{\neg}_{\forall}$)

- Negation in body and head
- Optional $\forall$-quantified variables in negated body atoms

Schema partitioned into

- *edb*(*P*) read-only input relations
- *eidb*(*P*) pre-initialized, updateable relations
- *idb*(*P*) initially empty, updateable relations (for output)

Semantics

- Interpret rules as *production rules*; are "fired" until fixpoint
- Non-deterministically choose rule instantiations
- Defer updates non-deterministically
  to model tuple sending across network (non-local rules).

Disorderly Semantics (III)

Model computation as transitions between *states*.
Each state $S = (I, U)$ is a pair:

- I ... database instance (set of positive facts)
- U ... bag of updates (bag of positive and negative facts)

State Transitions (relation $\rightsquigarrow$)

- **Request:** choose rule valuation with satified body & insert head into U
- **Insert/Delete:** choose an update in U and apply it to I

Definition

The result of applying P to a instance I is the set of instances J with:

- $(I, \emptyset) \rightsquigarrow^* (J, U)$, and
- there is no $J' \neq J$ with $(J, U) \rightsquigarrow^* (J', U')$

Roughly corresponds to Dedalus w/o location-specifiers and only async rules; result is set of quiescent models.

Disorderly Semantics (III)

Model computation as transitions between *states*.
Each state $S = (I, U)$ is a pair:

- I ... database instance (set of positive facts)
- U ... bag of updates (bag of positive and negative facts)

State Transitions (relation $\rightsquigarrow$)

- **Request:** choose rule valuation with satified body & insert head into U
- **Insert/Delete:** choose an update in U and apply it to I

Definition

The result of applying P to a instance I is the set of instances J with:

- $(I, \emptyset) \rightsquigarrow^* (J, U)$, and
- there is no $J' \neq J$ with $(J, U) \rightsquigarrow^* (J', U')$

Roughly corresponds to Dedalus w/o location-specifiers and only async rules; result is set of quiescent models.

Disorderly Semantics (III)

Model computation as transitions between *states*.
Each state $S = (I, U)$ is a pair:

- I ... database instance (set of positive facts)
- U ... bag of updates (bag of positive and negative facts)

State Transitions (relation $\rightsquigarrow$)

- **Request:** choose rule valuation with satified body & insert head into U
- **Insert/Delete:** choose an update in U and apply it to I

Definition

The result of applying P to a instance I is the set of instances J with:

- $(I, \emptyset) \rightsquigarrow^* (J, U)$, and
- there is no $J' \neq J$ with $(J, U) \rightsquigarrow^* (J', U')$

Roughly corresponds to Dedalus w/o location-specifiers and only async rules; result is set of quiescent models.

Disorderly Semantics (III)

Model computation as transitions between *states*.
Each state $S = (I, U)$ is a pair:

- I ... database instance (set of positive facts)
- U ... bag of updates (bag of positive and negative facts)

State Transitions (relation $\rightsquigarrow$)

- **Request:** choose rule valuation with satified body & insert head into U
- **Insert/Delete:** choose an update in U and apply it to I

Definition

The result of applying P to a instance I is the set of instances J with:

- $(I, \emptyset) \rightsquigarrow^* (J, U)$, and
- there is no $J' \neq J$ with $(J, U) \rightsquigarrow^* (J', U')$

Roughly corresponds to Dedalus w/o location-specifiers and only async rules; result is set of quiescent models.

Functionality

$$P = \boxed{\begin{array}{l} \texttt{t} \leftarrow \neg\texttt{a}. \\ \texttt{a} \leftarrow \texttt{b}. \end{array}}$$

$\text{Disorderly}_P(\{b\}) = \{\{a\}, \{a, t\}\}$.

Definition

Program P is functional if $|P(I)| \leq 1$ for any source instance I.

Functionality

$$P = \boxed{\begin{array}{l} \texttt{t} \leftarrow \neg\texttt{a}. \\ \texttt{a} \leftarrow \texttt{b}. \end{array}}$$

$$\text{Disorderly}_P(\{b\}) = \{\{a\}, \{a, t\}\}.$$

Definition

Program P is functional if $|P(I)| \leq 1$ for any source instance I.

Functionality

$$P = \boxed{\begin{array}{l} \texttt{t} \leftarrow \neg\texttt{a}. \\ \texttt{a} \leftarrow \texttt{b}. \end{array}}$$

$$\text{Disorderly}_P(\{b\}) = \{\{a\}, \{a, t\}\}.$$

Definition

Program P is functional if $|P(I)| \leq 1$ for any source instance I.

Termination

$$P_1 = \boxed{\begin{array}{l} \texttt{toggle} \leftarrow \texttt{a}. \\ \neg\texttt{toggle} \leftarrow \texttt{a}. \end{array}}$$

$\text{Disorderly}_{P1}(\{a\}) = \emptyset.$

$$P_2 = \boxed{\begin{array}{l} \texttt{toggle} \leftarrow \texttt{a}. \\ \neg\texttt{toggle} \leftarrow \texttt{a}. \\ \texttt{b} \leftarrow \texttt{c}. \\ \texttt{a} \leftarrow \neg\texttt{b}. \end{array}}$$

$\text{Disorderly}_{P2}(\{c\}) = \{\{a\}\}.$

Definition

A program P is terminating if for any fair trace

$$(I^0, \emptyset) \rightsquigarrow (I^1, U^1) \rightsquigarrow \ldots$$

there is an i such that I^i is in the result of applying P to I^0.

Termination

$P_1 =$

```
 toggle ← a.
¬toggle ← a.
```

$\text{Disorderly}_{P1}(\{a\}) = \emptyset.$

$P_2 =$

```
 toggle ← a.
¬toggle ← a.
 b ← c.
 a ← ¬b.
```

$\text{Disorderly}_{P2}(\{c\}) = \{\{a\}\}.$

Definition

A program P is terminating if for any fair trace

$$(I^0, \emptyset) \rightsquigarrow (I^1, U^1) \rightsquigarrow \ldots$$

there is an i such that I^i is in the result of applying P to I^0.

Termination

$P_1 =$

```
 toggle ← a.
¬toggle ← a.
```

$\text{Disorderly}_{P1}(\{a\}) = \emptyset.$

$P_2 =$

```
 toggle ← a.
¬toggle ← a.
 b ← c.
 a ← ¬b.
```

$\text{Disorderly}_{P2}(\{c\}) = \{\{a\}\}.$

Definition

A program P is terminating if for any fair trace

$$(I^0, \emptyset) \rightsquigarrow (I^1, U^1) \rightsquigarrow \ldots$$

there is an i such that I^i is in the result of applying P to I^0.

Eventual Consistency

Definition

P is eventually consistent if it is functional and terminating.

- The eventually consistent programs are those we can safely evaluate disorderly

Theorem

Functionality, termination, and eventual consistency is undecidable.
Even without universal quantification or eidb relations.

Note: this result does not require order
Proof idea: reduction from containment of positive Datalog.

Eventual Consistency

Definition

P is eventually consistent if it is functional and terminating.

- The eventually consistent programs are those we can safely evaluate disorderly

Theorem

Functionality, termination, and eventual consistency is undecidable.
Even without universal quantification or eidb relations.

Note: this result does not require order
Proof idea: reduction from containment of positive Datalog.

Eventual Consistency

Definition

P is eventually consistent if it is functional and terminating.

- The eventually consistent programs are those we can safely evaluate disorderly

Theorem

Functionality, termination, and eventual consistency is undecidable.
Even without universal quantification or eidb relations.

Note: this result does not require order
Proof idea: reduction from containment of positive Datalog.

Semi-Monotone Datalog (Well-Behaved Fragment)

Definition

P is semi-monotone if relation names in *idb*(*P*) occur only positively, and relation names in *eidb*(*P*) occur only negatively.

Examples

```
t ← ¬a.
¬a ← b.
c ← ¬b.
```
(✓)

```
a ← b.
c ← ¬a.
```
(×)

```
won(X) ← move(X,Y), ¬may_win(Y).
¬good_move(X,Y) ← won(Y), move(X,Y).
¬may_win(X) ← move(X,_), ∀Y ¬good_move(X,Y).
```
(✓)

Semi-Monotone Datalog (Well-Behaved Fragment)

Definition

P is semi-monotone if relation names in *idb*(*P*) occur only positively, and relation names in *eidb*(*P*) occur only negatively.

Examples

```
t ← ¬a.
¬a ← b.
c ← ¬b.
```
(✓)

```
a ← b.
c ← ¬a.
```
(×)

```
 won(X) ← move(X,Y), ¬may_win(Y).
¬good_move(X,Y) ← won(Y), move(X,Y).
¬may_win(X) ← move(X,_), ∀Y ¬good_move(X,Y).
```
(✓)

Theorem

Every semi-monotone Datalog$^{\neg}_{\forall}$ program is eventually consistent under the disorderly semantics.

Key observation: Set of satisfied body atoms is monotonically increasing during computation.

✓Time-Safe. What about Space-Safety?

Syntactic Restrictions for Space-Safety

Need to guarantee that

- All local inferences are valid globally.
- All global inferences be done with only local knowledge.

Syntactic restrictions on “Distributed Semi-Monotone Datalog”

- Consistently partition relations
- Co-locate joins (also negative joins, and the infinite $\forall$-join)
- The (joint) location-specifier must occur in at least one positive atom

Syntactic Restrictions for Space-Safety

Need to guarantee that

- All local inferences are valid globally.
- All global inferences be done with only local knowledge.

Syntactic restrictions on “Distributed Semi-Monotone Datalog”

- Consistently partition relations
- Co-locate joins (also negative joins, and the infinite $\forall$-join)
- The (joint) location-specifier must occur in at least one positive atom

Syntactic Restrictions for Space-Safety

Need to guarantee that

- All local inferences are valid globally.
- All global inferences be done with only local knowledge.

Syntactic restrictions on “Distributed Semi-Monotone Datalog”

- Consistently partition relations
- Co-locate joins (also negative joins, and the infinite $\forall$-join)
- The (joint) location-specifier must occur in at least one positive atom

Partition:

- `move` according to first: `m1`
- `move` according to second: `m2`
- `may_win`, `good_move`, `won` according to first

Syntactic Restrictions for Space-Safety

Need to guarantee that

- All local inferences are valid globally.
- All global inferences be done with only local knowledge.

Syntactic restrictions on "Distributed Semi-Monotone Datalog"

- Consistently partition relations
- Co-locate joins (also negative joins, and the infinite $\forall$-join)
- The (joint) location-specifier must occur in at least one positive atom

```
may_win(h(X),X) ← m1(h(X),X,_).
good_move(h(X),X,Y) ← m1(h(X),X,Y).
```

Syntactic Restrictions for Space-Safety

Need to guarantee that

- All local inferences are valid globally.
- All global inferences be done with only local knowledge.

Syntactic restrictions on "Distributed Semi-Monotone Datalog"

- Consistently partition relations
- Co-locate joins (also negative joins, and the infinite $\forall$-join)
- The (joint) location-specifier must occur in at least one positive atom

```
won(h(X),X) ← m2(h(Y),X,Y), ¬may_win(h(Y),Y).
¬good_move(h(X),X,Y) ← won(h(Y),Y), m2(h(Y),X,Y).
¬may_win(h(X),X) ← m1(h(X),X,_),
                   ∀Y ¬good_move(h(X),X,Y).
```

Outline

What is Coordination?

No good, accepted definition of coordination(-freeness).[1]

Intuition

- Coordination happens when a *consenus* needs to be achieved among all peers in the distributed system.
- Involves waiting for all hosts in the system.
- Examples
  - Establishing consenus that a fixpoint has been reached
  - Deciding whether to commit distributed transaction

Our win-move construction does not run consenus algorithms with all nodes involved (cf. alternating fixpoint).

[1] "They need to be worried about coordination. The question is: What definition of coordination do they need to be worried about?"
– Larry Noble, former FEC general counsel. NPR. January 6, 2012.

What is Coordination?

No good, accepted definition of coordination(-freeness).[1]

Intuition

- Coordination happens when a *consenus* needs to be achieved among all peers in the distributed system.
- Involves waiting for all hosts in the system.
- Examples
  - Establishing consenus that a fixpoint has been reached
  - Deciding whether to commit distributed transaction

Our win-move construction does not run consenus algorithms with all nodes involved (cf. alternating fixpoint).

[1] "They need to be worried about coordination. The question is: What definition of coordination do they need to be worried about?"
– Larry Noble, former FEC general counsel. NPR. January 6, 2012.

What is Coordination?

No good, accepted definition of coordination(-freeness).[1]

Intuition

- Coordination happens when a *consenus* needs to be achieved among all peers in the distributed system.
- Involves waiting for all hosts in the system.
- Examples
  - Establishing consenus that a fixpoint has been reached
  - Deciding whether to commit distributed transaction

Our win-move construction does not run consenus algorithms with all nodes involved (cf. alternating fixpoint).

[1] "They need to be worried about coordination. The question is: What definition of coordination do they need to be worried about?"
– Larry Noble, former FEC general counsel. NPR. January 6, 2012.

Relational Transducer Networks [ANVdB11]

Network of Relational Transducers

- bunch of nodes in a connected network
- each running a “program” (database queries)
- input partitioned arbitrarily among nodes
- they consume data from neighbors/local state, run program, send data to neighbors/local state
- output produced incrementally; cannot be revoked

Correctness: Relational Transducer should work regardless of

- network topology
- input partitioning
- non-deterministic execution trace

Coordination-Freeness

Definition [ANVdB11]

Relational transducer is called coordination-free if, for every network and input, can produce *complete answer* for *some partitioning* of input data even when all communication blocked.

- Not as easy as "partition all data to one node": node doesn't know it has all the data (needs communication to find this out)

Theorem (CALM) [ANVdB11]

Q can be distributedly computed by a transducer that is coordination-free if and only if Q is monotone.

Coordination-Freeness

Definition [ANVdB11]

Relational transducer is called coordination-free if, for every network and input, can produce *complete answer* for *some partitioning* of input data even when all communication blocked.

- Not as easy as "partition all data to one node": node doesn't know it has all the data (needs communication to find this out)

Theorem (CALM) [ANVdB11]

Q can be distributedly computed by a transducer that is coordination-free if and only if Q is monotone.

Orderly Input Distribution

Key idea

- What if we partition input data in a more orderly way?
  eg., according to a hash function, or horizontal partitioning
- Distribution strategy modeled as *Partitioning Policy*

Changes in Transducer Model

- Distribute input according to Partitioning Policy
- Transducer must compute result for all partitioning policies
- Transducer-access to Partitioning Policy:
  Can answer *"Am I responsible for a potential input fact?"*

Key insight

- A node "knows" for which input facts it is "responsible". If it sees that one of these is *not* among its input, it knows that no other node has it either! (increased reasoning power)

Orderly Input Distribution

Key idea

- What if we partition input data in a more orderly way?
  eg., according to a hash function, or horizontal partitioning
- Distribution strategy modeled as *Partitioning Policy*

Changes in Transducer Model

- Distribute input according to Partitioning Policy
- Transducer must compute result for all partitioning policies
- Transducer-access to Partitioning Policy:
  Can answer *"Am I responsible for a potential input fact?"*

Key insight

- A node "knows" for which input facts it is "responsible". If it sees that one of these is *not* among its input, it knows that no other node has it either! (increased reasoning power)

Orderly Input Distribution

Key idea

- What if we partition input data in a more orderly way?
  eg., according to a hash function, or horizontal partitioning
- Distribution strategy modeled as *Partitioning Policy*

Changes in Transducer Model

- Distribute input according to Partitioning Policy
- Transducer must compute result for all partitioning policies
- Transducer-access to Partitioning Policy:
  Can answer *"Am I responsible for a potential input fact?"*

Key insight

- A node "knows" for which input facts it is "responsible". If it sees that one of these is *not* among its input, it knows that no other node has it either! (increased reasoning power)

Variations of Network Transducers

Model $\mathcal{N}_0$ [ANVdB11]

- Partitioning arbitrary, no information about it is provided to nodes

Model $\mathcal{N}_1$

- Partitioning according to hash function h : ground atoms $\rightarrow$ node ids
- h is provided to nodes (and can be used in their queries)

Model $\mathcal{N}_2$

- Same as $\mathcal{N}_1$, but partitioning according to hash function h : attribute values $\rightarrow$ node ids
- This defines a mapping from ground atoms $\rightarrow$ node ids

Model $\mathcal{N}_3$

- Same as $\mathcal{N}_1$, but nodes also given access to a global active domain relation

Win-Move is Coordination-Free (Sometimes)

Denote with $\mathcal{F}[X]$ the class of Coordination-free Queries in model X.

Hierarchy of Coordination-Free Classes

$$\mathcal{M} = \mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$$

win-move $\in \mathcal{F}[\mathcal{N}_2]$

win-move $\notin \mathcal{F}[\mathcal{N}_1]$

next: formalization; interesting lemmas & proofs

Win-Move is Coordination-Free (Sometimes)

Denote with $\mathcal{F}[X]$ the class of Coordination-free Queries in model X.

Hierarchy of Coordination-Free Classes

$$\mathcal{M} = \mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$$

win-move $\in \mathcal{F}[\mathcal{N}_2]$

win-move $\notin \mathcal{F}[\mathcal{N}_1]$

next: formalization; interesting lemmas & proofs

Win-Move is Coordination-Free (Sometimes)

Denote with $\mathcal{F}[X]$ the class of Coordination-free Queries in model X.

Hierarchy of Coordination-Free Classes

$$\mathcal{M} = \mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$$

win-move $\in \mathcal{F}[\mathcal{N}_2]$

win-move $\notin \mathcal{F}[\mathcal{N}_1]$

next: formalization; interesting lemmas & proofs

Win-Move is Coordination-Free (Sometimes)

Denote with $\mathcal{F}[X]$ the class of Coordination-free Queries in model X.

Hierarchy of Coordination-Free Classes

$$\mathcal{M} = \mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$$

win-move $\in \mathcal{F}[\mathcal{N}_2]$

win-move $\notin \mathcal{F}[\mathcal{N}_1]$

next: formalization; interesting lemmas & proofs

$\mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{F}[\mathcal{N}_1]$

Consider set-difference: $\texttt{Q}(\bar{\texttt{X}}) \leftarrow \texttt{R}(\bar{\texttt{X}}), \neg\texttt{S}(\bar{\texttt{X}}).$

Coordination-free transducer $\mathcal{T}$ computing Q:

- Broadcast all tuples in $\texttt{R}$
- Compute $\texttt{Q}(\bar{\texttt{X}}) \leftarrow \texttt{R}(\bar{\texttt{X}}), \neg\texttt{S}(\bar{\texttt{X}}), \texttt{Responsible}(\texttt{S},\bar{\texttt{X}}).$
- Output $\texttt{Q}$.

Semi-Positive Datalog $\subseteq \mathcal{F}[\mathcal{N}_1]$

Theorem

If Q can be computed by a semi-monotone program with

- each rule has at most one negated eidb atom, and
- no variable is $\forall$-quantified

then Q is $\mathcal{F}[\mathcal{N}_1]$ coordination-free.

Corollary

Semi-positive Datalog is $\mathcal{F}[\mathcal{N}_1]$-coordination-free.

... our win-move had forall-quantification.

Semi-Positive Datalog $\subseteq \mathcal{F}[\mathcal{N}_1]$

Theorem

If Q can be computed by a semi-monotone program with

- each rule has at most one negated eidb atom, and
- no variable is $\forall$-quantified

then Q is $\mathcal{F}[\mathcal{N}_1]$ coordination-free.

Corollary

Semi-positive Datalog is $\mathcal{F}[\mathcal{N}_1]$-coordination-free.

... our win-move had forall-quantification.

Semi-Positive Datalog $\subseteq \mathcal{F}[\mathcal{N}_1]$

Theorem

If Q can be computed by a semi-monotone program with

- each rule has at most one negated eidb atom, and
- no variable is $\forall$-quantified

then Q is $\mathcal{F}[\mathcal{N}_1]$ coordination-free.

Corollary

Semi-positive Datalog is $\mathcal{F}[\mathcal{N}_1]$-coordination-free.

... our win-move had forall-quantification.

$\mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2]$

Theorem

If Q can be computed by a semi-monotone program with

- all negated eidb atoms in a rule share a common variable which is not universally quantified

then Q is $\mathcal{F}[\mathcal{N}_2]$ coordination-free.

```
won(X) ← move(X,Y), ¬may_win(Y).
¬good_move(X,Y) ← won(Y), move(X,Y).
¬may_win(X) ← move(X,_), ∀Y ¬good_move(X,Y).
```

With the local pre-processing, this yields win-move $\in \mathcal{F}[\mathcal{N}_2]$.

$\mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2]$

Theorem

If Q can be computed by a semi-monotone program with

- all negated eidb atoms in a rule share a common variable which is not universally quantified

then Q is $\mathcal{F}[\mathcal{N}_2]$ coordination-free.

```
 won(X) ← move(X,Y), ¬may_win(Y).
¬good_move(X,Y) ← won(Y), move(X,Y).
¬may_win(X) ← move(X,_), ∀Y ¬good_move(X,Y).
```

With the local pre-processing, this yields win-move $\in \mathcal{F}[\mathcal{N}_2]$.

$\mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2]$

Theorem

If Q can be computed by a semi-monotone program with

- all negated eidb atoms in a rule share a common variable which is not universally quantified

then Q is $\mathcal{F}[\mathcal{N}_2]$ coordination-free.

```
won(X) ← move(X,Y), ¬may_win(Y).
¬good_move(X,Y) ← won(Y), move(X,Y).
¬may_win(X) ← move(X,_), ∀Y ¬good_move(X,Y).
```

With the local pre-processing, this yields win-move $\in \mathcal{F}[\mathcal{N}_2]$.

$\mathcal{F}[\mathcal{N}_2] \neq \mathcal{C}$

Consider query

$$\boxed{\texttt{empty} \leftarrow \forall \texttt{X}\ \neg \texttt{R(X).}}$$

Intuition:

A node cannot test $\texttt{Responsible(R,}a\texttt{)}$ for all $a \in \mathit{dom}$.

$\mathcal{F}[\mathcal{N}_2] \neq \mathcal{C}$

Consider query

$$\boxed{\texttt{empty} \leftarrow \forall \texttt{X}\ \neg \texttt{R(X).}}$$

Intuition:

A node cannot test `Responsible(R,`a`)` for all $a \in dom$.

$\mathcal{F}[\mathcal{N}_3] = \mathcal{C}$

Model $\mathcal{N}_3$

- $\mathcal{T}$ is given access to active domain of global input

Key Idea

- For each edb relation, broadcast known positive and negative facts (if responsible for)
- If for all relations, all tuples over the active domain are known to be true or false, compute Q

Observations

- Only trivial Partitioning-Policy computes complete result w/o communication
- No incremental output

$\mathcal{F}[\mathcal{N}_3] = \mathcal{C}$

Model $\mathcal{N}_3$

- $\mathcal{T}$ is given access to active domain of global input

Key Idea

- For each edb relation, broadcast known positive and negative facts (if responsible for)
- If for all relations, all tuples over the active domain are known to be true or false, compute Q

Observations

- Only trivial Partitioning-Policy computes complete result w/o communication
- No incremental output

$\mathcal{F}[\mathcal{N}_3] = \mathcal{C}$

Model $\mathcal{N}_3$

- $\mathcal{T}$ is given access to active domain of global input

Key Idea

- For each edb relation, broadcast known positive and negative facts (if responsible for)
- If for all relations, all tuples over the active domain are known to be true or false, compute Q

Observations

- Only trivial Partitioning-Policy computes complete result w/o communication
- No incremental output

Outline

Contributions

Distributed Algorithm for Win-Move

- No global barriers

Beyond Win-Move: Semi-Monotone Datalog$^{\neg}$

- Well-suited for distributed evaluation
- Different levels of safety restrictions imply different level of coordination-freeness

Formal Investigation of Coordination-Free Query Classes

- Make precise a notion of coordination-freeness, parameterized by knowledge about input distribution
- Demonstrate that parameters give rise to strict hierarchy of coordination-free query classes:

$$\mathcal{M} = \mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$$

P. Alvaro, N. Conway, J.M. Hellerstein, and W.R. Marczak.

Consistency analysis in bloom: a calm and collected approach.
In *Biennial Conf. on Innovative DataSystems Research (CIDR)*, 2011.

Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell C Sears.

Dedalus: Datalog in time and space.
Technical Report EECS-2009-173, EECS Department, UC Berkeley, Dec 2009.

Tom J. Ameloot, Frank Neven, and Jan Van den Bussche.

Relational transducers for declarative networking.
In *PODS*, 2011.

Stefan Brass, Jürgen Dix, Burkhard Freitag, and Ulrich Zukowski.

Transformation-based bottom-up computation of the well-founded model.
Theory Pract. Log. Program., 1(5):497–538, September 2001.

Joseph M. Hellerstein.

The declarative imperative: Experiences and conjectures in distributed logic.
Technical Report UCB/EECS-2010-90, EECS Department, UC Berkeley, Jun 2010.

W. Marczak, P. Alvaro, N. Conway, J.M. Hellerstein, and D. Maier.

Confluence analysis for distributed programs: A model-theoretic approach.
Technical Report UCB/EECS-2011-154, EECS Department, UC Berkeley, 2011.

Victor Vianu.

Rule-based languages.
Annals of Mathematics and Artificial Intelligence, 19(1-2):215–259, 1997.

Daniel Zinn, Todd J. Green, and Bertram Ludäscher.

Win-move is coordination-free (sometimes).
In *ICDT*, 2012.

Daniel Zinn.

Weak forms of monotonicity and coordination-freeness.
Arxiv.org: CoRR, abs/1202.0242, 2012.

Outline

Exact Characterization of $\mathcal{F}[N_1]$ and $\mathcal{F}[N_2]$ [Zin12]

Theorem

$$\begin{array}{ccccccc} F[\mathcal{N}_0] & \subsetneq & \mathcal{F}[\mathcal{N}_1] & \subsetneq & \mathcal{F}[\mathcal{N}_2] & \subsetneq & F[\mathcal{N}_3] \\ || & & || & & || & & || \\ \mathcal{M} & \subsetneq & \mathcal{M}_{adom} & \subsetneq & \mathcal{M}_{weak-adom} & \subsetneq & \mathcal{C} \end{array}$$

Weak Forms of Monotonicity

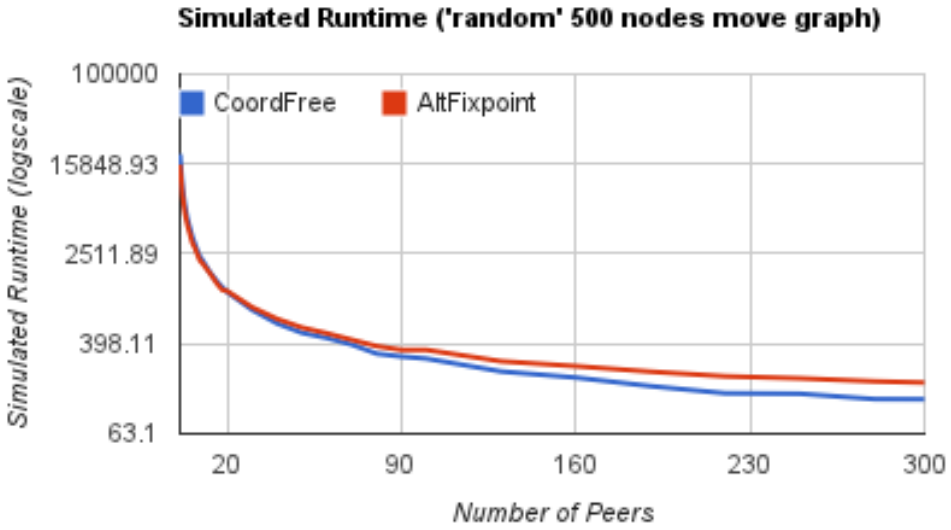
Definition

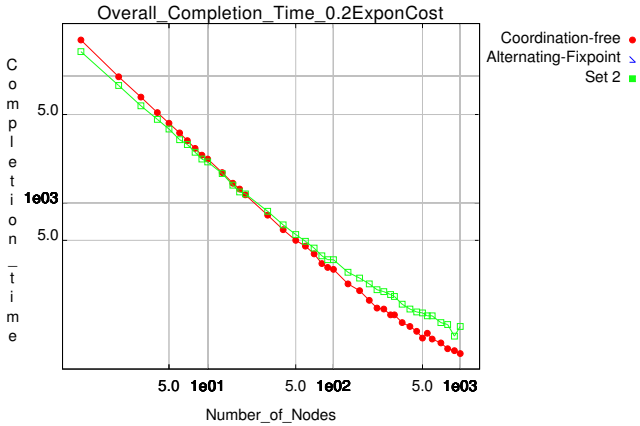
Let I be a database instance over a schema σ, and R a relation symbol in σ of arity n. A query is adom-monotone if $Q(I) \subseteq Q(I \cup \{f\})$ for all inputs I and facts $f = R(c_1, \ldots, c_n)$ that contain at least one constant c_i that does not occur in I, i.e., $c_i \notin adom(I)$ for some i. We denote the class of adom-monotone queries as $\mathcal{M}_{adom}$.

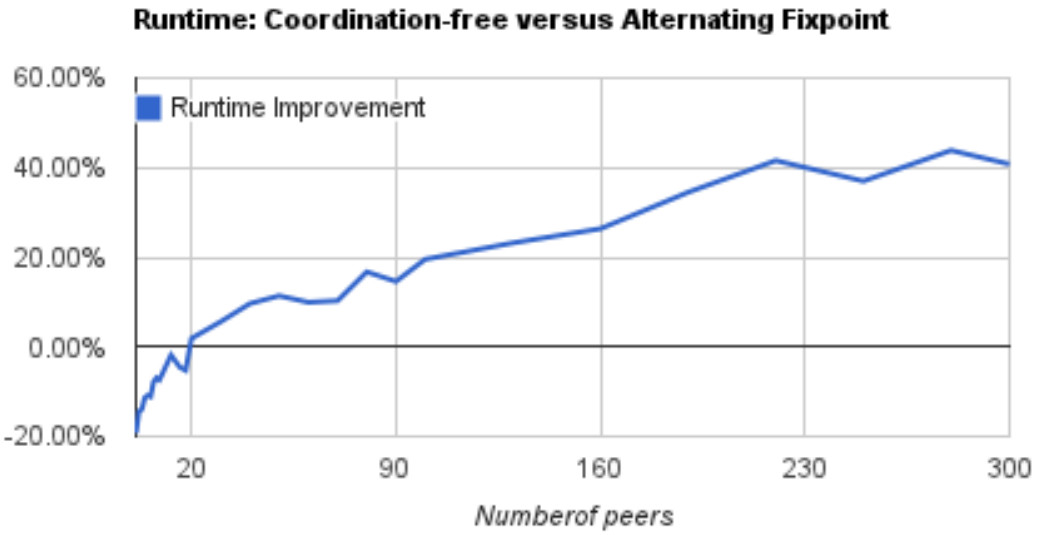
Definition

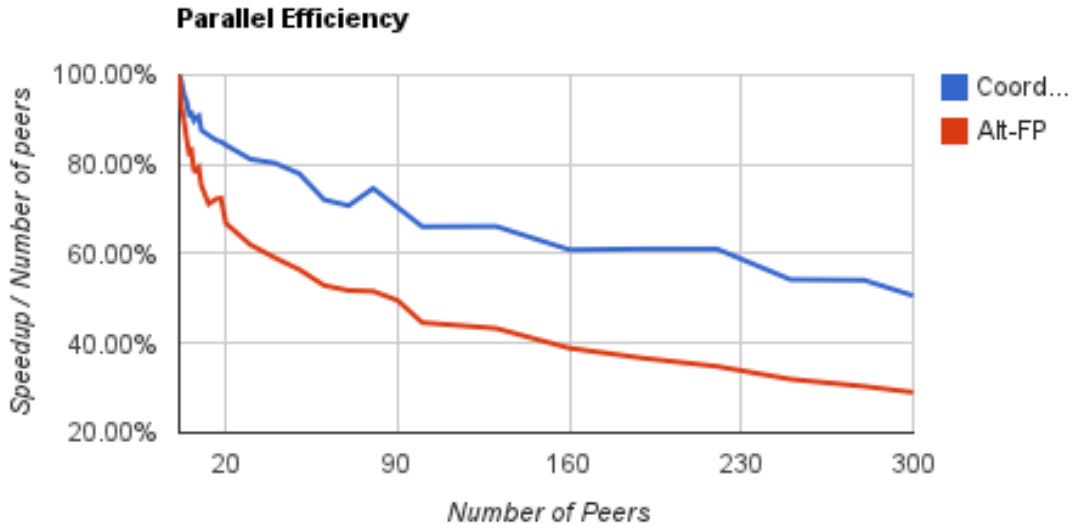
Let I be a database instance over a schema σ, and R a relation symbol in σ of arity $n \geq 1$. A query is weak-adom-monotone if $Q(I) \subseteq Q(I \cup \{t\})$ for all inputs I and facts $f = R(c_1, \ldots, c_n)$ that contain only constants c_i that do not occur in I, i.e., $c_i \notin adom(I)$ for all i. We denote the class of weak-adom-monotone queries as $\mathcal{M}_{weak-adom}$.

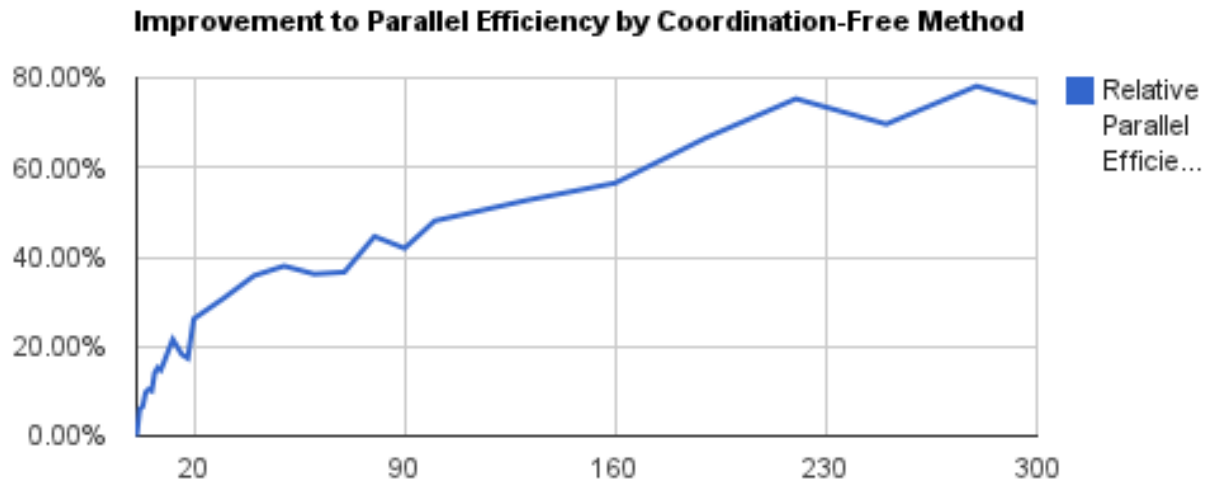
Outline


Simulated Runtime ('random' 500 nodes move graph)
Simulated Runtime (logscale)
100000
15848.93
2511.89
398.11
63.1
CoordFree
AltFixpoint
20
90
160
230
300
Number of Peers


Overall_Completion_Time_0.2ExponCost
Coordination-free
Alternating-Fixpoint
Set 2
Completion_time
5.0
1e03
5.0
5.0 1e01
5.0 1e02
5.0 1e03
Number_of_Nodes


Runtime: Coordination-free versus Alternating Fixpoint
60.00%
40.00%
20.00%
0.00%
-20.00%
Runtime Improvement
20
90
160
230
300
Numberof peers


Parallel Efficiency
Speedup / Number of peers
100.00%
80.00%
60.00%
40.00%
20.00%
20
90
160
230
300
Number of Peers
Coord...
Alt-FP


Improvement to Parallel Efficiency by Coordination-Free Method
80.00%
60.00%
40.00%
20.00%
0.00%
20
90
160
230
300
Relative Parallel Efficie...