

recursion

How expressive is SQL?

- Full programming languages can express all computable functions (C, Java, etc)

Can SQL express all computable queries?

A: YES B: NO

How expressive is SQL?

flight	from	to
	SD	LA
	SD	SF
	LA	NY
	...	...

Can SQL express the following query:
“Is there a way to get from City1 to City2?”

B: NO

Easier

“Is there a way to get from City1 to City2
by a direct flight?”

City1 $\longrightarrow$ City2

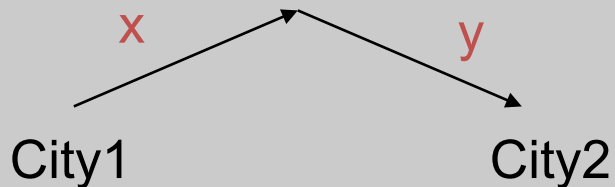
```
select * from flight  
where from = 'City1' and to = 'City2'
```

Easier

“Is there a way to get from City1 to City2
with at most one stopovers?”

City1 $\longrightarrow$ City2

OR



select * from flight
where from = 'City1' and to = 'City2'

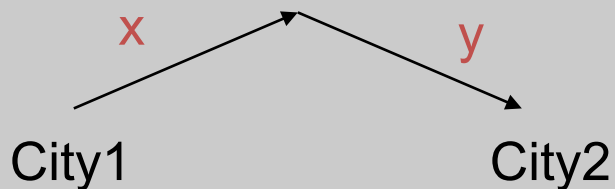
select x.from, y.to
from flight x, flight y
where x.from = 'City1' and
x.to = y.from and y.to = 'City2'

Easier

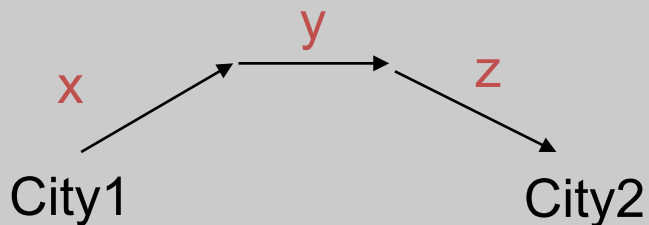
“Is there a way to get from City1 to City2
with at most two stopovers?”

City1 $\longrightarrow$ City2

OR



OR



select * from flight
where from = 'City1' and to = 'City2'

select x.from, y.to
from flight x, flight y
where x.from = 'City1' and
x.to = y.from and y.to = 'City2'

select x.from, z.to
from flight x, flight y, flight z
where x.from = 'City1' and x.to = y.from
and y.to = z.from and z.to = 'City2'

Easier

“Is there a way to get from City1 to City2
with at most k stopovers?”



Need $k+1$ tuple variables!

Now

“Is there a way to get from City1 to City2
with any number of stopovers?”

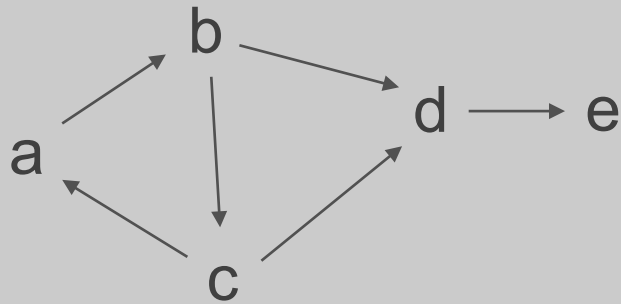
Cannot do in basic SQL!

Similar Examples

- Parts-components relation:
“Find all subparts of some given part A”
- Parent/child relation:
“Find all of John’ s descendants”

More general:

Transitive closure of graph



G	A	B
	A	B
	a	b
	c	a
	b	c
	...	...

Find the pairs of nodes $\langle x, y \rangle$ that are connected by some directed path

A	B
a	b
a	d
a	e
b	d
b	e
...	...

Computing transitive closure T of G

“Find the pairs of nodes $\langle a, b \rangle$ that are connected in G ”

Same as:

“find pairs of nodes $\langle a, b \rangle$ at distance 1” UNION

“find pairs of nodes $\langle a, b \rangle$ at distance at most 2” UNION

.....

“find pairs of nodes $\langle a, b \rangle$ at distance at most k ” UNION

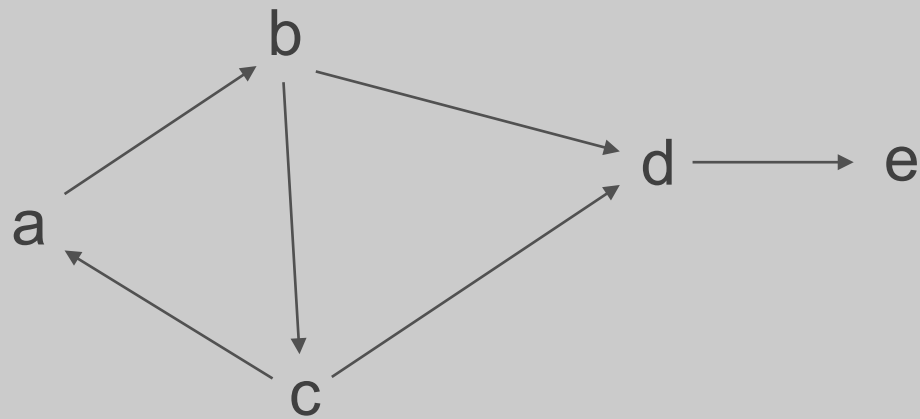
.....

When to stop?

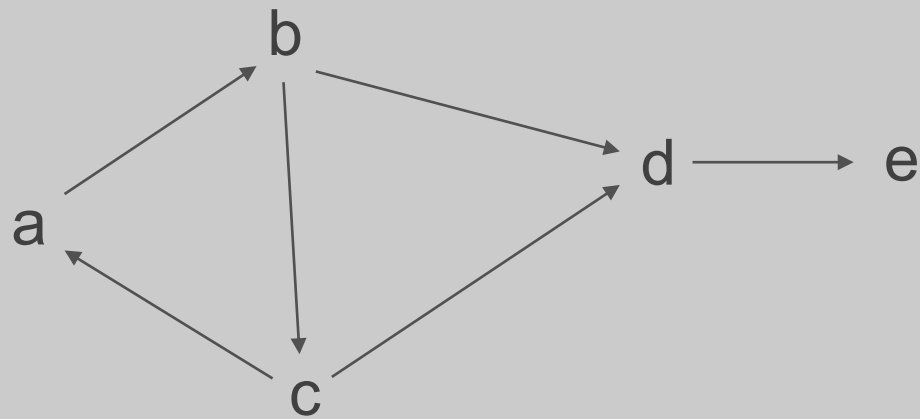
At some point, no new nodes are added.

Distance cannot be larger than total number of nodes in G .

Example

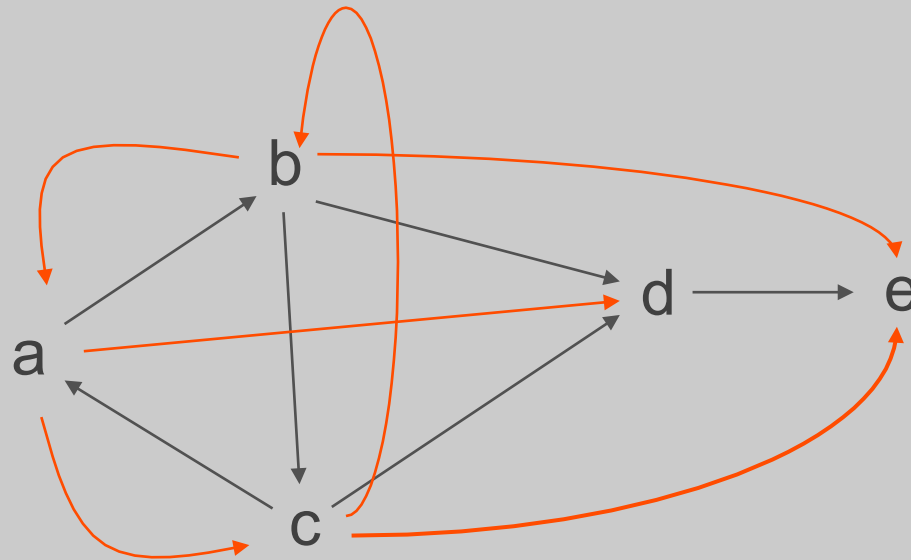


Example



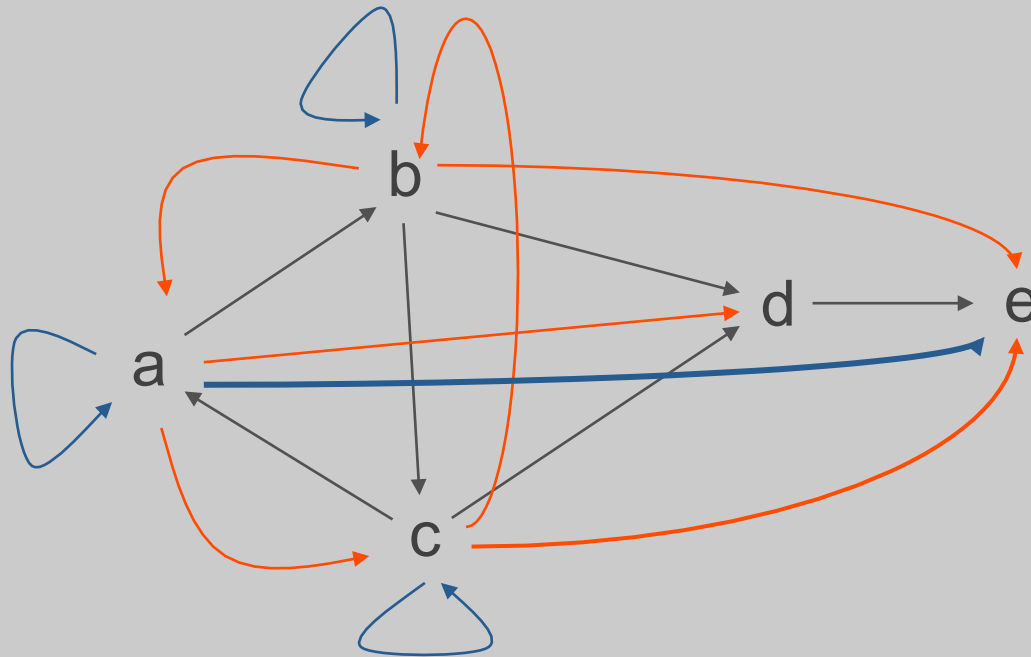
Distance 1

Example



Distance ≤ 2

Example



Distance ≤ 3

Algorithm

Denote by T_k the pairs of nodes at distance at most k

T_1 : “find pairs of nodes $\langle a, b \rangle$ at distance 1”

select * from G

T_k : “find the pairs of nodes $\langle a, b \rangle$ at distance at most k ”



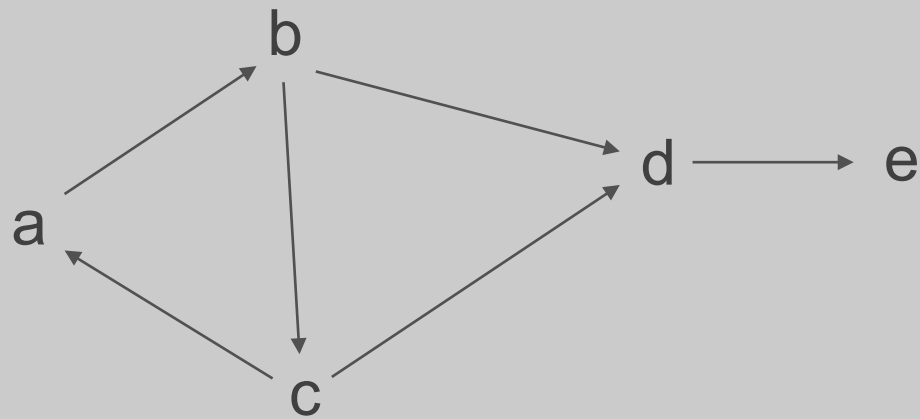
(select * from T_{k-1})

union

(select $x.A, y.B$ from G x, T_{k-1} y

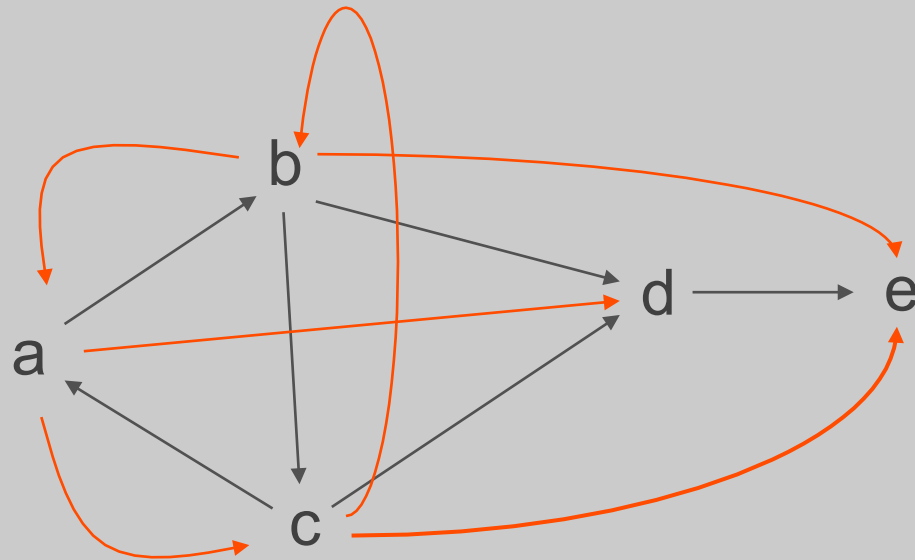
where $x.B = y.A$)

Example



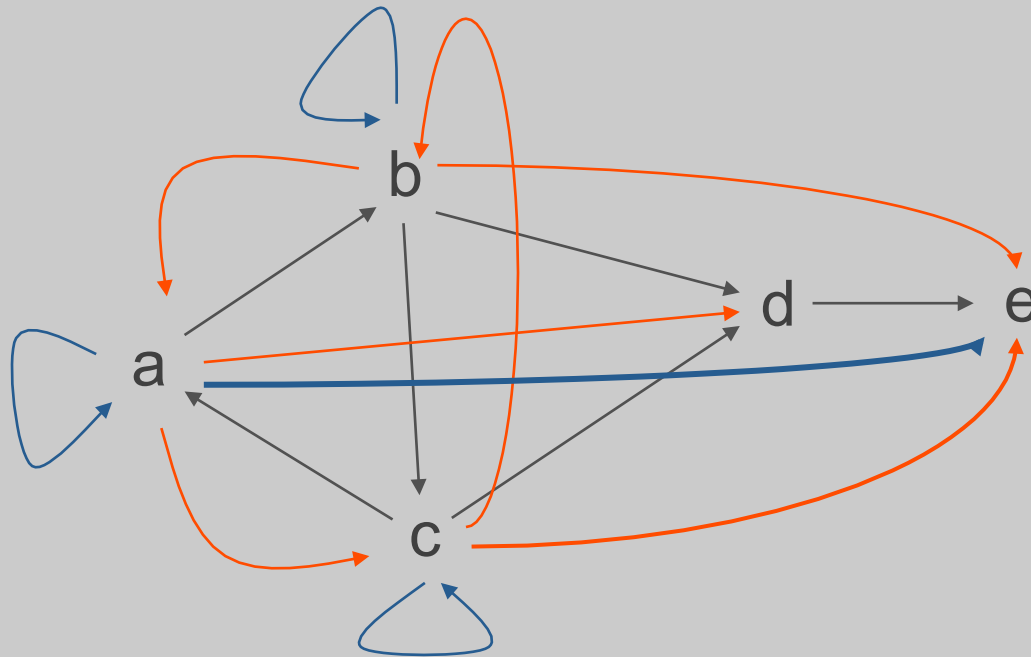
T_1

Example



T_2

Example



T_3

One Solution

Add recursion to SQL

(Not part of the standard)

```
create recursive view T as
(select * from G)
union
(select x.A, y.B
 from G x, T y
 where x.B = y.A)
```

Semantics:

1. Start with empty T
2. While T changes
{evaluate view with current T;
union result with T }

Note: This must terminate, since there are finitely many tuples one can add to T (if no new values are created)

One Solution

Add recursion to SQL

Alternative formulation:

```
with recursive T as
(select * from G)
union
(select x.A, y.B
 from G x, T y
 where x.B = y.A)
select * from T ;
```

Another Example

frequents	drinker	bar

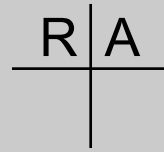
Friends: Drinkers who frequent the same bar

Find transitive closure of **Friends**

```
create recursive view T as  
(select f1.drinker as drinker1, f2.drinker as drinker2  
from frequents f1, frequents f2  
where f1.bar = f2.bar)  
union  
(select t1.drinker1, f2.drinker as drinker2  
from T t1, frequents f1, frequents f2  
where t1.drinker2 = f1.drinker and f1.bar = f2.bar)
```

Problematic example

```
create recursive view T as  
(select A, 0 as N from R)  
union  
(select A, N+1 as N  
from T)
```

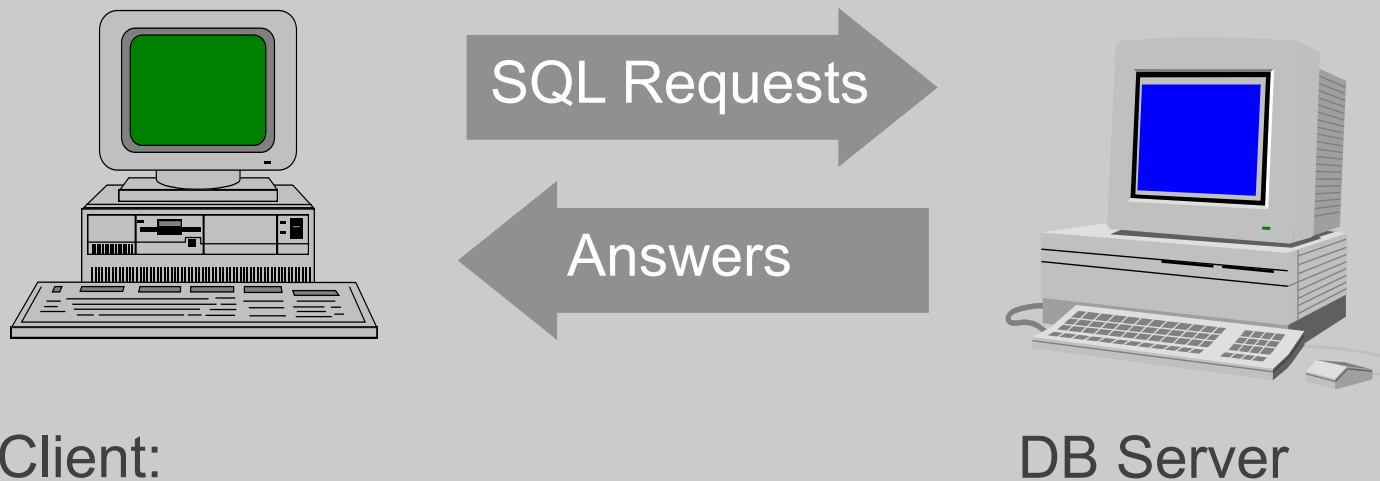


- Never terminates
- Arithmetic in selects, aggregate functions are forbidden in recursive definitions

Another Solution

Embedded SQL

Powerful way to overcome SQL limitations



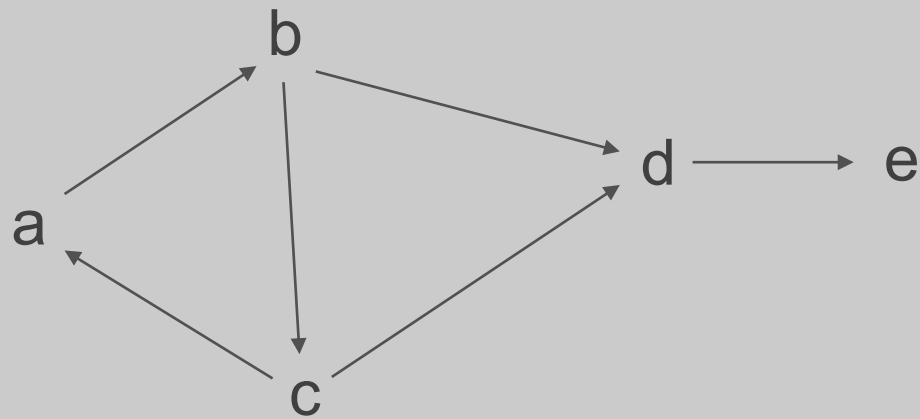
Client:
full programming language
(Java, C+, etc)

DB Server

Transitive Closure in embedded SQL

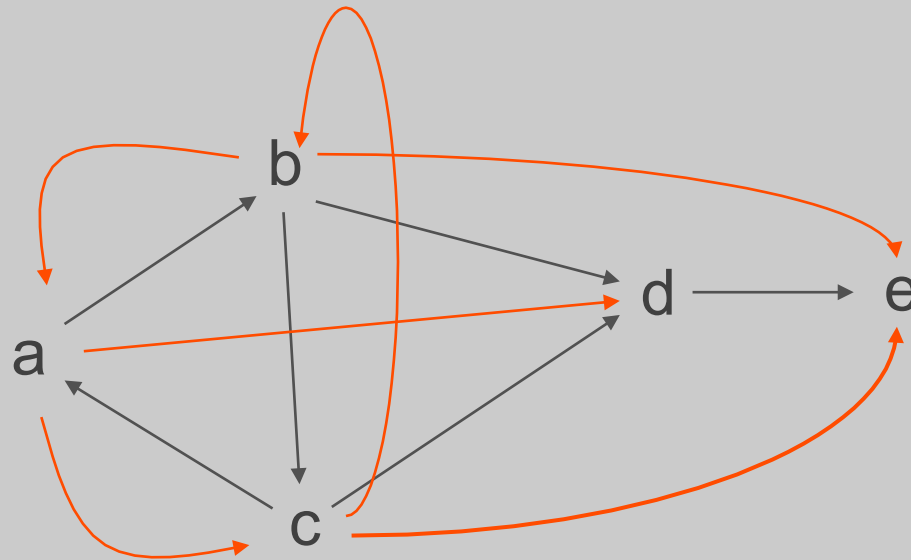
```
T := G
Δ := G
while Δ ≠ ∅ do
  { Told = T
    T := (select * from T)
         union
         (select x.A, y.B from G x, T y
          where x.B = y.A)
    Δ := T - Told }
Output T
```

Example



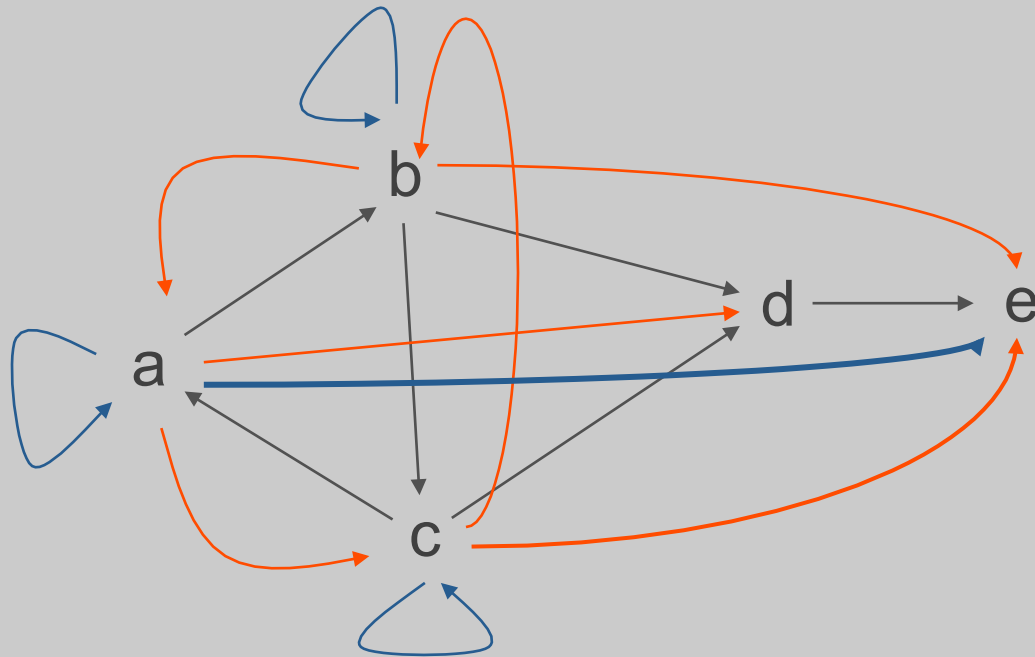
T_1 and Δ_1

Example



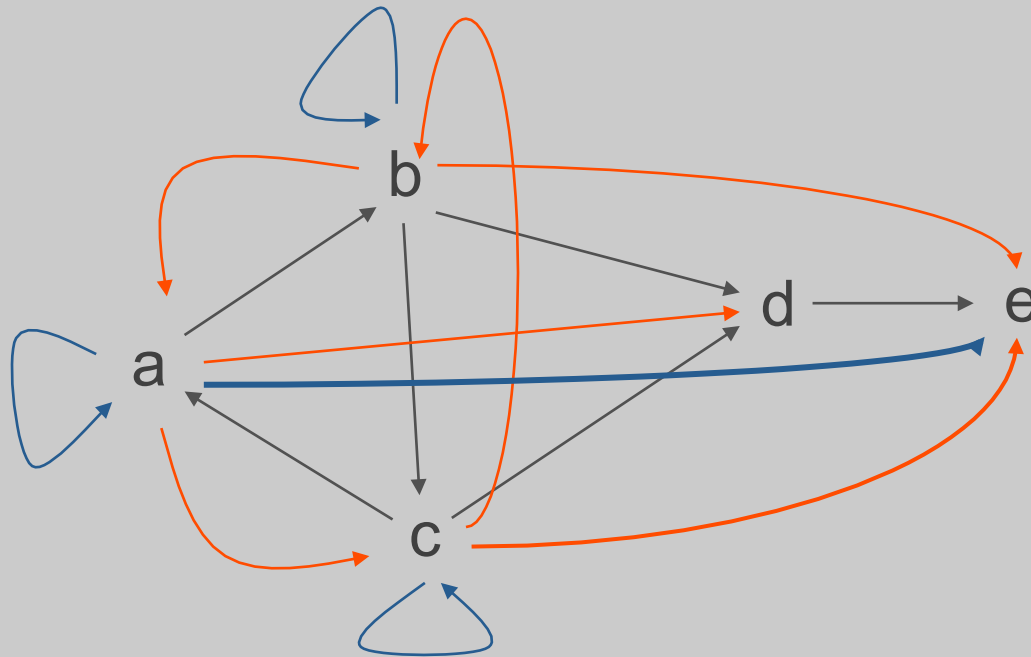
T_2 and Δ_2

Example



T_3 and Δ_3

Example



$T_4 = T_3$ and $\Delta_4 = \emptyset$: **Stop!**

Algorithm revisited

```
T := G
Δ := G
while Δ ≠ ∅ do
  { Told = T
    T := (select * from T)
         union
         (select x.A, y.B from G x, T y
          where x.B = y.A)
    Δ := T - Told }
Output T
```

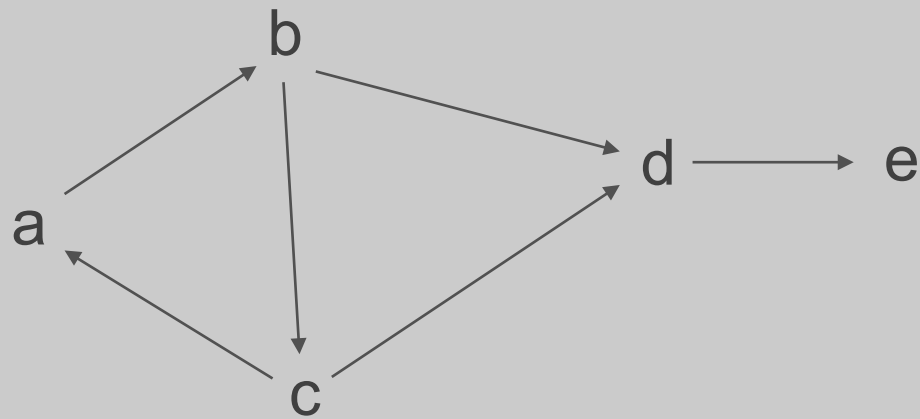
Converges in $\text{diameter}(G)$ iterations
(maximum distance between two nodes in G)

Optimization: "semi-naïve" evaluation

Use at least one new tuple (from Δ) every time!

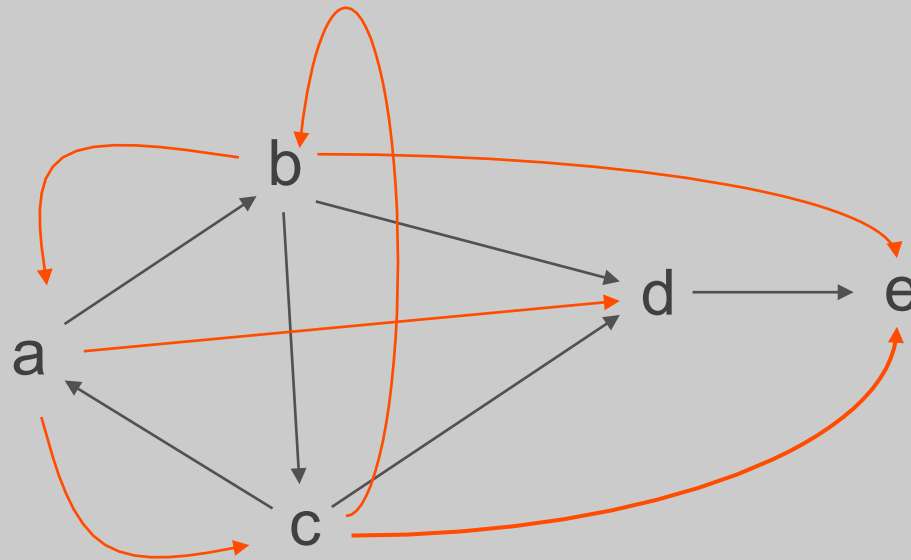
```
T := G
Δ := G
while Δ ≠ Φ do
    { Told = T
      T := (select * from T)
           union
           (select x.A, y.B from G x, Δ y
            where x.B = y.A)
      Δ := T - Told }
Output T
```

Example



T_1 and Δ_1

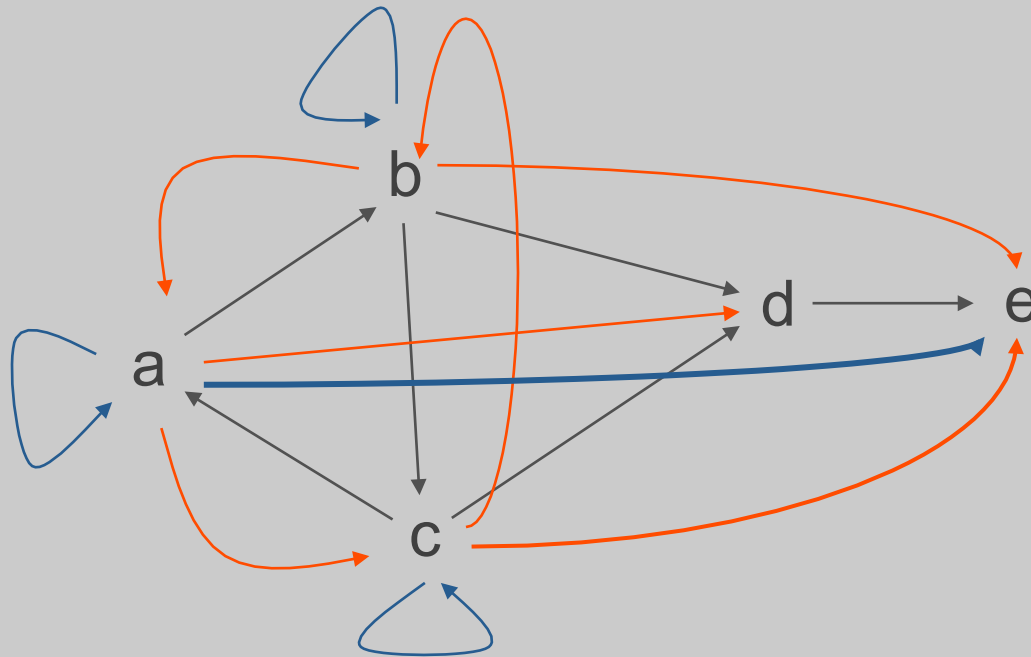
Example



T_1 and Δ_2

No longer recompute $\langle c, b \rangle$ but recompute $\langle c, d \rangle$

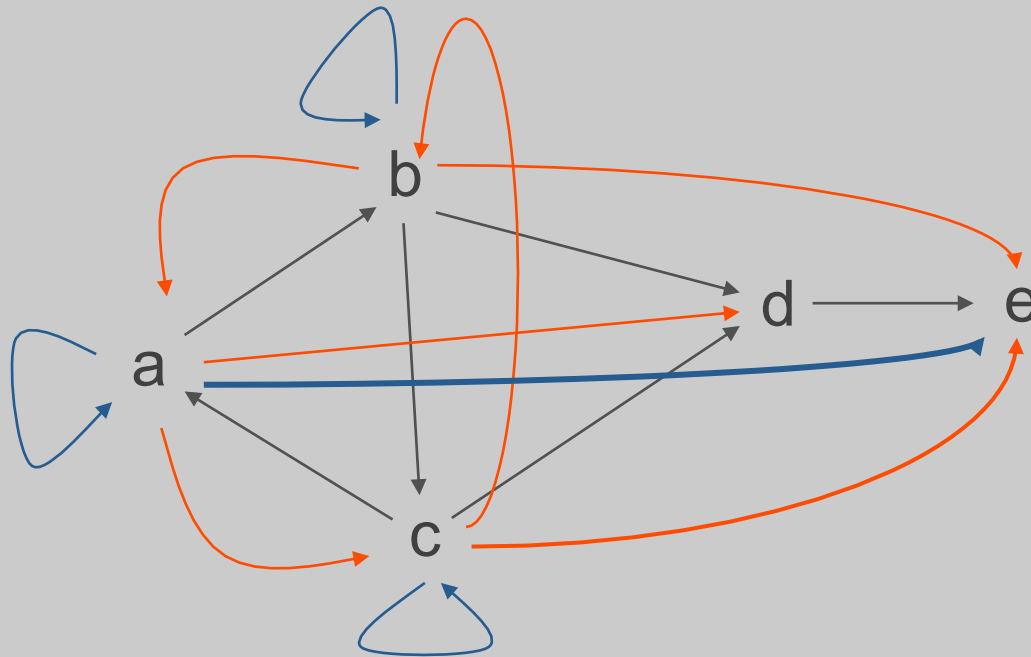
Example



T_1 and Δ_3

No longer recompute $\langle a, d \rangle$ but recompute $\langle c, e \rangle$

Example



$T_4 = T_3$ and $\Delta_4 = \emptyset$: **Stop!**

Faster Convergence (double recursion)

```
T := G
Δ := G
while Δ ≠ ∅ do
  { Told = T
    T := (select * from T)
         union
         (select x.A, y.B from T x, T y
          where x.B = y.A)
    Δ := T - Told }
Output T
```

Converges in $\log(\text{diameter}(G))$ iterations

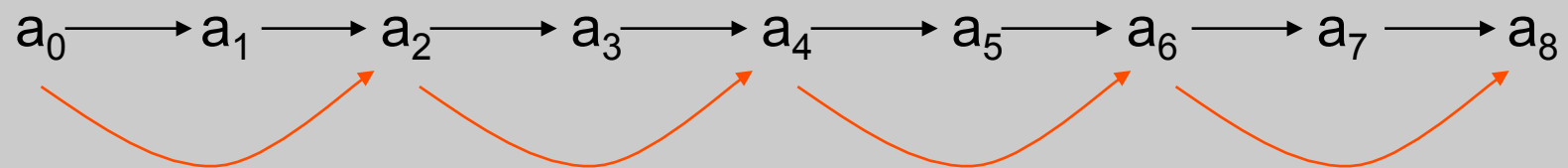
Example

Focus on computing $\langle a_0, a_8 \rangle$

$$a_0 \longrightarrow a_1 \longrightarrow a_2 \longrightarrow a_3 \longrightarrow a_4 \longrightarrow a_5 \longrightarrow a_6 \longrightarrow a_7 \longrightarrow a_8$$

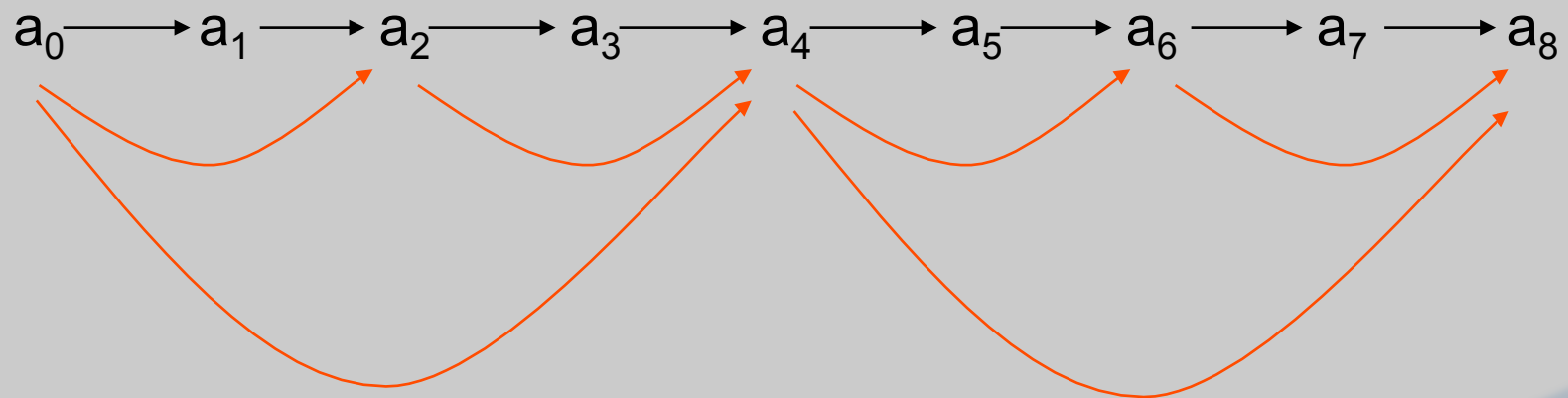
Example

Focus on computing $\langle a_0, a_8 \rangle$



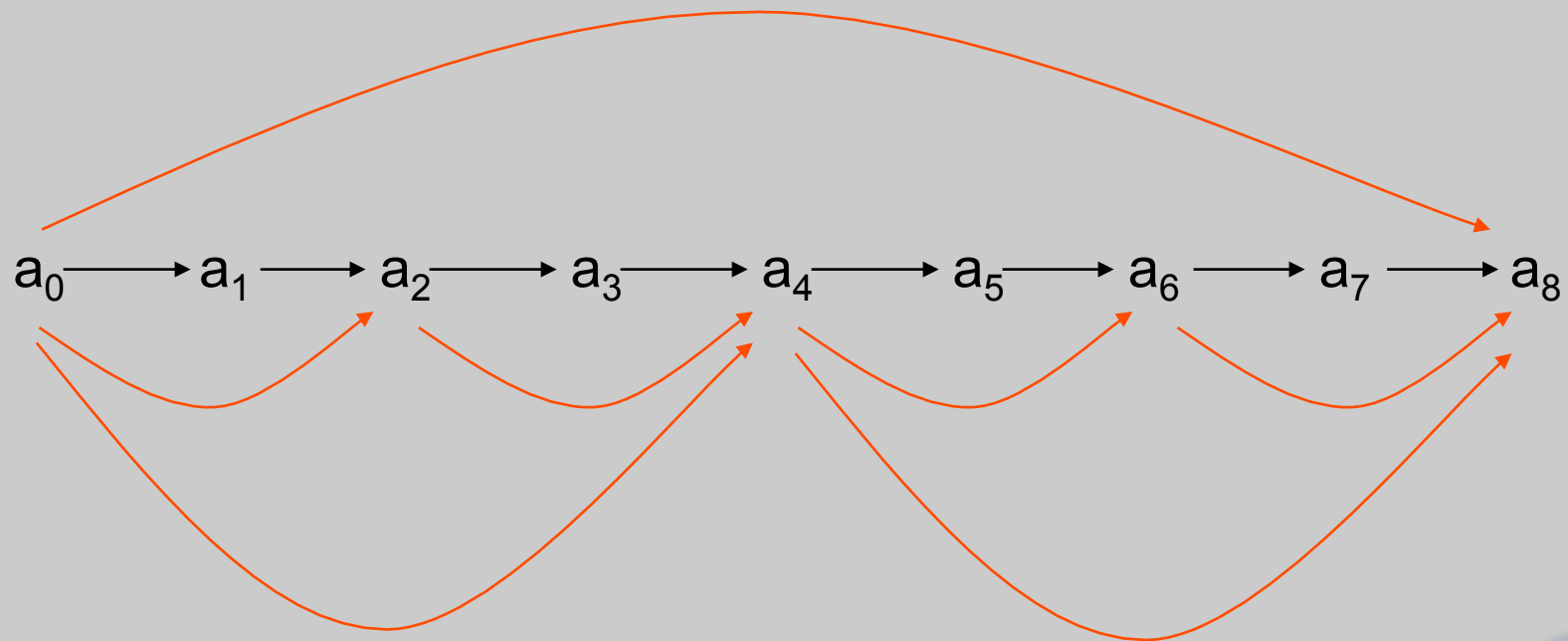
Example

Focus on computing $\langle a_0, a_8 \rangle$



Example

Focus on computing $\langle a_0, a_8 \rangle$



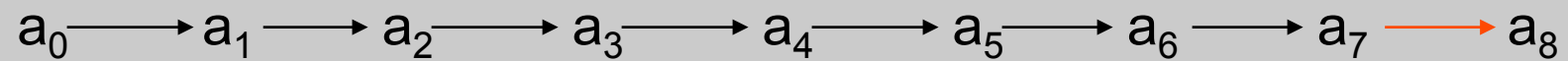
Example

Compare to linear recursion (first program)

$a_0 \longrightarrow a_1 \longrightarrow a_2 \longrightarrow a_3 \longrightarrow a_4 \longrightarrow a_5 \longrightarrow a_6 \longrightarrow a_7 \longrightarrow a_8$

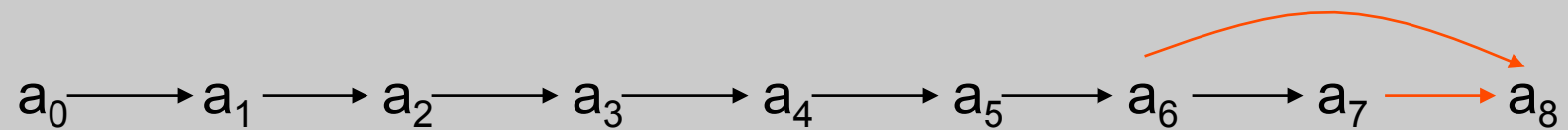
Example

Compare to linear recursion (first program)



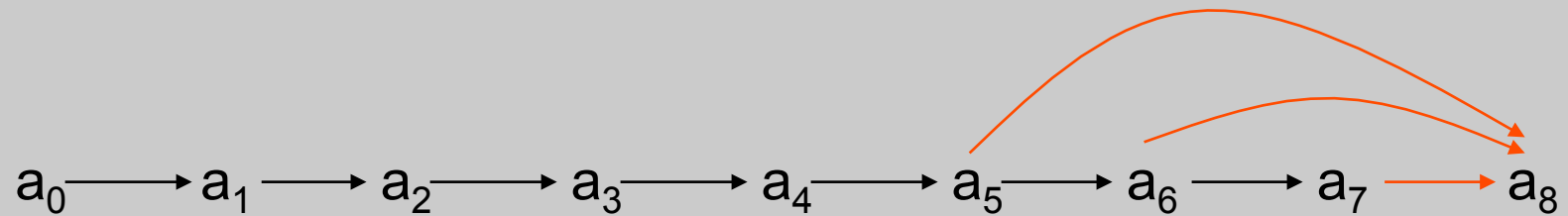
Example

Compare to linear recursion (first program)



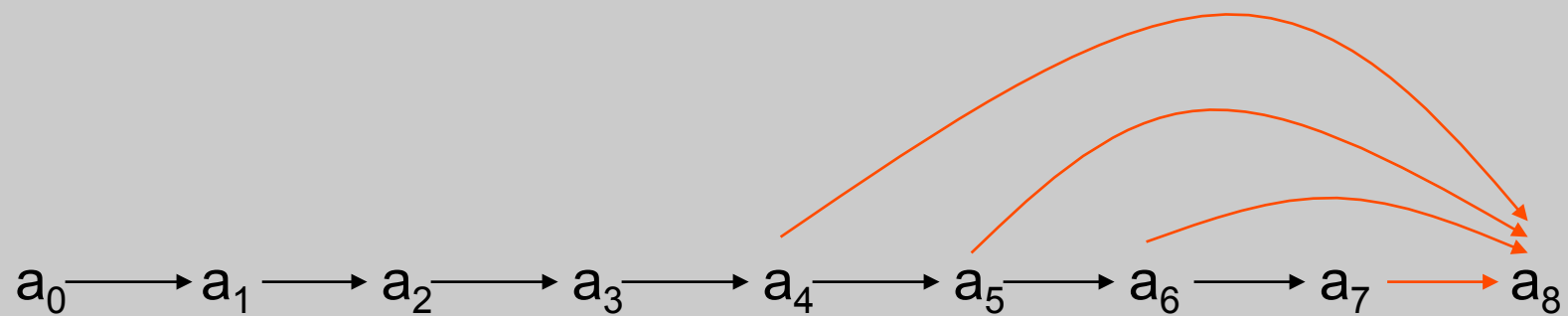
Example

Compare to linear recursion (first program)



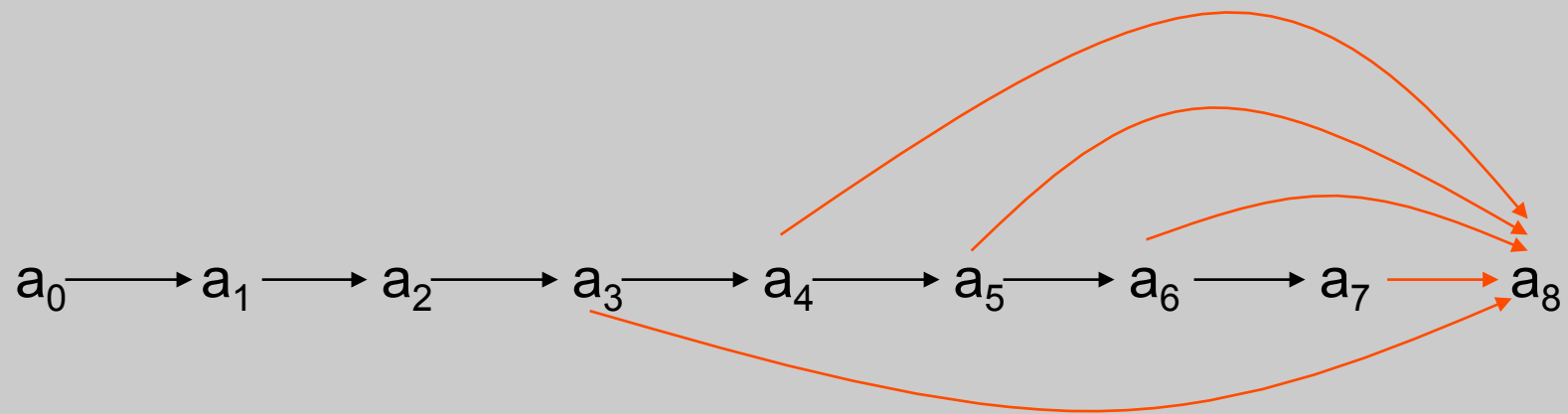
Example

Compare to linear recursion (first program)



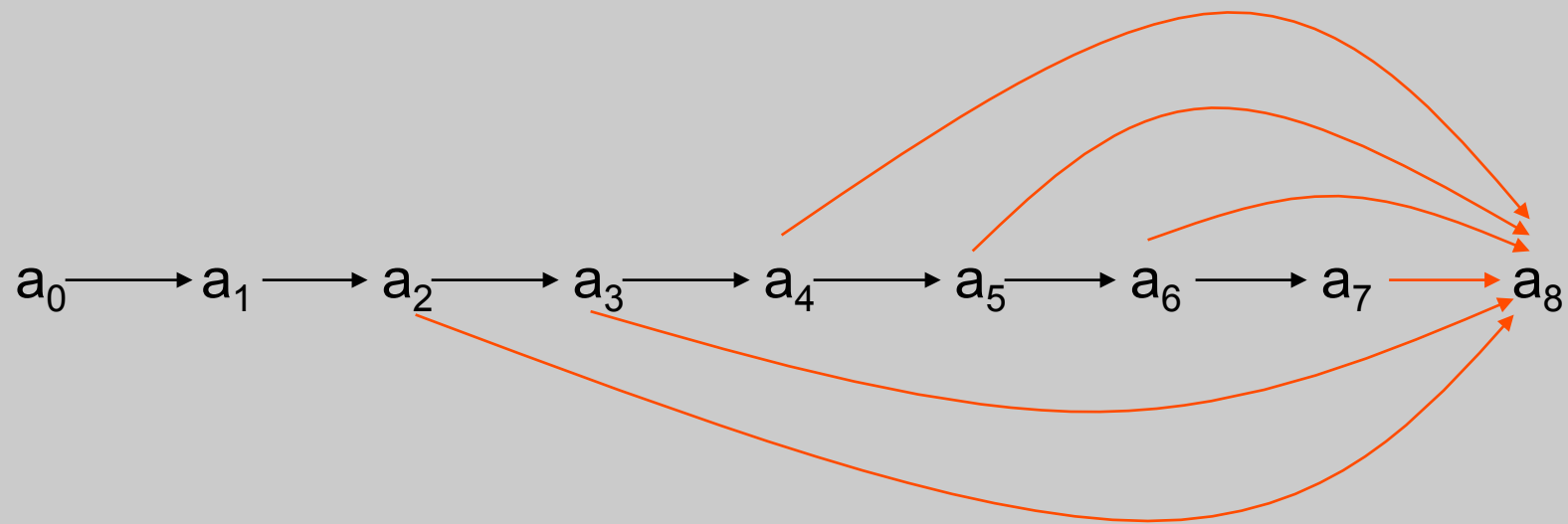
Example

Compare to linear recursion (first program)



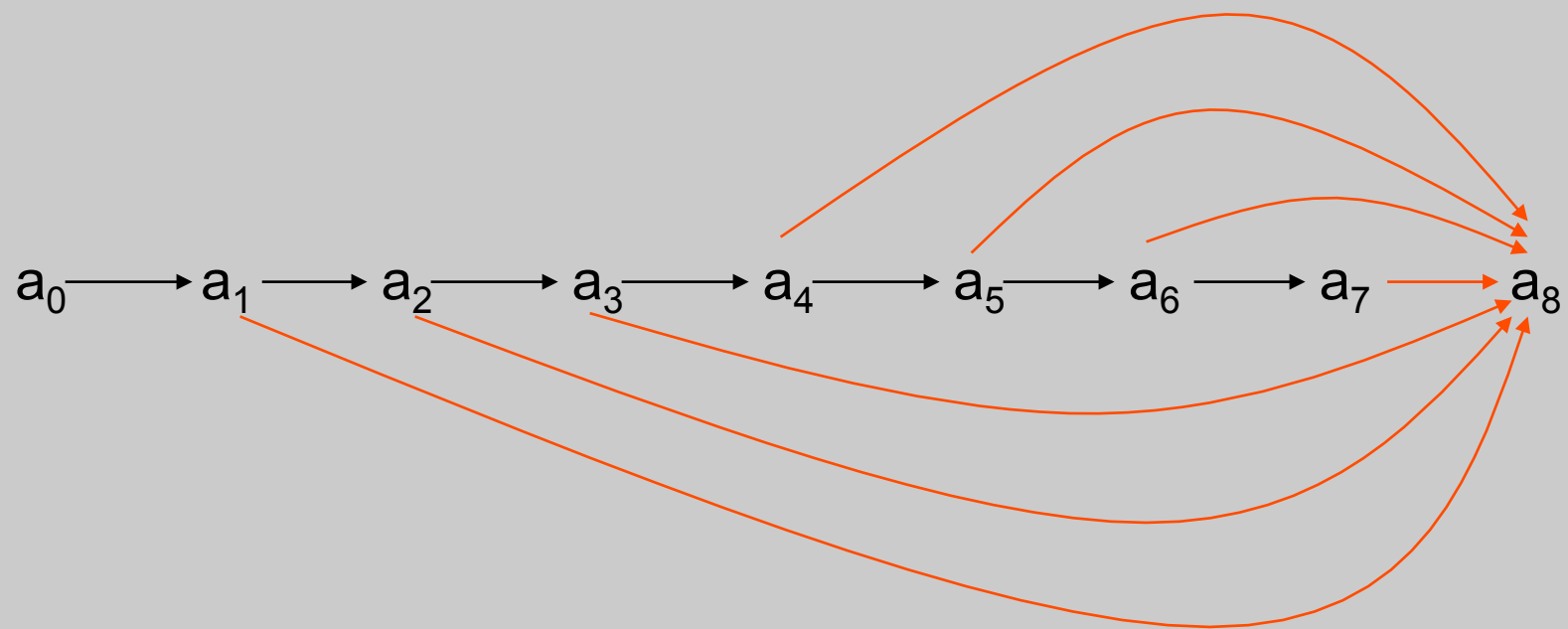
Example

Compare to linear recursion (first program)



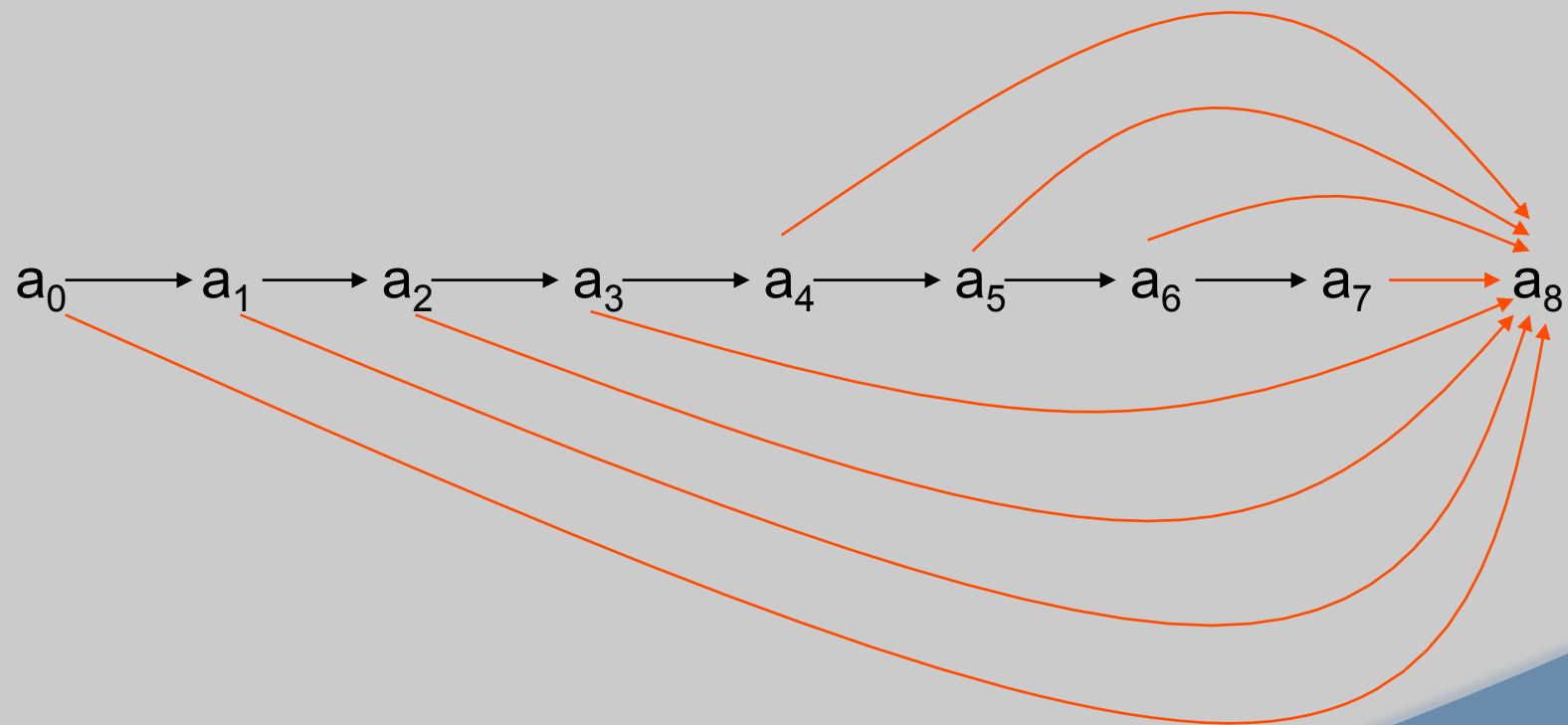
Example

Compare to linear recursion (first program)



Example

Compare to linear recursion (first program)



Optimization: "semi-naïve" evaluation

Again, use at least one new tuple (from Δ) every time!

```
T := G
Δ := G
while Δ ≠ ∅ do
  { Told = T
    T := (select * from T)
         union
         (select x.A, y.B from Δ x, T y
          where x.B = y.A)
         union
         (select x.A, y.B from T x, Δ y
          where x.B = y.A)
    Δ := T - Told }
Output T
```

JDBC

- Java Database Connectivity
- Allows SQL to be executed from within Java programs
- Similar to embedded SQL with the following difference:
 - Embedded SQL: SQL processed at compile time
 - JDBC: SQL interpreted at run-time