Limitations of SQL

flight	from	to
	SD	LA
	SD	ORD
	LA	NY
	…………..	

"Is there a way to get from City1 to City2" ?

Easier:

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

City1 ⟶ City2

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

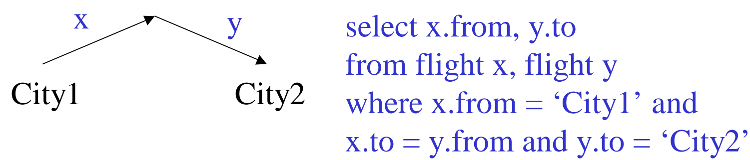
Easier:

"Is there a way to get from City1 to City2 with at most one stopover?"

City1 → City2

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

OR




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

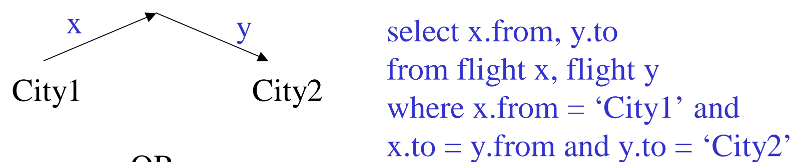
Easier:

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

City1 → City2

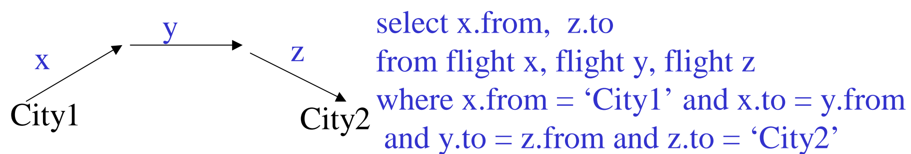
```
select * from flight
where from = 'City1' and to ='City2'
```

OR



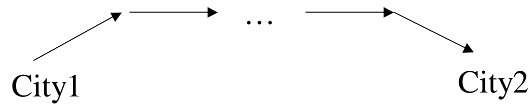

```
select x.from, y.to
from flight x, flight y
where x.from = 'City1' and
x.to = y.from and y.to = 'City2'
```

OR




```
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'
```

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



Need k+1 tuple variables!

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

Cannot do in SQL!

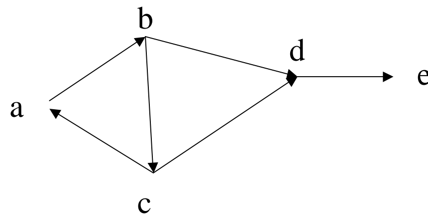
Similar examples

- 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 a graph

G	A	B
	a	b
	c	a
	b	c
	………	

Find the pairs of nodes <x,y> that are connected by some directed path



	a	b
	a	d
	a	e
	b	d
	b	e
	……..	

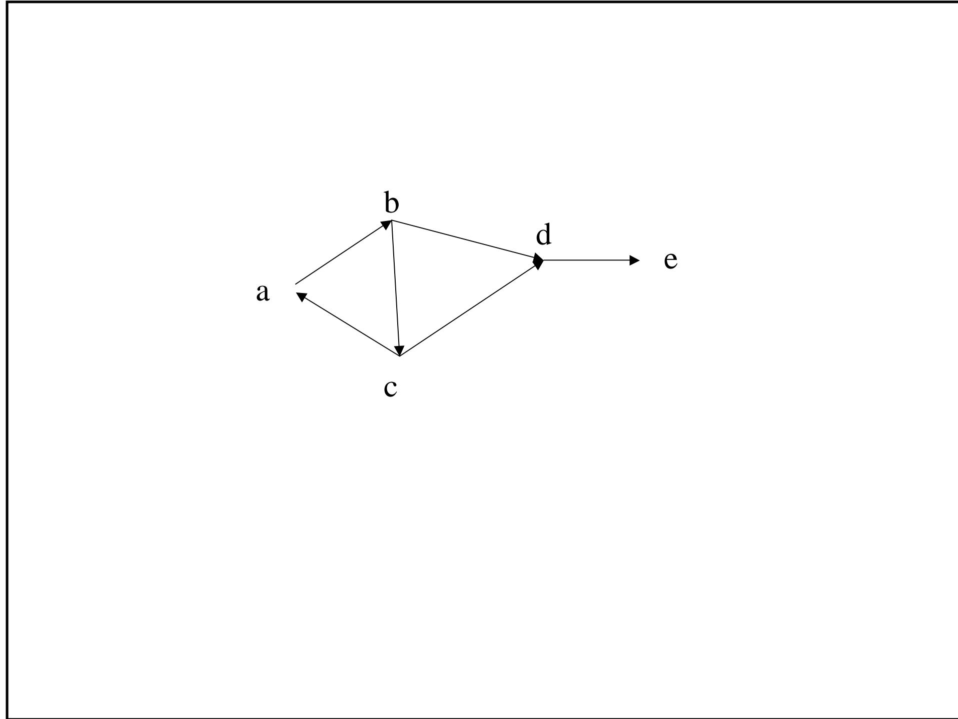
Computing transitive closure T of G

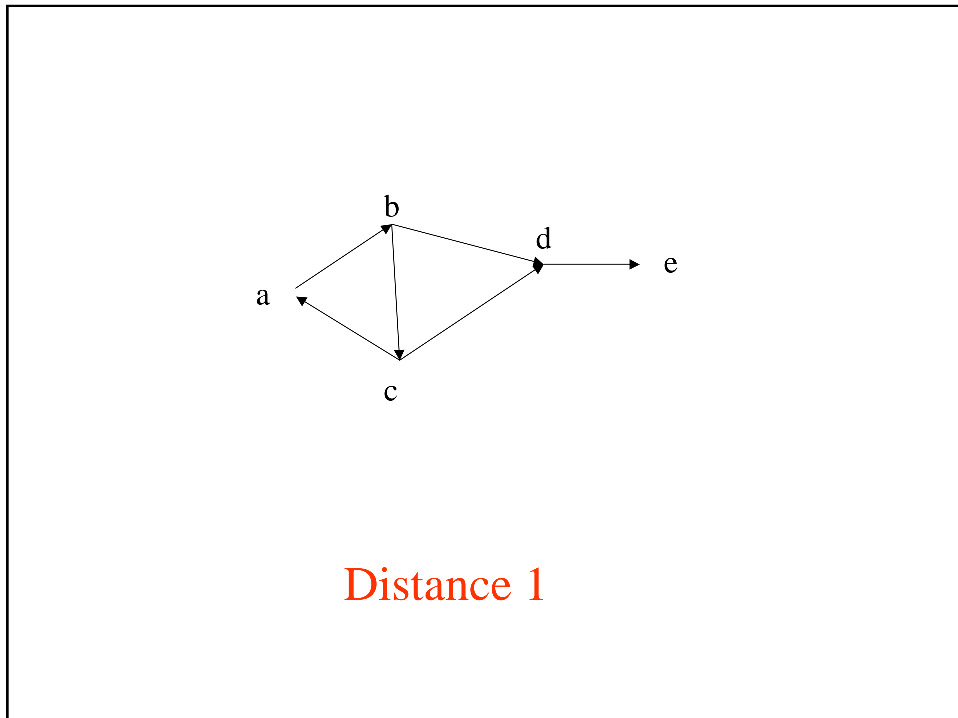
"Find the pairs of nodes <a,b> that are connected in G"

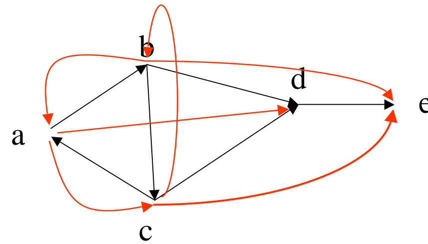
Same as

"find pairs of nodes <a,b> at distance 1" UNION
"find pairs of nodes <a,b> at distance at most 2" UNION
………..
"find pairs of nodes <a,b> at distance at most k" UNION
………..

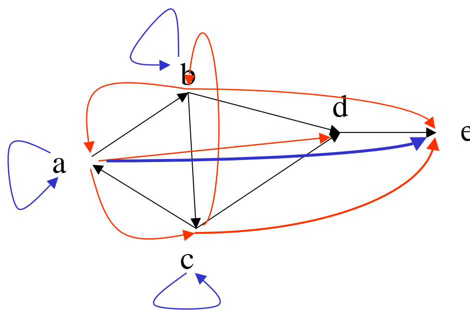
When to stop? At some point, no new nodes are added.
Distance cannot be larger than total number of nodes in G.


b
d
e
a
c


b
d
e
a
c
Distance 1



Distance ≤ 2



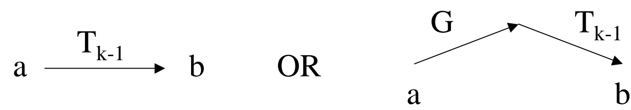
Distance ≤ 3

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

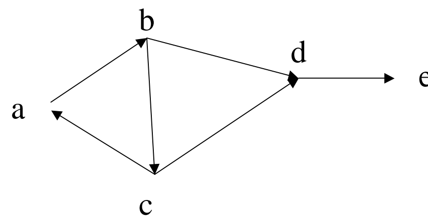
T_1 : “find pairs of nodes <a,b> at distance 1”

select * from G

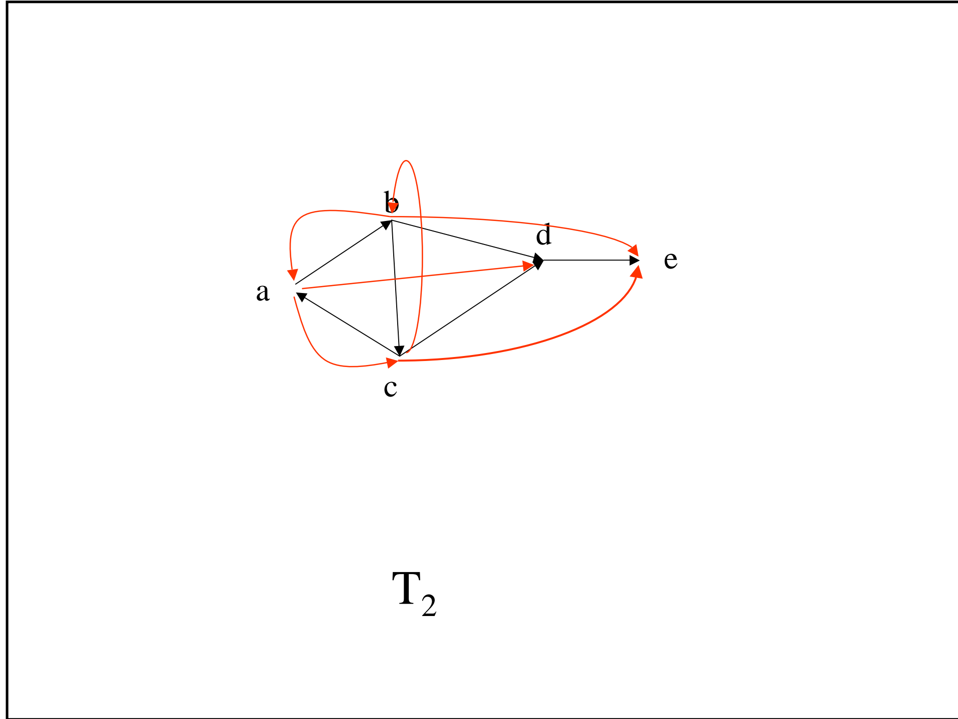
T_k : “find the pairs of nodes <a,b> at distance at most k”

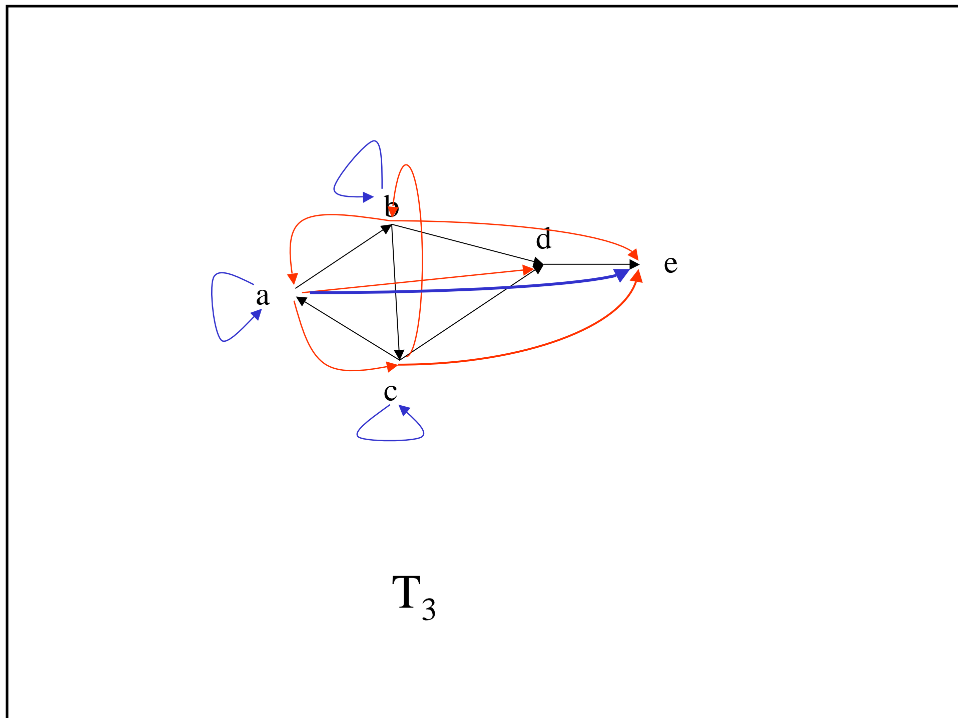


(select * from T_{k-1})
union
(select x.A, y.B from G x, T_{k-1} y
where x.B = y.A)



T_1


b
d
a
e
c
T2


b
d
a
e
c
T3

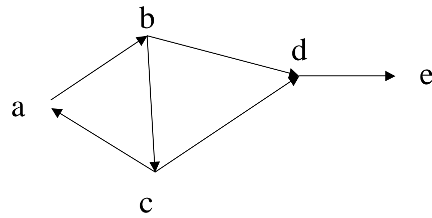
One solution: extend SQL with recursion

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

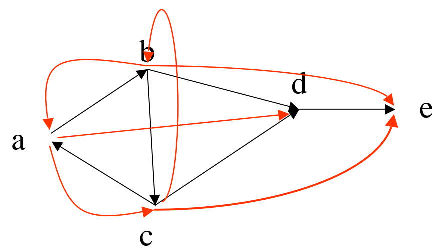
Another way: Embedded SQL

<pseudo-code>

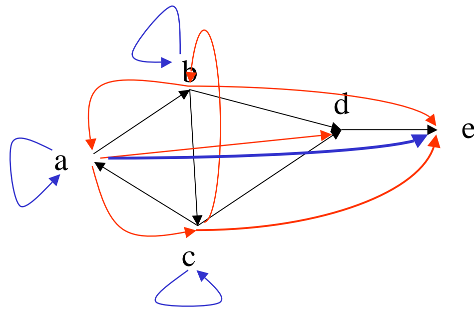
```
T := G
Δ := G
while Δ ≠ Φ do
    { T_old = T
      T := (select * from T)
           union
           (select x.A, y.B from G x, T y
           where x.B = y.A)
      Δ := T – T_old }
Output T
```



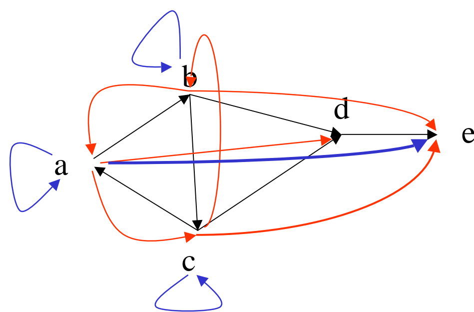
T_1 and Δ_1



T_2 and Δ_2



T_3 and Δ_3



$T_4 = T_3$ and $\Delta_4 = \Phi$

stop!

Another way: Embedded SQL

<pseudo-code>

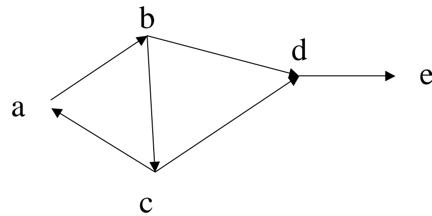
```
T := G
Δ := G
while Δ ≠ Φ do
    { T_old = T
      T := (select * from T)
           union
           (select x.A, y.B from G x, T y
           where x.B = y.A)
      Δ := T – T_old }
Output T
```

Converges in diameter(G) iterations
(maximum distance between two nodes in G)

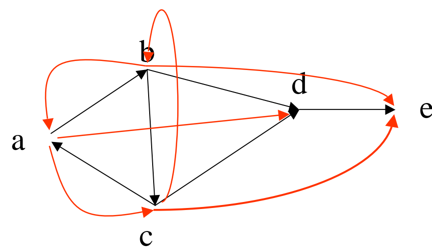
Optimization (“semi-naïve” evaluation): use at least one new tuple (from Δ) every time

<pseudo-code>

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

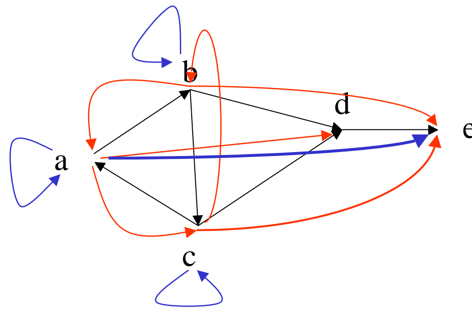


T_1 and Δ_1



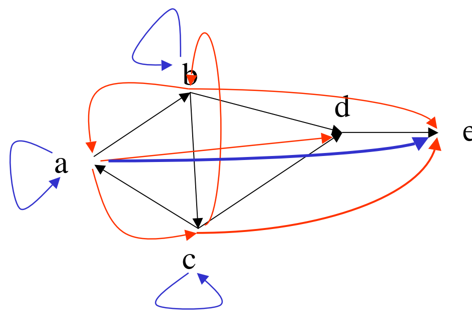
T_2 and Δ_2

No longer recompute <c,b> but recompute <c,d>



T_3 and Δ_3

No longer recompute <a,c> but recompute <c,e>



$T_4 = T_3$ and $\Delta_4 = \Phi$

stop!

Faster convergence (double recursion):

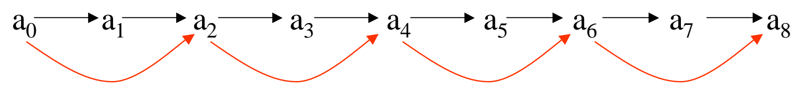
$T := G$
$\Delta := G$
while $\Delta \neq \Phi$ do
{ $T_{old} = T$
T := (select * from T)
union
(select x.A, y.B from T x, T y
where x.B = y.A)
$\Delta := T - T_{old}$ }
Output T

Converges in log(diameter(G)) iterations

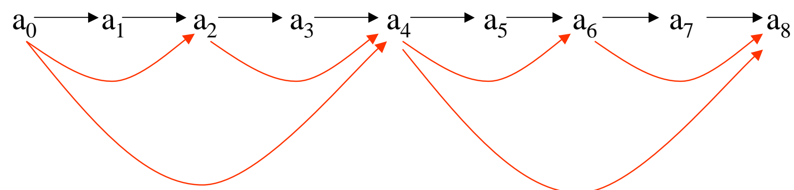
Example (focus on computing <a_0,a_8>)

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

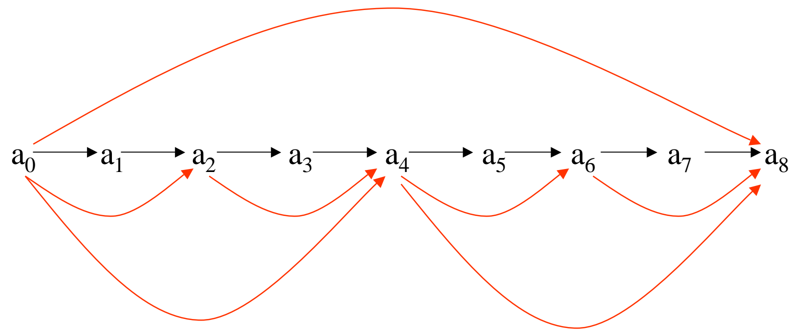
Example (focus on computing <a_0,a_8>)



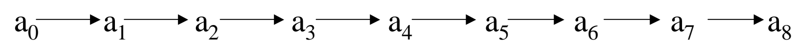
Example (focus on computing <a_0,a_8>)



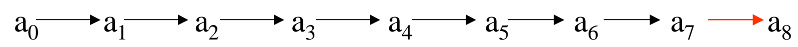
Example (focus on computing <a_0,a_8>)



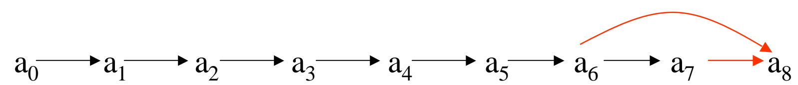
Compare to linear recursion (first program):



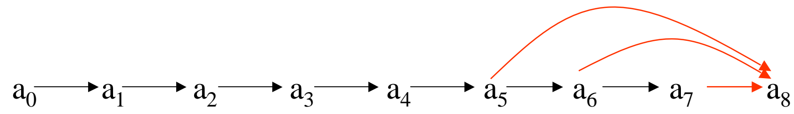
Compare to linear recursion (first program):



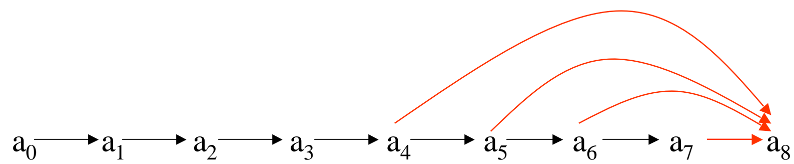
Compare to linear recursion (first program):



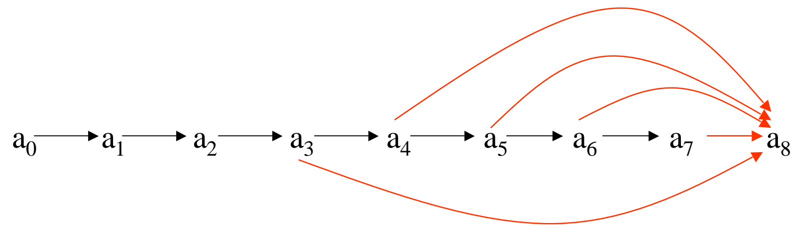
Compare to linear recursion (first program):



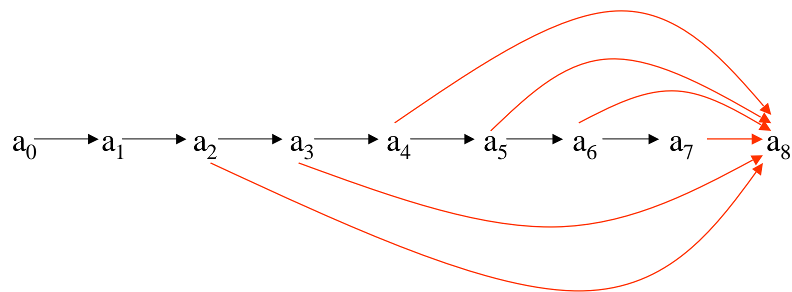
Compare to linear recursion (first program):



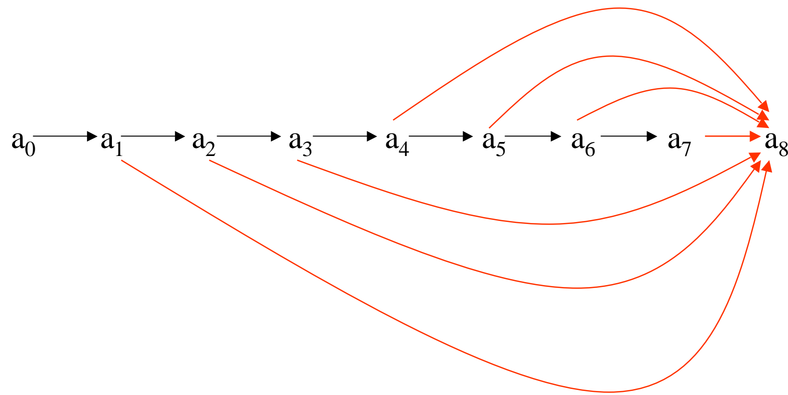
Compare to linear recursion (first program):



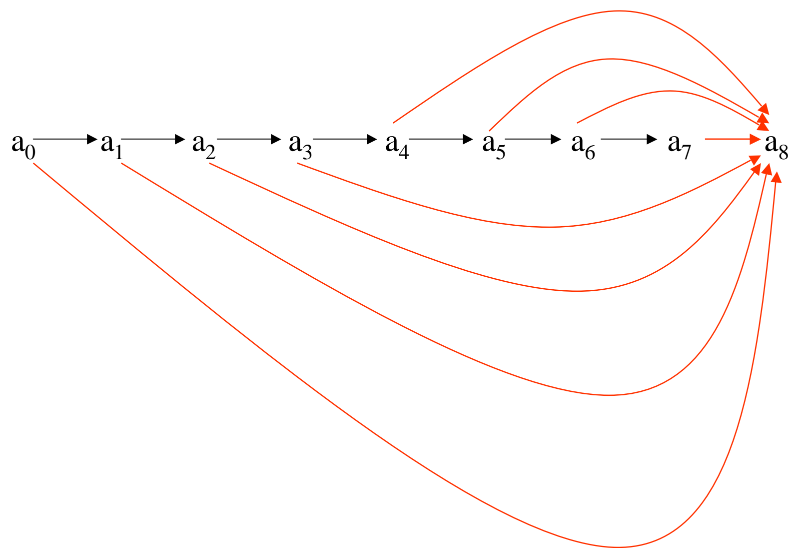
Compare to linear recursion (first program):



Compare to linear recursion (first program):



Compare to linear recursion (first program):



Optimization (“semi-naïve” evaluation): again, use at least one new tuple every time

```
T := G
Δ := G
while Δ ≠ Φ do
    { T_old = 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 − T_old
Output T
```