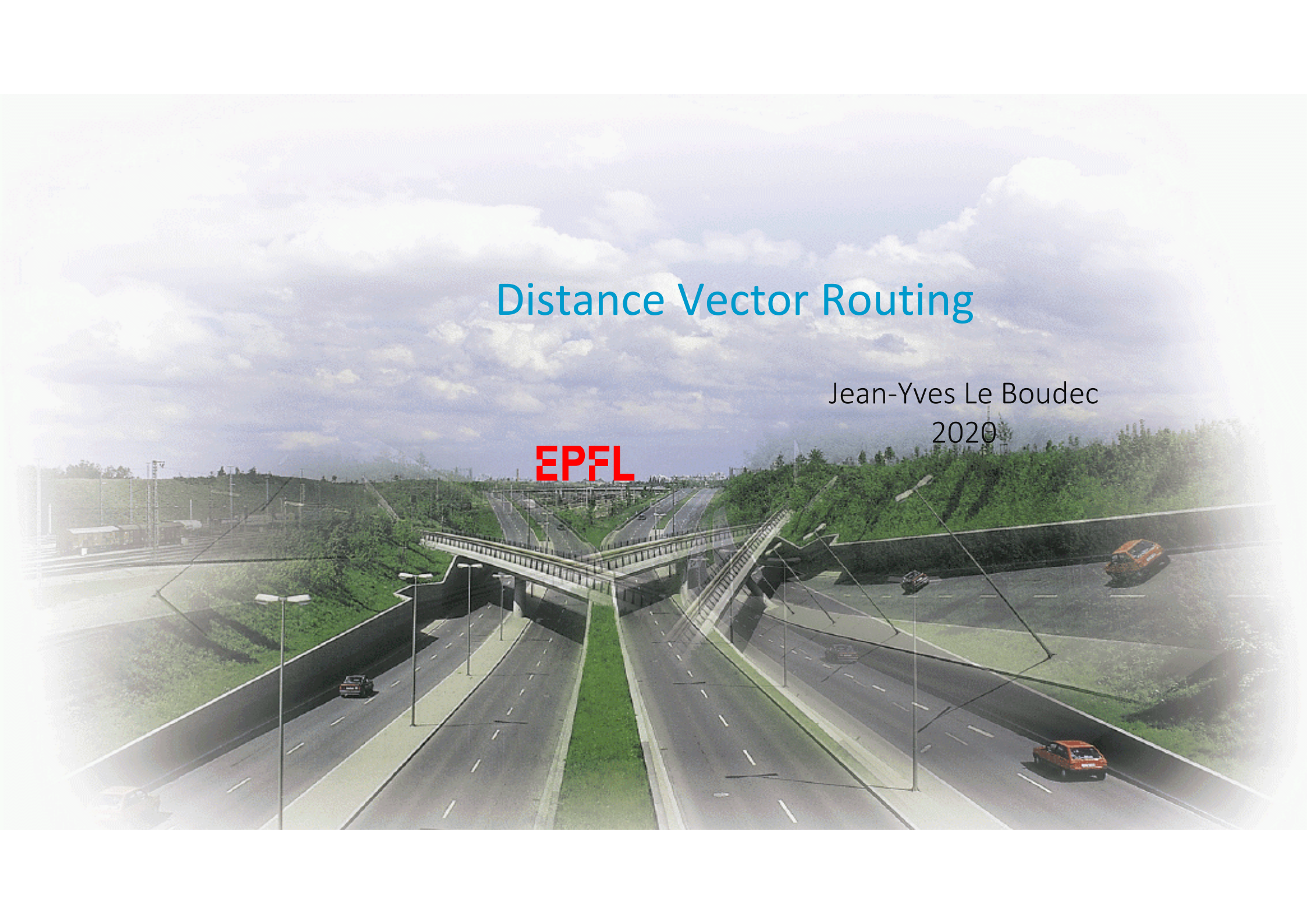


Distance Vector Routing

Jean-Yves Le Boudec
2020

EPFL

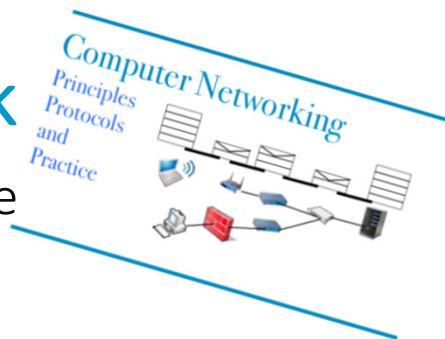


Contents

1. Distance vector: theory
2. Distance vector: practice
3. Braess paradox

Textbook

Section 5.1.1, The control plane



1. Distance Vector: Theory

What it does: it computes shortest paths to all destinations

Cost of a path = sum of the cost of network links along the path

Cost of a link is setup by configuration

Shortest path = path whose cost is minimal

Distance from node A to node B = cost of a shortest path from A to B

How it works

uses the “Distributed Bellman-Ford” algorithm

Fully distributed

Using as only information the distances from self to all destinations

The Centralized Bellman-Ford Algorithm

Algorithm BF-C

input: a directed graph with links costs $A(i,j)$; assume $A(i,j) > 0$ and $A(i,j) = \infty$ when nodes i and j are not connected.

output: vector p s.t. $p(i)$ = cost of best path from node i to node 1

$$p^0(1) = 0, \quad p^0(i) = \infty \text{ for } i \neq 1$$

for $k = 1, 2, \dots$ **do**

$$p^k(i) = \min_{j \neq i} [A(i,j) + p^{k-1}(j)] \text{ for } i \neq 1$$

$$p^k(1) = 0$$

until $p^k = p^{k-1}$

return(p^k)

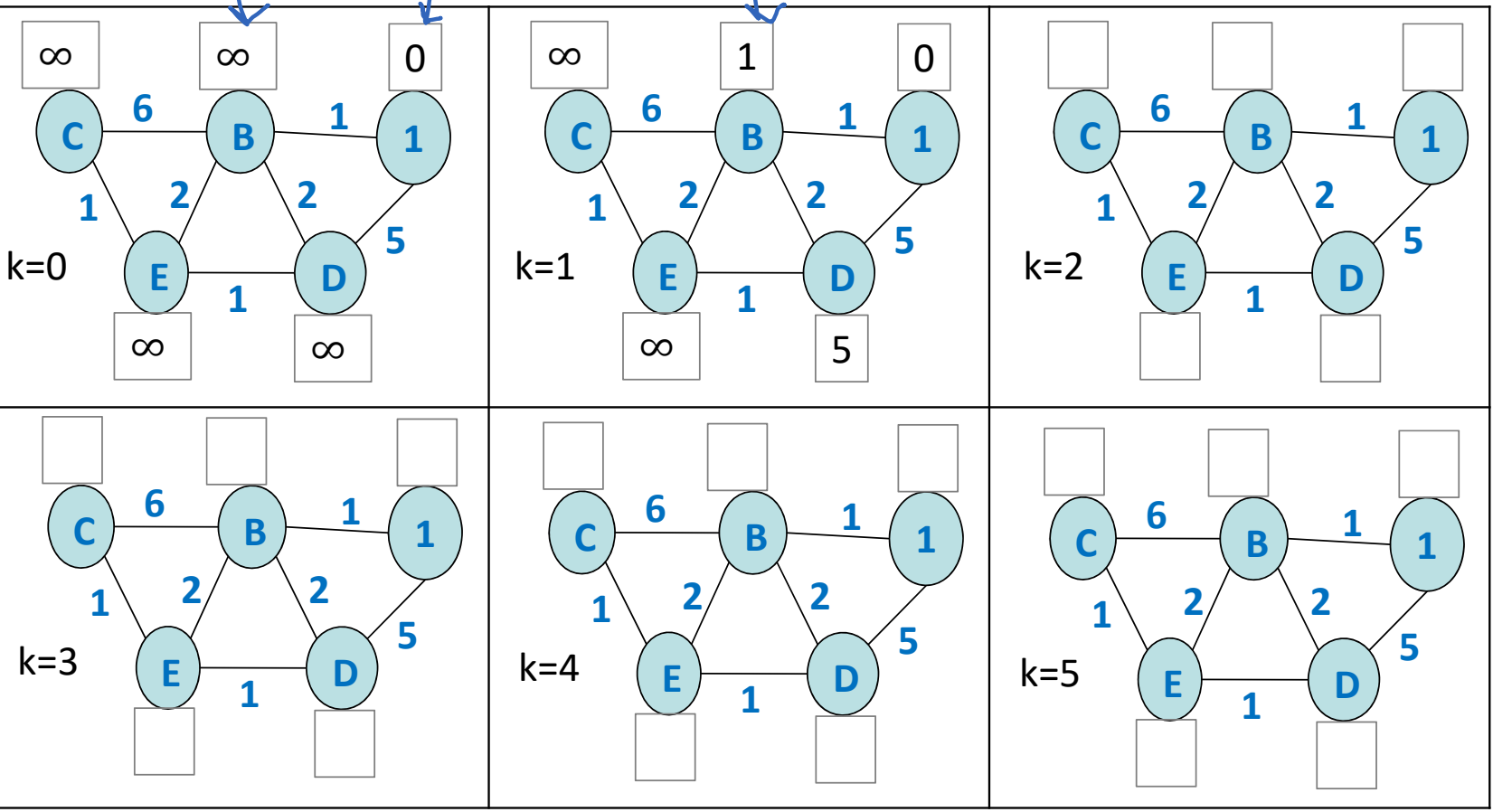
Example: shortest paths to node 1: run BF-C for $k = 1$

```

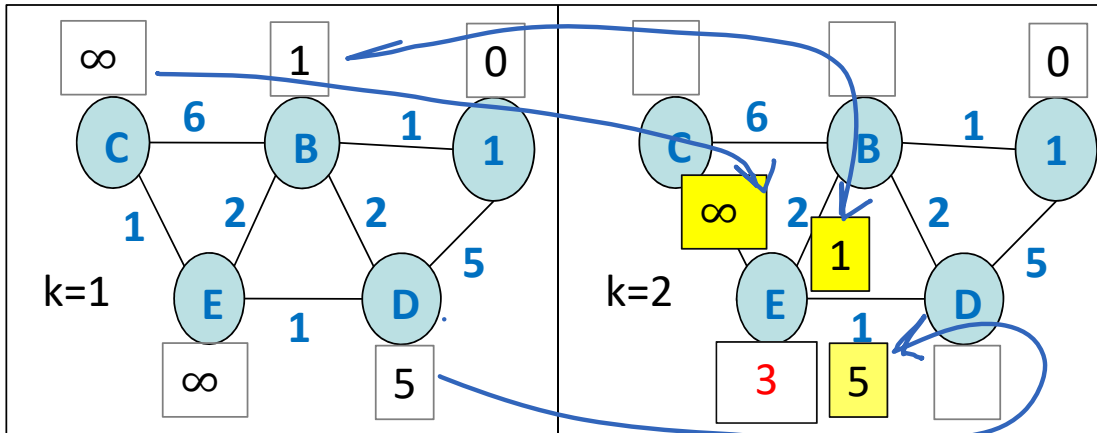
 $p^0(1) = 0, p^0(i) = \infty$  for  $i \neq 1$ 
for  $k = 1, 2, \dots$  do
     $p^k(i) = \min_{j \neq i} [A(i, j) + p^{k-1}(j)]$  for  $i \neq 1$ 
     $p^k(1) = 0$ 
until  $p^k = p^{k-1}$ 
return( $p^k$ )
    
```

BF-C

$p^0(B) = \infty$ $p^0(1) = 0$ $p^1(B) = 1$



Message Passing Interpretation of Bellman-Ford



```

 $p^0(1) = 0, p^0(i) = \infty$  for  $i \neq 1$ 
for  $k = 1, 2, \dots$  do
     $p^k(i) = \min_{j \neq i} [A(i, j) + p^{k-1}(j)]$  for all  $i$ 
     $p^k(1) = 0$ 
until  $p^k = p^{k-1}$ 
return( $p^k$ )
    
```

With BF-C it is as if :

at iteration k , node i receives from neighbors j their previous costs $p^{k-1}(j)$ and updates own cost by picking the best

BF-C can be interpreted as a *synchronous* message passing algorithm

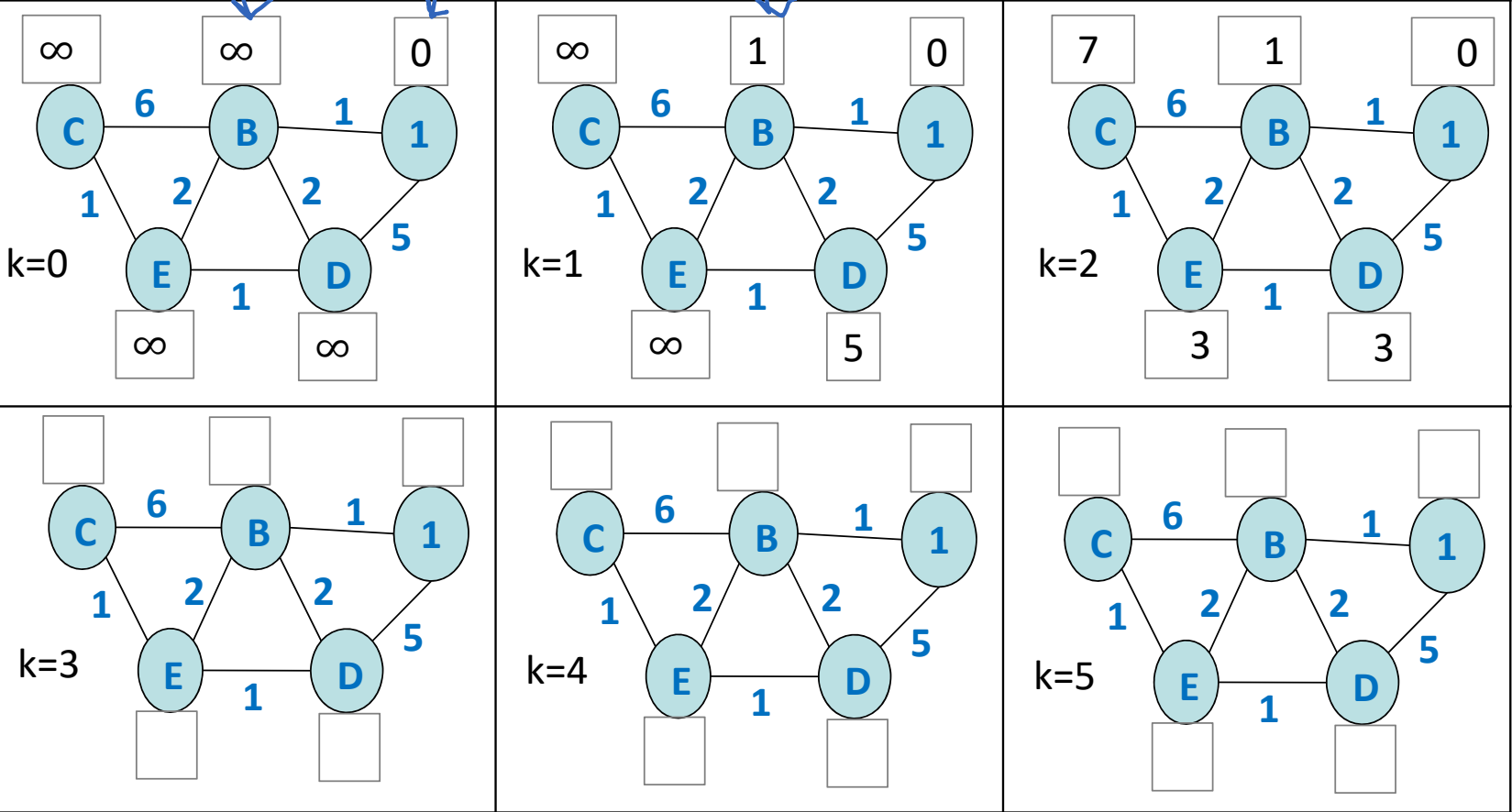
synchronous = nodes wait until iteration $k - 1$ is finished before starting iteration k

Example: shortest paths to node 1:
run BF-C, $k = 2, 3$

```

 $p^0(1) = 0, p^0(i) = \infty \text{ for } i \neq 1$ 
for  $k = 1, 2, \dots$  do
     $p^k(i) = \min_{j \neq i} [A(i, j) + p^{k-1}(j)]$  for  $i \neq 1$ 
     $p^k(1) = 0$ 
until  $p^k = p^{k-1}$  BF-C
return( $p^k$ ) BF-C
    
```

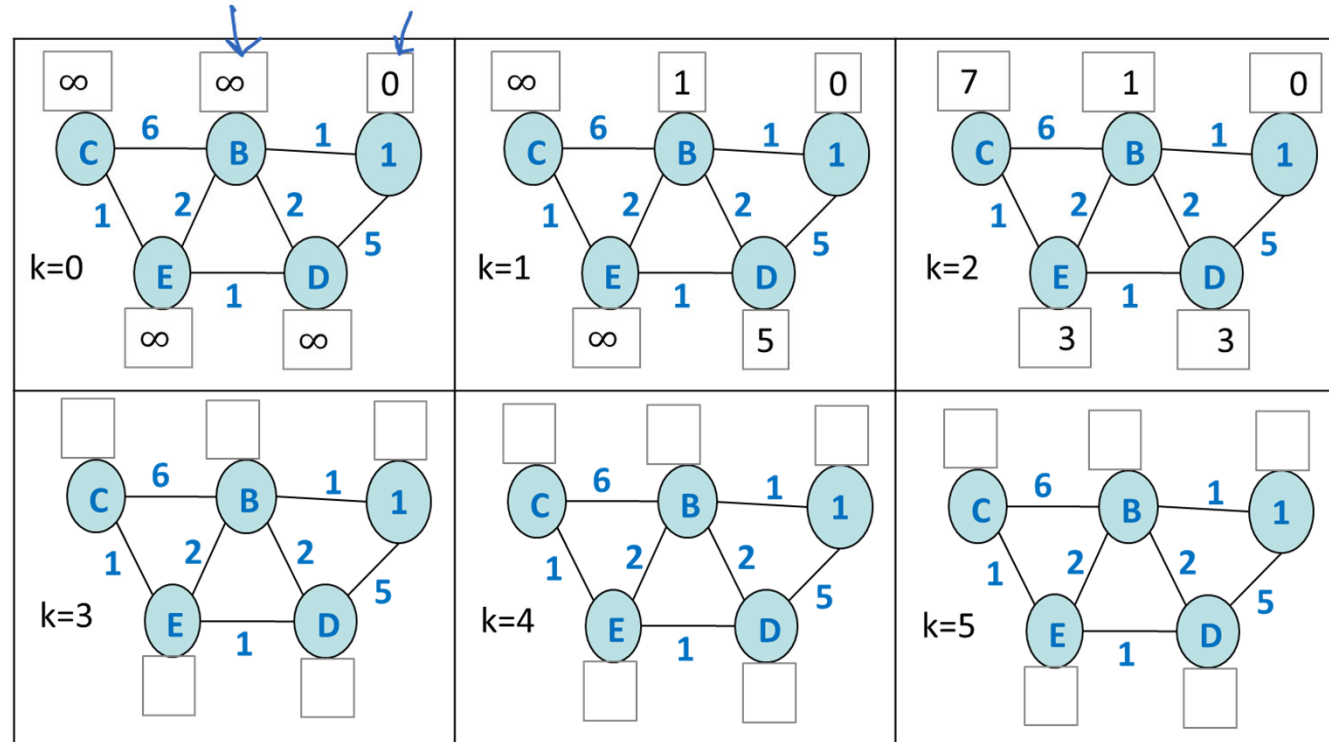
$p^0(B) = \infty$ $p^0(1) = 0$ $p^1(B) = 1$



At which k does the algorithm stop ?

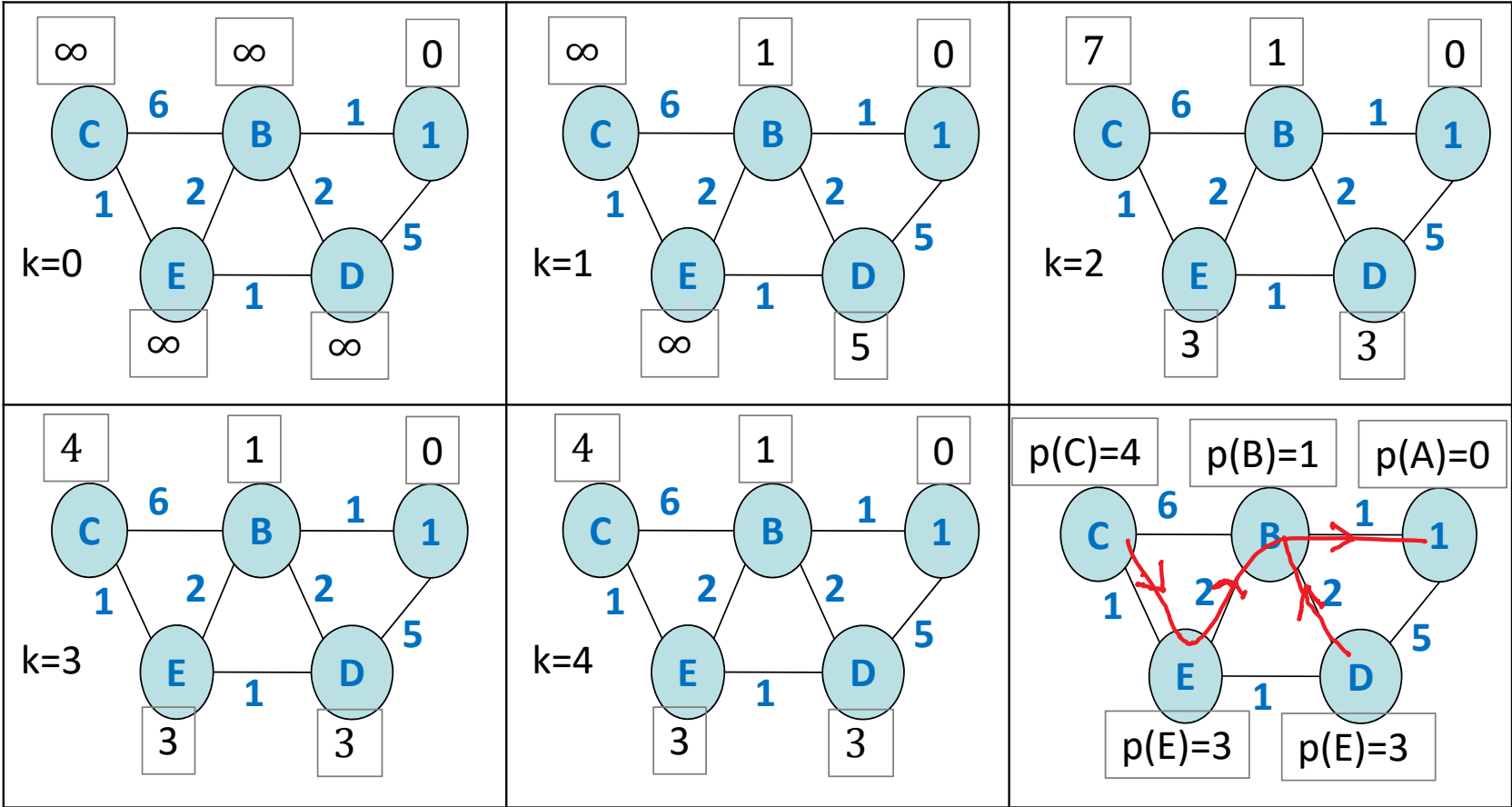
(i.e. what is the smallest k such that $p^k = p^{k-1}$?)

- A. $k = 1$
- B. $k = 2$
- C. $k = 3$
- D. $k = 4$
- E. $k = 5$
- F. $k \geq 6$
- G. I don't know



Solution

Answer D: $p^3 = p^4$; algorithm stops at $k = 4$



Correctness of BF-C

Algorithm BF-C

input: a directed graph with links costs $A(i,j)$; assume $A(i,j) > 0$ and $A(i,j) = \infty$ when nodes i and j are not connected.

output: vector p s.t. $p(i)$ = cost of best path from node i to node 1

$$p^0(1) = 0, \quad p^0(i) = \infty \text{ for } i \neq 1$$

for $k = 1, 2, \dots$ **do**

$$p^k(i) = \min_{j \neq i} [A(i,j) + p^{k-1}(j)] \text{ for } i \neq 1$$

$$p^k(1) = 0$$

until $p^k = p^{k-1}$

return(p^k)

Theorem 1

1. If the network is fully connected, BF-C stops at the latest at $k = n$ (where n = number of nodes) and outputs the distance along shortest path from i to 1, for all i
2. $p^k(i)$ is the distance from i to 1 in at most k hops
3. The next hop along a shortest path from $i \neq 1$ to 1 is found by $\text{nextHop}(i) \in \text{Argmin}_{j \neq i} [A(i,j) + p(j)]$

Computation of shortest path tree

3. The next hop along a shortest path from $i \neq 1$ to 1 is found by $\text{nextHop}(i) \in \text{Argmin}_{j \neq i} [A(i,j) + p(j)]$

When algorithm has converged, the next hop of node C is obtained by looking for j that minimizes $A(C,j) + p(j)$

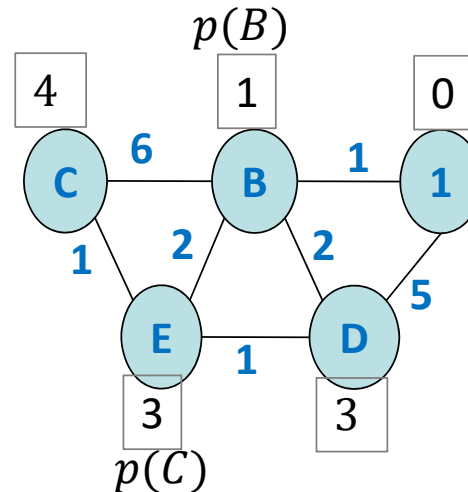
$$j = B: 6 + 1 = 7$$

$$j = E: 1 + 3 = 4$$

Other j : ∞

Argmin is $\{j = E\}$

$$\text{nextHop}(C) = E$$



Assume we start from other initial conditions. What will happen ?

~~$p^0(1) = 0, p^0(i) = \infty$ for $i \neq 1$~~
 replaced by values on graph \rightarrow

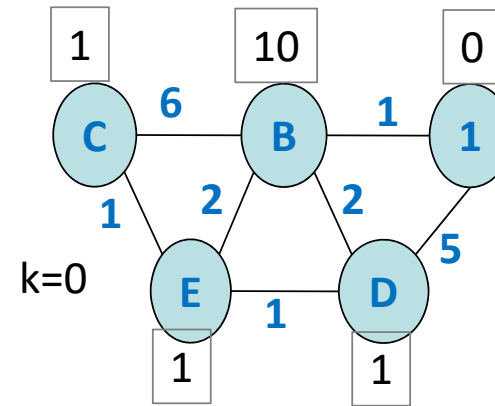
for $k = 1, 2, \dots$ do

$$p^k(i) = \min_{j \neq i} [A(i, j) + p^{k-1}(j)] \text{ for } i \neq 1$$

$$p^k(1) = 0$$

until $p^k = p^{k-1}$

return(p^k)



- A. The algorithm converges to the correct distances
- B. The algorithm does not converge
- C. The algorithm converges but not to the correct distances
- D. I don't know

Solution

Answer A

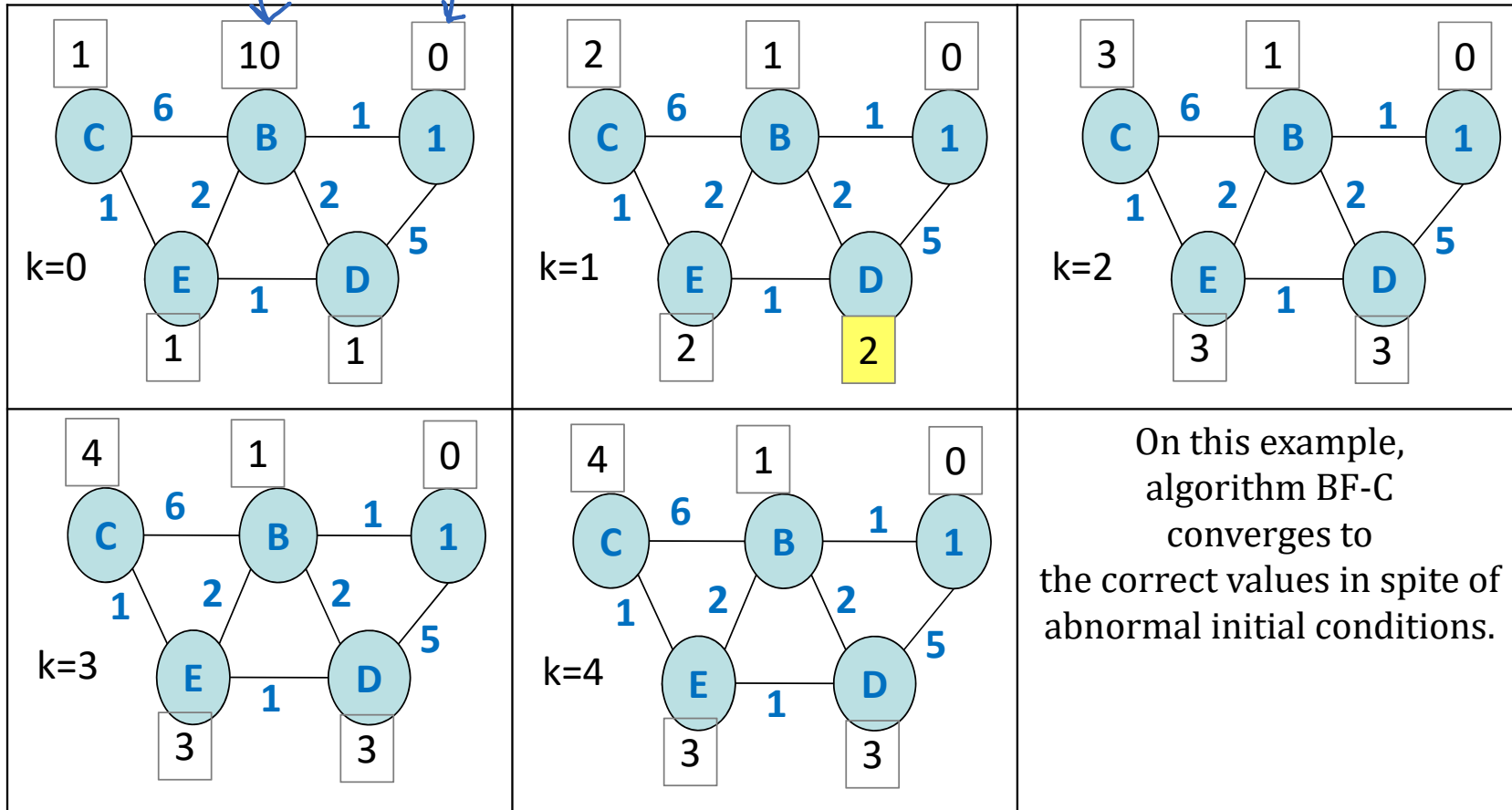
$p^0(B) = 10$ $p^0(0) = 0$ $p^1(B) = 1$

```

 $p^0(1) = 0, p^0(i) = \infty$  for  $i \neq 1$ 

for  $k = 1, 2, \dots$  do
     $p^k(i) = \min_{j \neq i} [A(i, j) + p^{k-1}(j)]$  for  $i \neq 1$ 
     $p^k(1) = 0$ 
until  $p^k = p^{k-1}$ 
return( $p^k$ )
    
```

BF-C



Assume we break some links and start from these initial conditions. What will happen ?

$$p^0(1) = 0, \quad p^0(i) = \infty \text{ for } i \neq 1$$

replaced by values on graph \rightarrow

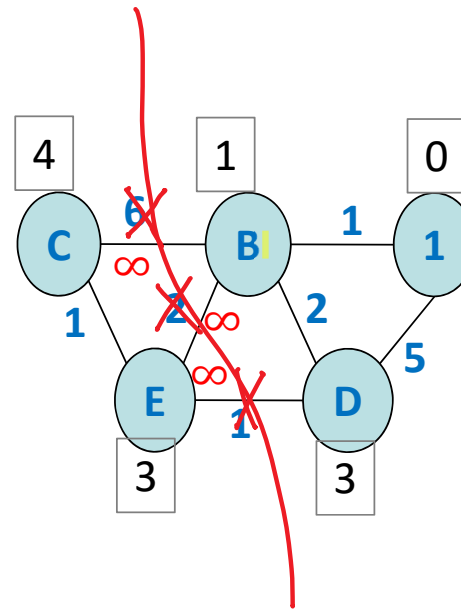
for $k = 1, 2, \dots$ **do**

$$p^k(i) = \min_{j \neq i} [A(i, j) + p^{k-1}(j)] \text{ for } i \neq 1$$

$$p^k(1) = 0$$

until $p^k = p^{k-1}$

return(p^k)



- The algorithm converges to the correct distances
- The algorithm does not converge
- The algorithm converges but not to the correct distances
- I don't know

Solution

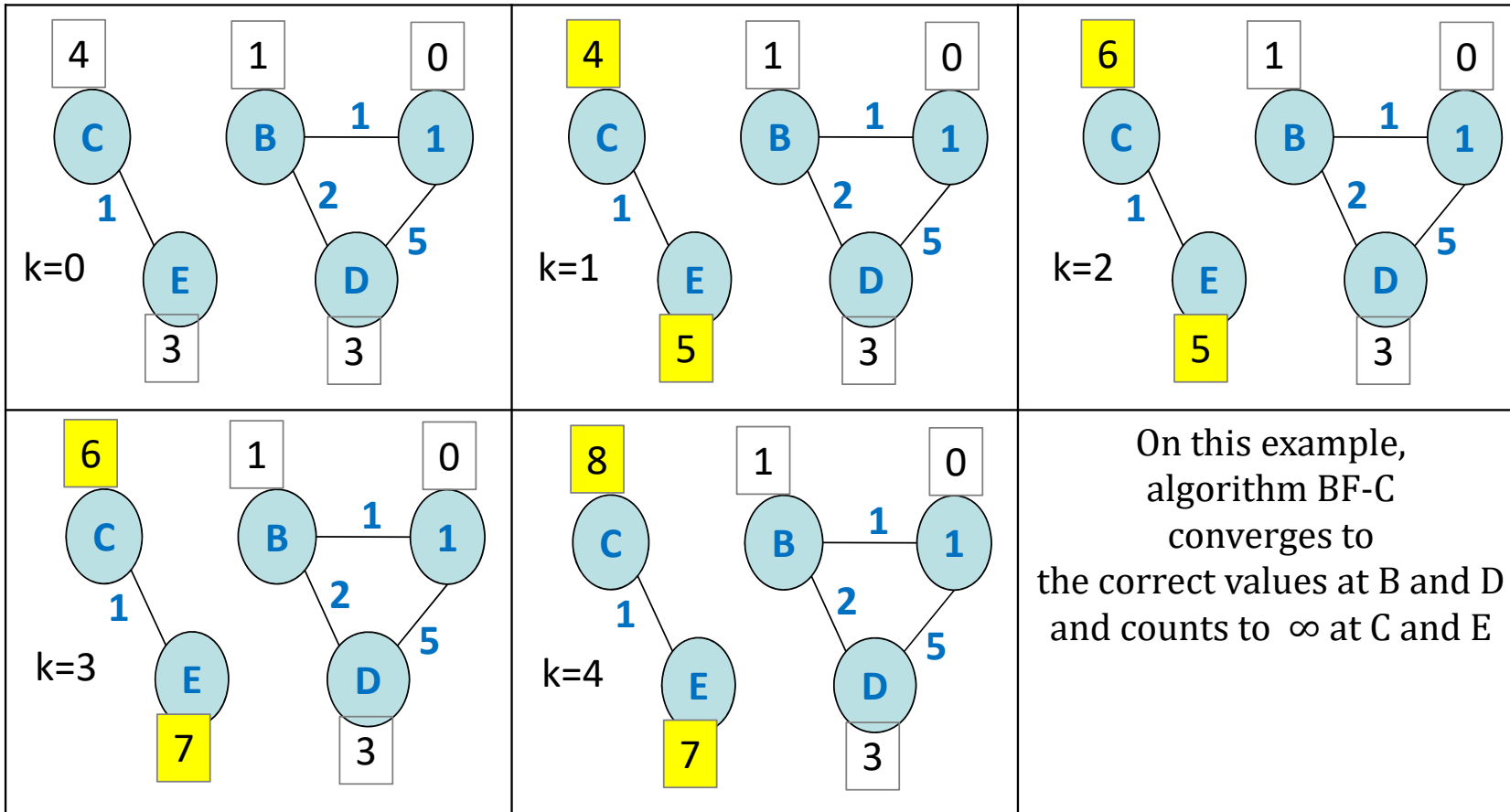
Answer B.

```

 $p^0(1) = 0, p^0(i) = \infty$  for  $i \neq 1$ 

for  $k = 1, 2, \dots$  do
     $p^k(i) = \min_{j \neq i} [A(i, j) + p^{k-1}(j)]$  for  $i \neq 1$ 
     $p^k(1) = 0$ 
until  $p^k = p^{k-1}$ 
return( $p^k$ )
    
```

BF-C



Correctness of BF-C with arbitrary initial condition and arbitrary network

Algorithm BF-C

input: a directed graph with links costs $A(i,j)$; assume $A(i,j) > 0$ and $A(i,j) = \infty$ when nodes i and j are not connected.

output: vector p s.t. $p(i)$ = cost of best path from node i to node 1

~~$p^0(1) = 0, p^0(i) = \infty$ for $i \neq 1$~~
 $p^0(i) = \text{any nonnegative value}$

for $k = 1, 2, \dots$ **do**

$$p^k(i) = \min_{j \neq i} [A(i,j) + p^{k-1}(j)] \text{ for } i \neq 1$$

$$p^k(1) = 0$$

until $p^k = p^{k-1}$

return(p^k)

Theorem 2

1. If there is a path from i to 1, then for k large enough $p^k(i) = p(i) \forall i$ where $p(i)$ is the distance from i to 1. It follows that if the network is fully connected, BF-C with arbitrary initial conditions terminates and eventually computes the correct distances. However, in general, when k is small $p^k(i)$ is *not* the distance from i to 1 in $\leq k$ hops.

2. If there is no path from i to 1 then $\lim_{k \rightarrow \infty} p^k(i) = \infty$. It follows that if the network is not fully connected, BF-C does not terminate ("count to infinity").

Proof

We do the proof assuming all nodes are connected.

1. Let p^k be the vector $p^k[i]$, $i=2, \dots$. Let B be the mapping that transforms an array $x[i]_{i=2, \dots}$ into the array Bx defined for $i \neq 1$ by

$$Bx[i] = \min_{j \neq i, j \neq 1} [A(i,j) + x(j)]$$

Let b be the array defined for $i \neq 1$ by

$$b[i] = A(i,1)$$

The algorithm can be rewritten in vector form as

$$(1) p^k = B p^{k-1} \wedge b$$

where \wedge is the pointwise minimum

2. Eq (1) is a min-plus linear equation and the operator B satisfies $B(x \wedge y) = Bx \wedge By$. Thus, Eq(1) can be solved using min-plus algebra into

$$(2) p^k = B^k p^0 \wedge B^{k-1} b \wedge \dots \wedge B b \wedge b$$

3. Define the array e for $i \neq 1$ by $e[i] = \infty$. Let $p^0 = e$. Eq (2) becomes

$$(3) p^k = B^{k-1} b \wedge \dots \wedge B b \wedge b. \text{ Now we have the Bellman Ford algorithm with classical initial conditions, thus, by Theorem 1:}$$

$$(4) \text{ for } k \geq n-1: B^{k-1} b \wedge \dots \wedge B b \wedge b = q$$

where $q[i]$ is the distance from i to 1.

4. We can rewrite Eq(2) for $k \geq n-1$ as

$$(5) p^k = B^k p^0 \wedge q$$

5. $B^k p^0[i]$ can be written as $A[i, i_1] + A[i_1, i_2] + \dots + A[i_{k-1}, i_k] + p[i_k]$ thus

(6) $B^k p^0[i] \geq k a$, where a is the minimum of all $A[i, j]$. Thus $B^k p^0[i]$ tends to ∞ when k grows. Thus for k large enough, $B^k p^0$ is larger than q and can be ignored in Eq(5). In other words, for k large enough :

$$(6) p^k = q$$

Distributed Bellman Ford

BF-C can be used to compute $p(i)$ i.e. find the shortest path. However, this is not its main interest, because there is a better algorithm (Dijkstra) that can be used in a centralized way.

But: it can be distributed as follows.

Distributed Bellman-Ford Algorithm , BF-prelim

node i maintains an estimate $q(i)$ of the true distance $p(i)$ to node 1;
node i also keeps a record of latest values $q(j)$ for all neighbors j
initial conditions are arbitrary but $q(1) = 0$ at all steps;

from time to time, i sends its value $q(i)$ to all neighbours

when i receives an updated value $q(j)$ from neighbor j , node i recomputes $q(i)$:

$$eq(1) \quad q(i) \leftarrow \min_{j \text{ neighbor}} (A(i,j) + q(j))$$

$nextHop(i)$ is set to a value of j that achieves the min in eq(1)

This is *asynchronous* message passing, i.e. nodes do not wait for complete rounds of iteration to be performed.

Correctness of BF-prelim

Theorem: Assume at least one message is reliably sent to all neighbors by every node every T time units and the network is fully connected. BF-prelim converges to the same result as the centralized version BF-C (i.e. to the correct distances) in $\leq mT$ time units, where m is the number of steps performed by BF-C with same initial conditions.

Distributed Bellman-Ford Algorithm , BF-prelim

node i maintains an estimate $q(i)$ of the true distance $p(i)$ to node 1;
node i also keeps a record of latest values $q(j)$ for all neighbors j
initial conditions are arbitrary but $q(1) = 0$ at all steps;

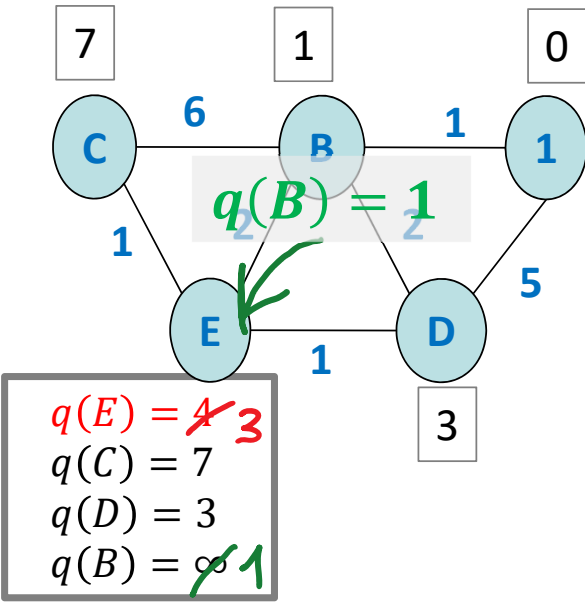
from time to time, i sends its value $q(i)$ to all neighbours

when i receives an updated value $q(j)$ from neighbor j , node i recomputes $q(i)$:

$$eq (1) \quad q(i) \leftarrow \min_{j \text{ neighbour}} (A(i, j) + q(j))$$

$nextHop(i)$ is set to a value of j that achieves the min in eq(1)

Distributed Bellman-Ford BF-prelim



A possible run of algorithm BF-prelim. The table shows the successive values of $q(i)$

$q(i)$	1	B	C	D	E
	0	∞	∞	∞	∞
1 -> B	0	1	∞	∞	∞
B -> D	0	1	∞	3	∞
B -> C	0	1	7	3	∞
C -> E	0	1	7	3	8
D -> E	0	1	7	3	4
1 -> D	0	1	7	3	4
B -> E	0	1	7	3	3
E -> B	0	1	7	3	3
E -> C	0	1	4	3	3

$$q(E) \leftarrow \min(1 + 2, 7 + 1, 3 + 1) = 3$$

Converged !

Super-Duper Bellman-Ford

BF-prelim requires a node to remember all estimates $q(j)$ received from all neighbors j , even for j not on the shortest path to destination; can we do better ?

A possible modification would be to replace eq(1) by

Super-Duper Bellman-Ford:

when i receives an updated value $q(j)$ from neighbor j , node i recomputes $q(i)$:

$$eq(1s) \quad q(i) \leftarrow \min \{A(i,j) + q(j), q(i)\}$$

Node i now needs to store only its own estimate $q(i)$

Q. does this work ?

Is Super-Duper Bellman-Ford correct ?

when i receives an updated value $q(j)$ from neighbor j , node i recomputes $q(i)$:

$$eq(1s) \quad q(i) \leftarrow \min \{A(i,j) + q(j), q(i)\}$$

- A. Yes, regardless of initial condition
- B. Only if initial conditions are $q(i) = \infty, i \neq 0$ and $q(1) = 0$ at time 0
- C. It may fail, even with initial conditions as in 2.
- D. I don't know

Solution

Super-Duper can only decrease $q(i)$. So if we start from initial conditions as in example « Impact of Initial Conditions », the algorithm will not converge to the right value. It gets « stuck » with a low value.

It is possible to show that it works if all initial conditions are above the final values, for example $q(j) = \infty$ and $q(1) = 0$ initially.

But even then, it will not work if there is a topology change, since this is equivalent to starting from different initial conditions. Indeed, requiring that initial conditions are correct means that we are able to reset the entire network whenever there is a topology change (which is not feasible in practice).

Answer B.

Distributed Bellman-Ford

Requires only to remember distance from self to destination + the best neighbor ($\text{nextHop}(i)$)

and works for all initial conditions

Distributed Bellman-Ford Algorithm, BF-D

node i maintains an estimate $q(i)$ of the distance $p(i)$ to node 1;
node i remembers the best neighbor $\text{nextHop}(i)$

initial conditions are arbitrary but $q(1) = 0$ at all steps;

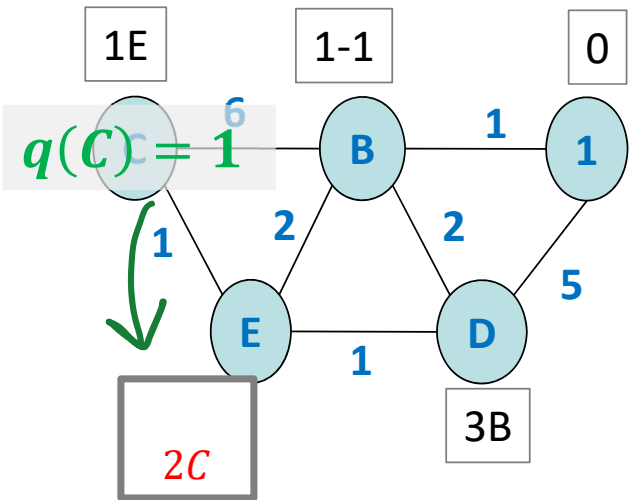
from time to time, i sends its value $q(i)$ to all neighbors

when i receives an updated value $q(j)$ from j , node i recomputes $q(i)$:

eq (2) if $j == \text{nextHop}(i)$
 then $q(i) \leftarrow A(i, j) + q(j)$
 else $q(i) \leftarrow \min(A(i, j) + q(j), q(i))$

if eq(2) causes $q(i)$ to be modified, $\text{nextHop}(i) \leftarrow j$

Distributed Bellman-Ford BF-D

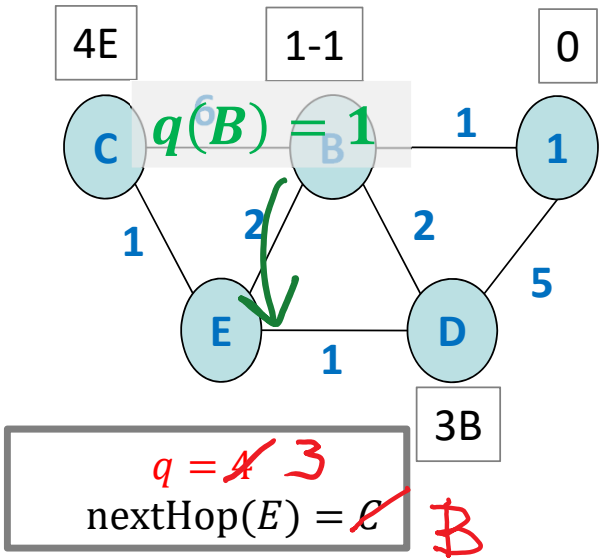


A possible run of algorithm BF-D. The table shows the successive values of $q(i)$ and nextHop

i	1	B	C	D	E
	0	10C	1E	6E	1C
1 -> B	0	1-1	1E	6E	1C
B -> D	0	1-1	1E	3B	1C
B -> C	0	1-1	1E	3B	1C
B -> E	0	1-1	1E	3B	1C
D -> F	0	1-1	1E	3B	1C
C -> E	0	1-1	1E	3B	2C

$$q(E) \leftarrow 1 + 1 = 2$$

Distributed Bellman-Ford BF-D, cont'd



$q(E) \leftarrow \min(1 + 2, 4) = 3$
 $\text{nextHop}(E) \leftarrow B$

A possible run of algorithm BF-D. The table shows the successive values of $q(i)$

i	1	B	C	D	E
	0	10C	1E	6E	1C
1 -> B	0	1-1	1E	6E	1C
B -> D	0	1-1	1E	3B	1C
B -> C	0	1-1	1E	3B	1C
B -> E	0	1-1	1E	3B	1C
D -> E	0	1-1	1E	3B	1C
C -> E	0	1-1	1E	3B	2C
E -> C	0	1-1	2E	3B	2C
C -> E	0	1-1	2E	3B	3C
E -> C	0	1-1	3E	3B	3C
C -> E	0	1-1	3E	3B	4C
E -> C	0	1-1	4E	3B	4C
B -> E	0	1-1	4E	3B	3B

Correctness of BF-D

Theorem: Assume at least one message is reliably sent to all neighbors by every node every T time units and the network is fully connected. BF-D converges to the same result as the centralized version BF-C (i.e. to the correct distances) in $\leq mT$ time units, where m is the number of steps performed by BF-C with same initial conditions.

Comment: The main difference with BF-prelim is that eq(2) replaces eq(1). Assume we use BF-D, and we start from a condition such that $q(i)$ is indeed equal to the minimum given by eq (1) (which is what, intuitively, is true most of the time).

When j is not equal to $\text{nextHop}(i)$, both eq(1) and eq(2) have the same effect: the new value of $q(i)$ is the same in both cases. In contrast, if $j == \text{nextHop}(i)$, then eq (2) sets $q(i)$ to the new value $A(i,j)+q(j)$, whereas eq(1) sets it to $\min_{j \text{ neighbor}} (A(i,j)+q(j))$. Eq(2) provides an upper bound on eq(1), in this case. It turns out that the algorithm still works, by the same mechanism that makes the algorithm work even when the initial conditions are arbitrary. Indeed, node i will send its new value to all remaining neighbors, who will in turn do an update and eventually, node i will receive values of $q(j)$ that will correct the problem. In other words, if the new value of $q(i)$ is too high (compared to what would be obtained with eq (1)), this is repaired in one round of exchanges with the neighbors.

Recap

Bellman-Ford, Distributed (BF-D), allows a node to compute distances to one destination

Stored information is:

- Next hop to destination

- Distance to destination

BF-D works by message passing: every node informs its neighbors of its current distance

BF-D works even when initial conditions are not as expected (which occurs due to topology changes)

If destination is unreachable (distance = ∞) then the algorithm counts to infinity.

2. Distance Vector routing in practice: RIP

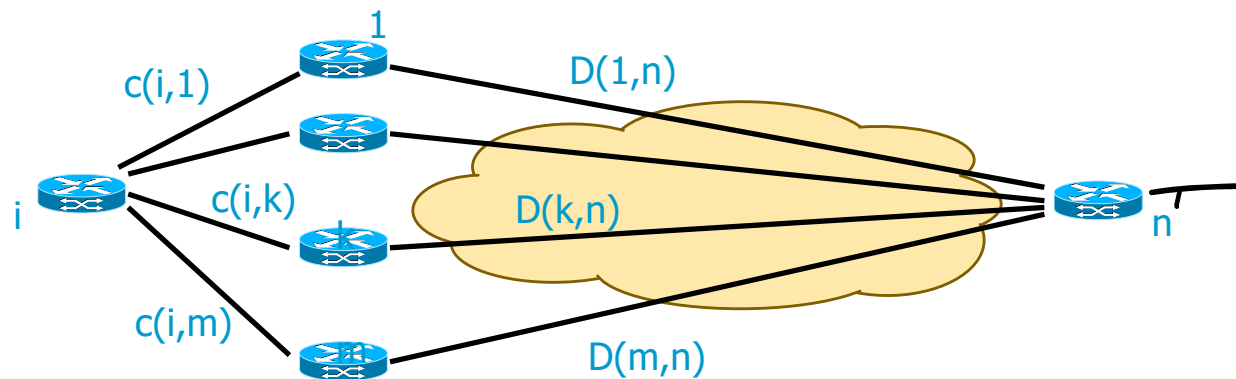
Distance vector routing in RIP = based on BF-D

BF-D is run in all routers *for all possible destination prefixes*

Router i computes shortest path and next hop for all network prefixes n that it heard of.

Initially: $D(i,n) = 1$ if i directly connected to n and implicitly $D(i,n) = +\infty$ for any n that was never heard of.

Node i receives from neighbour k latest values of $D(k,n)$ for all n (this is the **distance vector**). Node i computes the best estimates according to algorithm BF-D

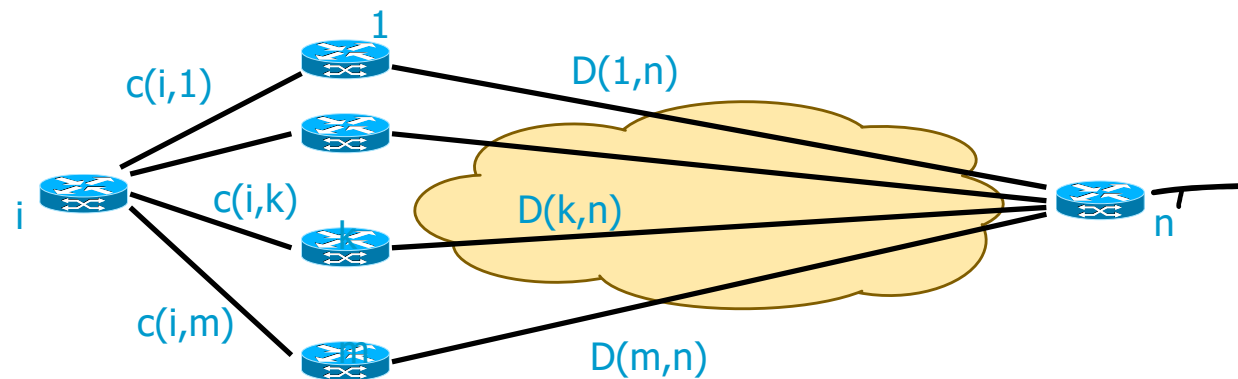


Periodic Updates

Updates are sent periodically (e.g. every 30 sec)

If neighbour k is no longer present, node i will no longer receive hello messages, and after a timeout, this has the same effect as if node i would receive the message from k : $D(k, n) = \infty$ for all n . Then algorithm BF-D is run:

It follows that if k was the next hop to n , then i declares n unreachable.



Example 1

All link costs = 1

A

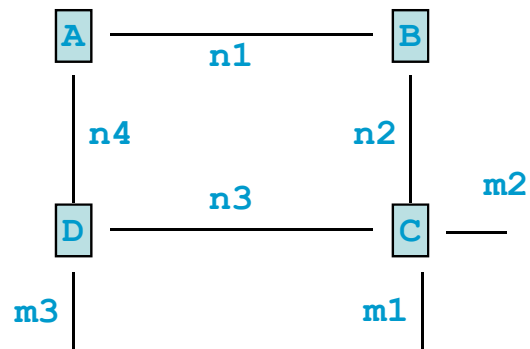
net	dist	nxt
n1	1	n1,A
n4	1	n4,A

B

net	dist	nxt
n1	1	n1,B
n2	1	n2,B

D

net	dist	nxt
n3	1	n3,D
n4	1	n4,D
m3	1	m3,D



C

net	dist	nxt
n2	1	n2,C
n3	1	n3,C
m1	1	m1,C
m2	1	m2,C

Example 1

All link costs = 1

A

net	dist	nxt
n1	1	n1,A
n4	1	n4,A

B

net	dist	nxt
n1	1	n1,B
n2	1	n2,B
n4	2	n1,A

from A
n1 1
n4 1



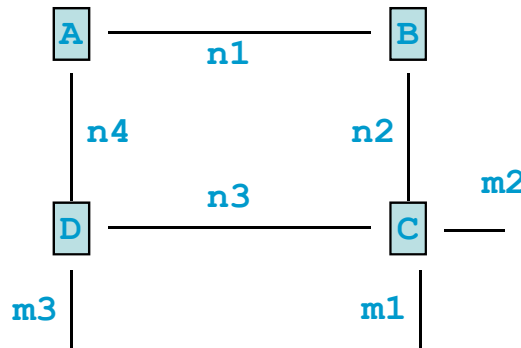
D

net	dist	nxt
n3	1	n3,D
n4	1	n4,D
m3	1	m3,D

C

net	dist	nxt
n2	1	n2,C
n3	1	n3,C
m1	1	m1,C
m2	1	m2,C
n4	2	n3,D
m3	2	n3,D

from D
n3 0
n4 0
m3 0



Example 1

All link costs = 1

A

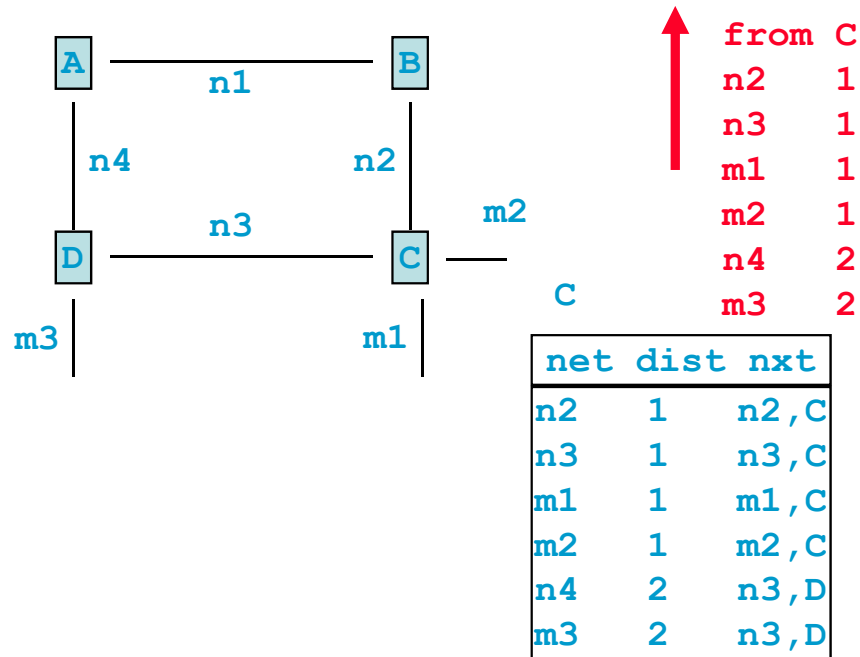
net	dist	nxt
n1	1	n1,A
n4	1	n4,A

D

net	dist	nxt
n3	1	n3,D
n4	1	n4,D
m3	1	m3,D

B

net	dist	nxt
n1	1	n1,B
n2	1	n2,B
n3	2	n2,C
n4	2	n1,A
m1	2	n2,C
m2	2	n2,C
m3	3	n2,C



Example 1 - Final

A

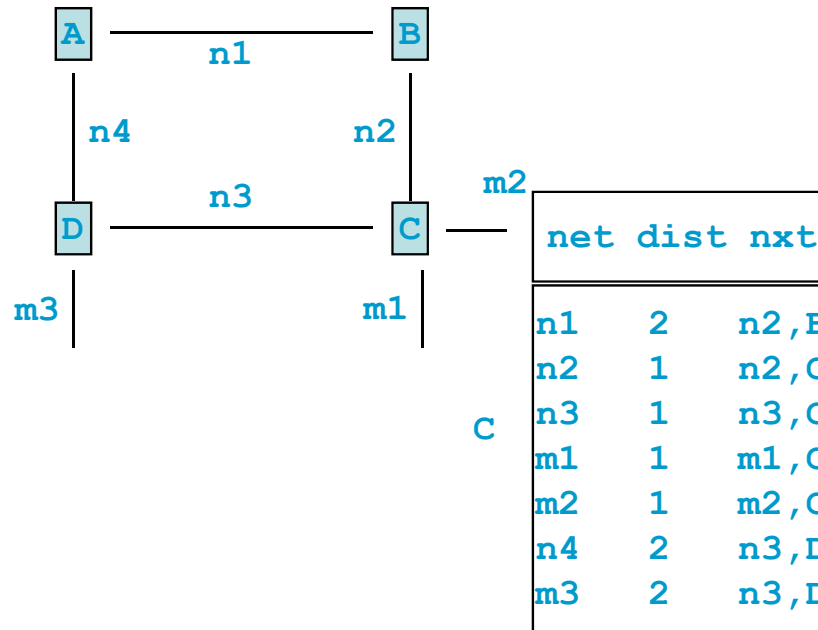
	net	dist	nxt
n1	1		n1,A
n2	2		n1,B
n3	2		n4,D
n4	1		n4,A
m1	3		n4,D
m2	3		n4,D
m3	2		n4,D

B

	net	dist	nxt
n1	1		n1,B
n2	1		n2,B
n3	2		n2,C
n4	2		n1,A
m1	2		n2,C
m2	2		n2,C
m3	3		n2,C

D

	net	dist	nxt
n1	2		n4,A
n2	2		n3,C
n3	1		n3,D
n4	2		n4,D
m1	2		n3,C
m2	2		n3,C
m3	1		m3,D



C

	net	dist	nxt
n1	2		n2,B
n2	1		n2,C
n3	1		n3,C
m1	1		m1,C
m2	1		m2,C
n4	2		n3,D
m3	2		n3,D

Example 1 – Router Failure

All link costs = 1

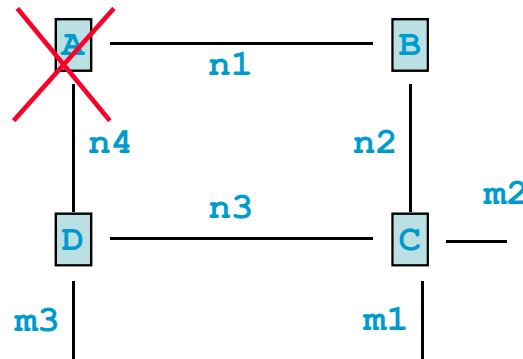
We show only the router in the next hop field

B

net	dist	nxt
n1	1	B
n2	1	B
n3	2	C
n4	2	A
m1	2	C
m2	2	C
m3	3	C

D

net	dist	nxt
n1	2	A
n2	2	C
n3	1	D
n4	1	D
m1	2	C
m2	2	C
m3	1	D



C

net	dist	nxt
n1	2	B
n2	1	C
n3	1	C
m1	1	C
m2	1	C
n4	2	D
m3	2	D

Example 1 - Failure

All link costs = 1

B

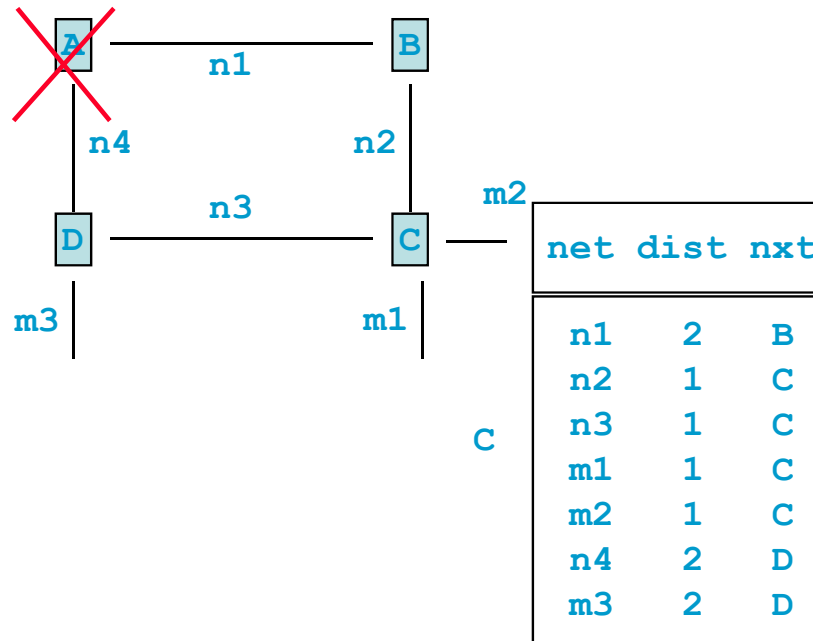
net	dist	nxt
n1	1	B
n2	1	B
n3	2	C
n4	2	A
m1	2	C
m2	2	C
m3	3	C

timeout

D

net	dist	nxt
n1	2	A
n2	2	C
n3	1	D
n4	1	D
m1	2	C
m2	2	C
m3	1	D

timeout

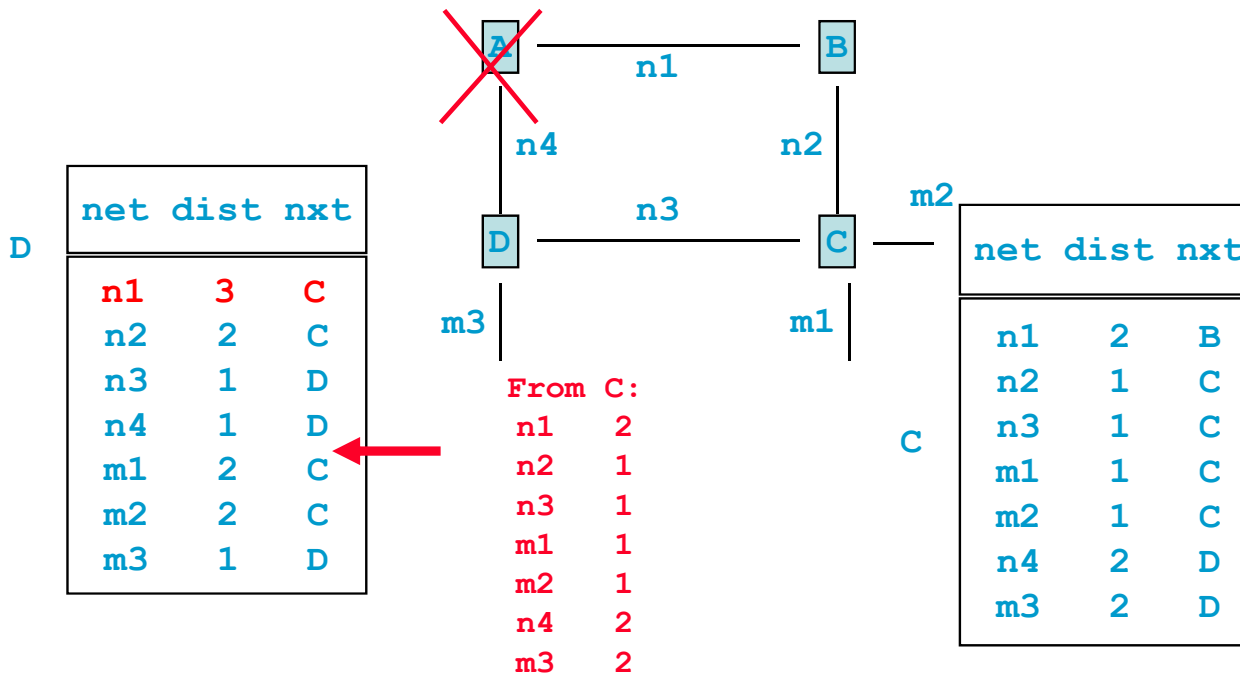


Example 1 - Failure

All link costs = 1

B

net	dist	nxt
n1	1	B
n2	1	B
n3	2	C
m1	2	C
m2	2	C
m3	3	C

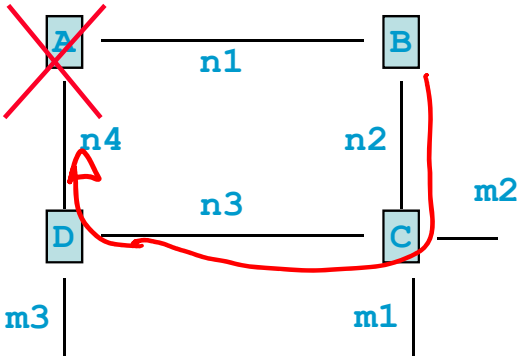


Example 1 - After Failure

All link costs = 1

B

net	dist	nxt
n1	1	B
n2	1	B
n3	2	C
n4	3	C
m1	2	C
m2	2	C
m3	3	C



D

net	dist	nxt
n1	3	C
n2	2	C
n3	1	D
n4	1	D
m1	2	C
m2	2	C
m3	1	D

C

net	dist	nxt
n1	2	B
n2	1	C
n3	1	C
m1	1	C
m2	1	C
n4	2	D
m3	2	D

Example 2 Assume RIP has converged

A All link costs = 1

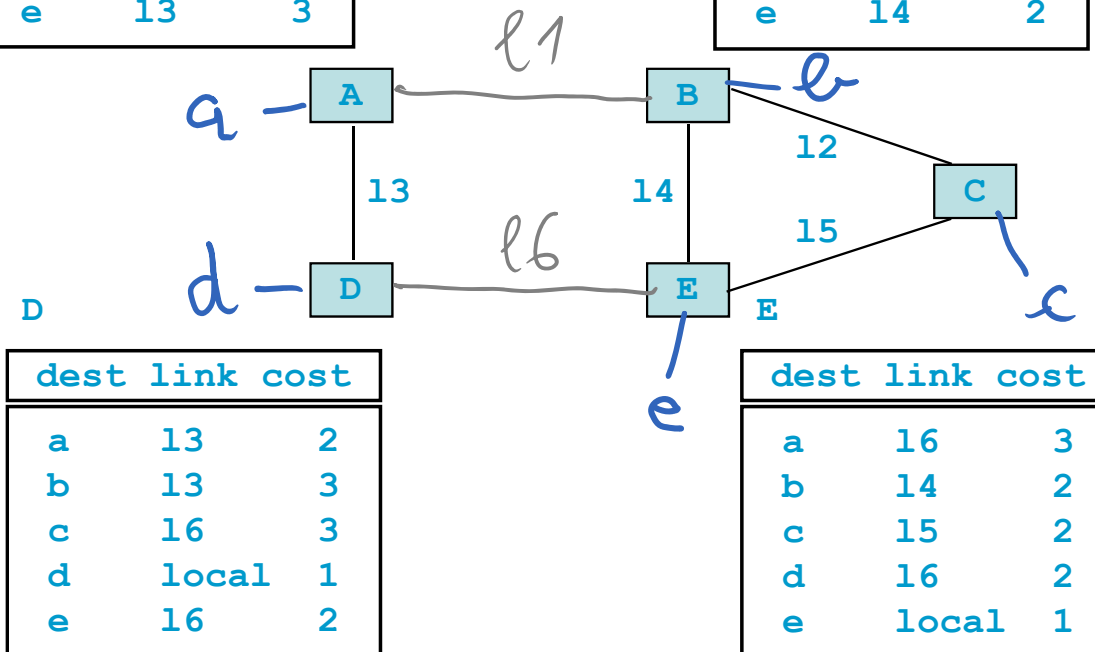
dest link cost		
a	local	1
b	11	2
c	11	3
d	13	2
e	13	3

B

dest link cost		
a	11	2
b	local	1
c	12	2
d	14	3
e	14	2

C

dest link cost		
a	12	3
b	12	2
c	local	1
d	15	3
e	15	2



D

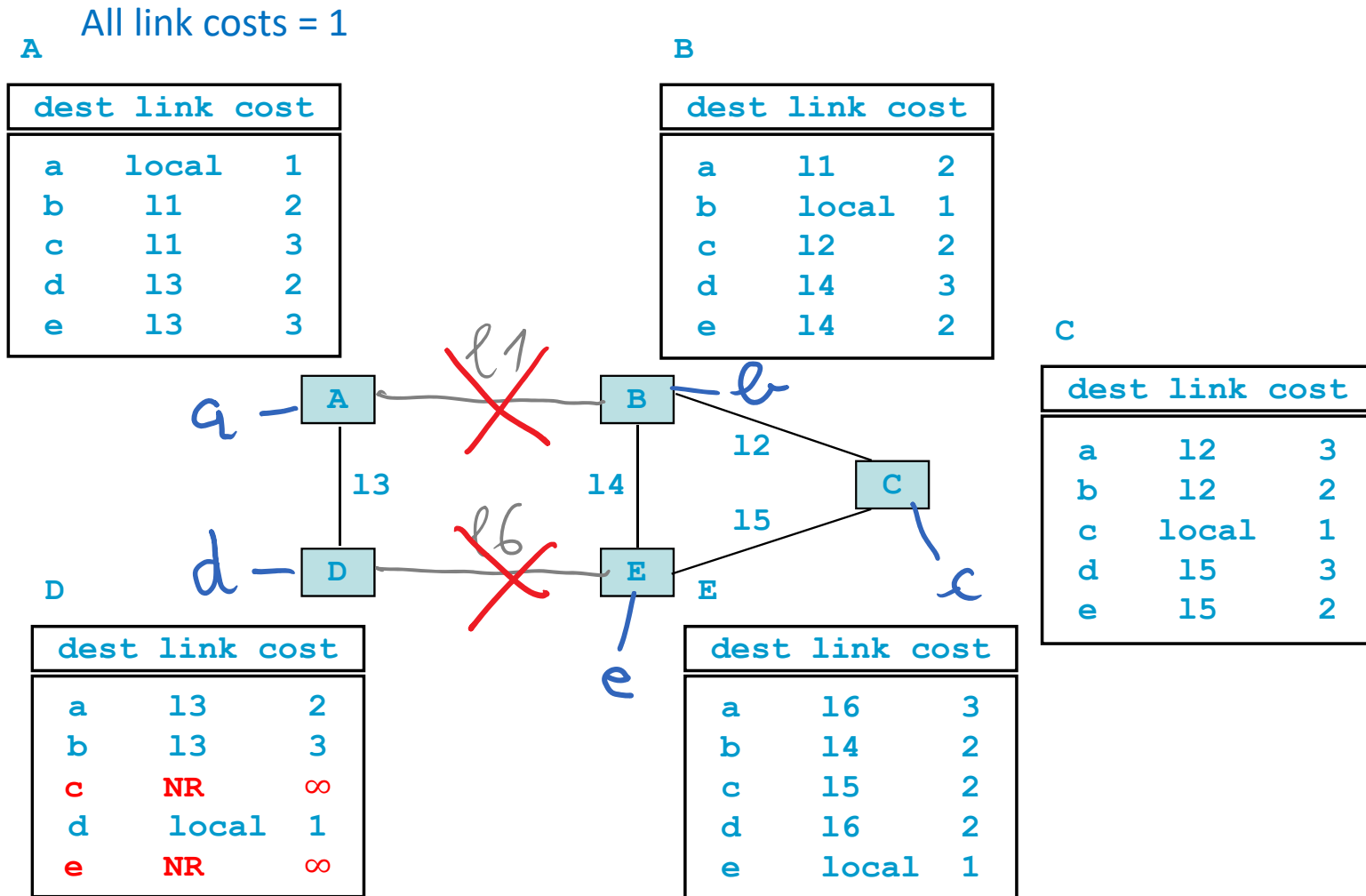
dest link cost		
a	13	2
b	13	3
c	16	3
d	local	1
e	16	2

E

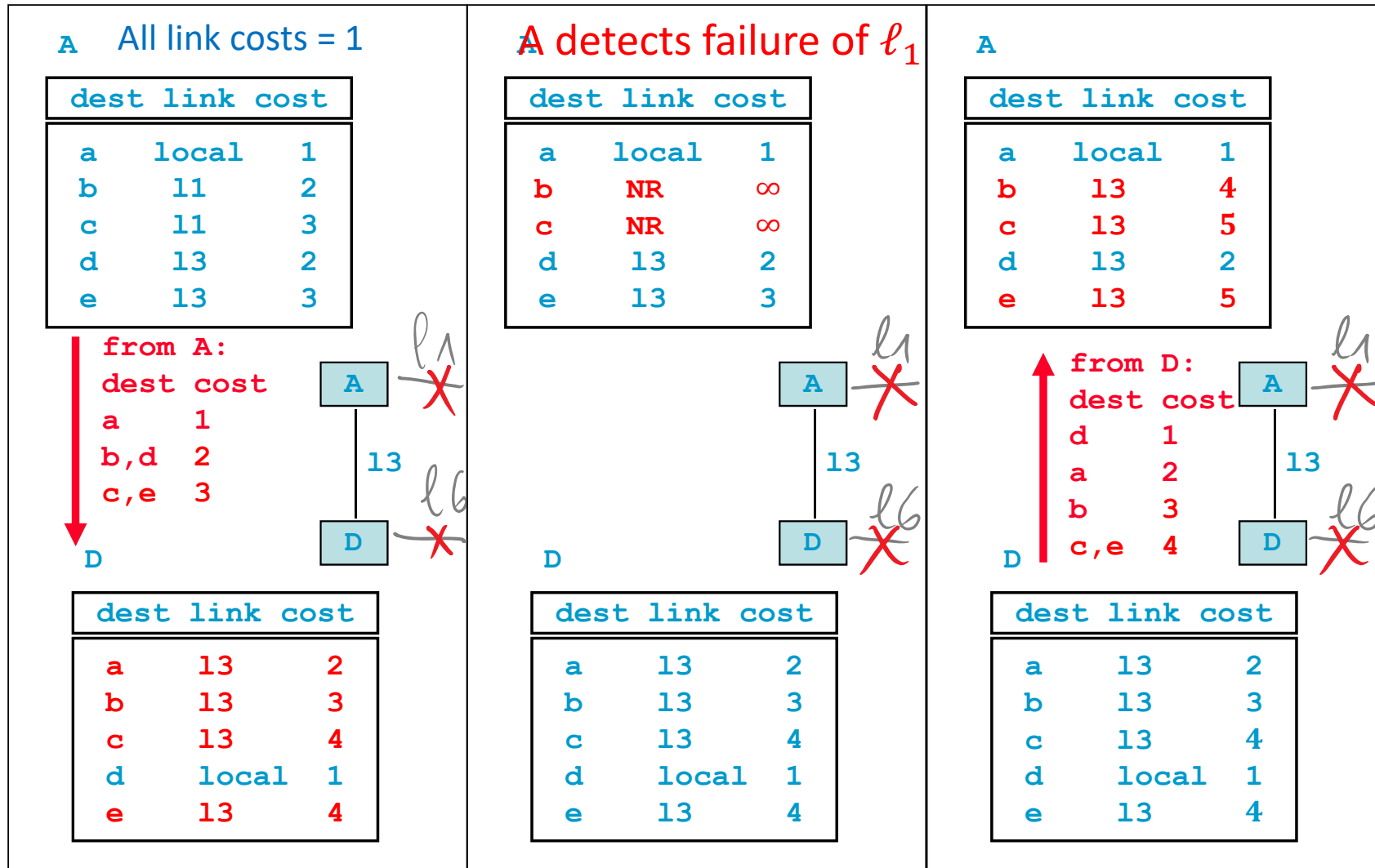
dest link cost		
a	16	3
b	14	2
c	15	2
d	16	2
e	local	1

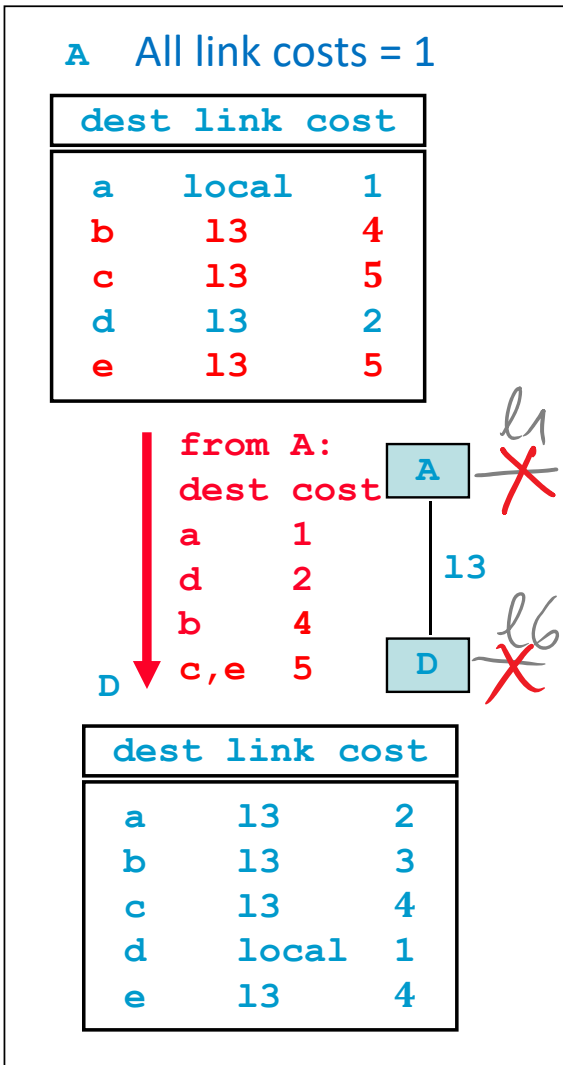
Example 2

Links l1 and l6 fail simultaneously,
D detects failure



Example 2 : Zoom on A and B

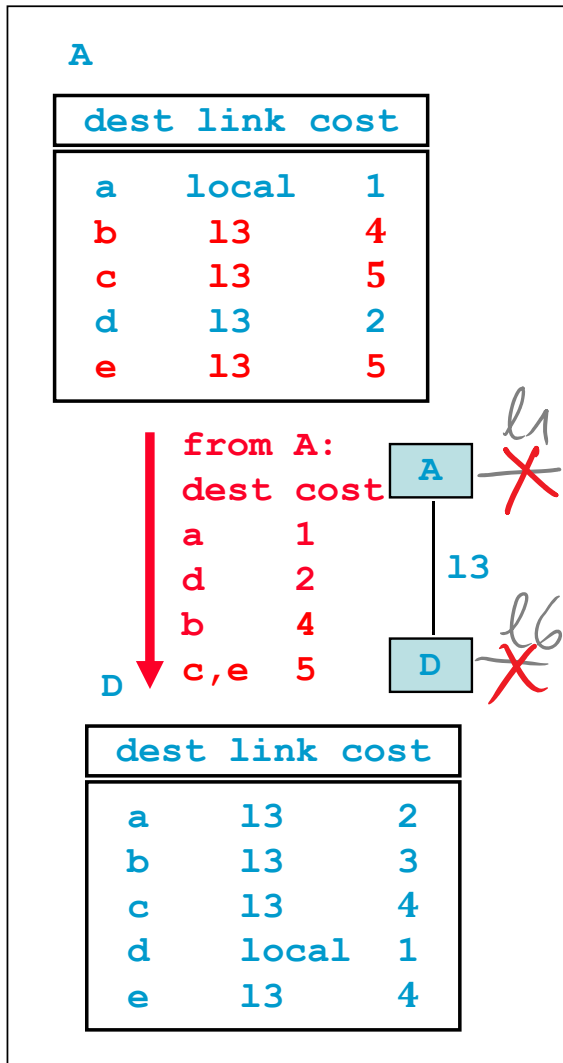




After processing this update, what is the distance at D to e ?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6
- G. I don't know

Solution



Answer F

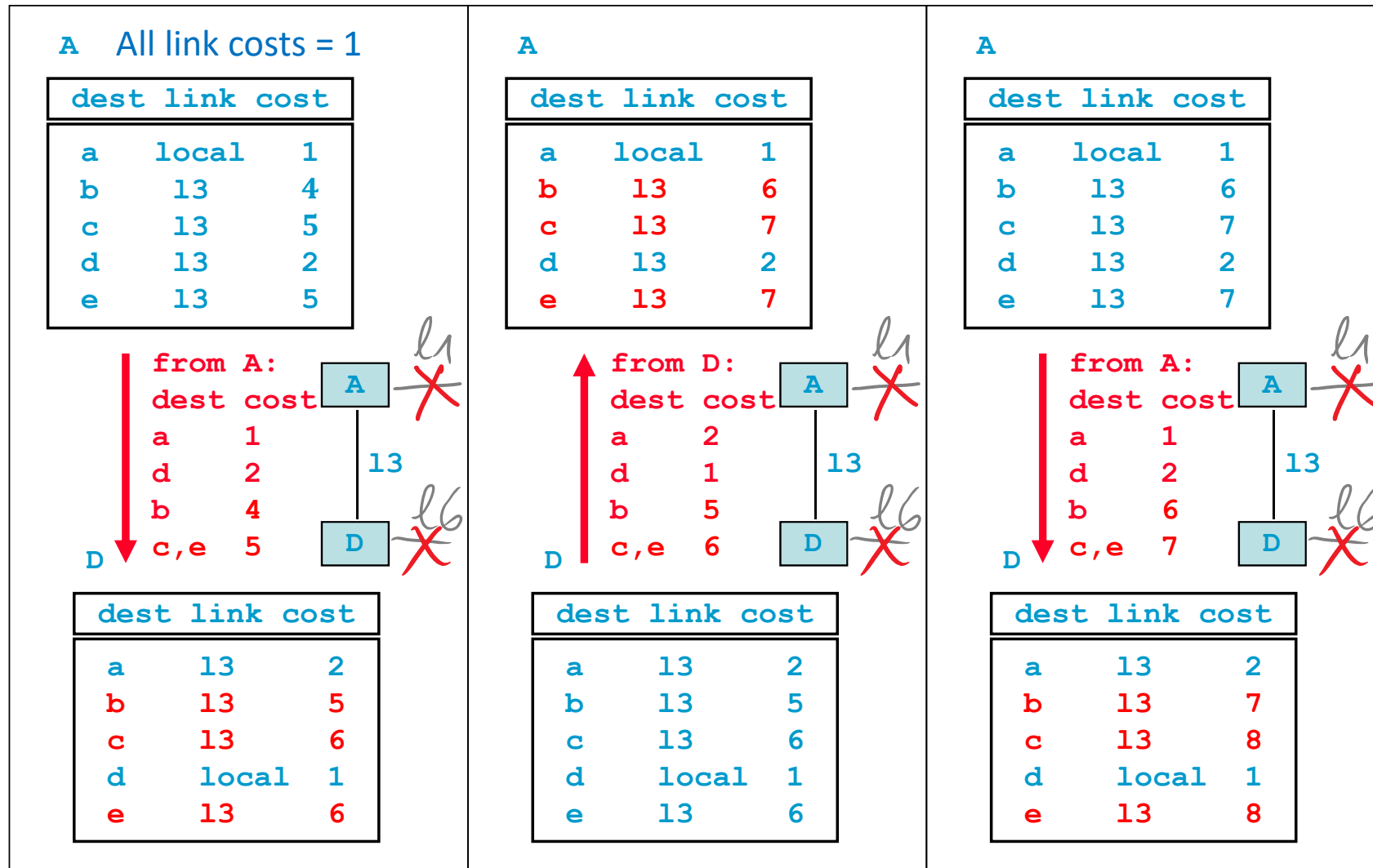
At D:

Distributed Bellman-Ford (BF-D) \Rightarrow
 update is received from next hop to e
 therefore new distance is accepted
 D sets distance to e to $5 + 1 = 6$

when i receives an updated value $q(j)$ from j , node i recomputes $q(i)$:

eq (2) if $j == \text{nextHop}(i)$
 then $q(i) \leftarrow A(i, j) + q(j)$
 else $q(i) \leftarrow \min(A(i, j) + q(j), q(i))$

Count to Infinity



Conclusion from Example 2

The costs to b,c,e grow unbounded “Count to Infinity”

the true costs are infinite

this is the expected behaviour of Bellman-Ford

RIP enforces convergence by setting $\infty = \text{large number} = 16$

Several optimizations (route poisoning, split horizon) exist to accelerate convergence:

RIP optimizations:

Route poisoning and Split Horizon

Two practical heuristics in order to

- Avoid count to infinity

- Reduce occurrence of ping-pong loops

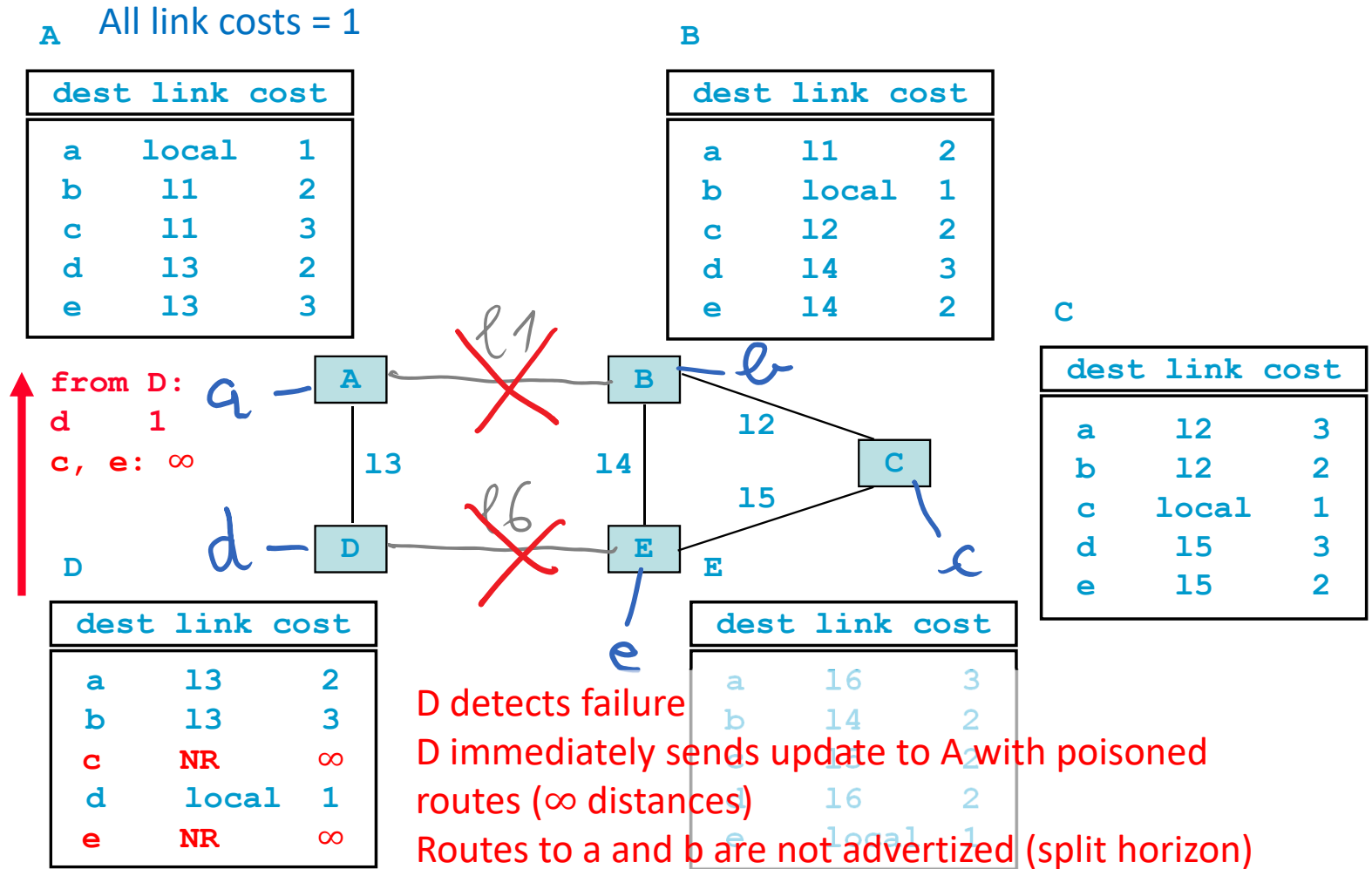
Route poisoning : if A detects that route to x is not reachable, A immediately sends «distance to x = ∞ » to neighbours

Further, when a distance = ∞ is received for x, any further update about x is ignored for some time (holddown timer)

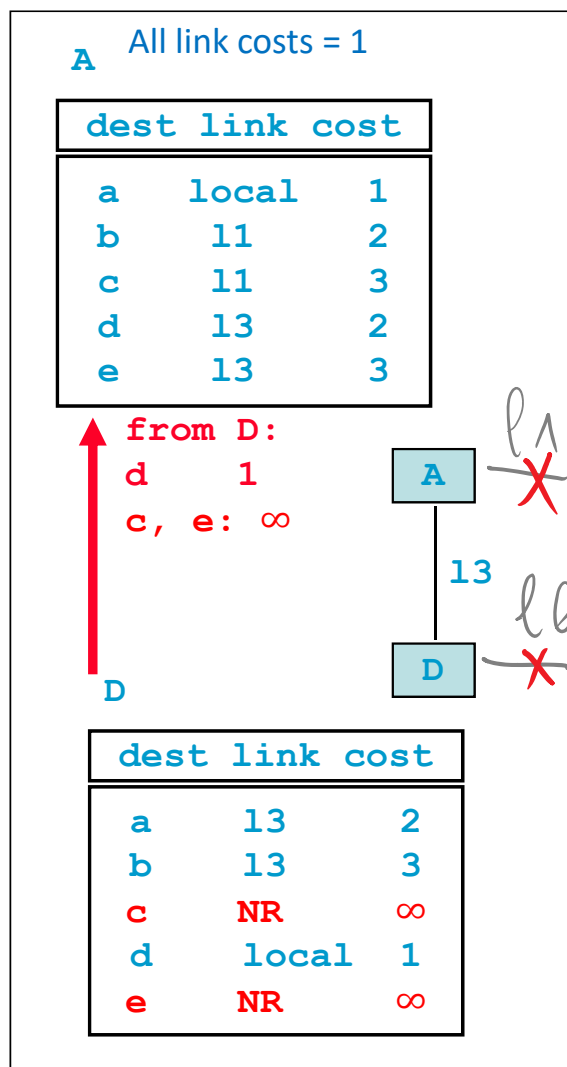
Split Horizon : if A routes packets to x via B, A does not announce this route to B

Example 2

with Route Poisoning and Split Horizon

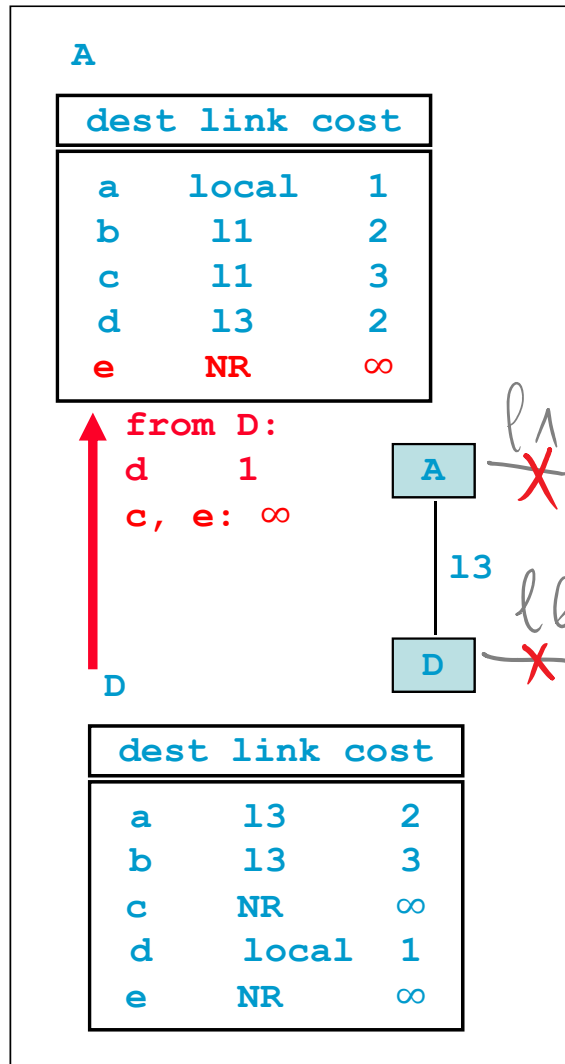


After processing this update, what are the distances at A to c and e ?



- A. $d(c) = 3, d(e) = 3$
- B. $d(c) = \infty, d(e) = 3$
- C. $d(c) = 3, d(e) = \infty$
- D. $d(c) = \infty, d(e) = \infty$
- E. None of the above
- F. I don't know

Solution



Answer C

A applies BF-D:

D is not the next hop to c, therefore A computes

$$\min(3, \infty + 1) = 3$$

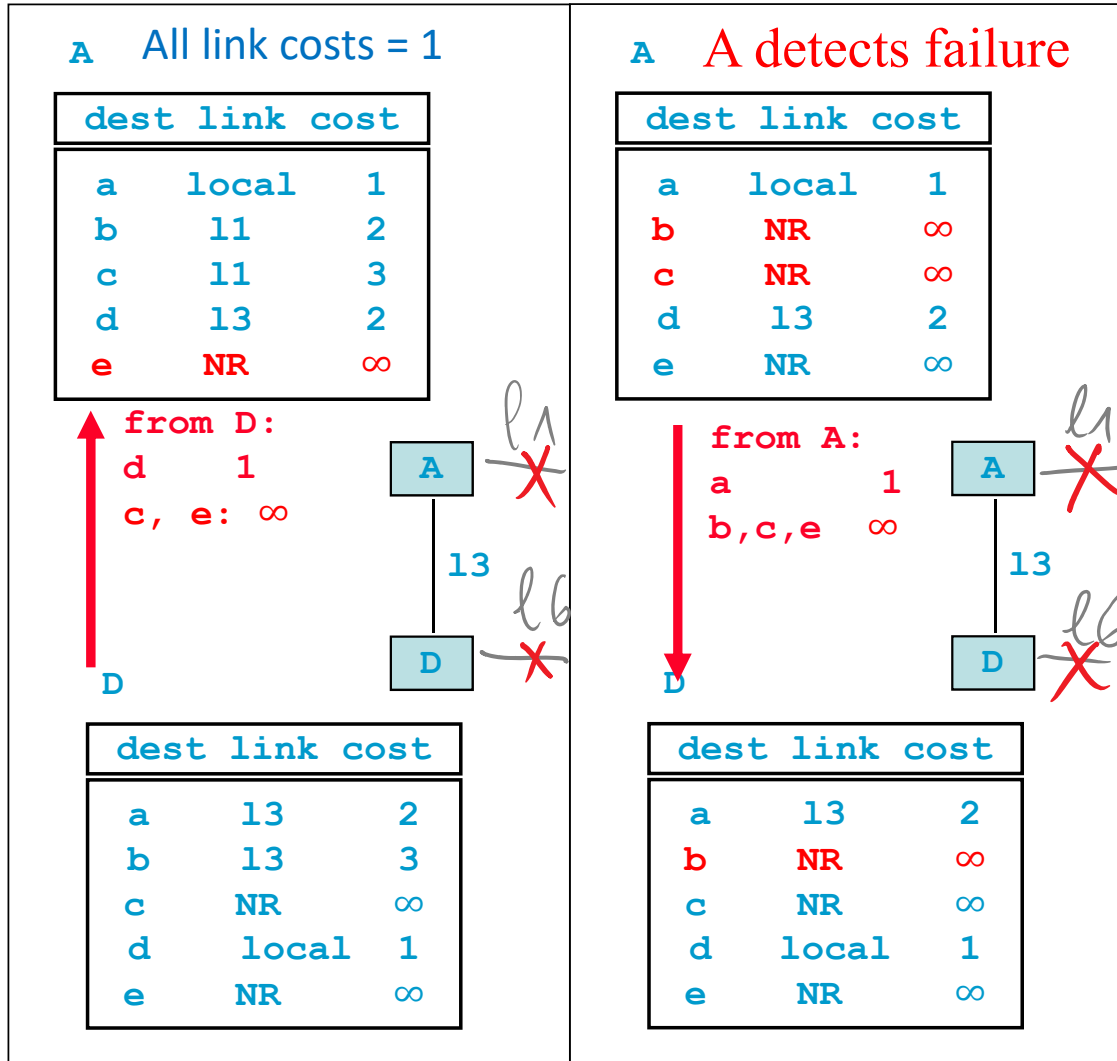
distance is 3, no change

D is the next hop to e, therefore A computes

$$\infty + 1 = \infty$$

distance is ∞, i.e. e is unreachable

Example 2 with Route poisoning and Split Horizon



No count to infinity
 After some time, the ∞ distances are removed by timeout

RIP (Routing Information Protocol)

Implements BF-D

$\infty = 16$; therefore network span limited to 15

Split horizon and route poisoning

UDP port 520 and multicast address 224.0.0.9 / ff02::9

Sends distance vector update every 30 seconds (by default) or when change is detected (“triggered update”)

Route not announced during 3 minutes becomes invalid

Authentication by shared secret

No configuration required (all costs equal to 1).

Other Distance Vector Protocols

EIGRP (Cisco):

- uses Bellman-Ford prelim + additional heuristics to avoid loops (therefore no limit of 15 as in RIP)

- keeps a Routing Information Base (RIB) which is much more than with RIP but less than with OSPF

- also uses dynamic metrics

Babel, AODV add a sequence number put by destination to avoid routing loops and count to infinity (Destination Sequenced Distance Vector).

Spanning Tree Protocol (STP, for bridges) computes the distance to the root using Bellman Ford.

OSPF is link state in one area but uses distance vectors between areas. Count to infinity does not exist here (no area loop: areas must be connected in a logical star topology, centred on area 0).

3. Dynamic Metrics

Does a routing protocol maximize network utility ?

- A. Yes, because it minimizes the cost to destination
- B. Yes if TCP is used because it ensures fairness
- C. No
- D. I don't know

Solution

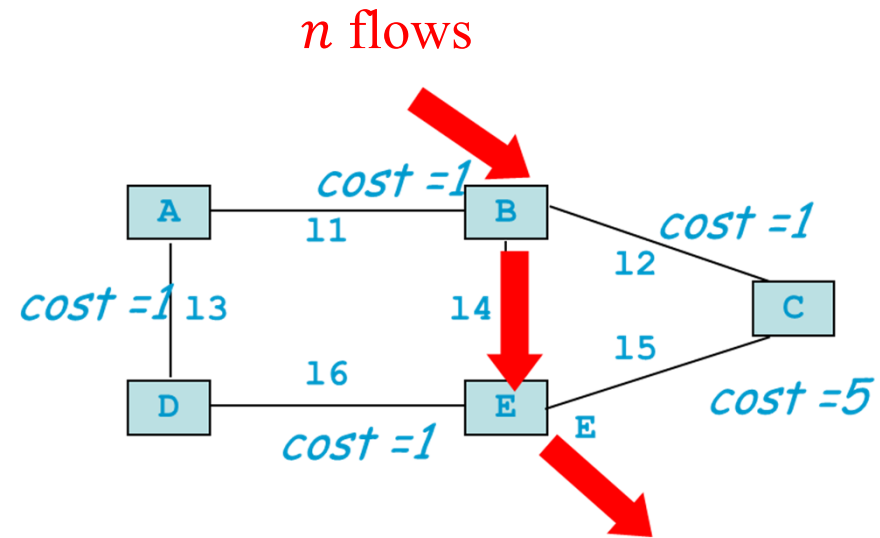
We would need to define the utility.
For example:

$$U = \sum_i \log x_i$$

Assume n flows, all link capacities are 1. Shortest path routing gives rate $1/n$ to all sources.

Deflection routing could deviate $2/3$ of the flows and give $3/n$ to all sources

Answer C



Dynamic Metrics

Some routing protocols use dynamic metrics for improving over shortest path

high load on a link => high cost => link is less used

Example (Cisco)

Cost of a link is

$$m = [K_1 * \text{Trans} + (K_2 * \text{Trans}) / (256 - \text{load}) + K_3 * \text{delay}] * [K_5 / (\text{Reliability} + K_4)]$$

Trans = 10000000/Bandwidth (time to send 10 Kb)

delay = (sum of Delay)/10

default: K1=1, K2=0, K3=1, K4=0, K5=0

But there may be some issues → Braess paradox

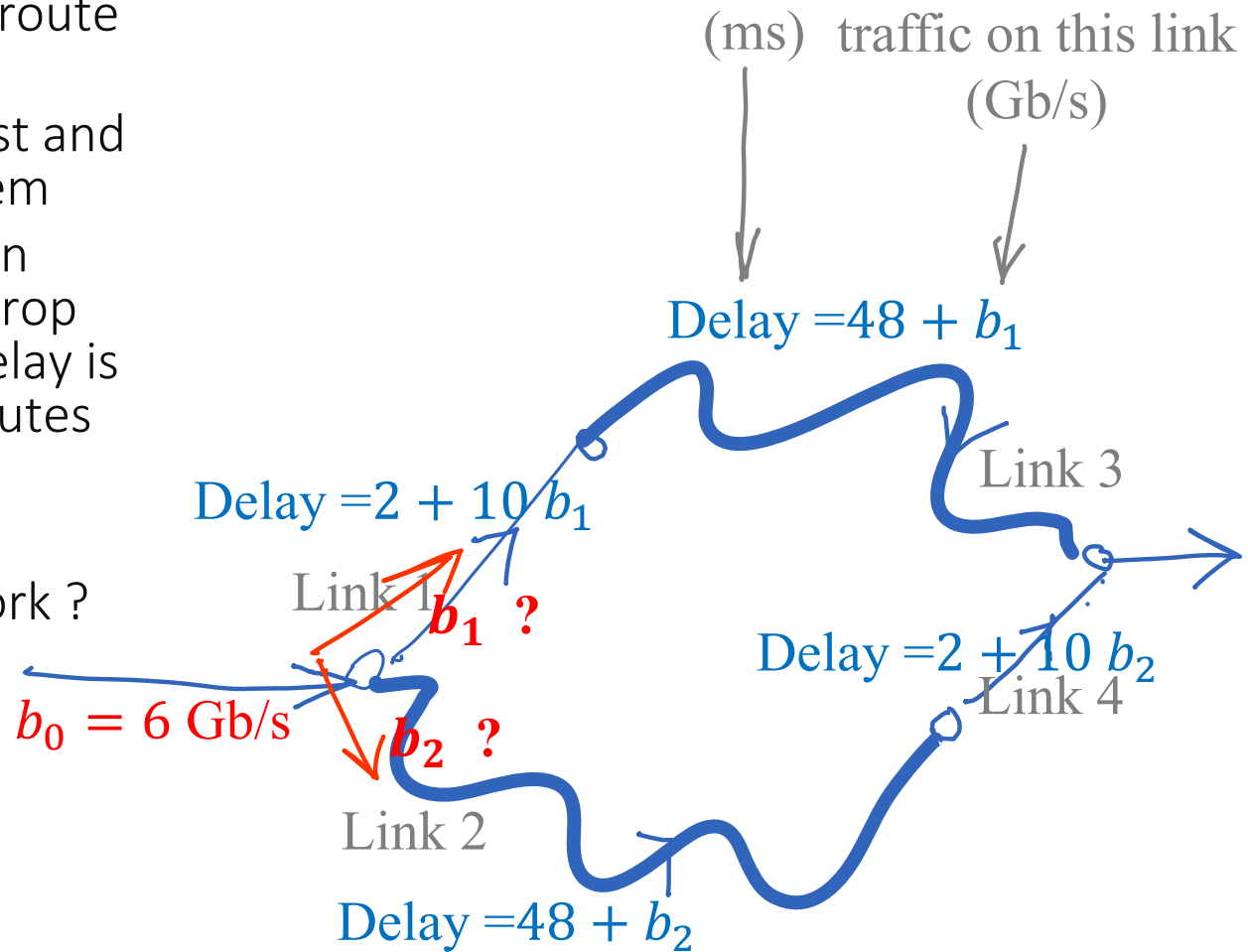
Least Delay Routing and Wardrop Equilibrium

Assume all flows pick the route with shortest delay

Assume parallel paths exist and flows can make use of them

Eventually, there will be an equilibrium (called "Wardrop Equilibrium") such that delay is equal on all competing routes

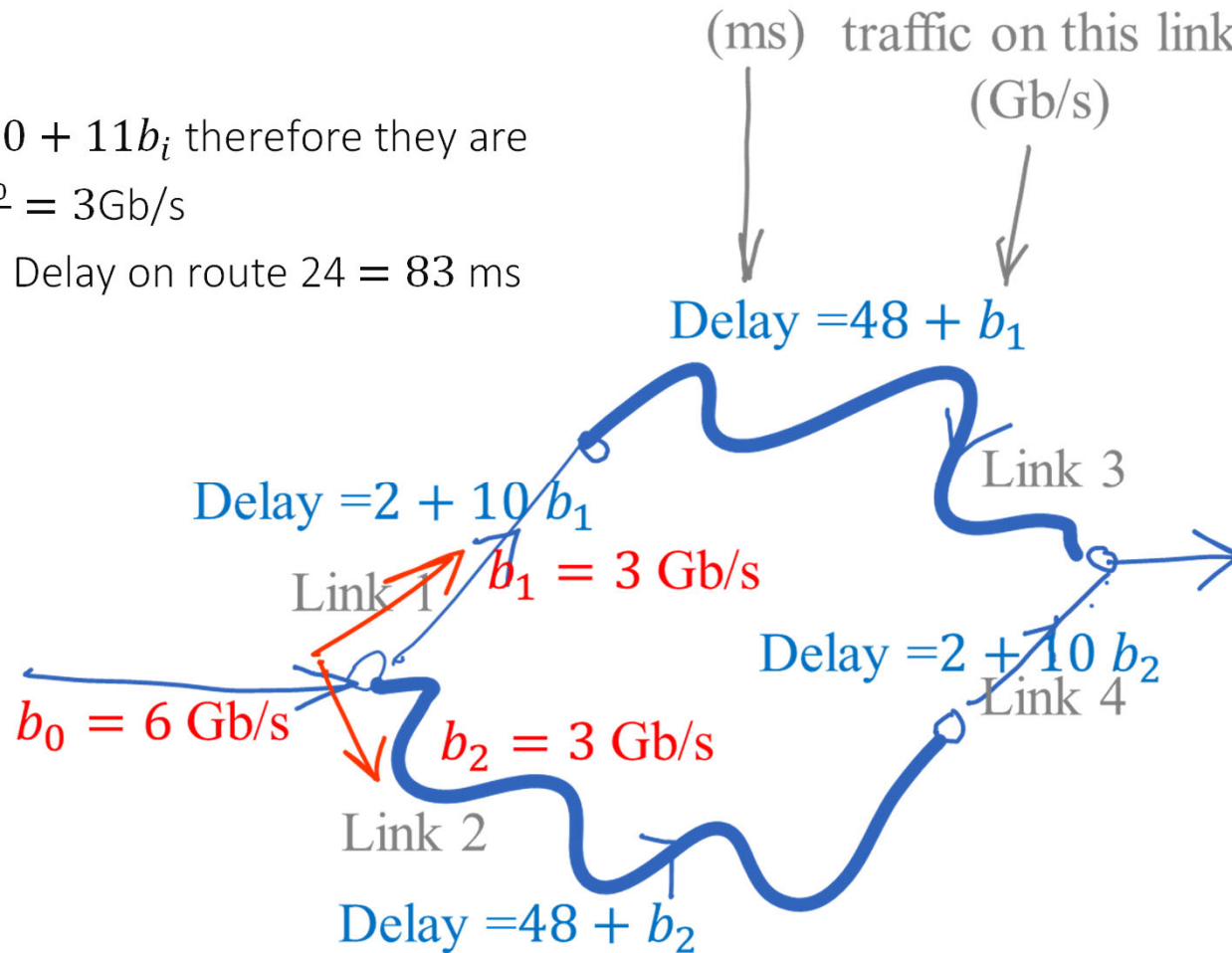
What is the Wardrop equilibrium for this network ?



Solution

Delay on both routes is $50 + 11b_i$ therefore they are equal when $b_1 = b_2 = \frac{b_0}{2} = 3\text{Gb/s}$

Then delay on route 13 = Delay on route 24 = 83 ms



Now introduce link 5

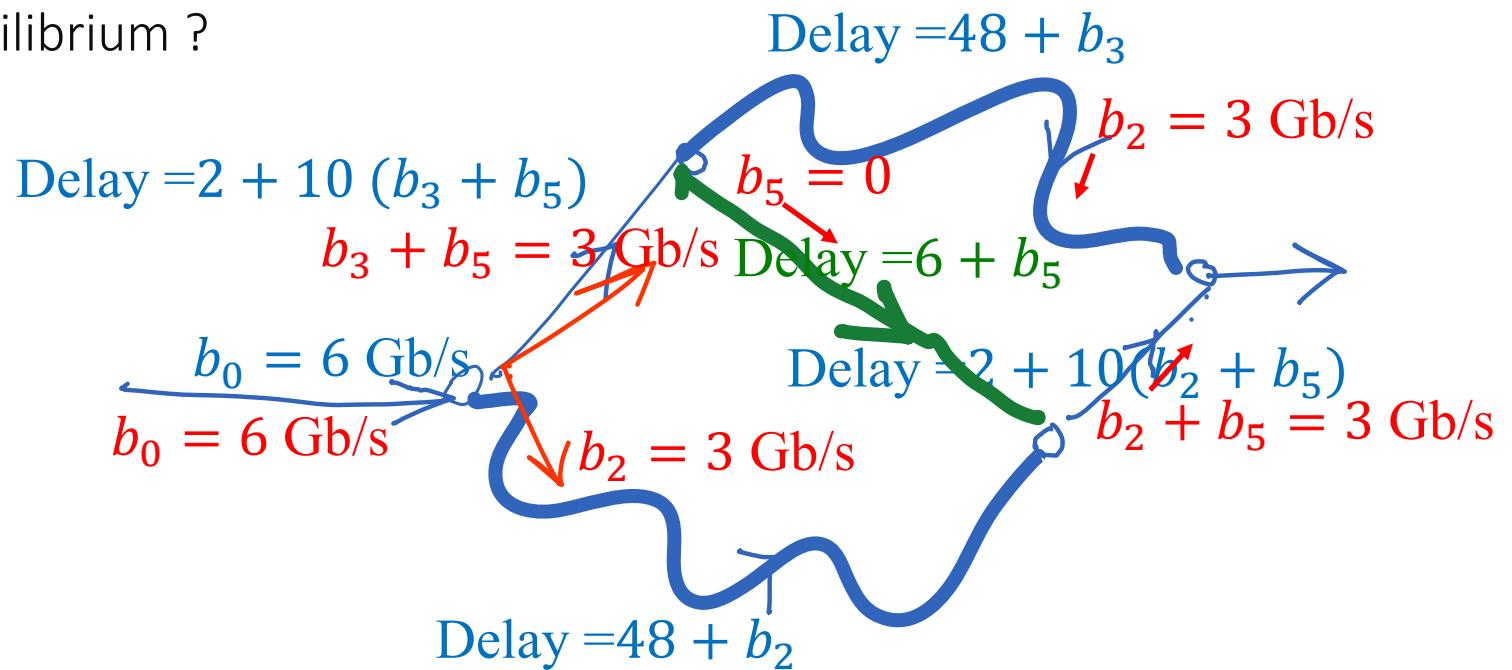
Link 5 has delay function $6 + b_5$ i.e. short delay and high capacity

There are now 3 paths: 13, 154 and 24

Assume we start from previous equilibrium

$$b_1 = b_2 = b_3 = b_4 = 3, b_5 = 0$$

Is this a Wardrop equilibrium ?



Solution

Assume we start from previous equilibrium; $b_1 = b_2 = b_3 = b_4 = 3, b_5 = 0$

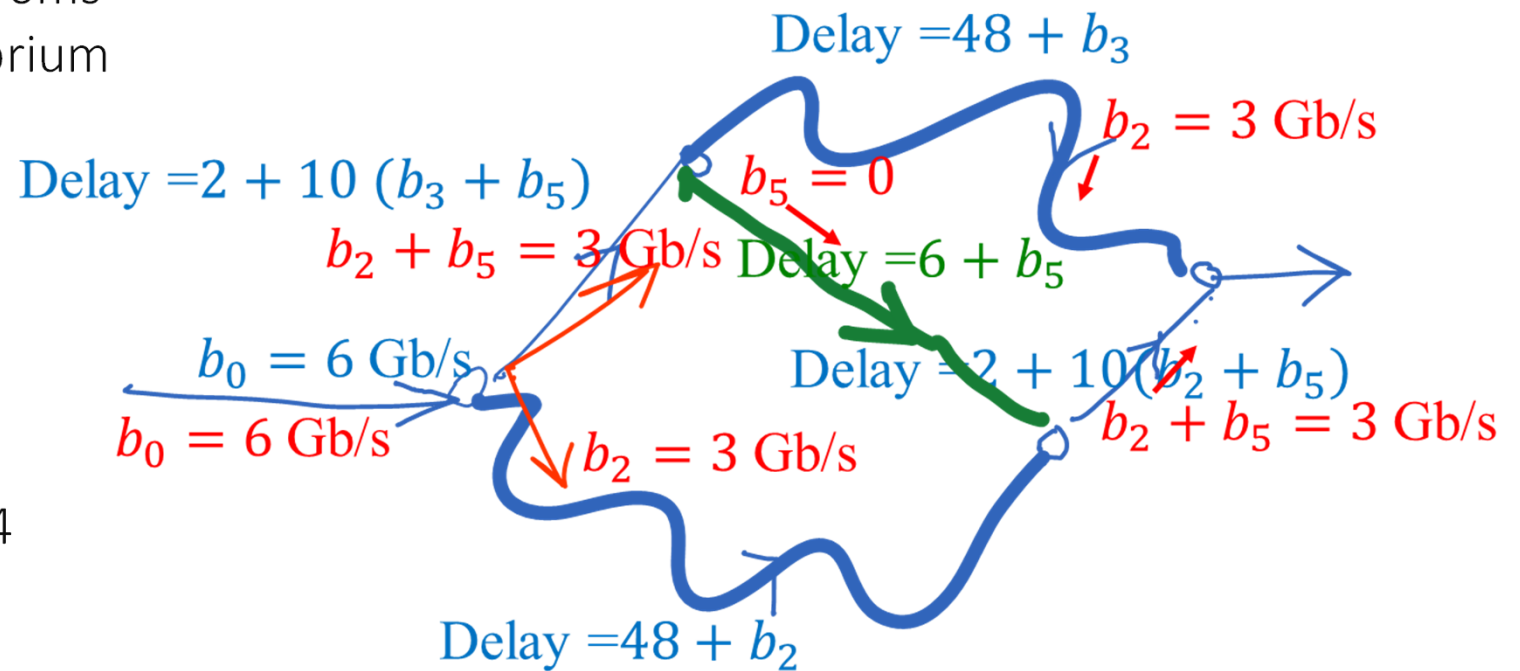
Delays are:

route 13: 83 ms

route 24: 83 ms

route 154: $32+6+32=70$ ms

Not a Wardrop equilibrium



Some traffic
will move to route 154

What is the Wardrop Equilibrium now ?

delay equations

$$50 + 11b_3 + 10b_5 = 50 + 11b_2 + 10b_5$$

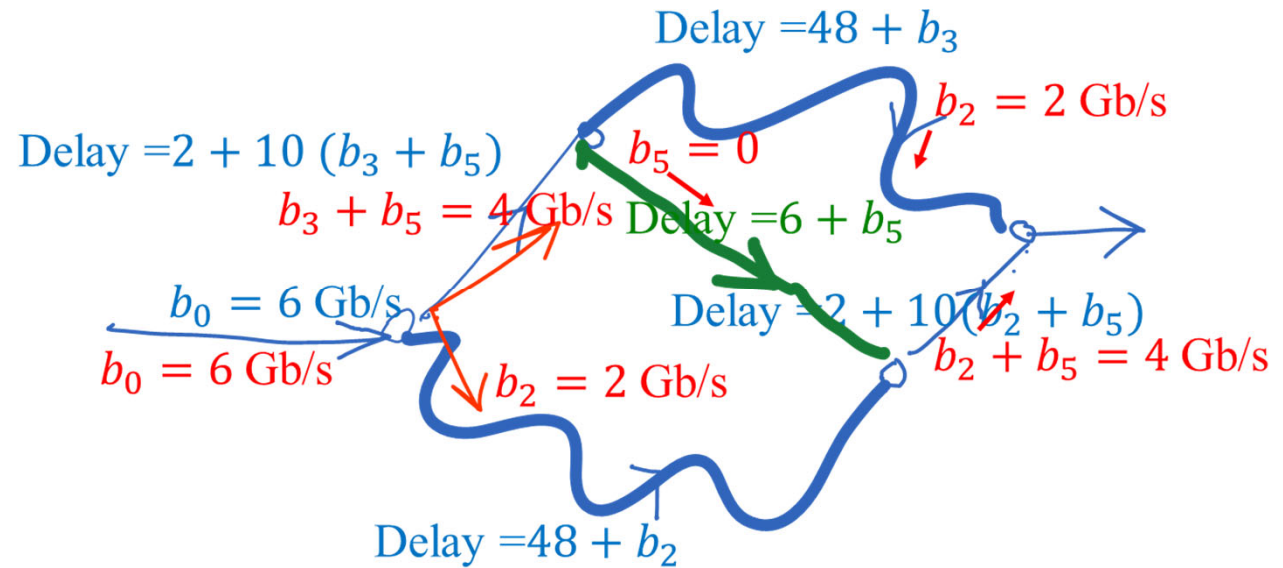
$$= 10 + 10b_3 + 10b_2 + 21b_5$$

total flow

$$b_1 + b_2 + b_3 = 6$$

Solution : $b_3 = b_5 = b_2 = 2$ Gb/s

Delay now is 92 ms on all routes



Braess Paradox

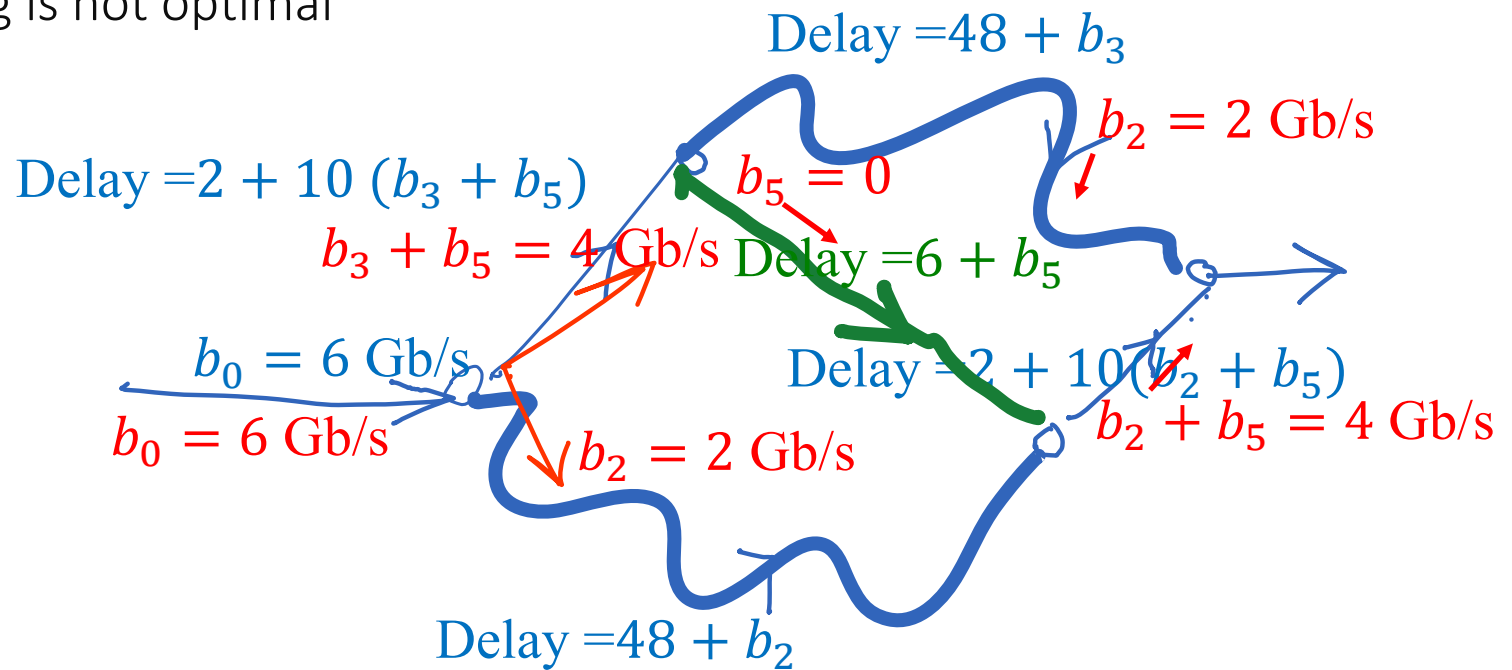
With shortest delay routing:

disable link 5: delay = 83 ms; enable link 5 : delay = 92 ms

Adding capacity made things worse

This is called Braess paradox

Shortest delay routing is not optimal



Optimal Routing

One can change the objective of routing: instead of computing shortest paths, one can solve a global optimization problem maximizing a utility function:

- minimize total delay subject to flow constraints

- this is a well posed optimization problem

- the optimal solution depends on all flows

- but it can be implemented in a distributed algorithm similar to TCP congestion control, see [BertsekasGallager92]

This can be solved using an offline optimization procedure that computes optimal link costs and downloads them into all routers.

Conclusion

Distance vector is simple and smart: Fully distributed, little information stored, little or zero configuration.

But: relatively slow

- Not suited for large and complex networks

- Link State protocols are used instead

Shortest Path routing is better than STP but is generally not optimal. Optimal routing requires another layer of optimization.

Taking link delay as link cost is also not optimal and is subject to Braess paradox.

