Computer Networks

Sándor Laki
ELTE-Ericsson Communication Networks Laboratory
ELTE FI – Department Of Information Systems
[*lakis@elte.hu*](mailto:lakis@elte.hu)
[*http://lakis.web.elte.hu*](http://lakis.web.elte.hu)

Eötvös Loránd
University

Based on the slides of Laurent Vanbever.
Further inspiration: Scott Shenker & Jennifer Rexford & Phillipa Gill

Last week on
Computer Networks

Each layer provides a service to the layer above by using the services of the layer directly below it

Applications

...built on...

Reliable (or unreliable) transport

...built on...

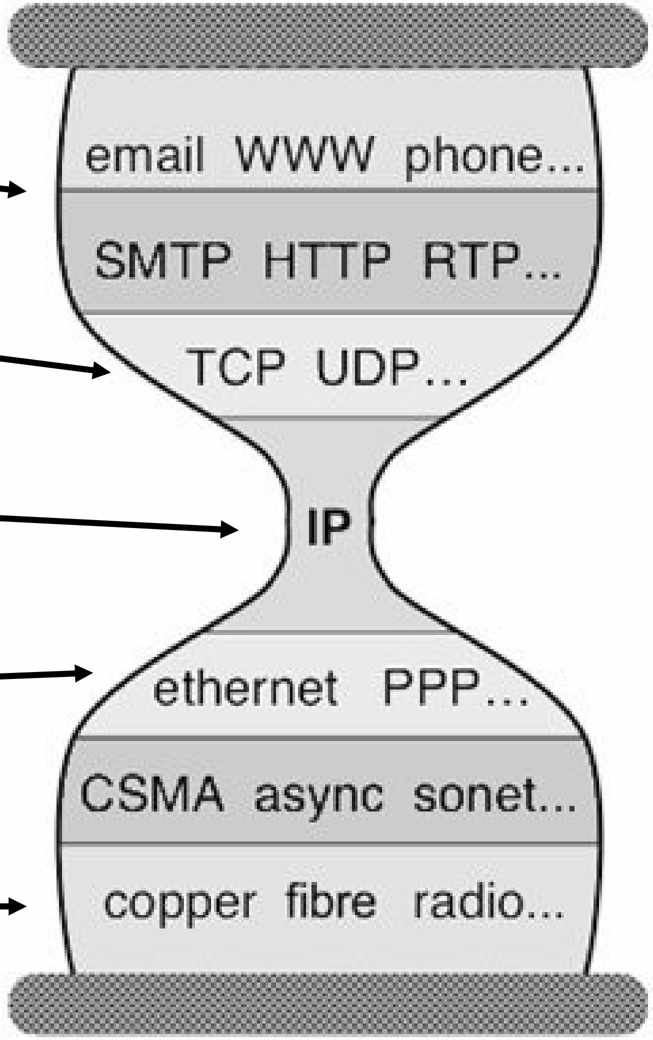
Best-effort global packet delivery

...built on...

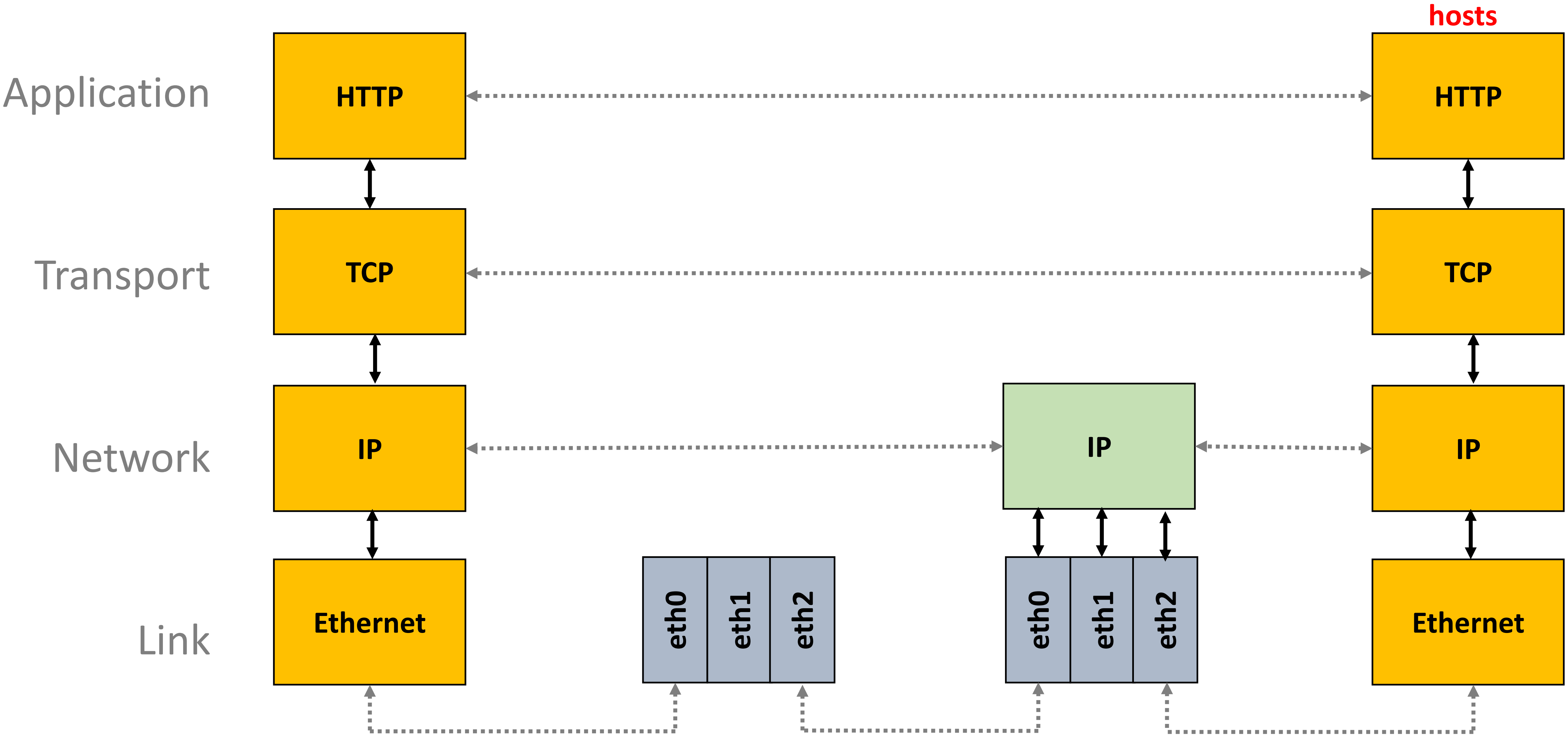
Best-effort local packet delivery

...built on...

Physical transfer of bits



Since when bits arrive they must make it to the application, all the layers exist on a host



A network *connection* is characterized by its delay, loss rate and throughput

delay

loss

throughput

How long does it take for a packet to reach the destination

What fraction of packets sent to a destination are dropped?

At what rate is the destination receiving data from the source?

This week

Fundamental challenges – Part I

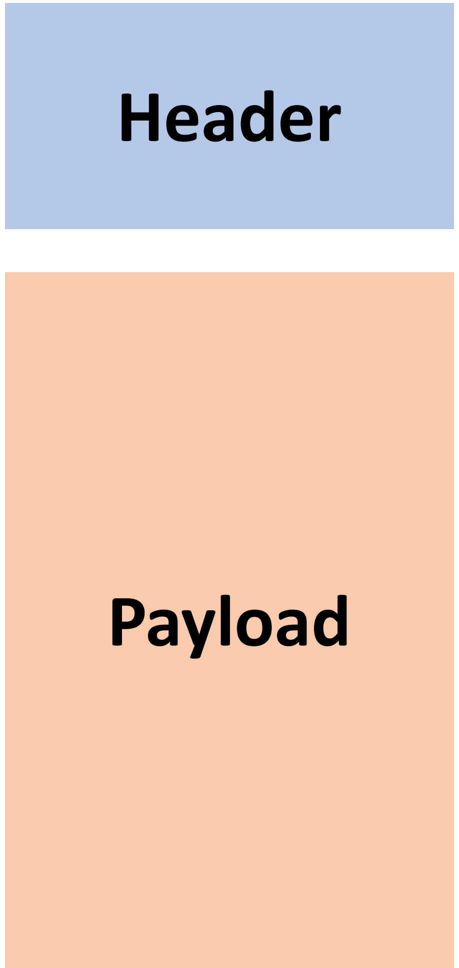
Routing

How do you deliver packet from a source to destination?

Think of IP packets as envelopes

Packet

Think of IP packets as envelopes



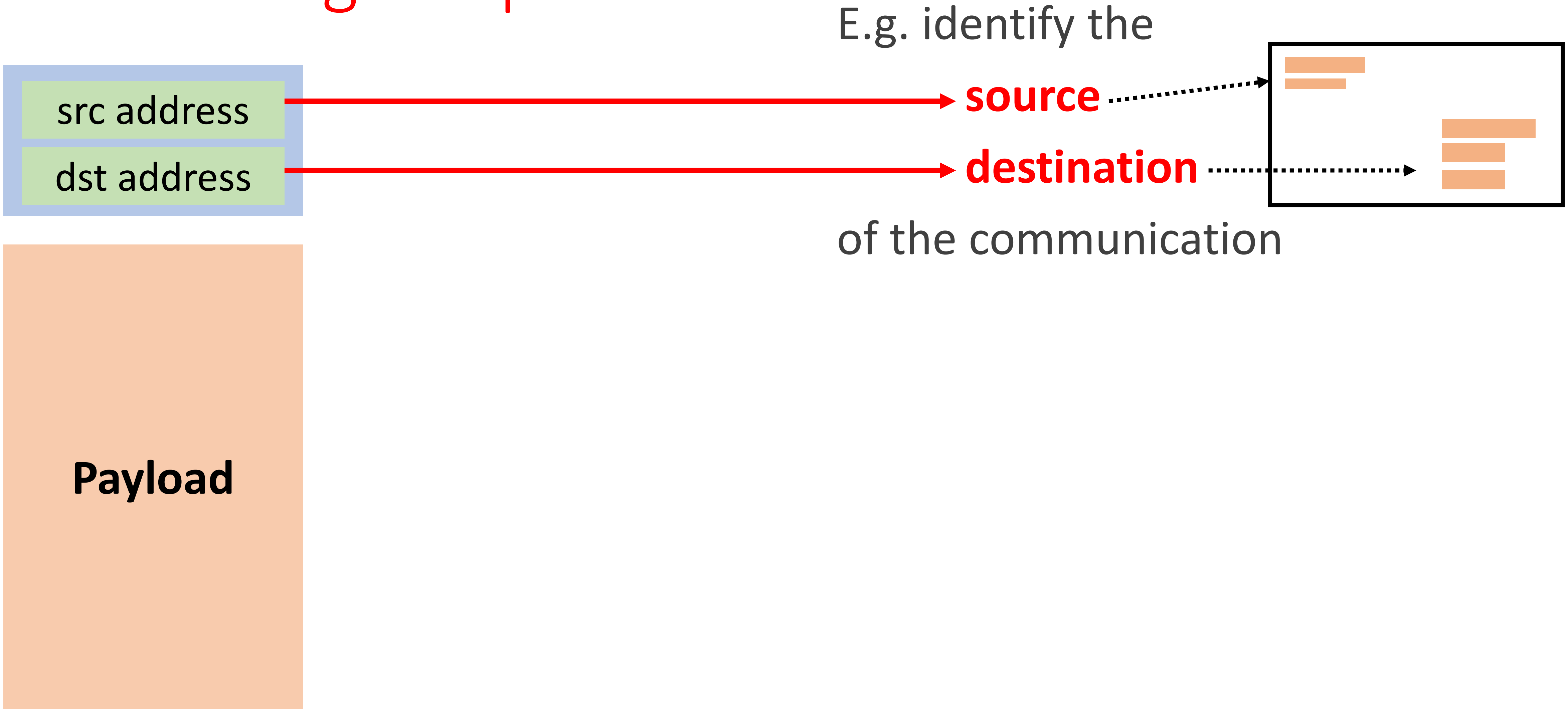
They have

a header

&

a payload

The header contains metadata needed for forwarding the packet



src address

dst address

Payload

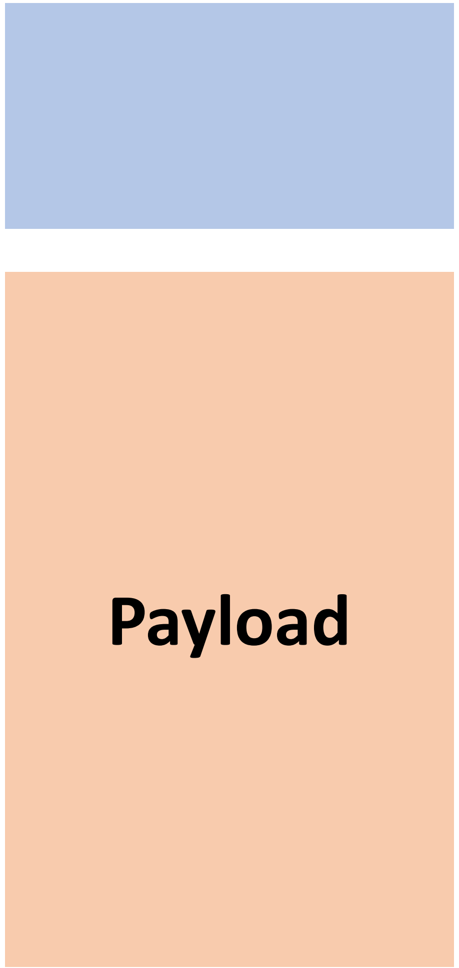
E.g. identify the

source

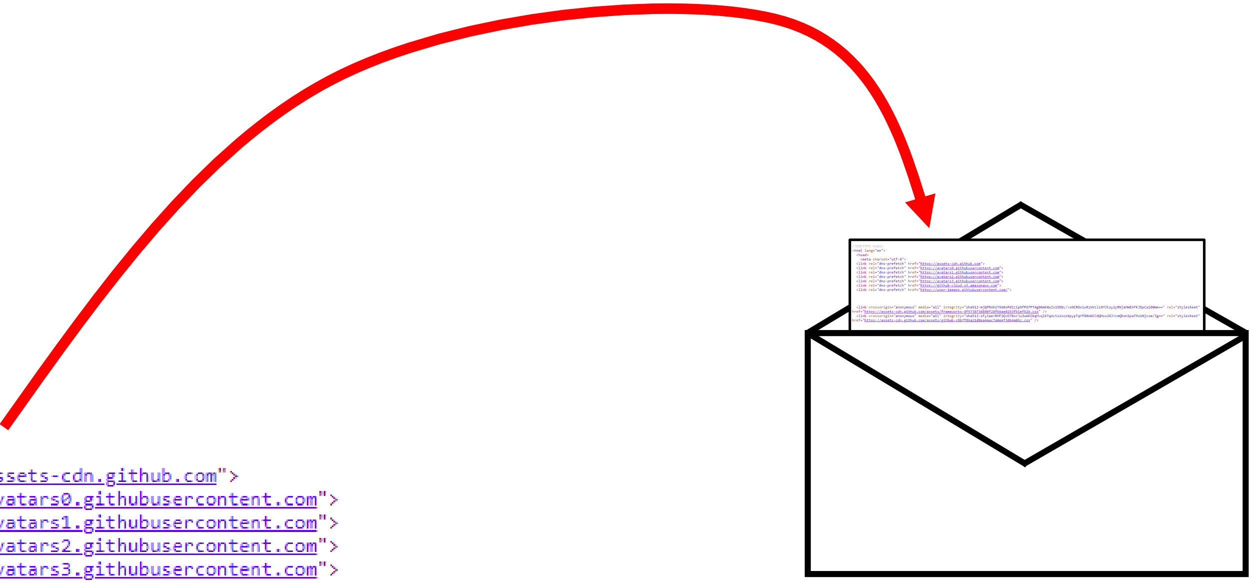
destination

of the communication

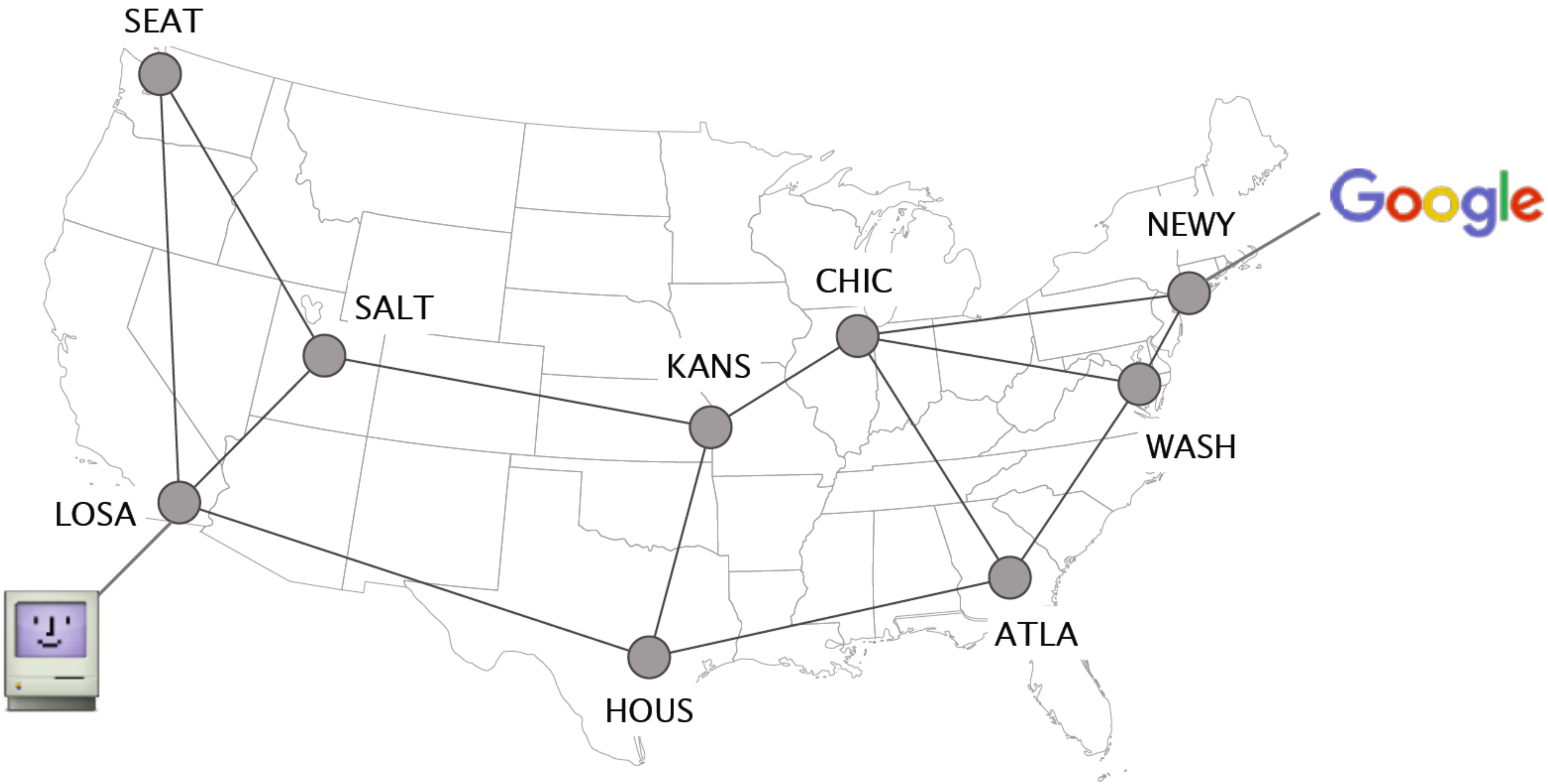
The payload contains the data to be delivered



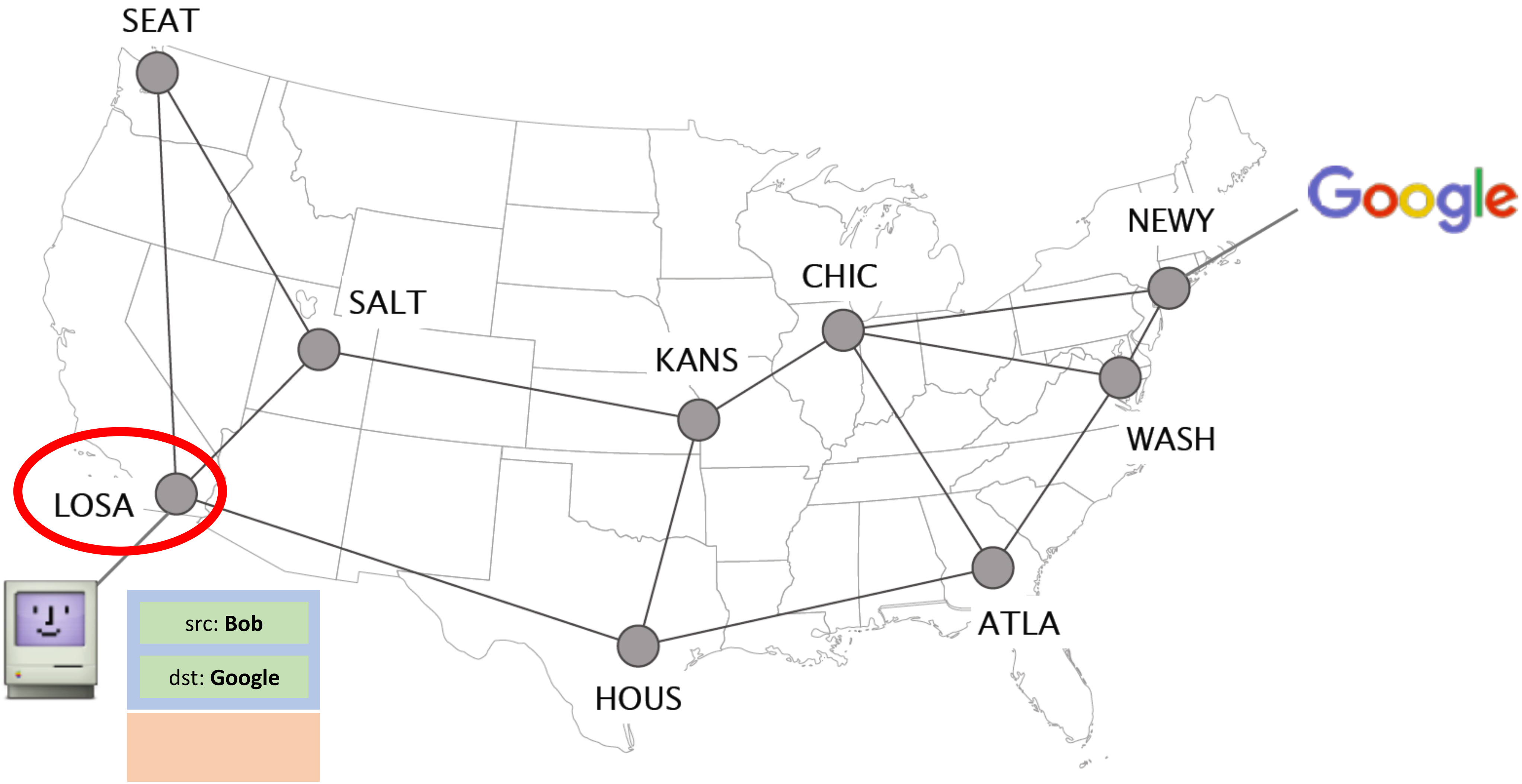
```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
  <link rel="dns-prefetch" href="https://assets-cdn.github.com">
  <link rel="dns-prefetch" href="https://avatars0.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars1.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars2.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars3.githubusercontent.com">
  <link rel="dns-prefetch" href="https://github-cloud.s3.amazonaws.com">
  <link rel="dns-prefetch" href="https://user-images.githubusercontent.com/">



  <link crossorigin="anonymous" media="all" integrity="sha512-mjQPRAh2Y9A0sPdZzipNfPO7PT4g06mk0uZs15DbL/vsNCRGx1uRzWVzls9MJCoy2yRNjaMmEVFKJDpCui00mA==" rel="stylesheet"
href="https://assets-cdn.github.com/assets/frameworks-df973073d880f28fbbae0263fb1ef62b.css" />
  <link crossorigin="anonymous" media="all" integrity="sha512-sFylaerRMF2QvD7BxrJw3uWMZbqMvqlbTqActs2xcnXpypTqYf80W6OJdQHsx2GJrcmQhxU1paT9sUNjcsm/3g==" rel="stylesheet"
href="https://assets-cdn.github.com/assets/github-c8b7f8ba21d8ea4aac7a0e4f3db4a01c.css" />





  <meta name="viewport" content="width=device-width">

  <title>Juniper/grpc-c: C implementation of gRPC layered on top of core library</title>
    <meta name="description" content="C implementation of gRPC layered on top of core library - Juniper/grpc-c">
    <link rel="search" type="application/opensearchdescription+xml" href="/opensearch.xml" title="GitHub">
  <link rel="fluid-icon" href="https://github.com/fluidicon.png" title="GitHub">
```



Routers forward IP packets hop-by-hop towards their destination



Routers forward IP packets hop-by-hop towards their destination



src: **Bob**

dst: **Google**

Routers forward IP packets hop-by-hop towards their destination

Routers forward IP packets hop-by-hop towards their destination

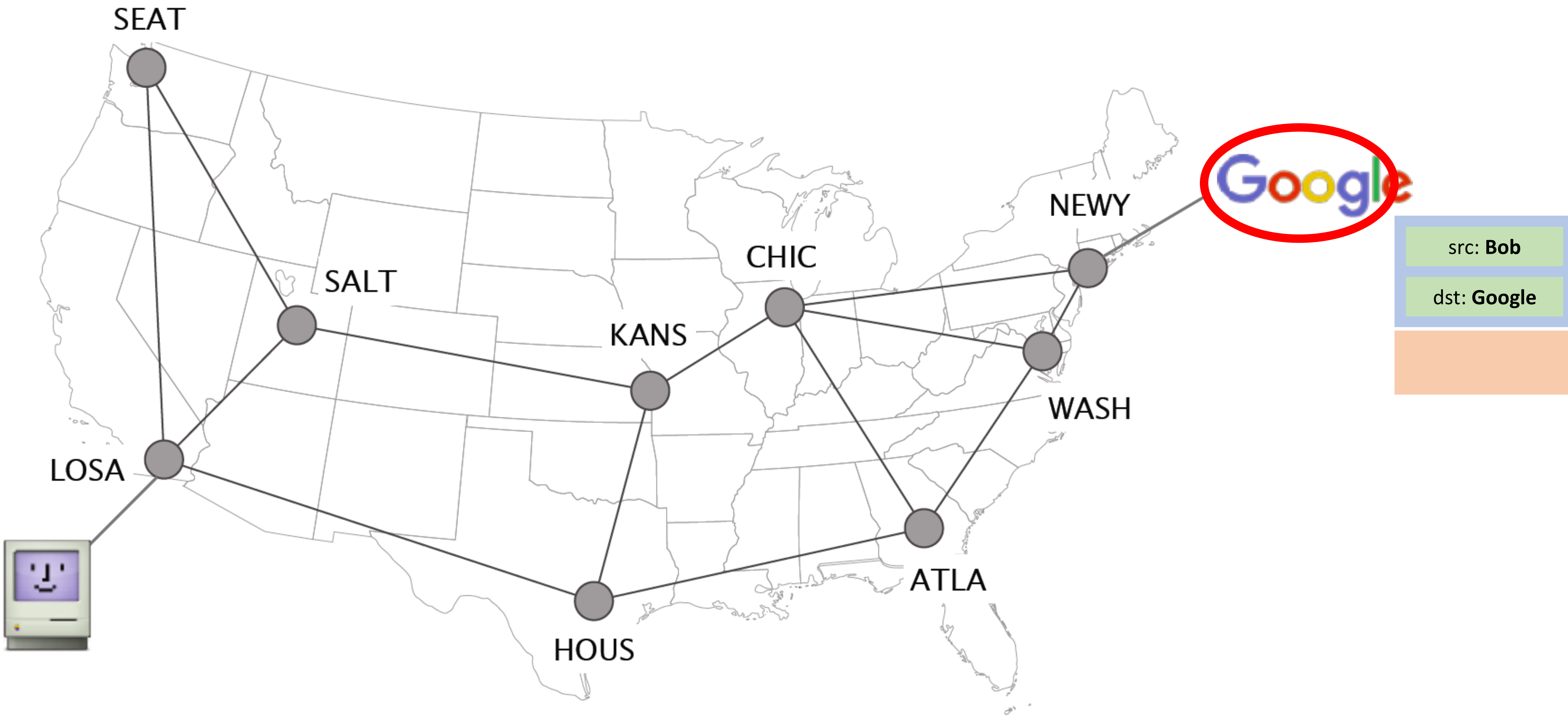
Routers forward IP packets hop-by-hop towards their destination

src: **Bob**

dst: **Google**

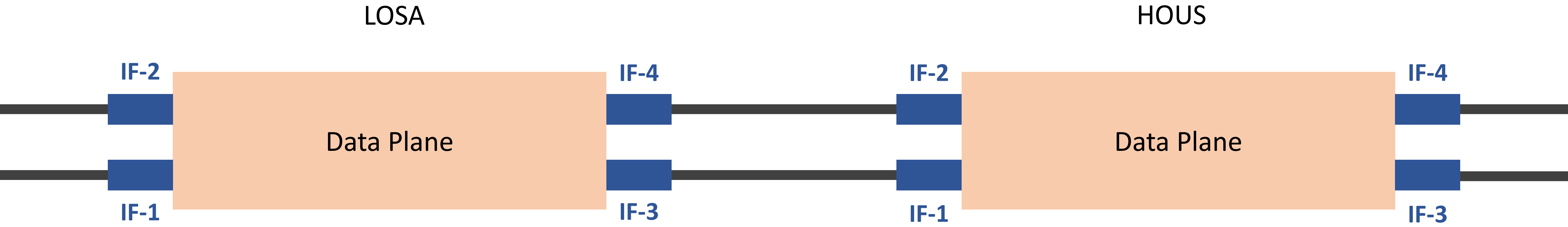
Routers forward IP packets hop-by-hop towards their destination

Routers forward IP packets hop-by-hop towards their destination

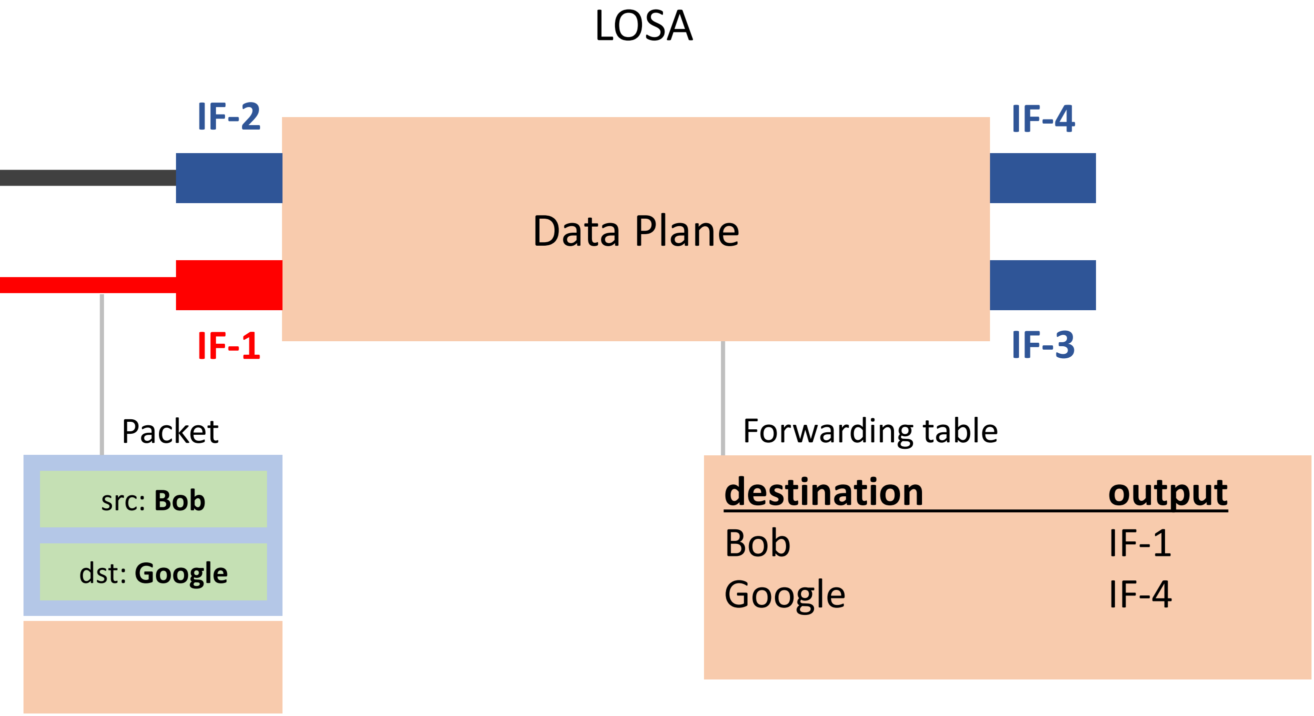


Let's check what is going on between two neighboring routers

Two neighboring routers



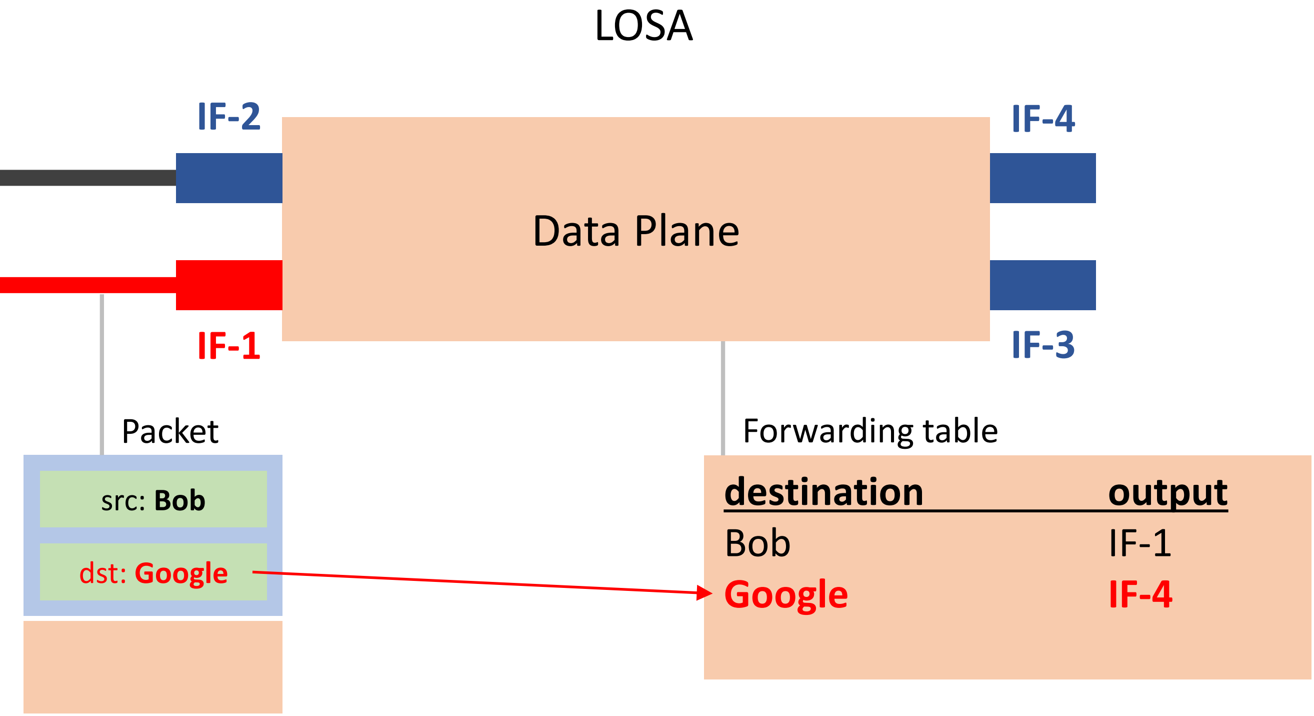
Upon packet reception, routers locally lookup their forwarding table to know where to send it next



Forwarding table

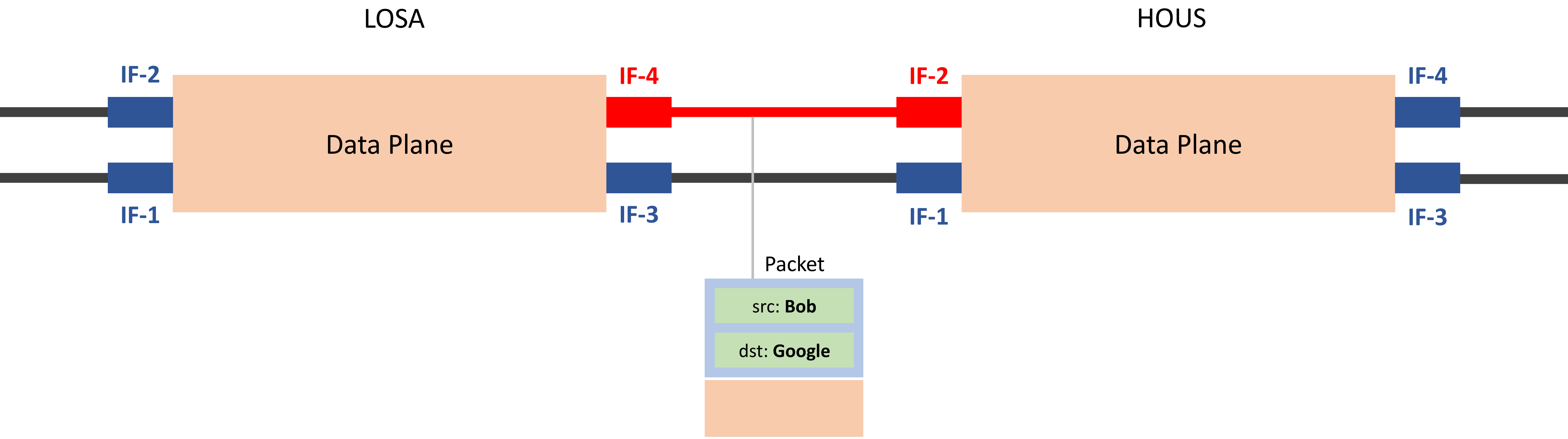
destination	**output**
Bob	IF-1
Google	IF-4

According to the fwd table, the packet should be directed to IF-4

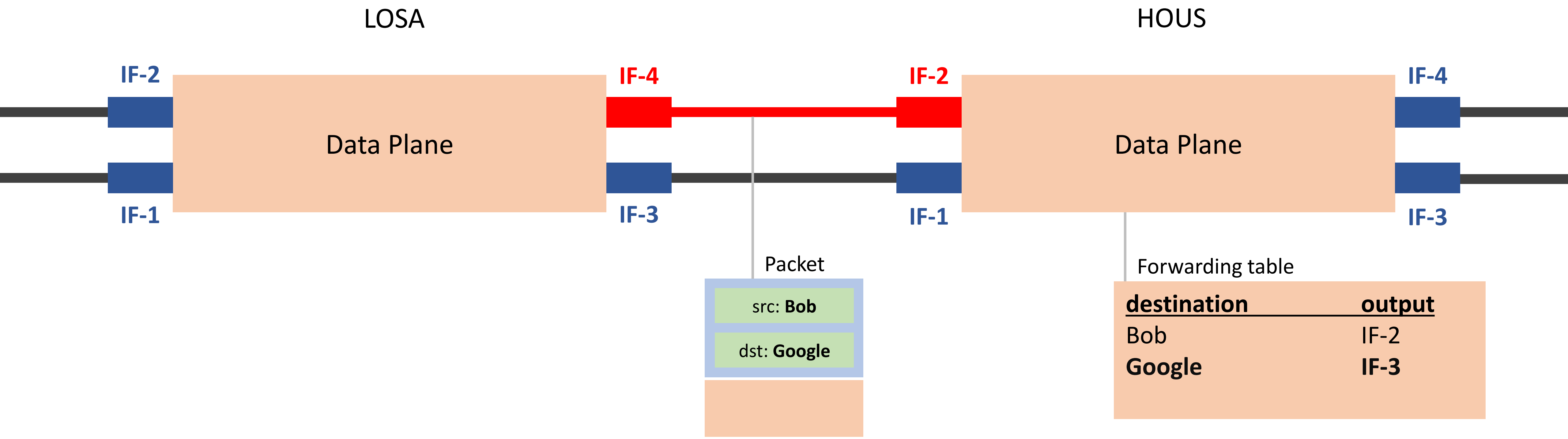


destination	**output**
Bob	IF-1
Google	**IF-4**

According to the fwd table, the packet should be directed to IF-4



Forwarding is repeated at each router until the destination is reached



Packet

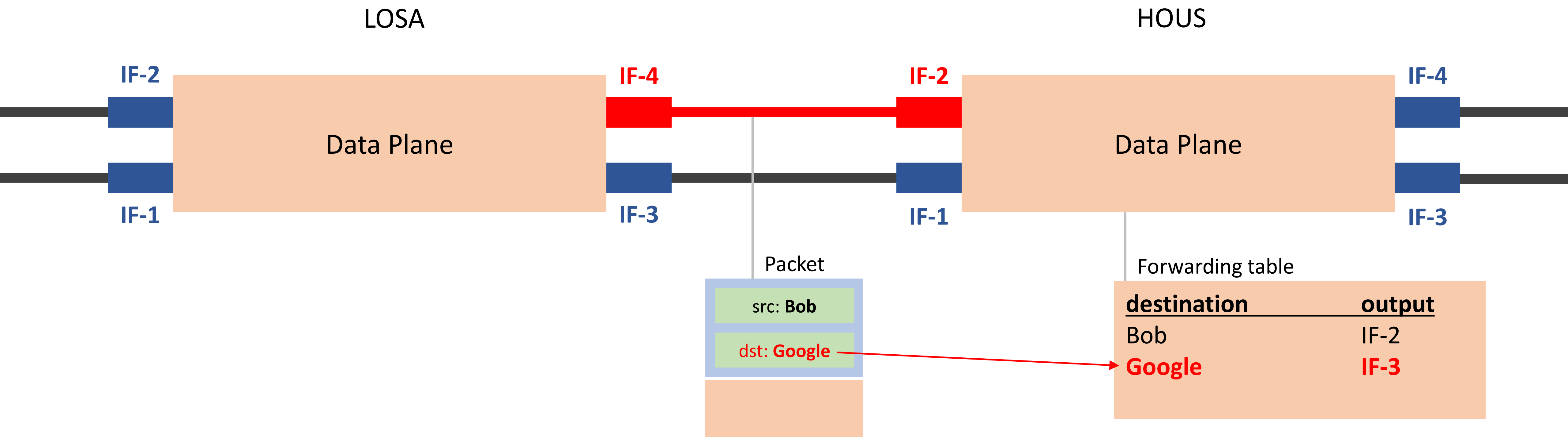
src: **Bob**

dst: **Google**

Forwarding table

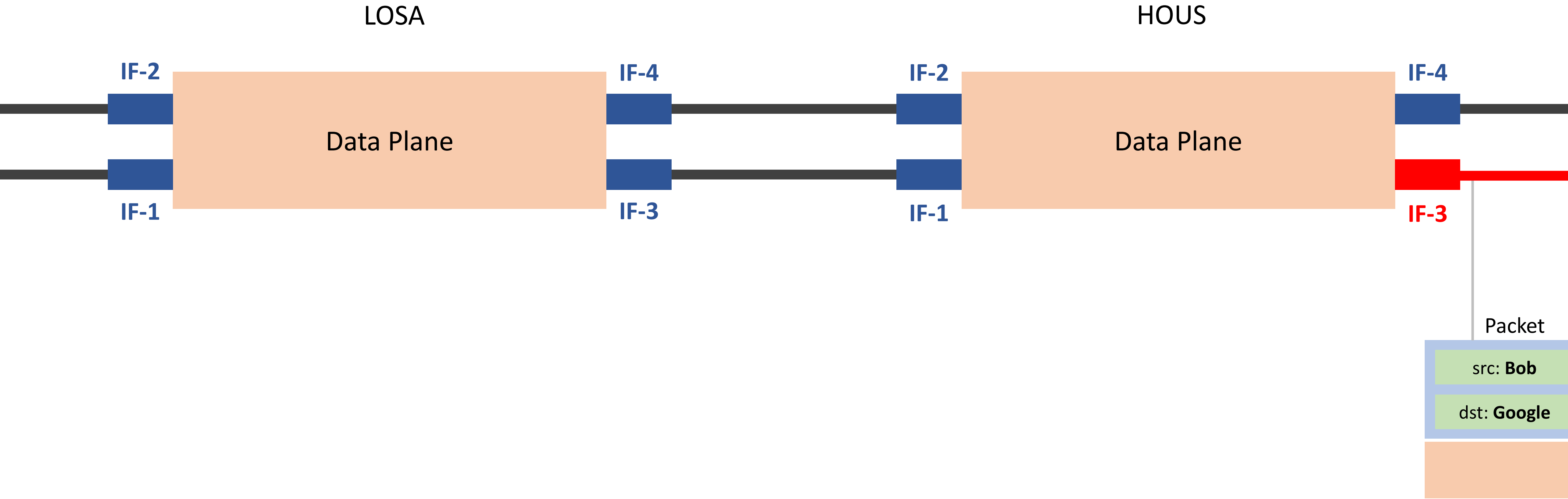
destination	output
Bob	IF-2
Google	**IF-3**

Forwarding is repeated at each router until the destination is reached



destination	**output**
Bob	IF-2
Google	**IF-3**

Forwarding is repeated at each router until the destination is reached



Nowadays network equipments can have Terabits per second of forwarding capacity

Forwarding decisions necessarily depend on the destination, but can also depend on other criteria

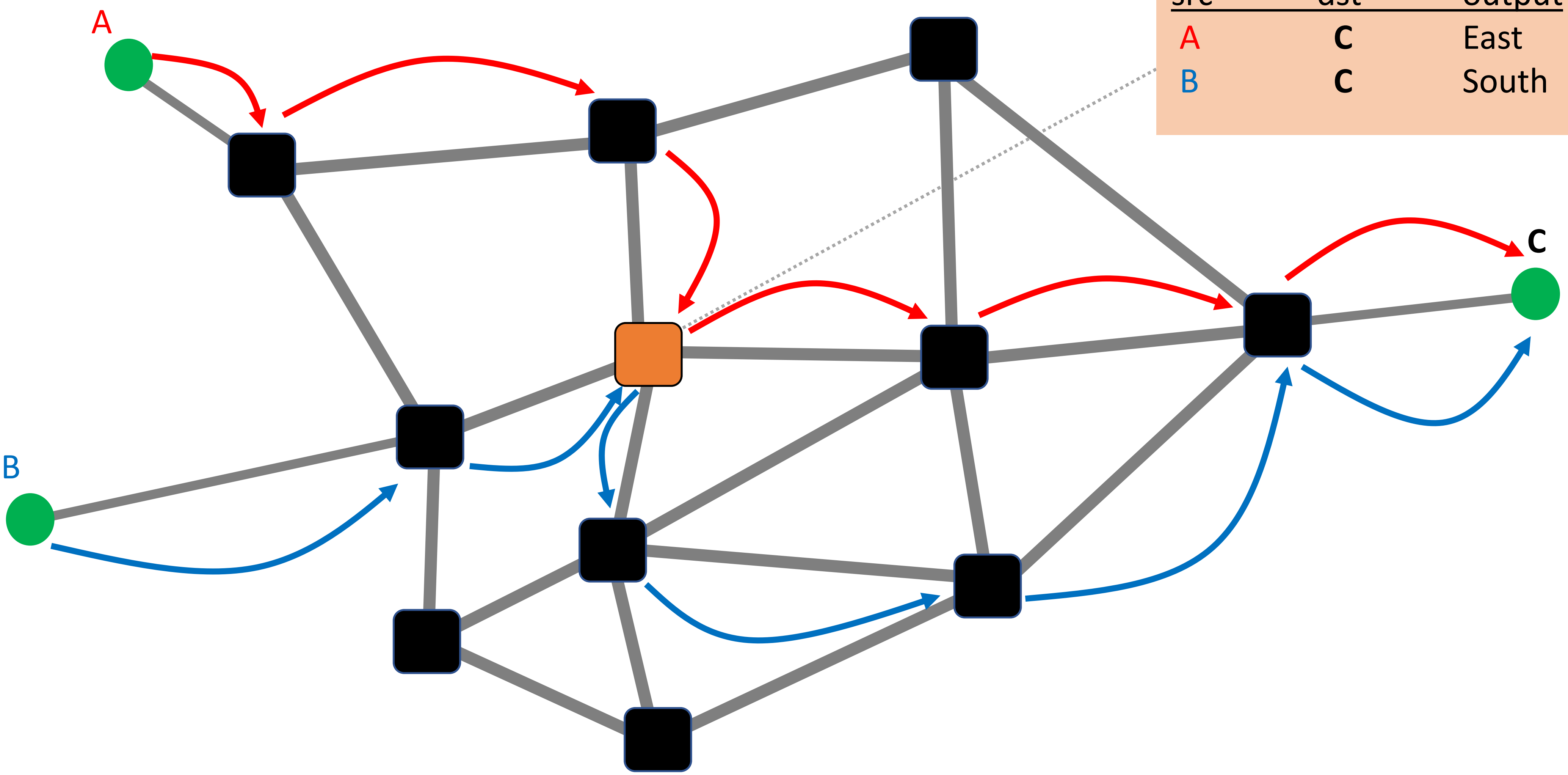
criteria	**destination**	mandatory (why?)
	source	requires n^2 states
	input port	traffic engineering
	+any other header fields	

Forwarding decisions necessarily depend on the destination, but can also depend on other criteria

criteria	**destination**	mandatory (why?)
	source	requires n^2 states
	input port	traffic engineering
	+any other header fields	

Let’s consider source- and destination-based routing

Paths from different sources can differ

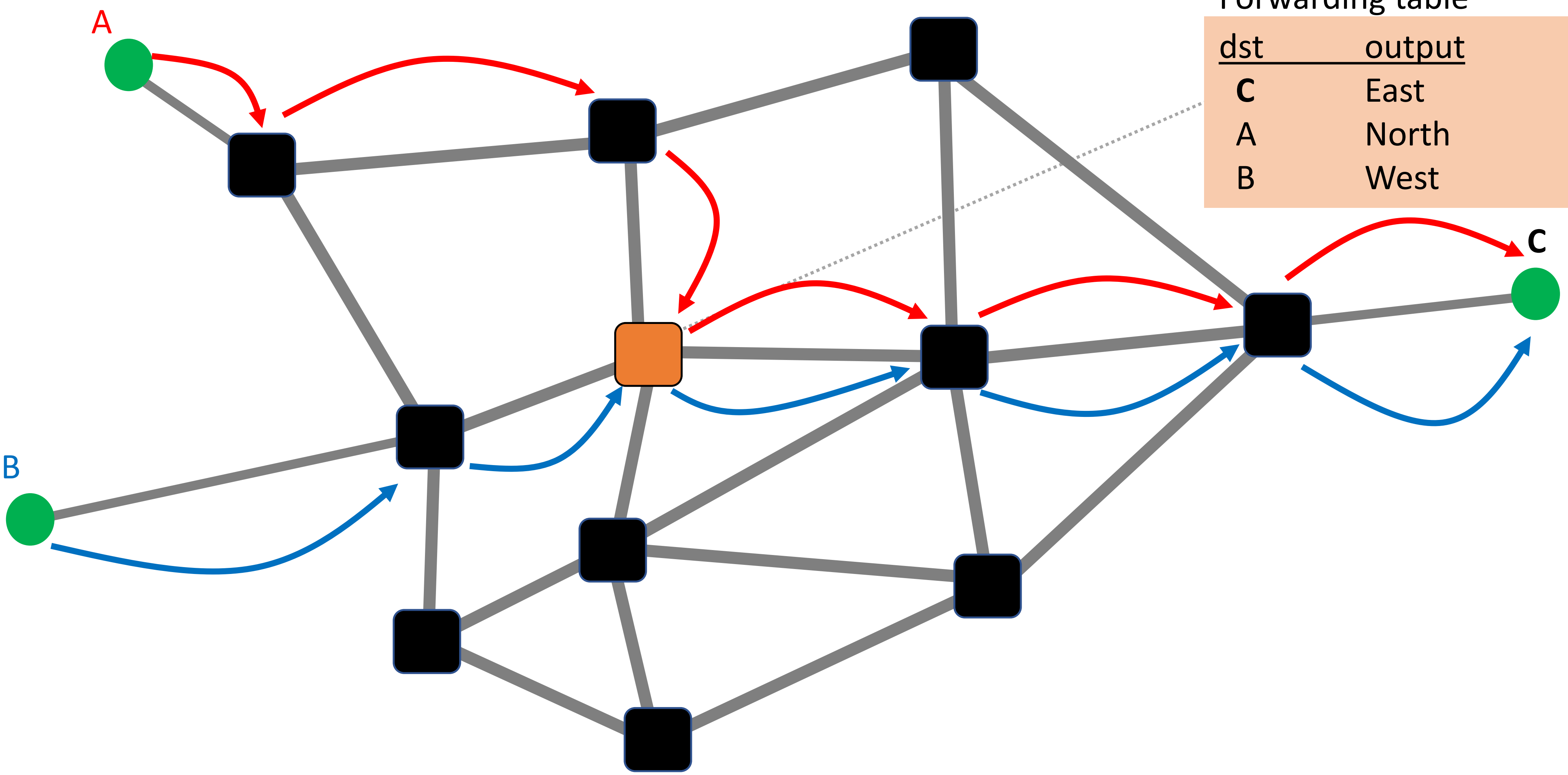


Forwarding table

src	dst	output
A	**C**	East
B	**C**	South

With destination-based routing

Paths from different sources coincide once they overlap



Forwarding table

dst	output
C	East
A	North
B	West

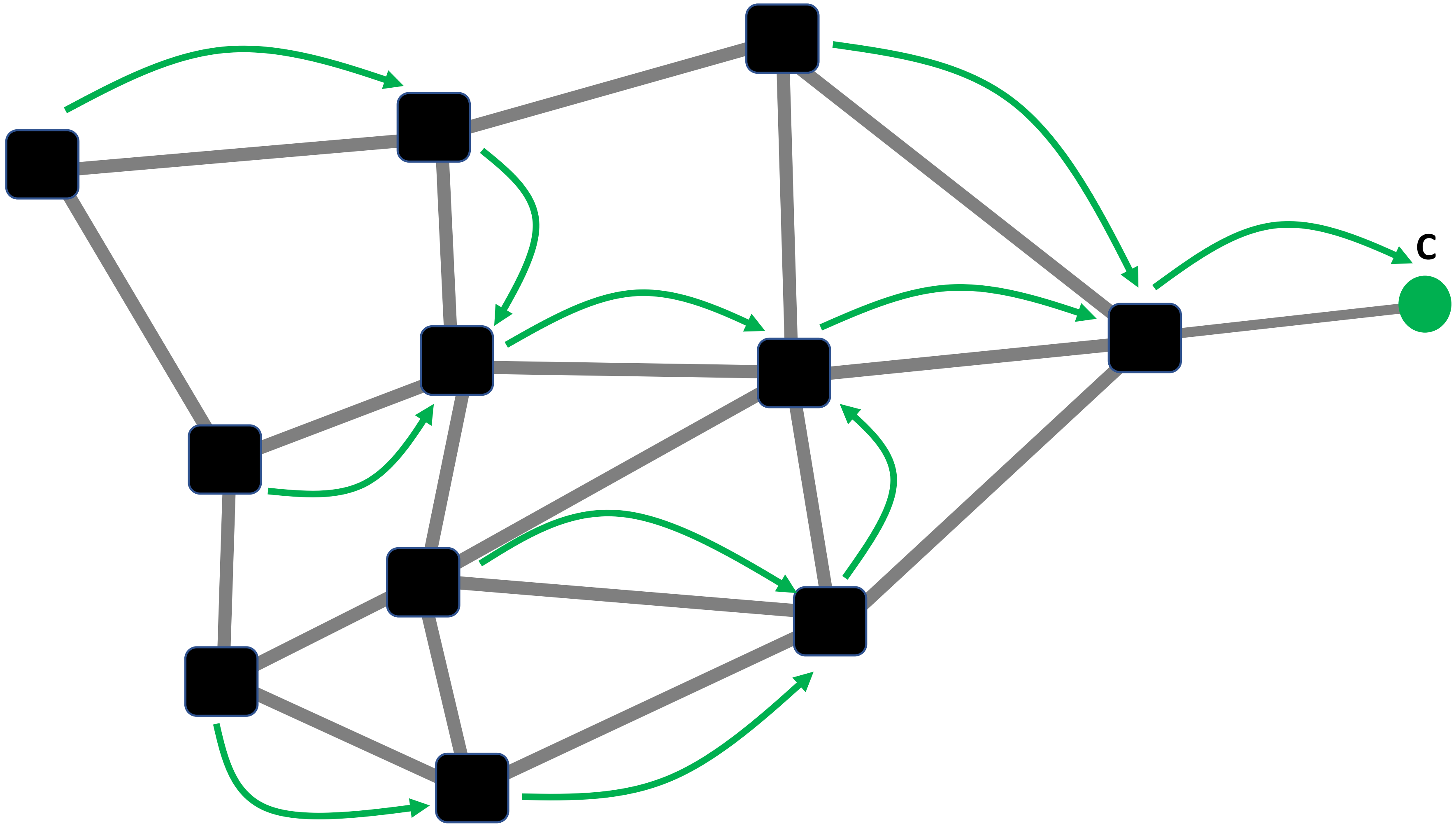
Once paths to destination meet,
they will never split

Set of paths to the destination produce
a spanning tree rooted at the destination:

cover every router exactly once

only one outgoing arrow at each router

An example of a spanning tree for destination **C**



In the rest of the lecture,
we’ll consider **destination-based** routing

The default in the Internet

Where are these forwarding tables coming from?

Forwarding table

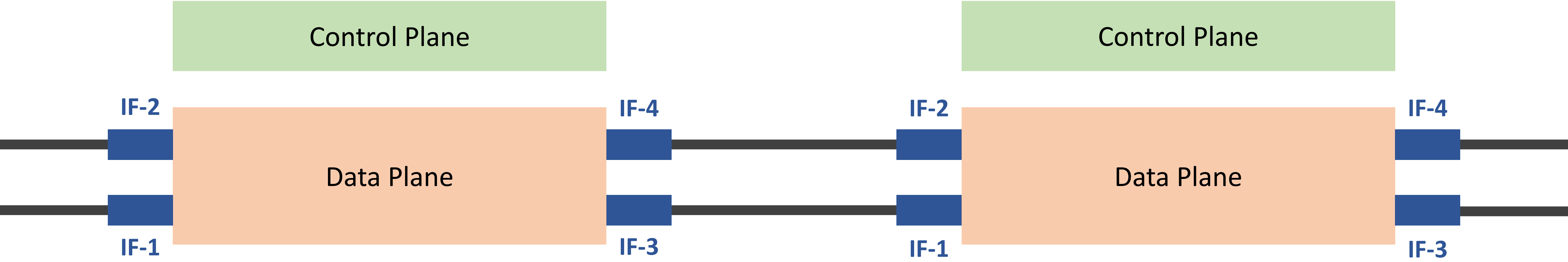
destination	output
Bob	IF-1
Google	IF-4

Forwarding table

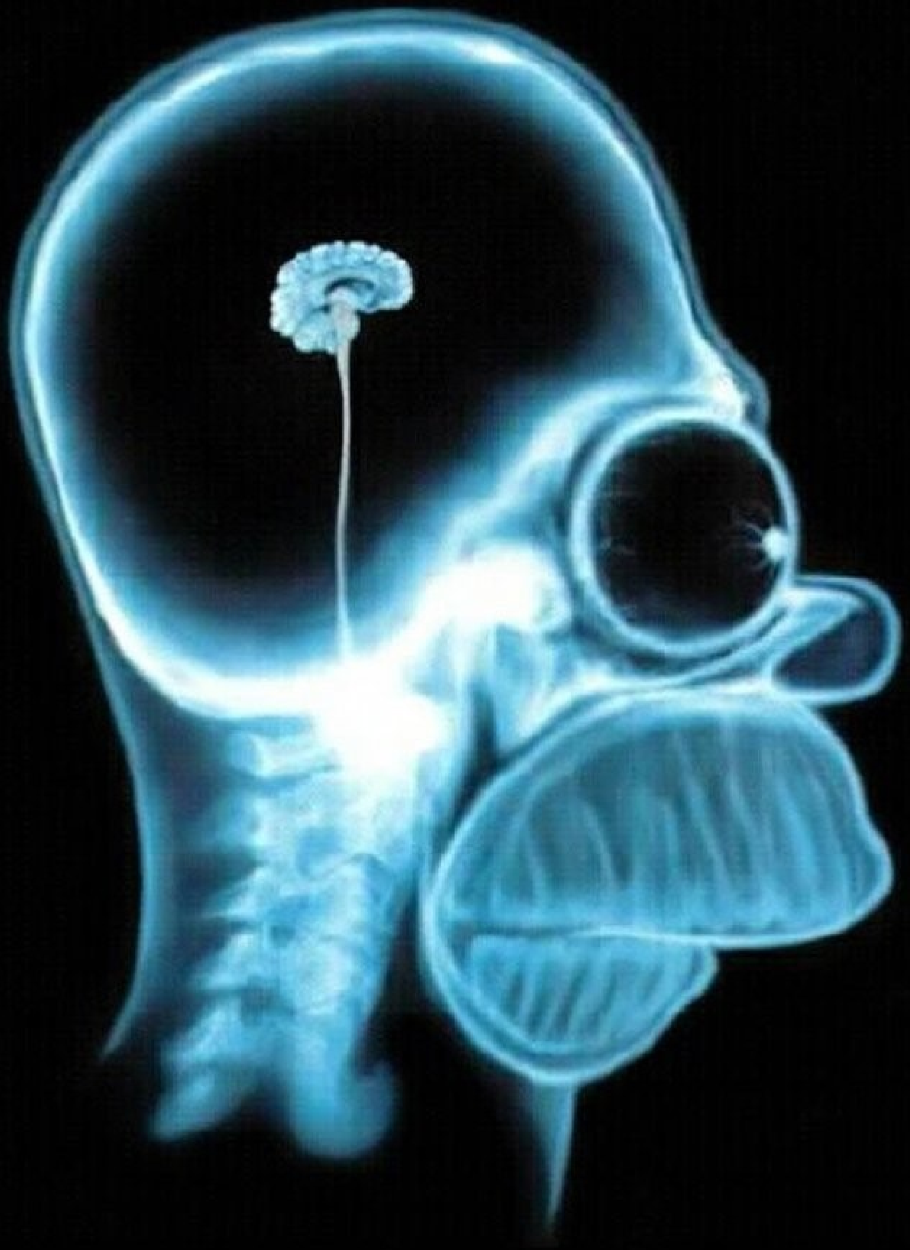
destination	output
Bob	IF-2
Google	IF-3

In addition to a data plane …

In addition to a data plane, routers are also equipped with a **control plane**



Control plane = the router’s brain



Control plane = the router’s brain

Roles

Routing

Configuration

Statistics (counters, meters, etc.)

…

Routing is the control plane process that computes and populates the forwarding tables

Control Plane

Forwarding table

destination	output
Bob	IF-1
Google	IF-4

Control Plane

Forwarding table

destination	output
Bob	IF-2
Google	IF-3

While forwarding is a **local** process, routing is inherently a **global** process

A router should know how the network looks like
for directing the packet towards the destination.

Forwarding vs routing

	forwarding	routing
Goal	**directing a packet to an outgoing link**	**computing the path packets will follow**
Scope	**local**	**global, network wide**
Implementation	**hardware (usually)** **(software is also possible)**	**software (always)**
Timescale	**nanoseconds**	**10s of milliseconds**

The goal of routing is to compute valid global forwarding state

[Definition]

A global forwarding state is valid if

it always delivers packets to the correct destination

Valid states

[Theorem]

A global forwarding state is valid iff (iff = <u>if and only if</u>)

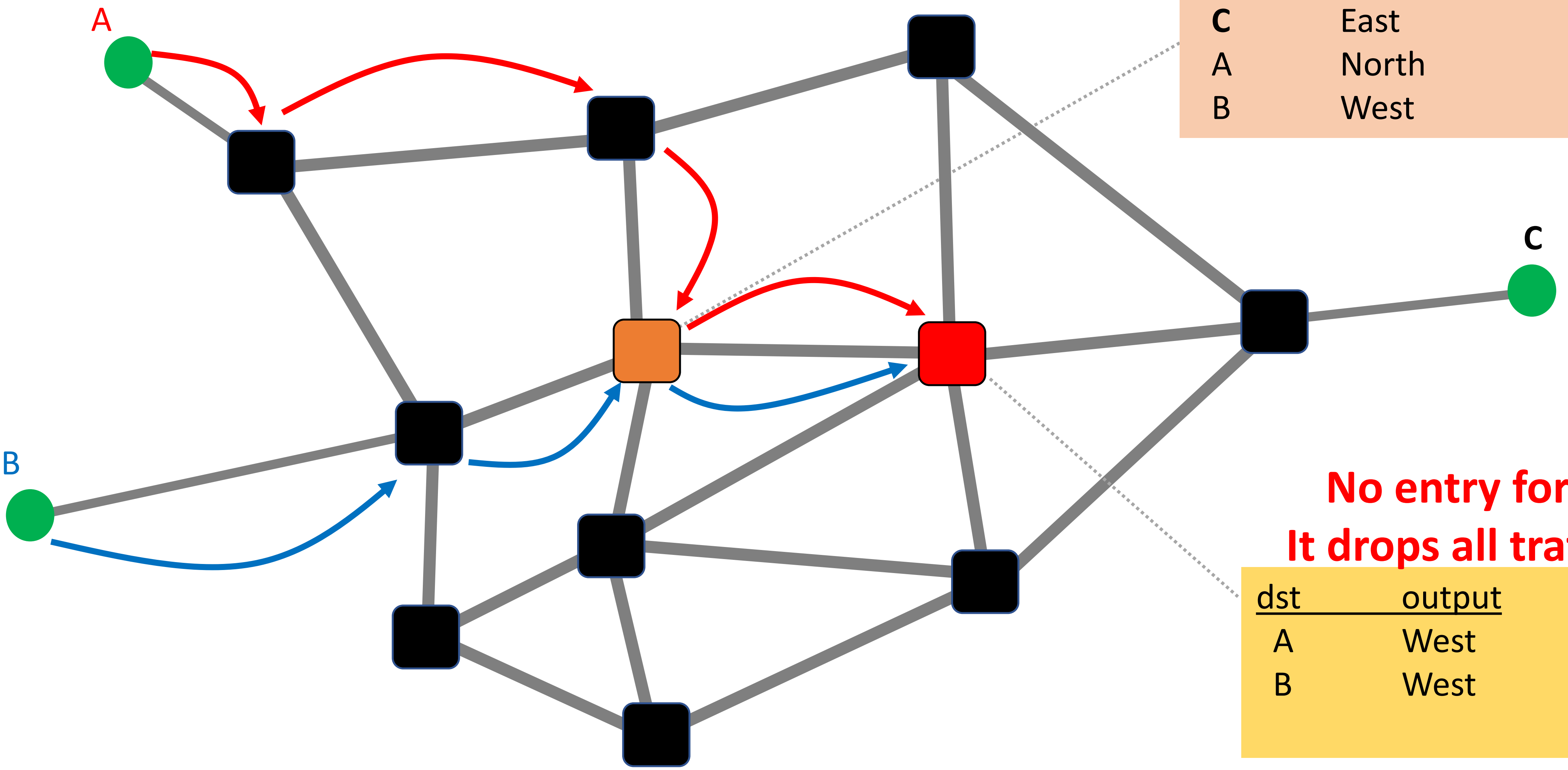
A) there are no dead ends

dead end = i.e. no outgoing port defined in the table for a given dst

B) there are no loops

loop = i.e. packets going around the same set of nodes

A global forwarding state is valid if and only if there are **no dead ends**



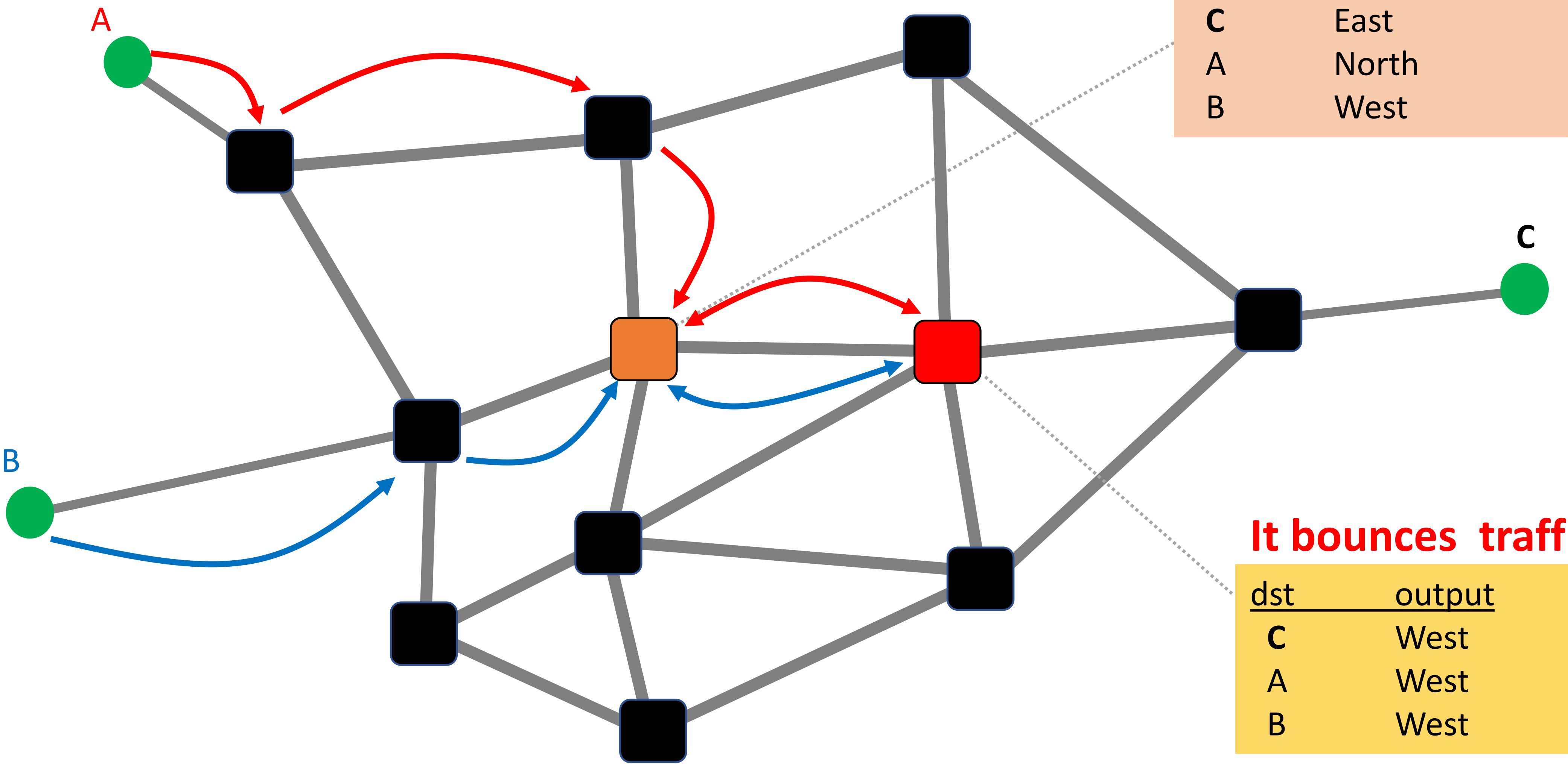
dst	output
C	East
A	North
B	West

No entry for dst C

It drops all traffic to C

dst	output
A	West
B	West

A global forwarding state is valid if and only if there are **no forwarding loops**



dst	output
C	East
A	North
B	West

It bounces traffic back

dst	output
C	West
A	West
B	West

Proving the necessary condition is easy

If a routing state is valid
then there are no loops or dead-end

[Proof]

If you run into a dead-end or a loop

you'll ***never*** *reach the destination*

Proving the sufficient condition is more subtle

If a routing state has no dead end and no loop
then it is valid

[Proof]

A) Assumption: there is only a finite number of ports to visit

B) A packet can never enter a switch via the same port, otherwise it is a loop (which does not exist by assumption)

C) As such, the packet must ***eventually*** *reach the destination*

How do we verify that a forwarding state is valid?

Verifying that a routing state is valid is easy

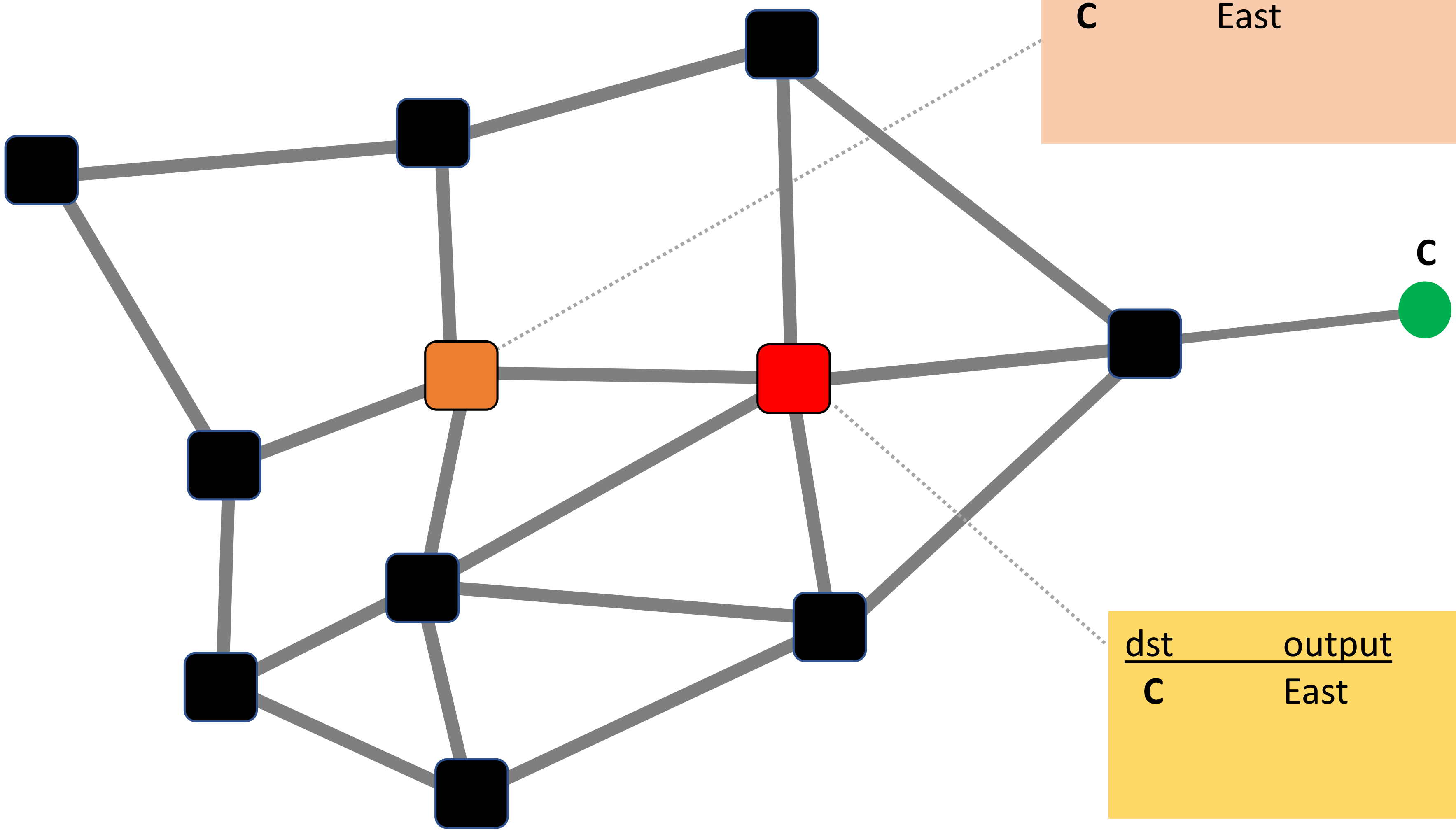
A simple algorithm for one destination

1) **Mark all outgoing ports with an arrow**

2) **Eliminate all links with no arrow**

3) **State is valid *iff* the remaining graph is a spanning-tree**

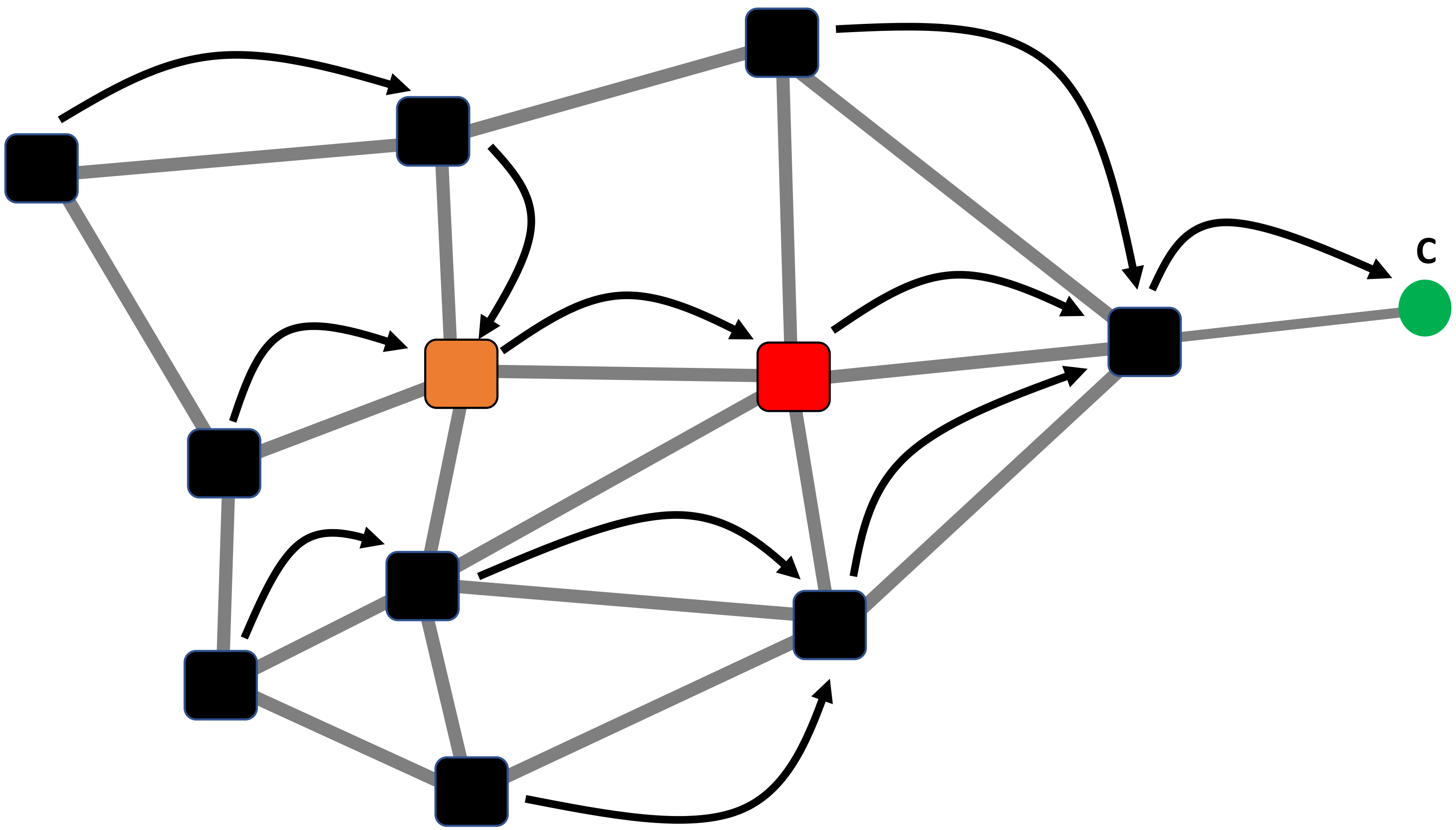
Given a graph



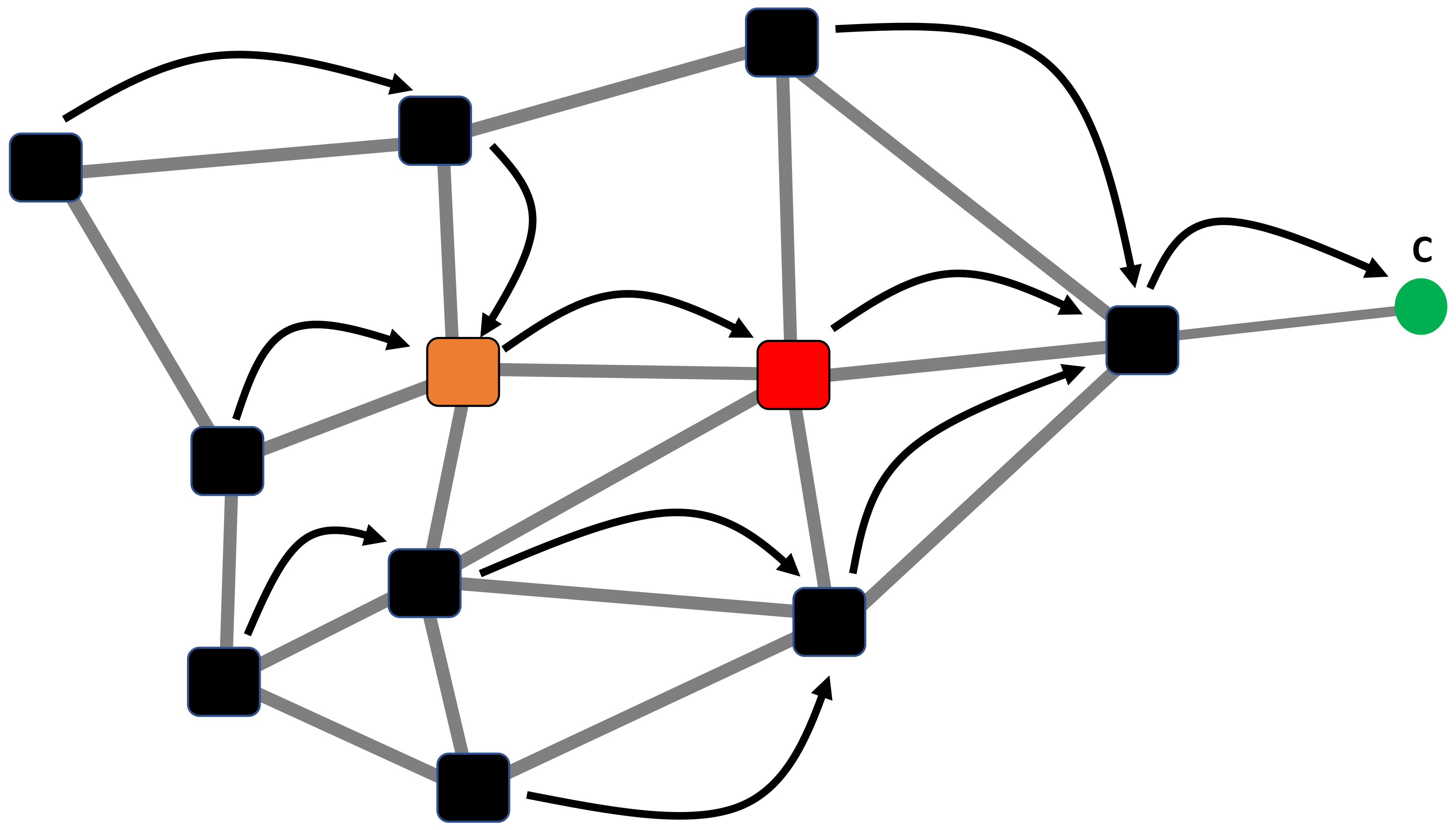
dst	output
C	East

dst	output
C	East

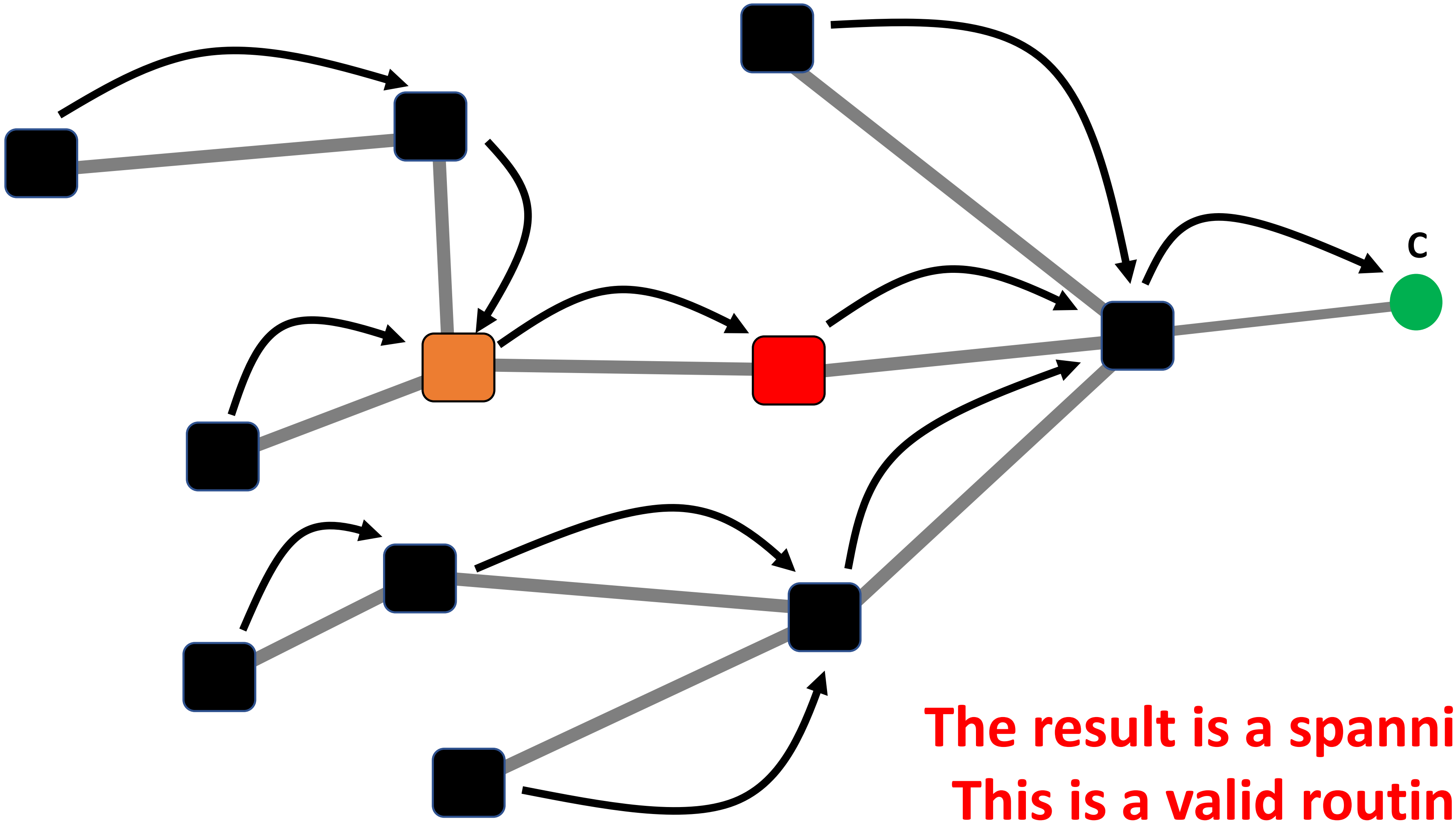
Mark all outgoing ports with an arrow



Eliminate links with no arrow

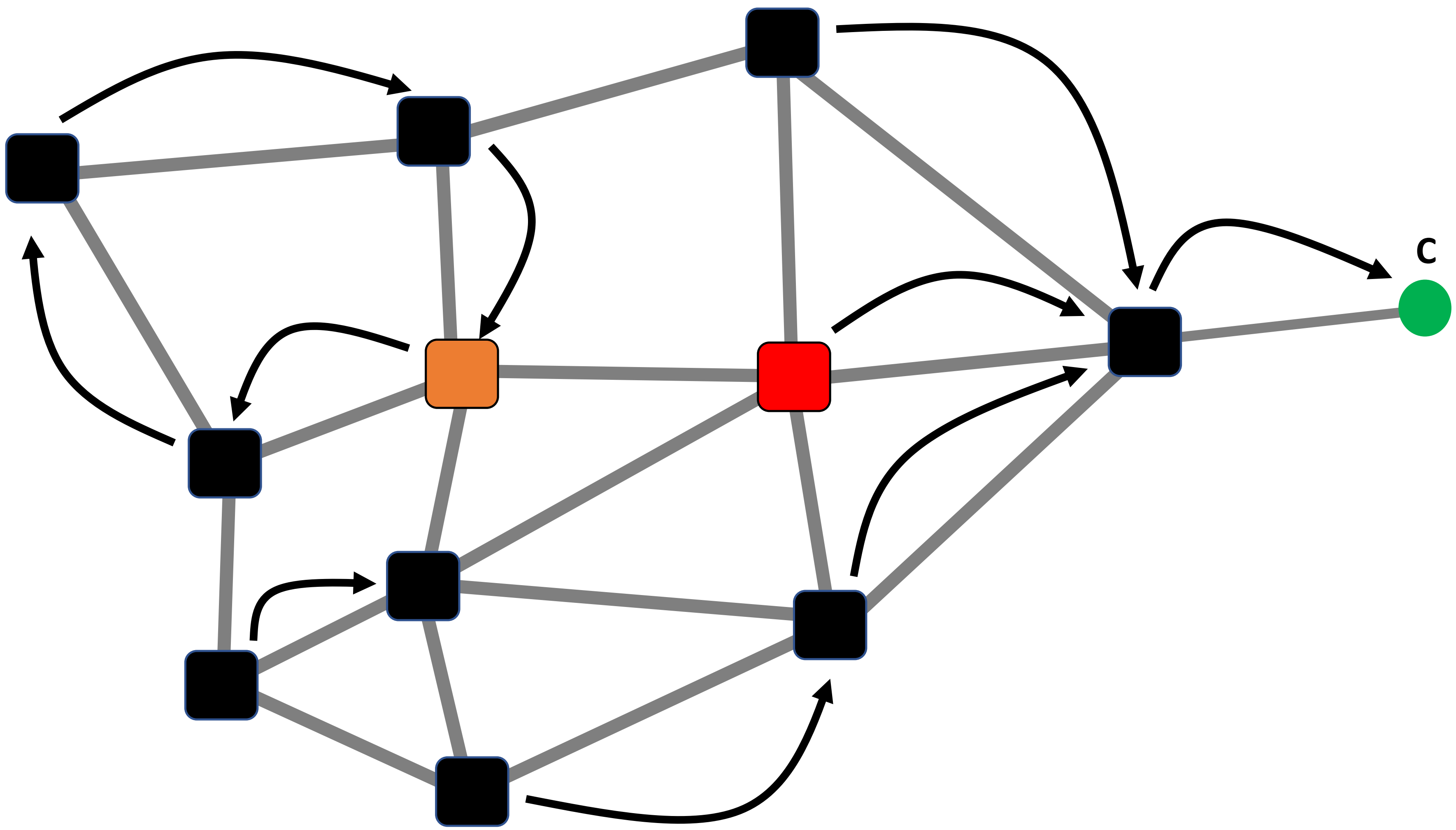


Eliminate links with no arrow

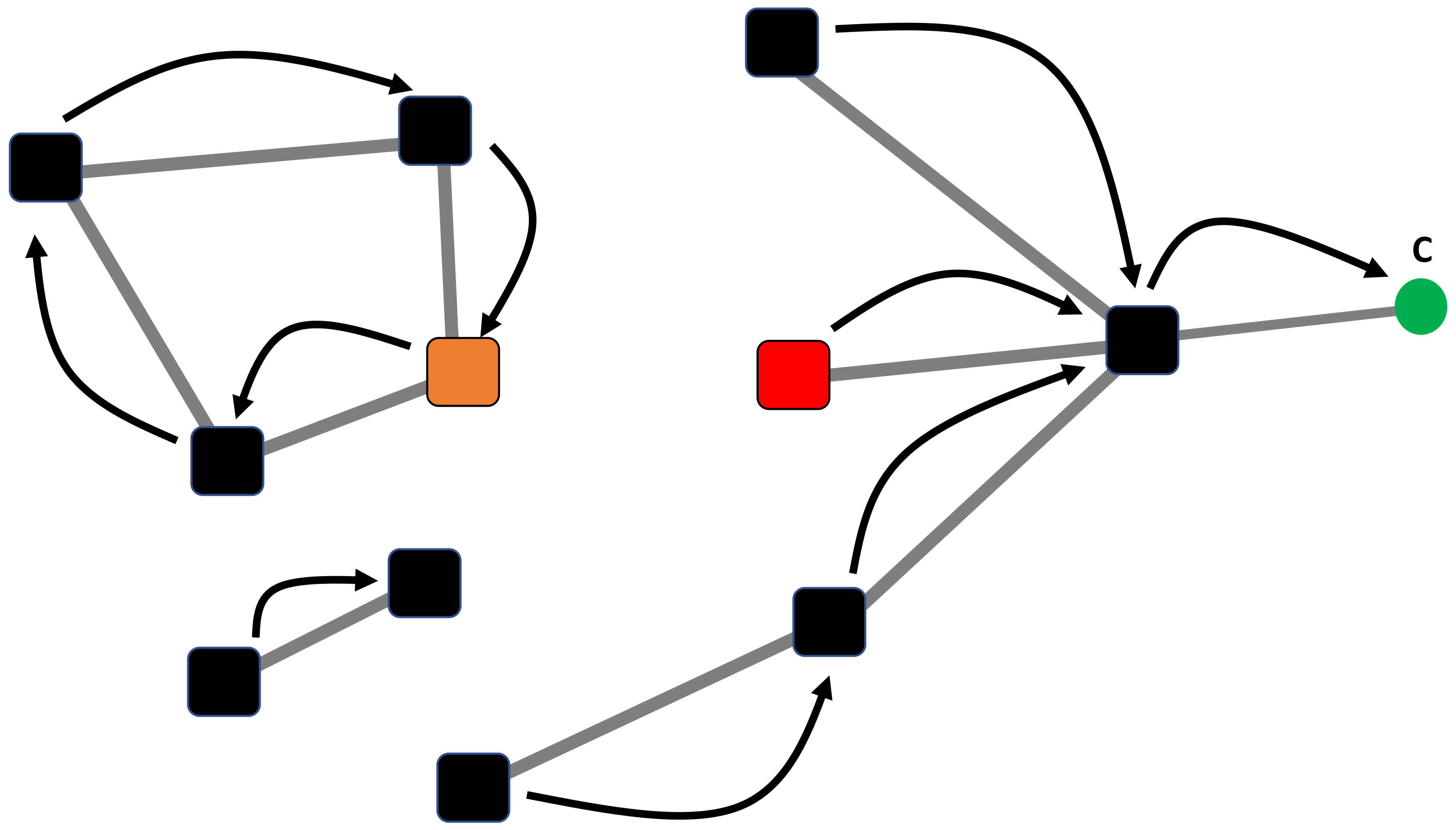


The result is a spanning tree.
This is a valid routing state

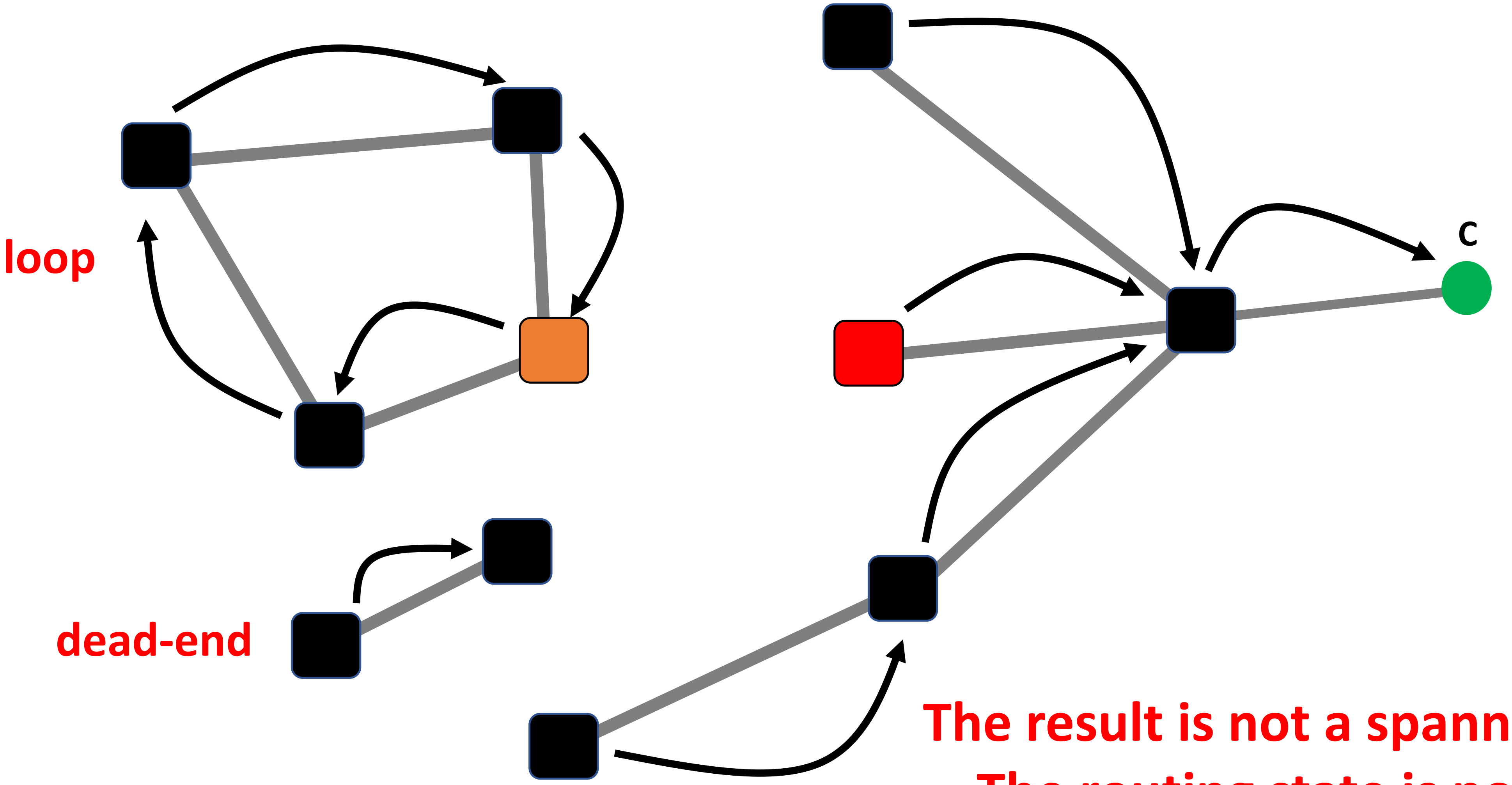
Example 2
Mark all outgoing ports with an arrow



Example 2
Eliminate links with no arrow



Example 2
Eliminate links with no arrow



The result is not a spanning-tree.
The routing state is not valid

How do we compute valid forwarding state?

Producing valid routing state is hard, but doable

Preventing dead-ends

easy

Preventing loops

harder – we will focus on this…

Existing routing protocols differ in how they avoid loops

Essentially, there are three ways to compute valid routing state

	Intuition	Example
1)	**Use tree-like topologies**	*Spanning-tree*
2)	**Rely on global network view**	*Link-state routing* *SDN*
3)	**Rely on distributed computation**	*Distance vector routing* *BGP*

1) Use tree-like topologies *Spanning-tree*

The easiest way to avoid loops is to route traffic on a loop-free topology

A simple algorithm

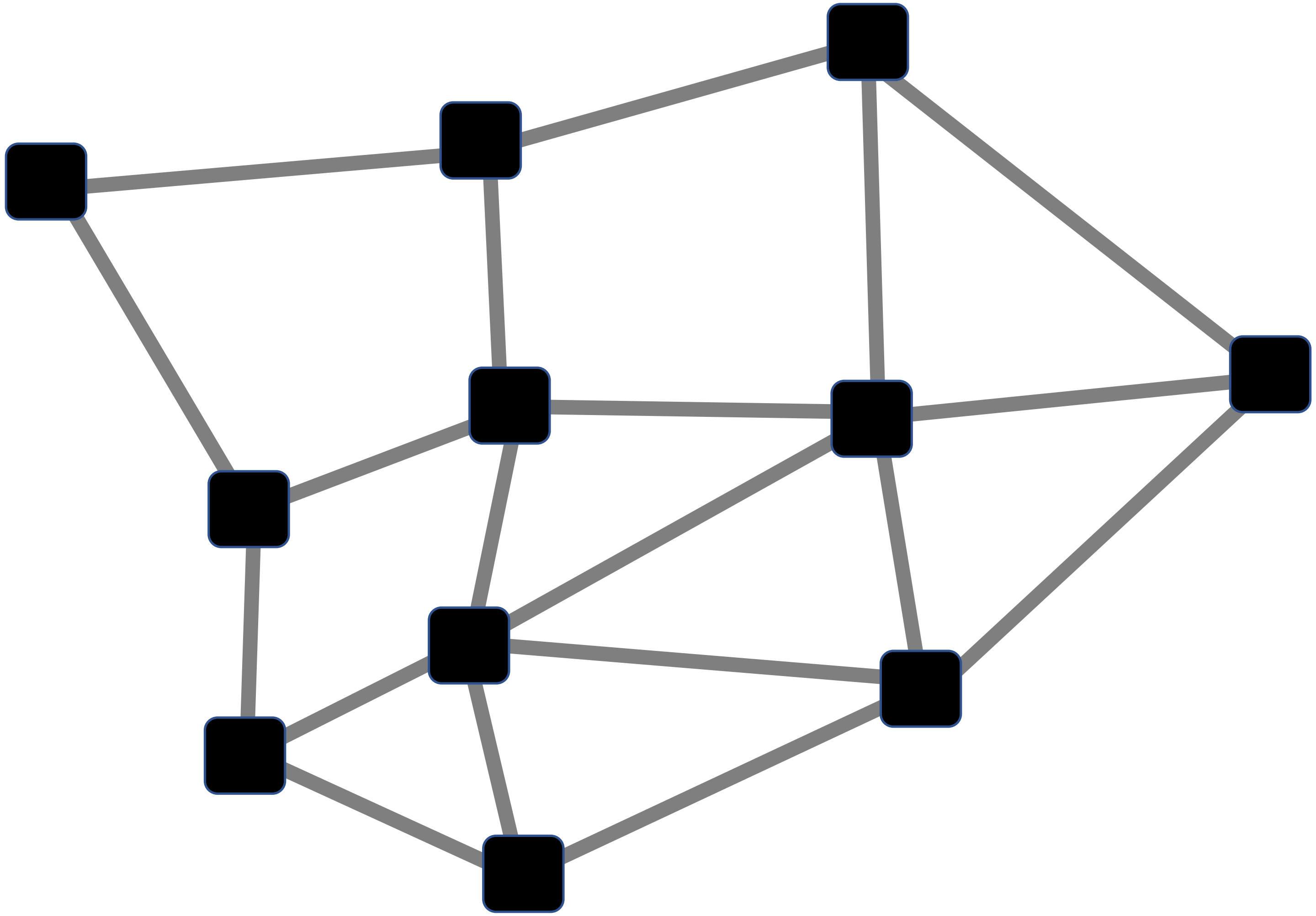
1) **Take an arbitrary topology**
2) **Build a spanning tree and ignore all other links**
3) **Done!**

Why does it work?

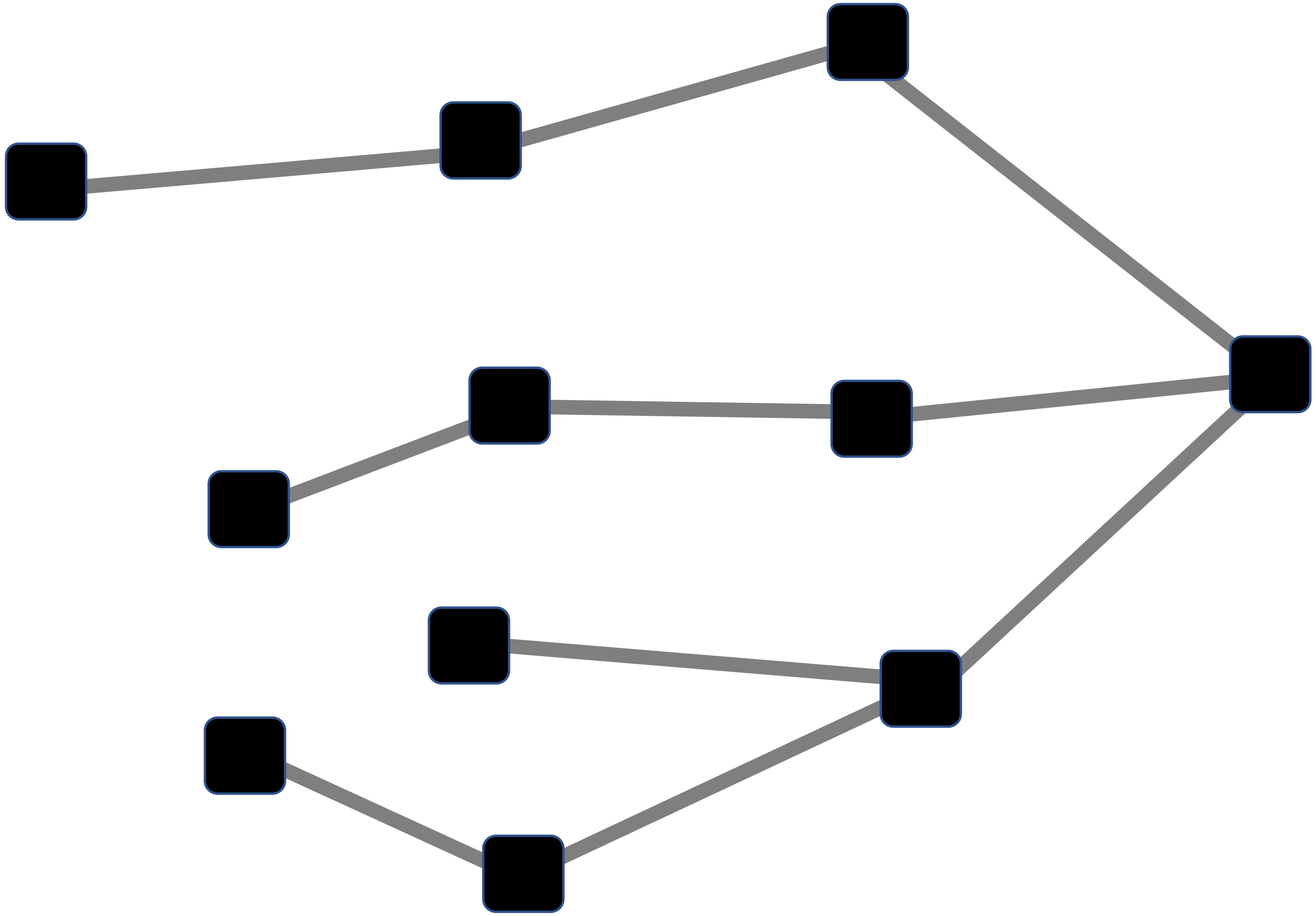
Spanning-trees have only one path

between any two nodes

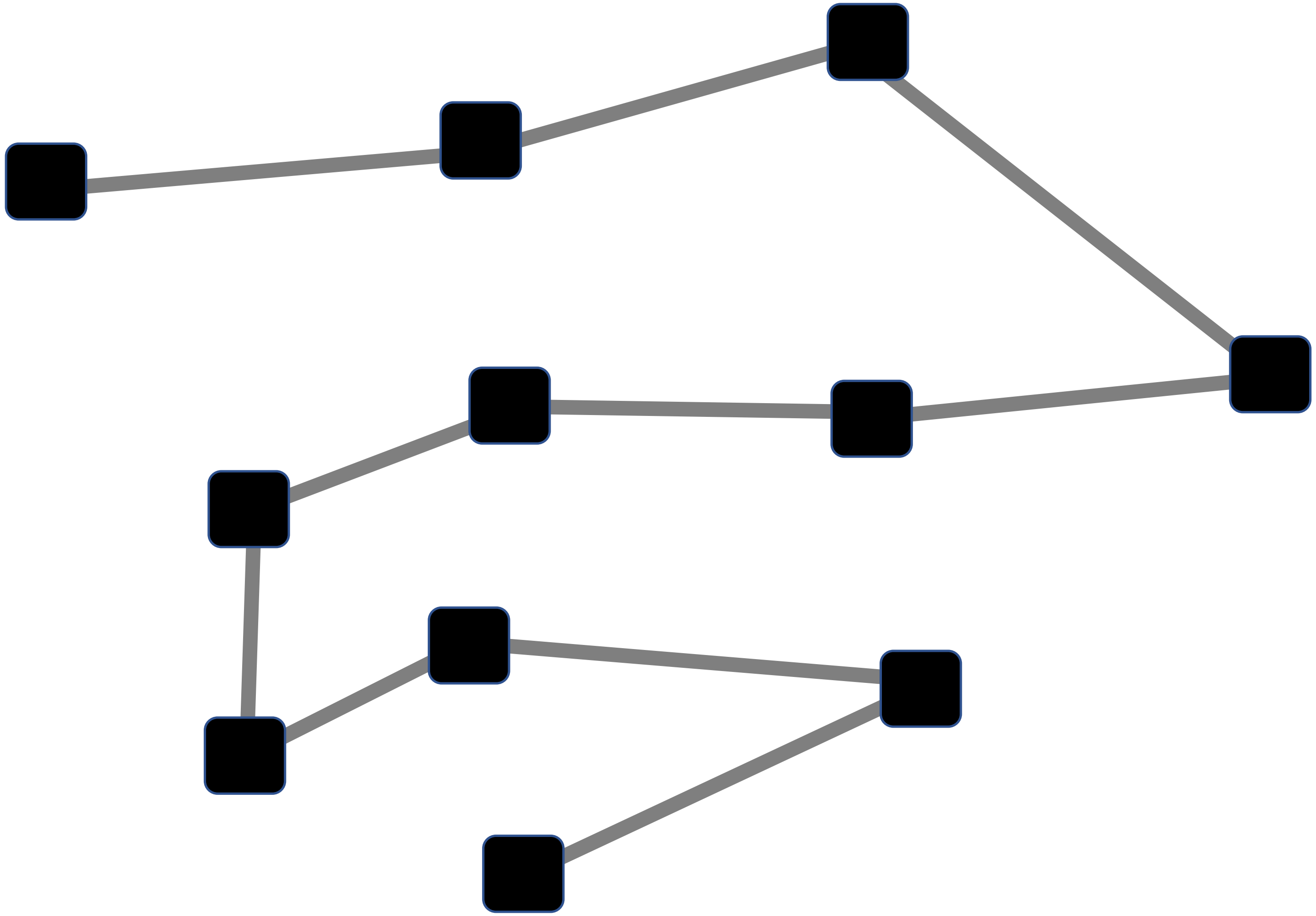
In practice, there can be many spanning-trees for a given topology



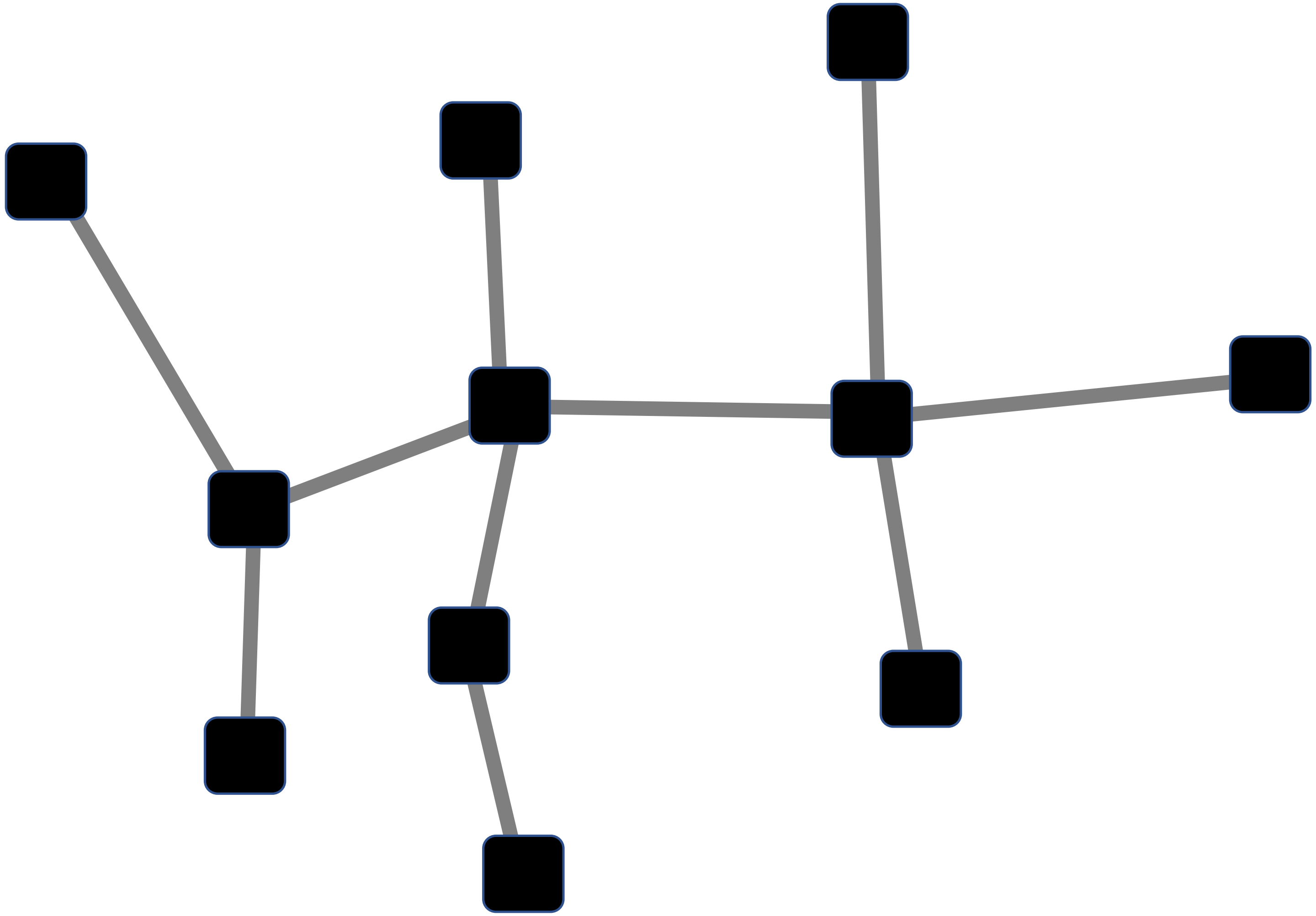
Spanning tree #1



Spanning tree #2



Spanning tree #3

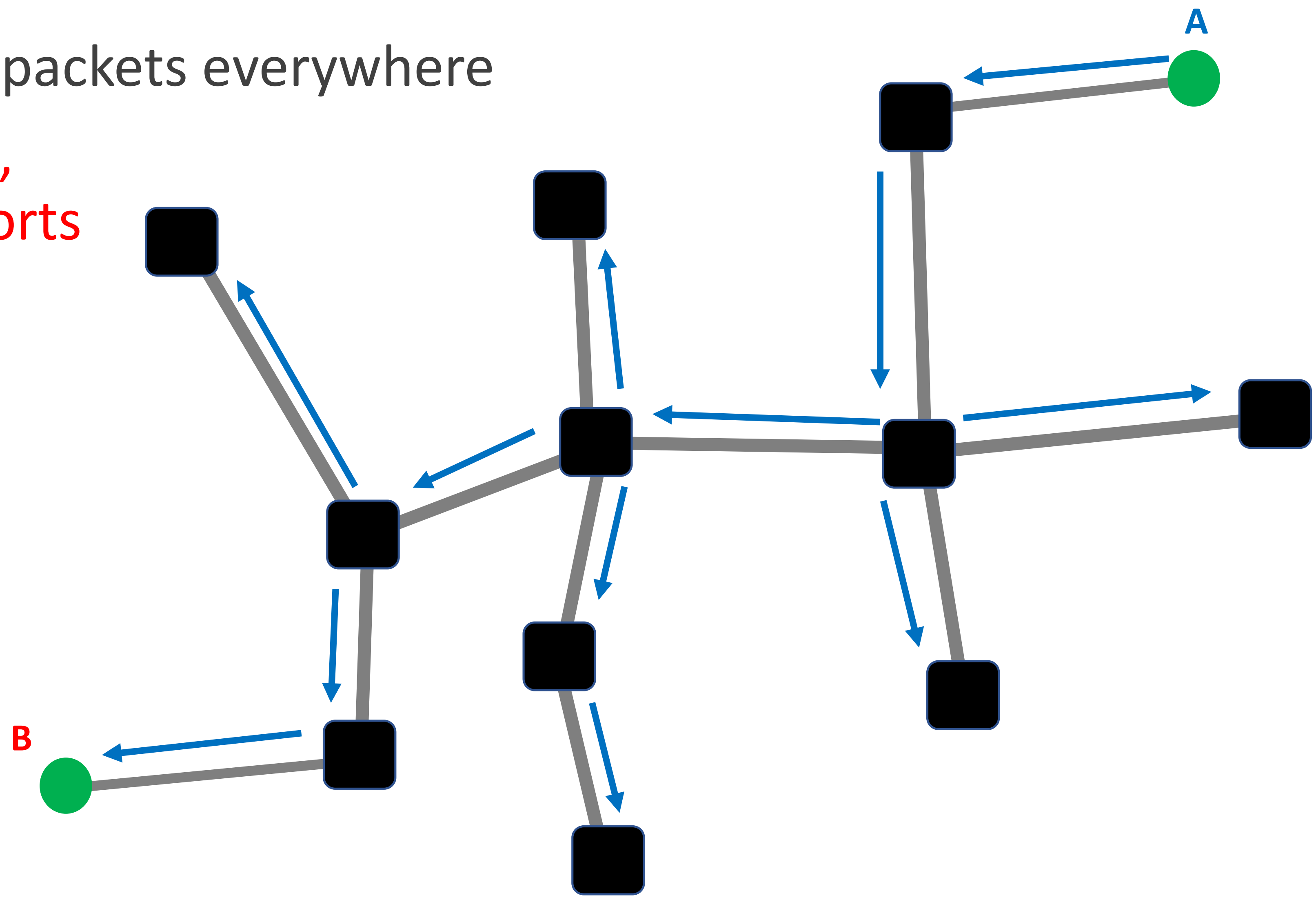


We’ll see how to compute spanning-trees in 2 weeks.
For now, assume it is possible

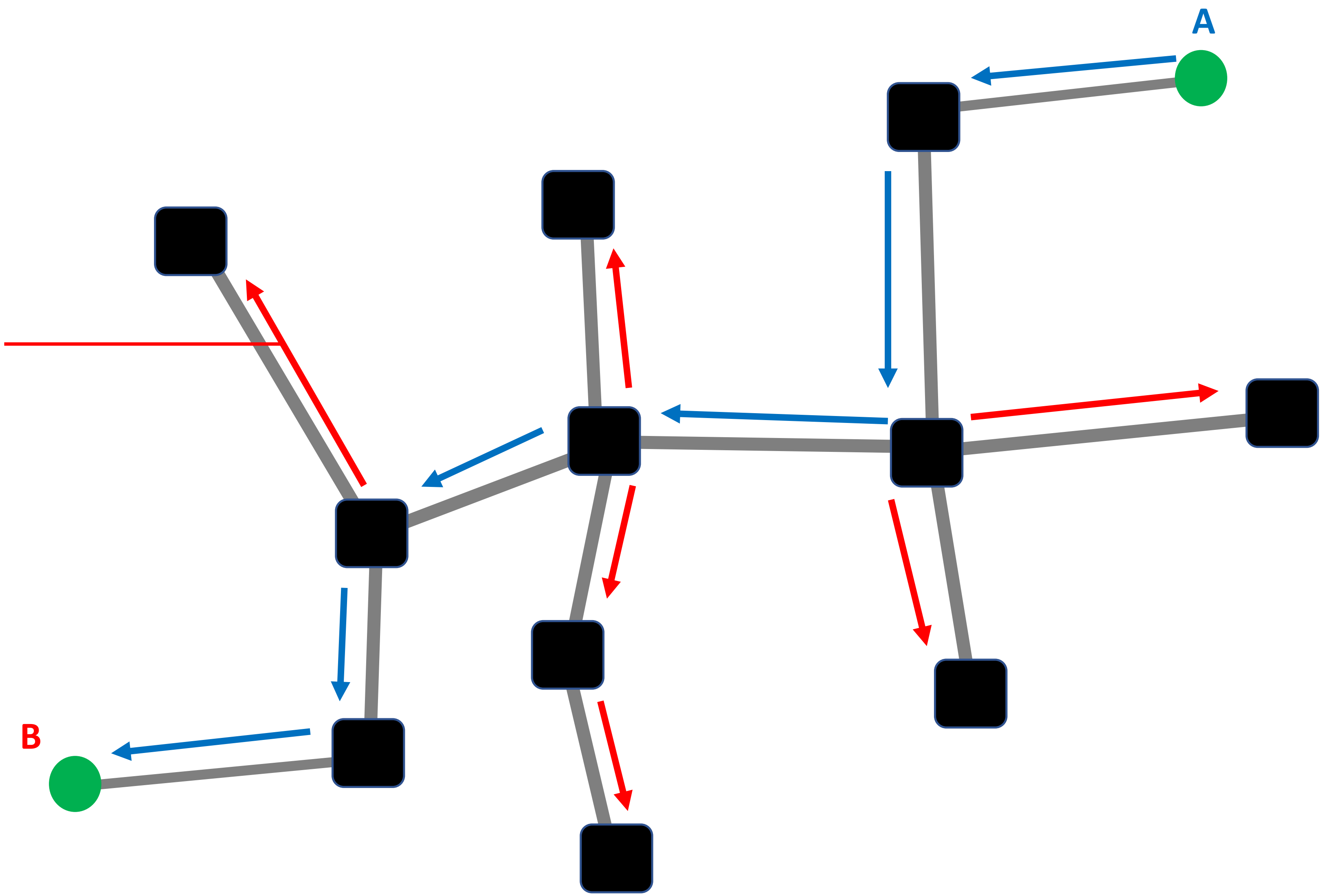
Once we have a spanning tree, forwarding on it is easy

Literally just flood the packets everywhere

When a packet arrives,
simply send it on all ports



Flooding is quite wasteful



useless transmissions

Problem

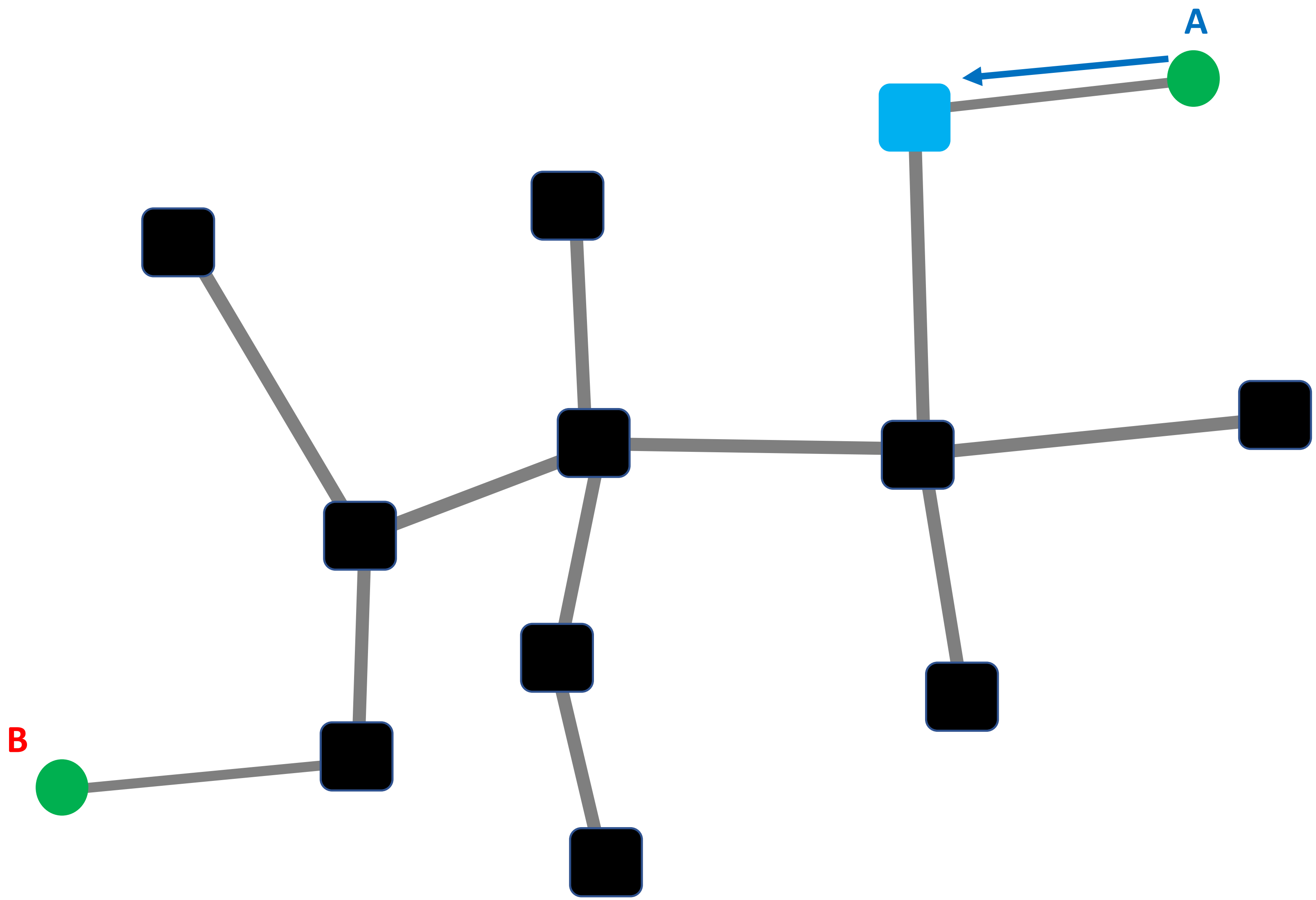
The issue is that nodes do not know their respective locations

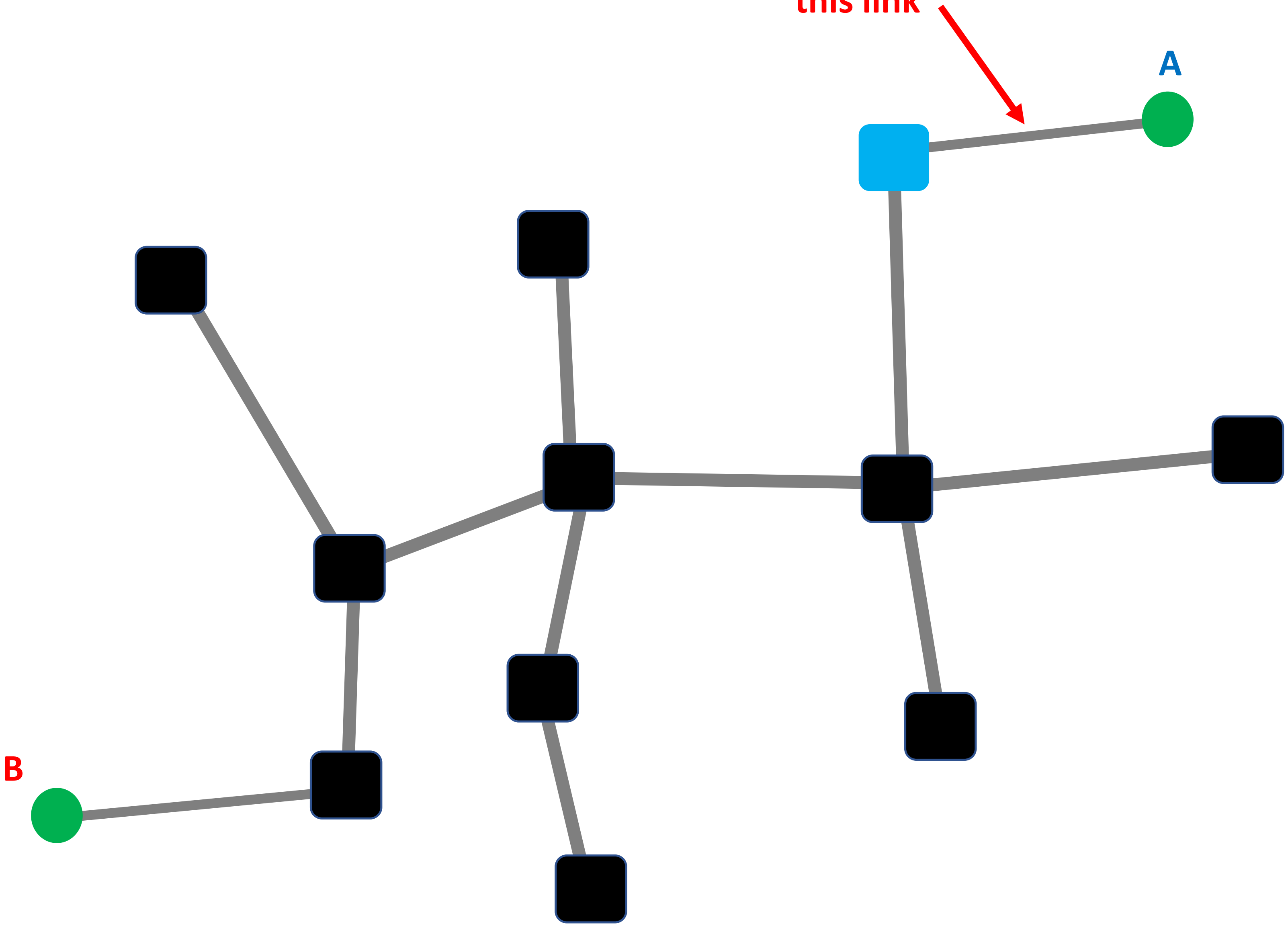
Solution

Nodes can learn how to reach nodes by remembering where packets came from

Intuition ***if*** *flood packet from node A entered switch X on port 4*

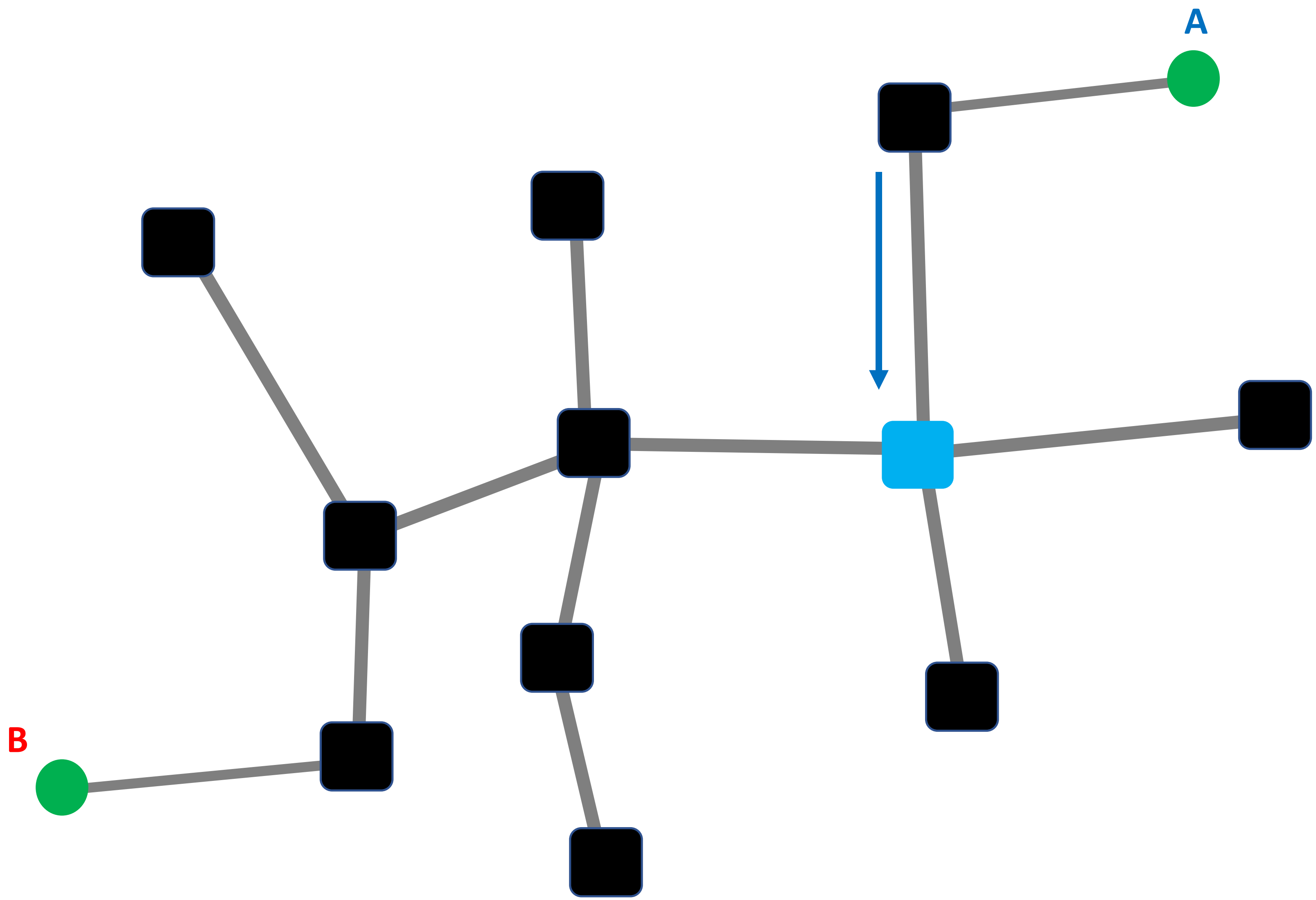
then *switch X can use port 4 to reach node A*


A
B

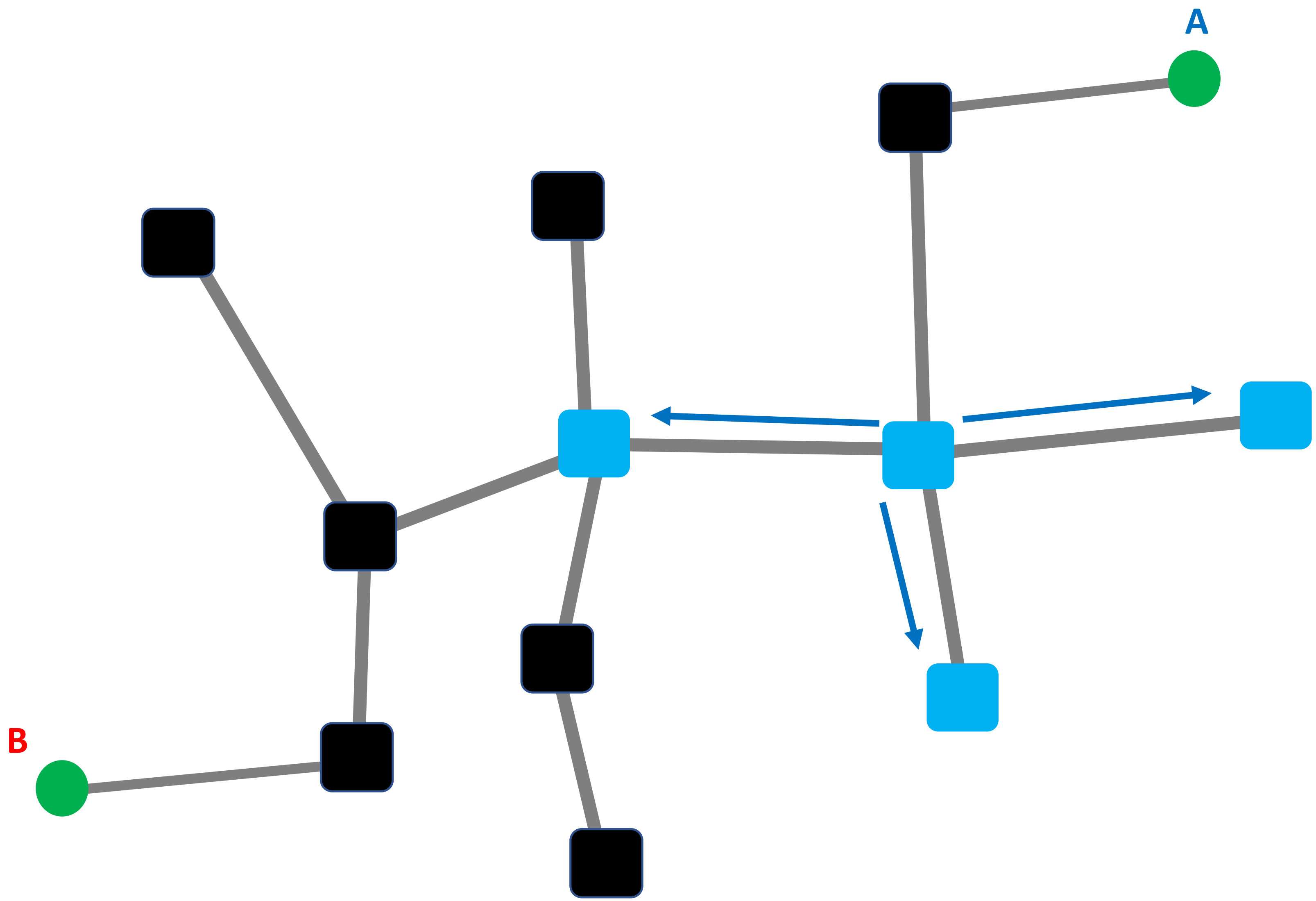


Node A can be reached through this link

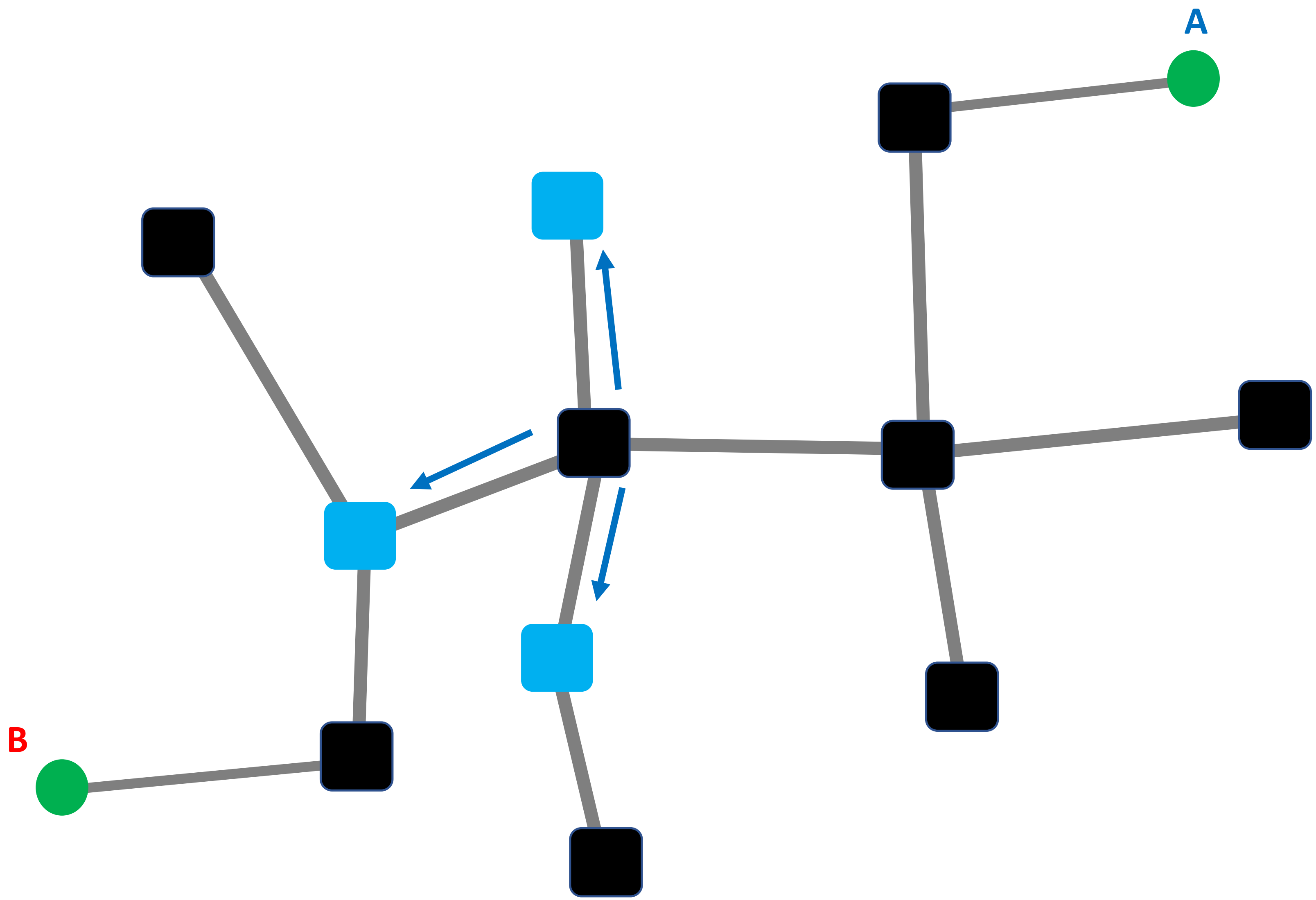
Blue nodes learn
how to reach node A



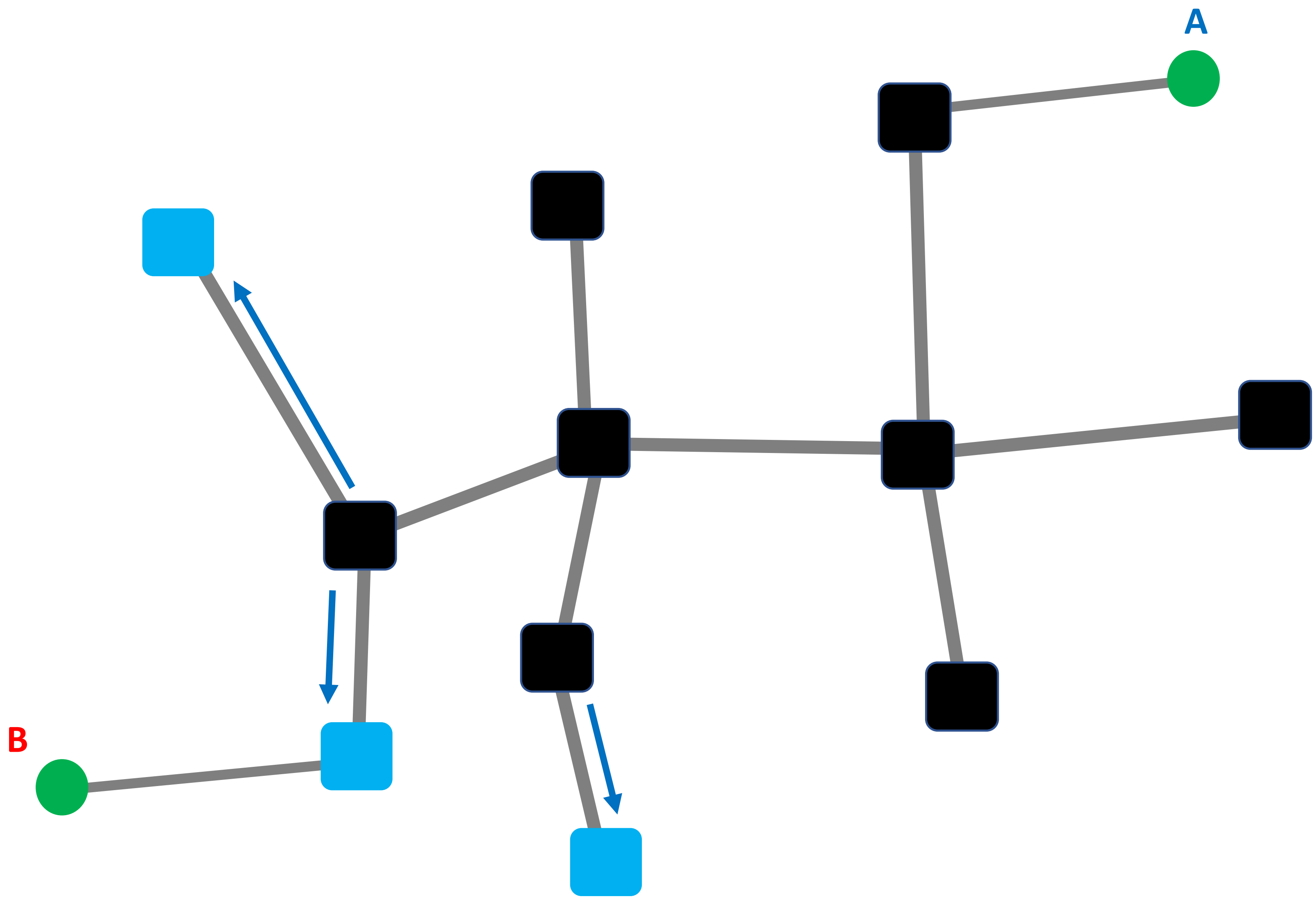
Blue nodes learn how to reach node A



Blue nodes learn how to reach node A

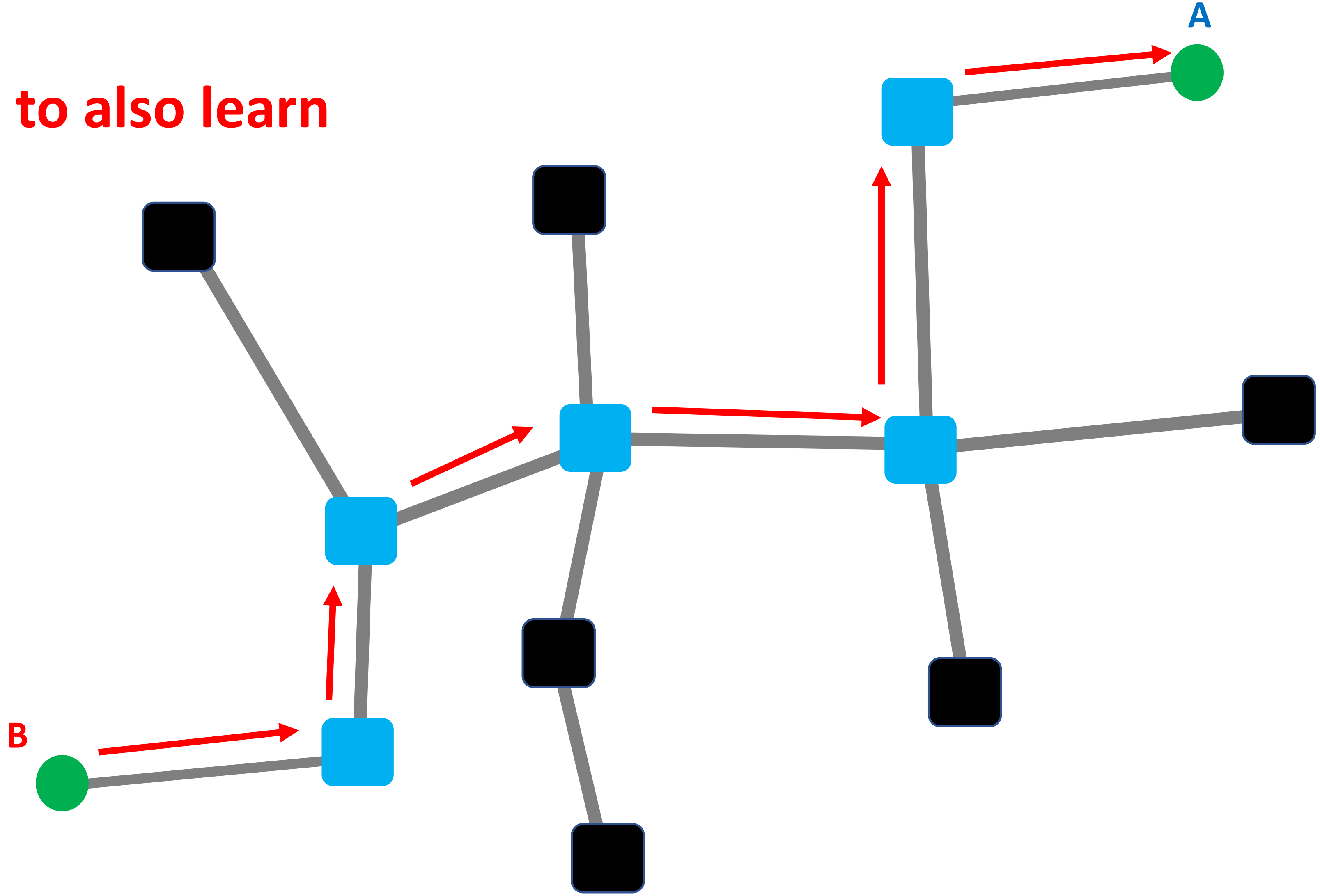


All nodes know how to reach node A



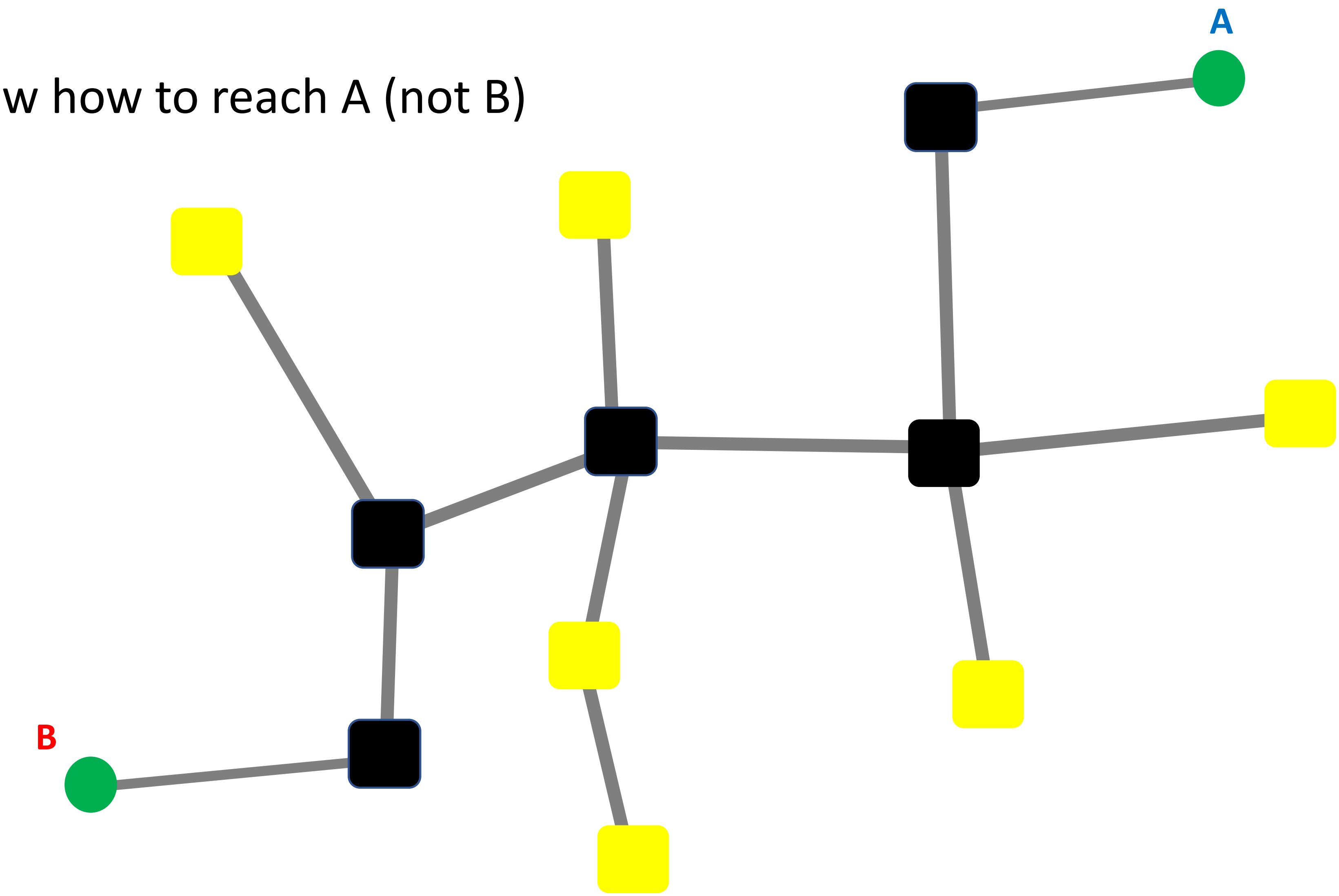
B answers back to A
No need for flooding anymore

enabling the blue nodes to also learn where B is



Learning is topology- dependent

The yellow nodes only know how to reach A (not B)



Routing by flooding on a spanning-tree
in a nutshell

Flood first packet to node you're trying to reach

all switches learn where you are

When destination answers, some switches learn where it is

some because packet to you is not flooded anymore

The decision to flood or not is done on each switch

depending on who has communicated before

Spanning-tree in practice

used in Ethernet

advantages	disadvantages
plug-and-play	**only use the links of the spanning-tree**
configuration-free	eliminate many links from the topology
	inefficient
automatically adapts to moving host	**slow to react to failures** **slow to react to host movement**

2) **Rely on global network view** *Link-state routing*

If each router knows the entire graph, it can **locally** compute paths to all other nodes

Once a node *u* knows the entire topology,

it can compute shortest-paths using Dijkstra's algorithm

Initialization

```
S = {u}
for all nodes v:
    if (v is adjacent to u):
        D(v) = c(u,v)
    else:
        D(v) = ∞
```

Loop

```
while not all nodes in S:
    add w with the smallest D(w) to S
    update D(v) for all adjacent v (to w) not in S:
        D(v) = min{D(v), D(w) + c(w,v)}
```

If each router knows the entire graph, it can **locally** compute paths to all other nodes

Once a node *u* knows the entire topology,

it can compute shortest-paths using Dijkstra's algorithm

Initialization

u is the node running the algotrithm

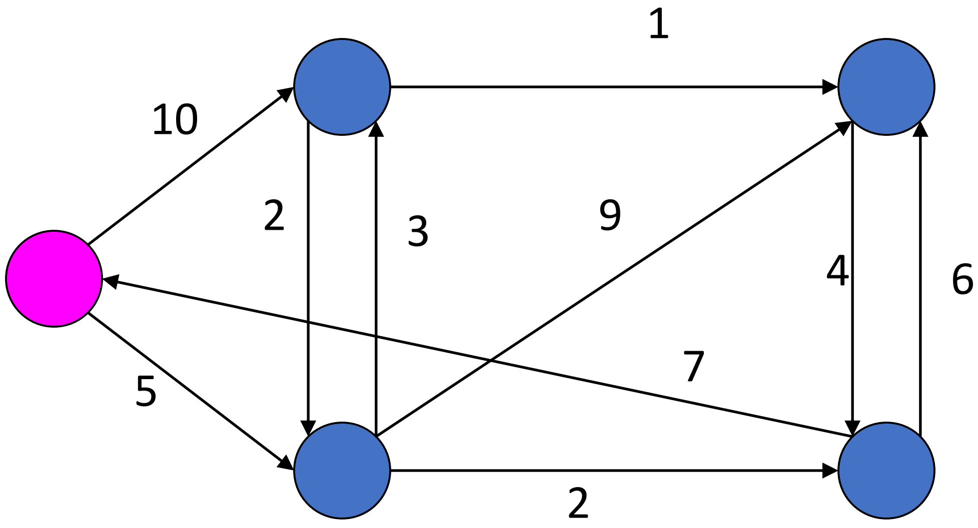
```
S = {u}
while not all nodes in S:
    for all nodes v:
        if (v is adjacent to u):
            $D(v) = c(u,v)$
        else:
            $D(v) = \infty$
```

The weight of link connecting u and v

Loop

```
while not all nodes in S:
    add w with the smallest $D(w)$ to S
    update $D(v)$ for all adjacent v (to w) not in S:
        $D(v) = \min\{D(v), D(w) + c(w,v)\}$
```

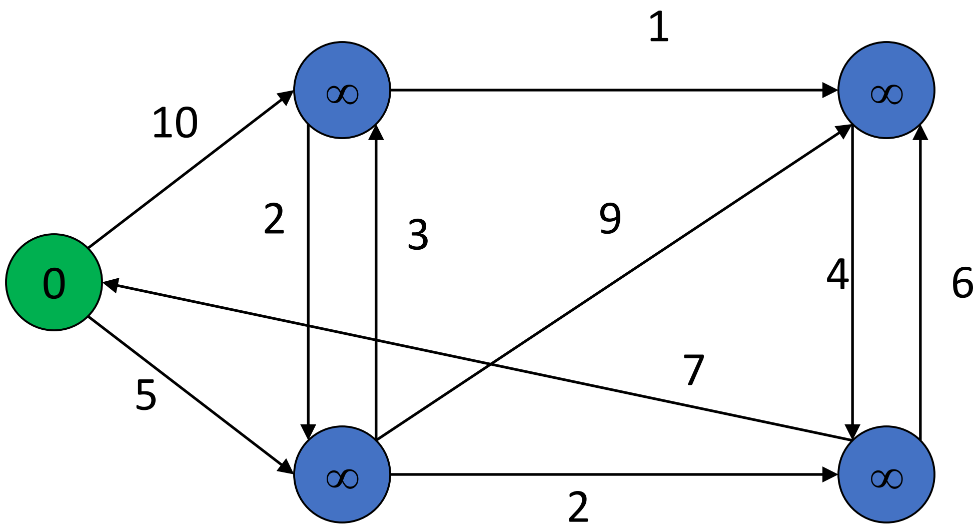
Dijkstra’s Algorithm - Example



S set is marked by green

D values are written inside the nodes

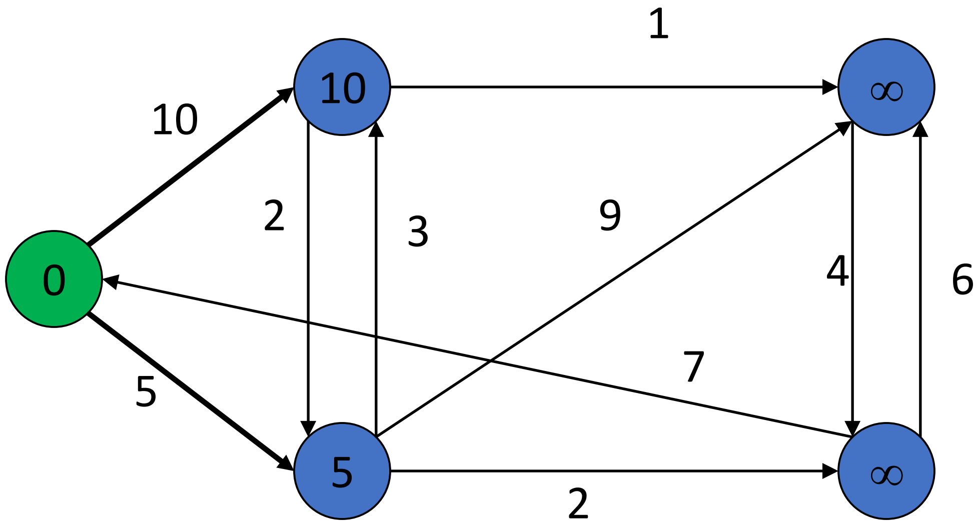
Dijkstra’s Algorithm - Example



S set is marked by green

D values are written inside the nodes

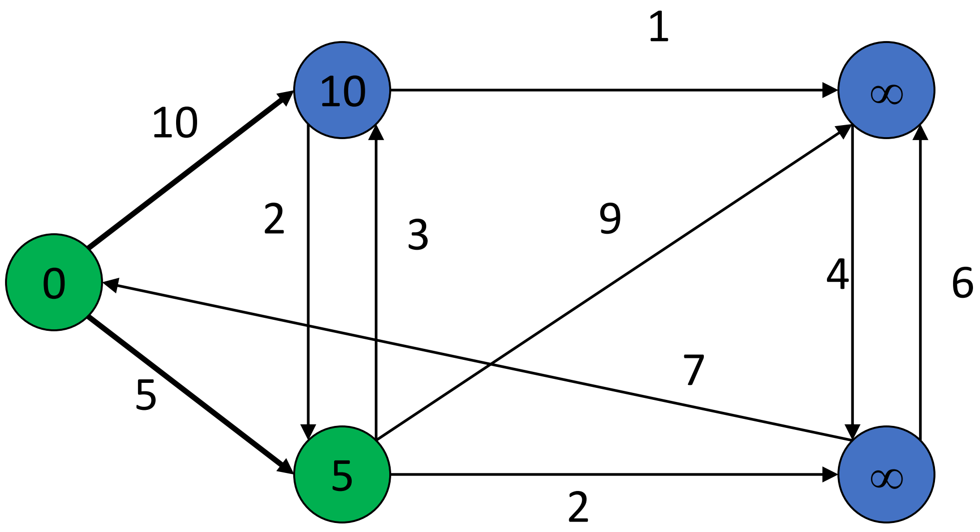
Dijkstra’s Algorithm - Example



S set is marked by green

D values are written inside the nodes

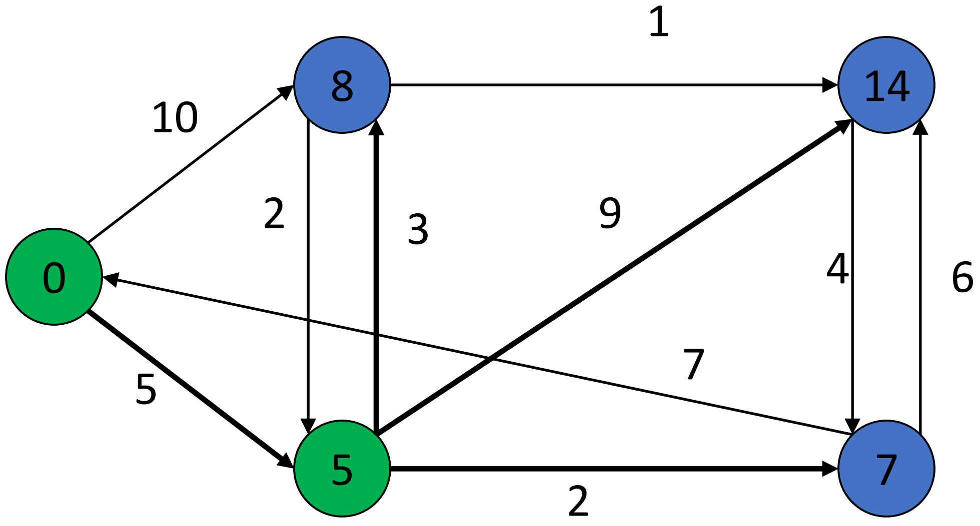
Dijkstra’s Algorithm - Example



S set is marked by green

D values are written inside the nodes

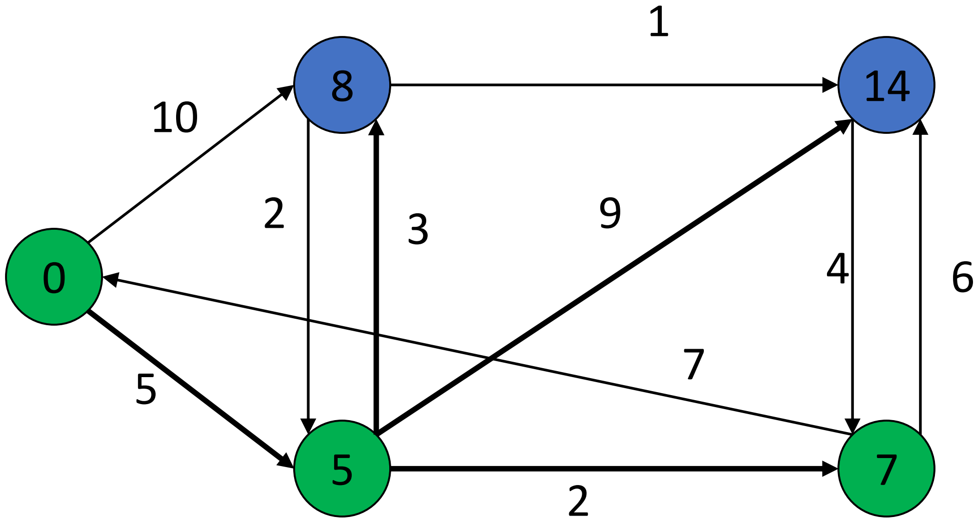
Dijkstra’s Algorithm - Example



S set is marked by green

D values are written inside the nodes

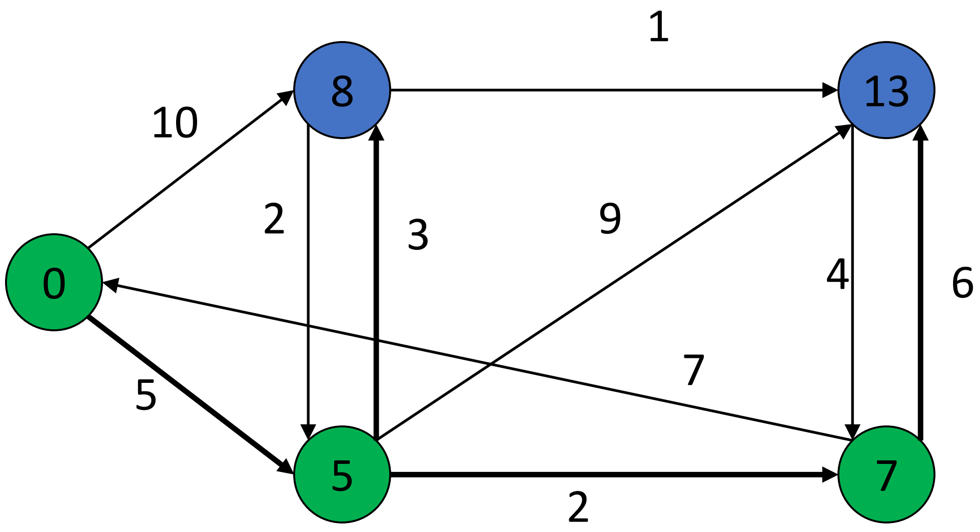
Dijkstra’s Algorithm - Example



S set is marked by green

D values are written inside the nodes

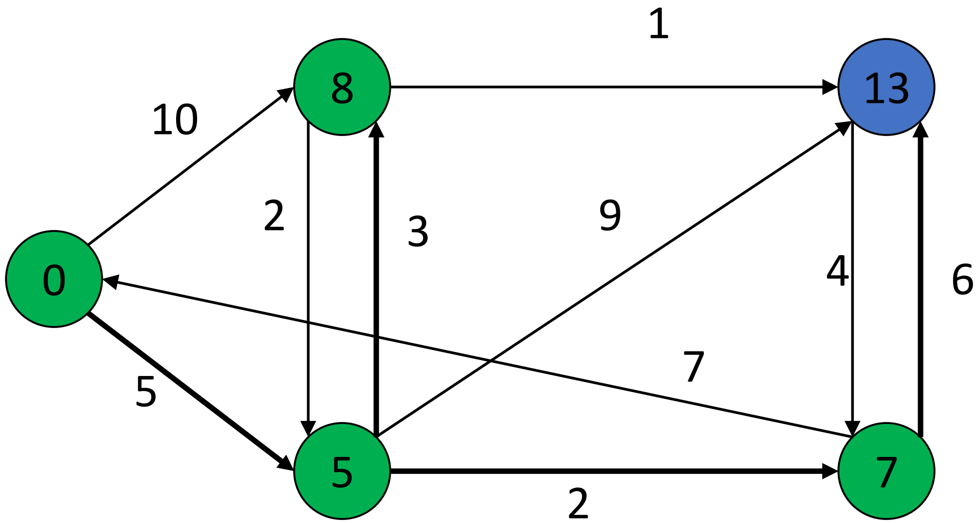
Dijkstra's Algorithm - Example



S set is marked by green

D values are written inside the nodes

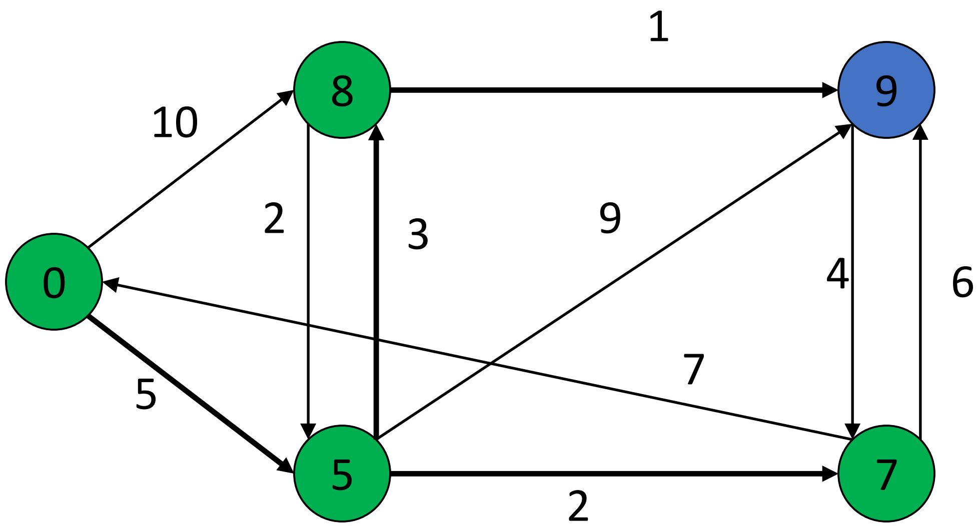
Dijkstra’s Algorithm - Example



S set is marked by green

D values are written inside the nodes

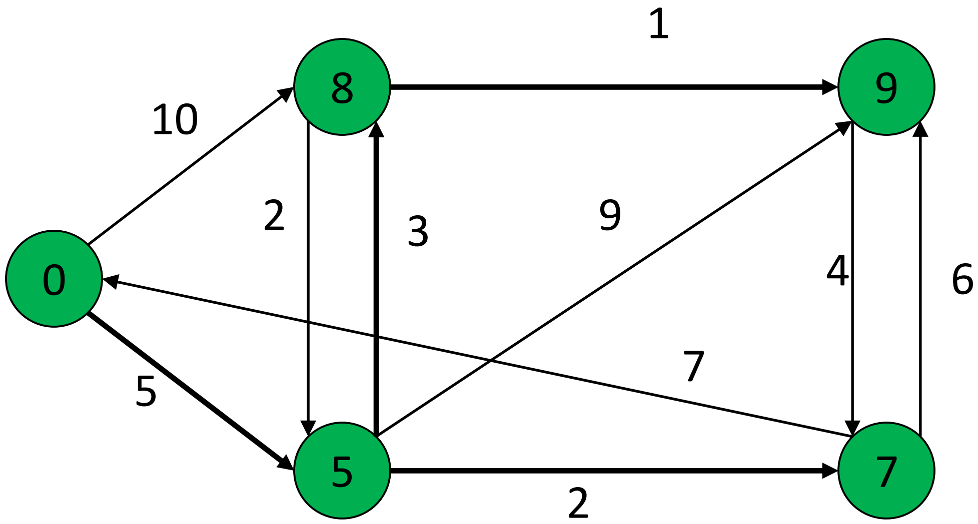
Dijkstra's Algorithm - Example



S set is marked by green

D values are written inside the nodes

Dijkstra's Algorithm - Example



S set is marked by green

D values are written inside the nodes

This algorithm has a $O(n^2)$ complexity

where n is the number of nodes in the graph

iteration #1	search for minimum through n nodes
iteration #2	search for minimum through n-1 nodes
iteration #n	search for minimum through 1 node

$\frac{n(n+1)}{2}$ operations => $O(n^2)$

This algorithm has a *O*(n^2) complexity

where n is the number of nodes in the graph

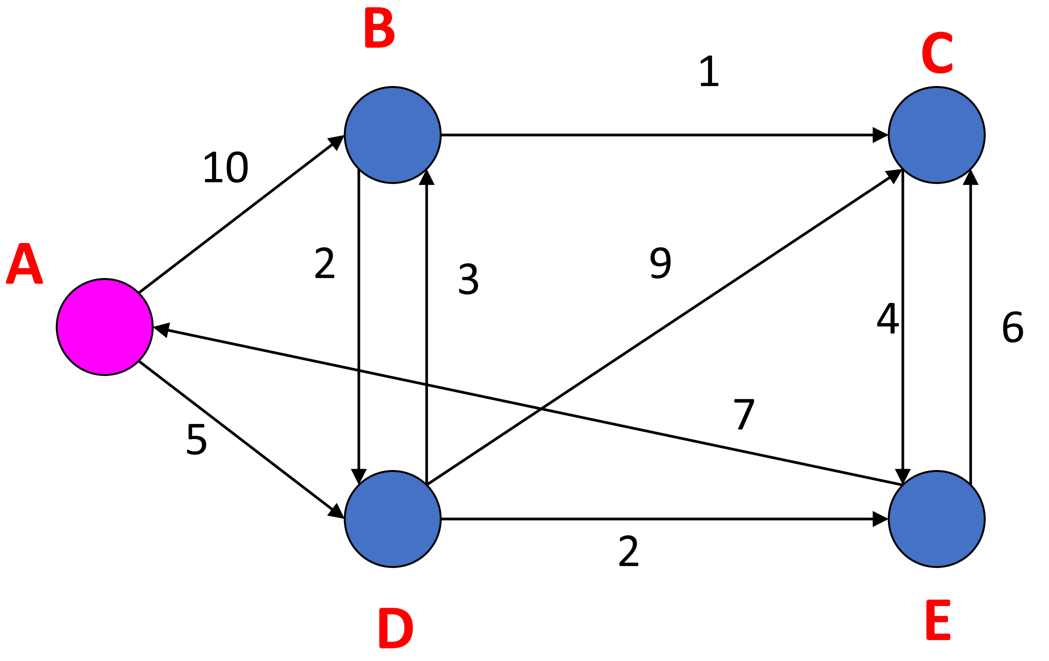
iteration #1	search for minimum through n nodes
iteration #2	search for minimum through n-1 nodes
iteration #n	search for minimum through 1 node

$\frac{n(n+1)}{2}$ operations => *O*(n^2)

Better implementations rely on a heap to find the next node to expand, bringing down the complexity to O(*n* log *n*)

Building a global view is essentially equal to solving jigsaw puzzle

Initially,
routers only know their ID and their neighbors

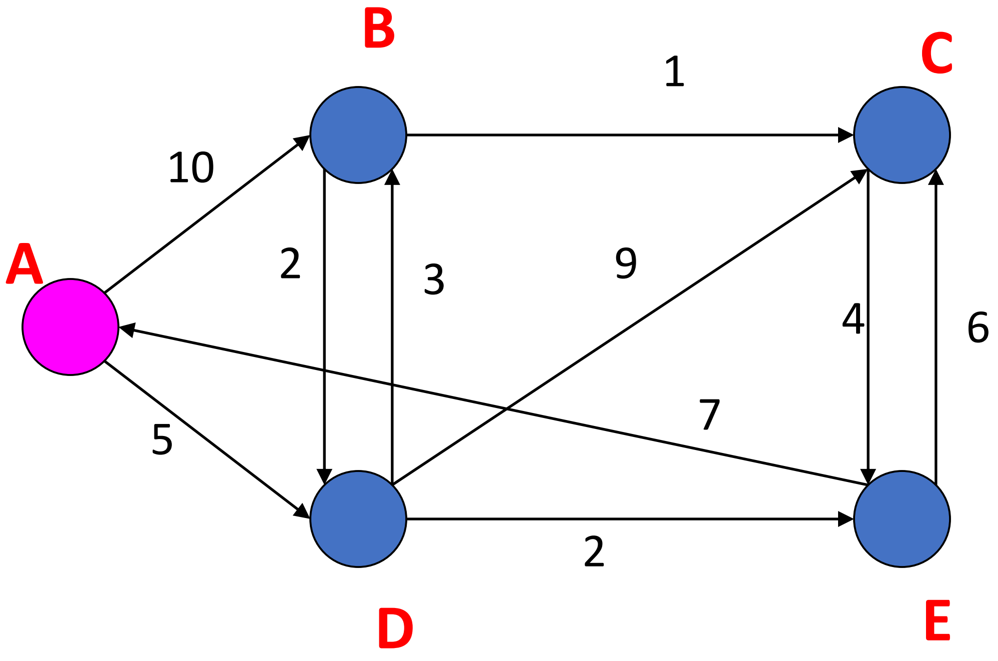


Node A only knows:

A) it is connected to B and D

B) the weigths to reach them (by configuration).

Each routers builds a message (known as Link-State Advertisement (LSA)) and floods it (reliably) in the entire network



Node A’s advertisement

edge(A,B); cost=10

edge(A,D); cost=5

At the end of the flooding process,

everybody share the exact same view of the network

Dijkstra will always converge to a unique stable state when run on static weights

Dynamically changing weights can lead to oscillations

The problem of oscillation is fundamental to congestion-based routing with local decisions

Solution #1	**Use static weights** i.e. don’t do congestion-aware routing
Solution #2	**Use randomness to break self-synchronization** `wait(random(0,50ms)); send(new_link_weight);`
Solution #3	**Have the routers agree on the paths to use** essentially meaning to rely on circuit-switching

3) Rely on distributed computation *Distance-vector routing*

Instead of locally compute paths based on the graph, paths can be computed in a distributed fashion

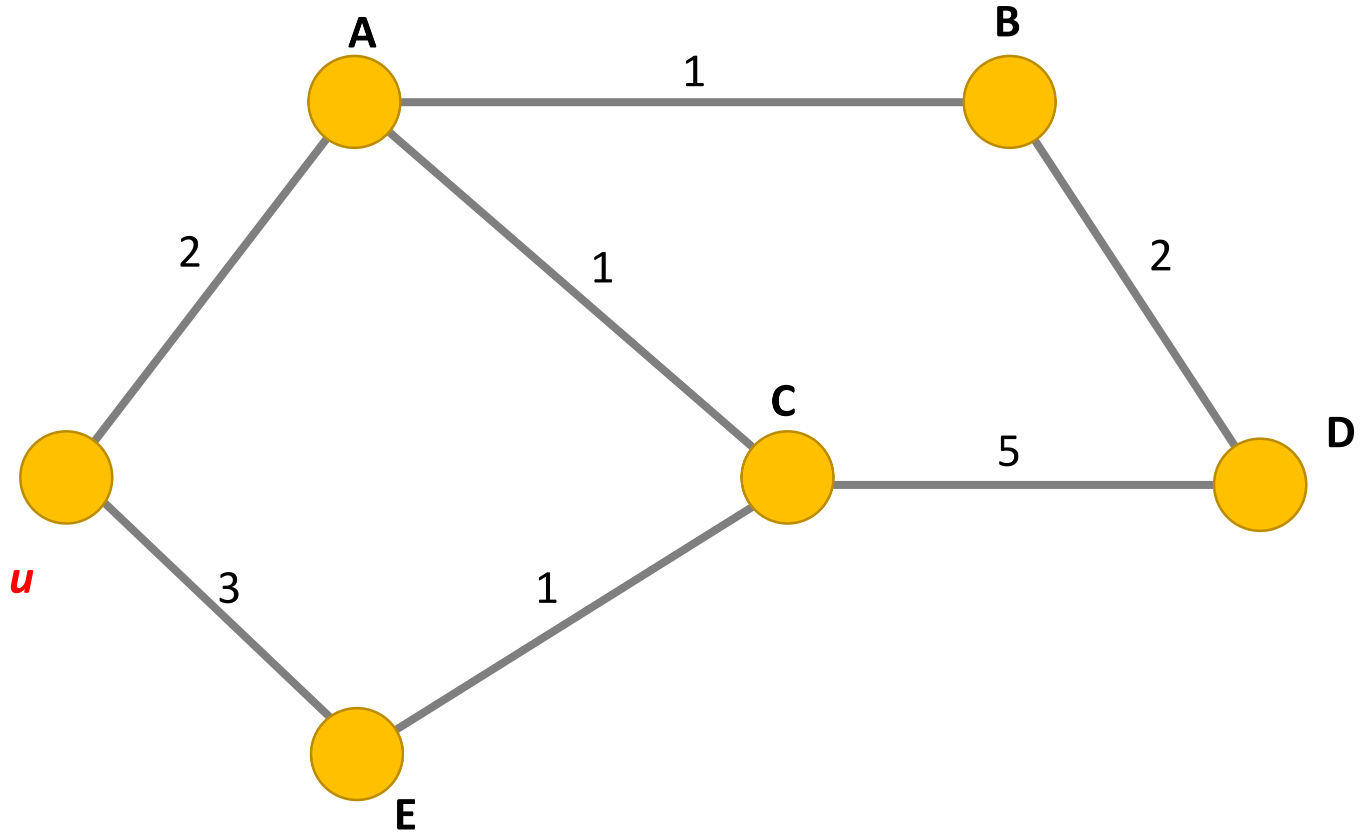
Let **$d_x(y)$** be the cost of the least-cost path known by x to reach y

1) Each node bundles these distances into one message (called a vector) that it repeatedly sends until convergence to all its neighbors

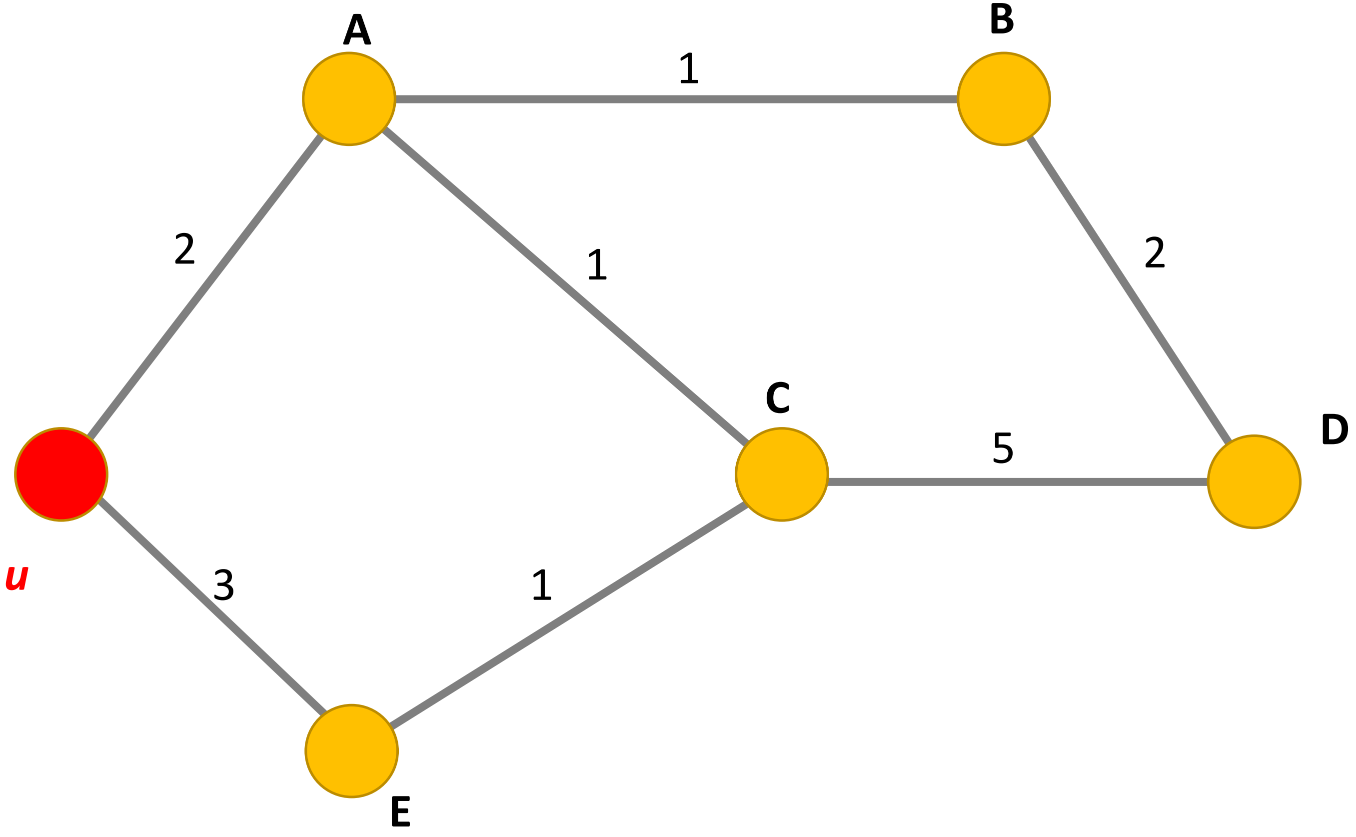
2) Each node updates its distances based on neighbors' vectors:

$$d_x(y) = \min\{\ c(x,v) + d_v(y)\ \}$$

Let’s compute the shortest-path from *u* to **D**



The values computed by a node *u* depends on what it learns from its neighbors (**A** and **E**)

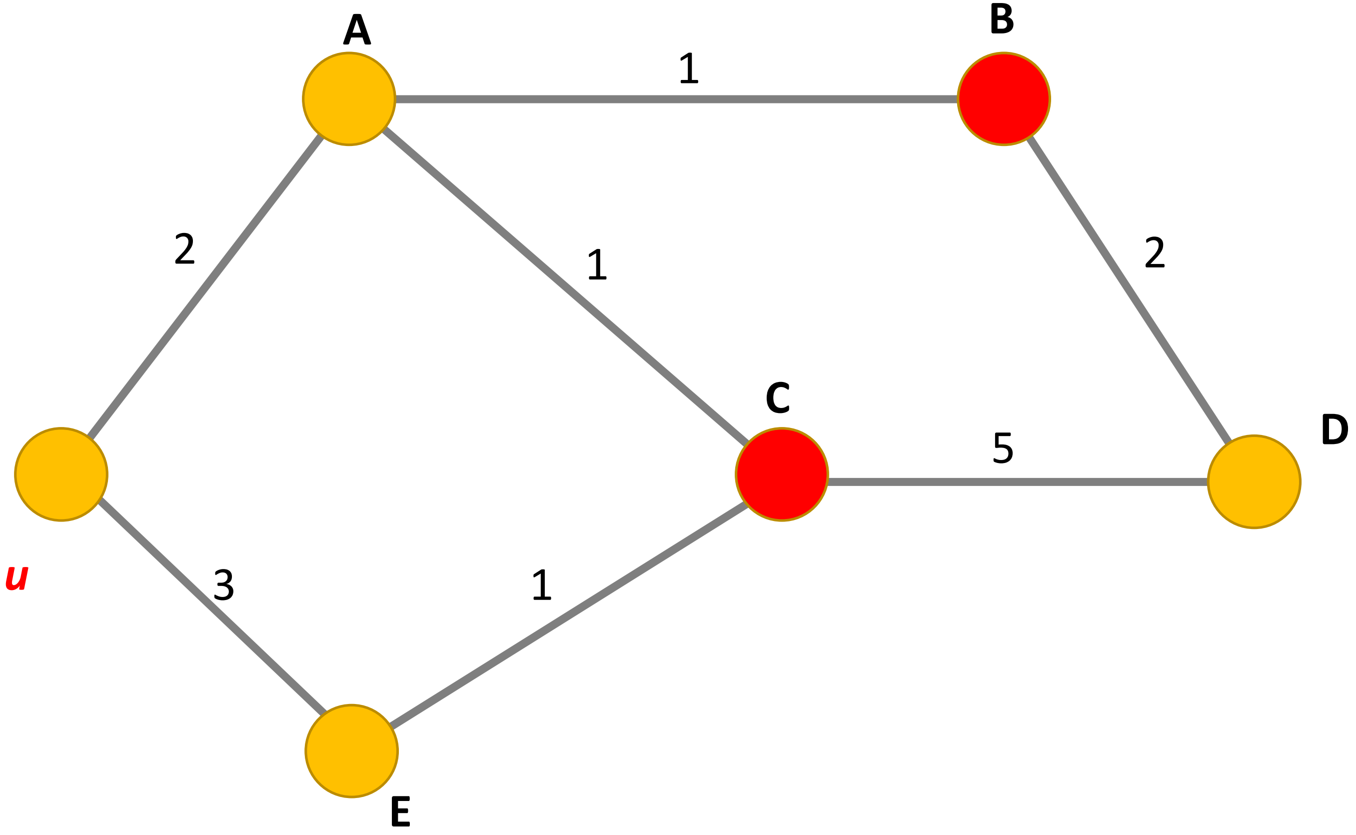


$d_x(y) = \min\{ c(x,v) + d_v(y) \}$
over all neighbors *v*

Now:

$d_u(\mathbf{D}) = \min\{ c(u, \mathbf{A}) + d_A(\mathbf{D}),$
$c(u, \mathbf{E}) + d_E(\mathbf{D}) \}$

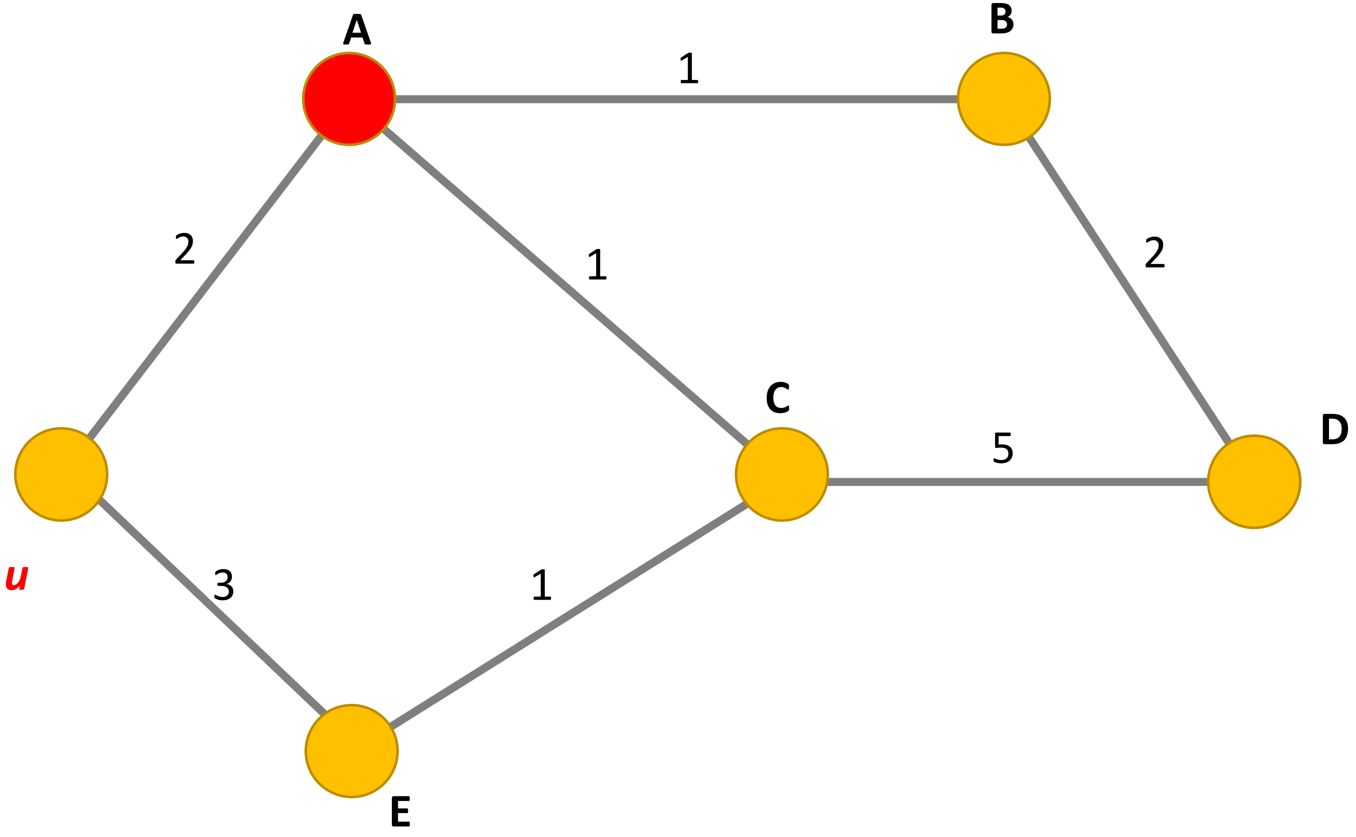
To unfold the recursion, let's start with the direct neighbor of **D**



$d_B(\mathbf{D}) = 2$

$d_C(\mathbf{D}) = 5$

B and **C** announce their vector to their neighbors, enabling **A** to compute its shortest-path

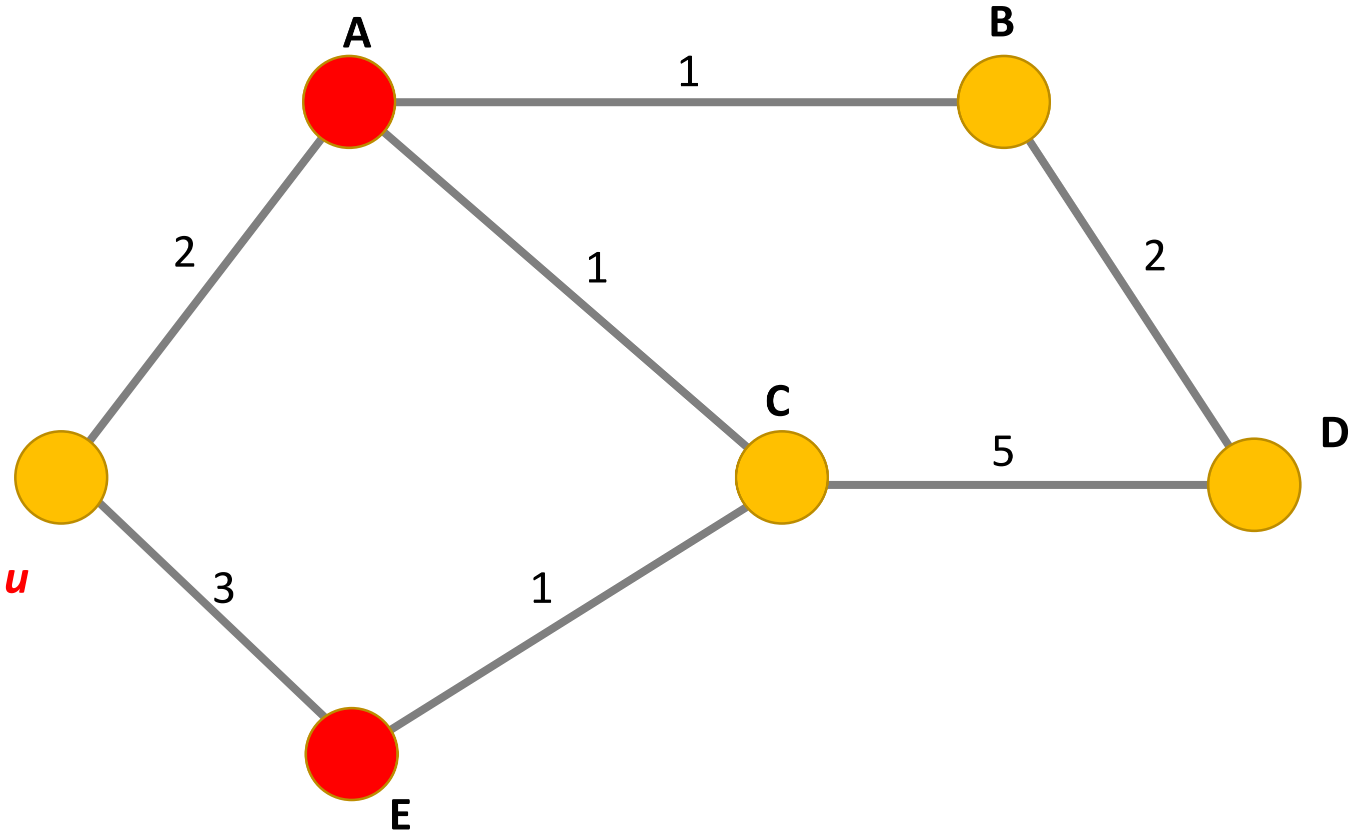


$d_A(\mathbf{D}) = \min \{\, 1 + d_B(\mathbf{D}),$
$1 + d_C(\mathbf{D}) \,\}$

$= \min \{\, 1 + 2,\ 1 + 5 \,\}$

$= 3$

B and **C** announce their vector to their neighbors, enabling **A** to compute its shortest-path



$$d_A(\mathbf{D}) = \min \{\, 1 + d_B(\mathbf{D}),\ 1 + d_C(\mathbf{D}) \,\}$$

$$= \min \{\, 1 + 2,\ 1 + 5 \,\}$$

$$= 3$$

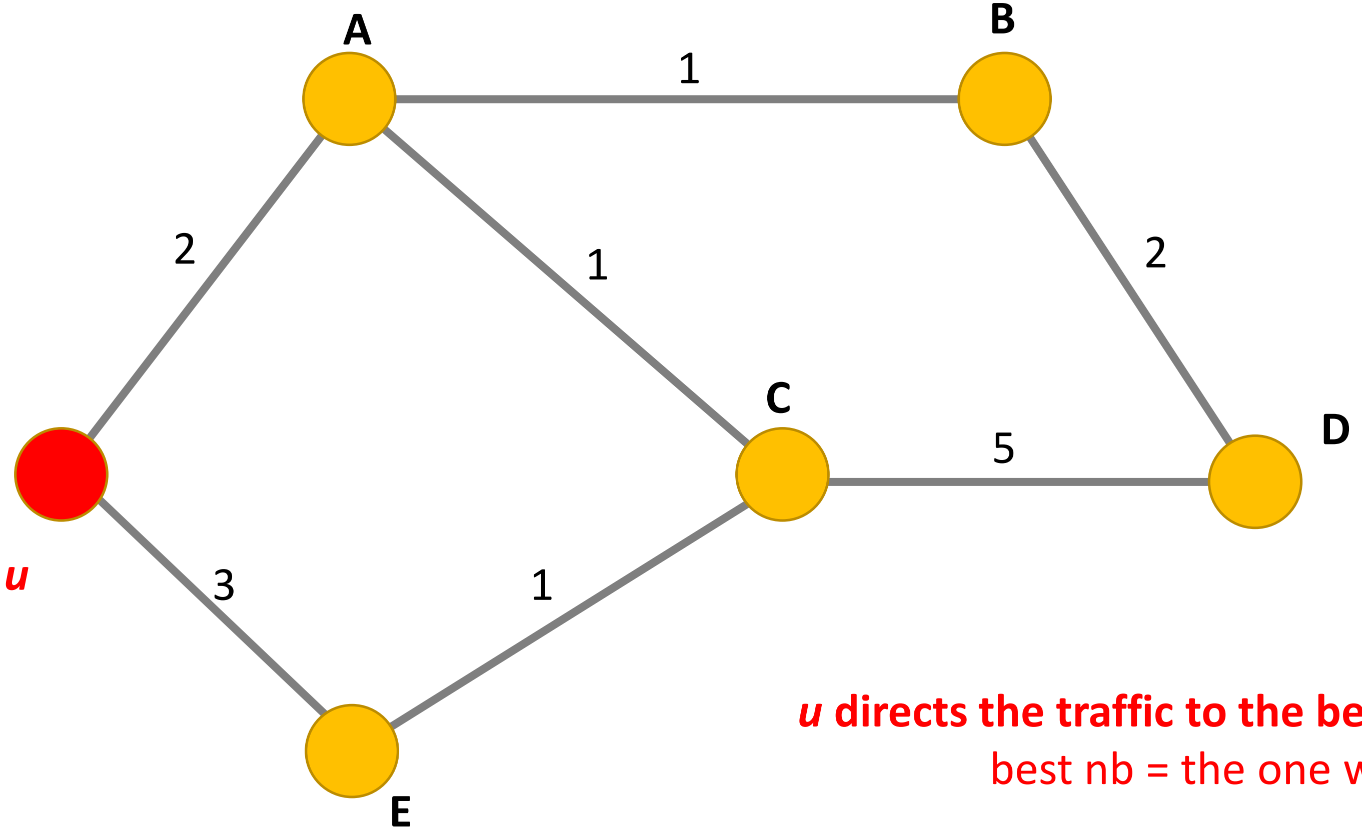
$$d_E(\mathbf{D}) = \min \{\, 1 + d_C(\mathbf{D}) \,\}$$

$$= \min \{\, 1 + 5 \,\}$$

$$= 6$$

As soon as a distance vector changes, each node propagates it to its neighbor

Eventually, the process converges to the shortest-path distance to each destination



$d_u(\mathbf{D}) = \min\{\, 2 + d_A(\mathbf{D}),$
$3 + d_E(\mathbf{D}) \,\}$

$= \min\{\, 2 + 3,\ 3 + 6 \,\}$

$= 5$

***u* directs the traffic to the best neighbor**

best nb = the one with the smallest cost in the forwarding table

Evaluating the complexity of DV is harder, we'll get back to that in a couple of weeks

To be continued…