

Computer Networks

Jiaqi Zheng

Material with thanks Mosharaf Chowdhury, and many other colleagues.

Agenda

- Network layer recap
- Routing fundamentals

ROUTING FUNDAMENTALS

Goal of routing

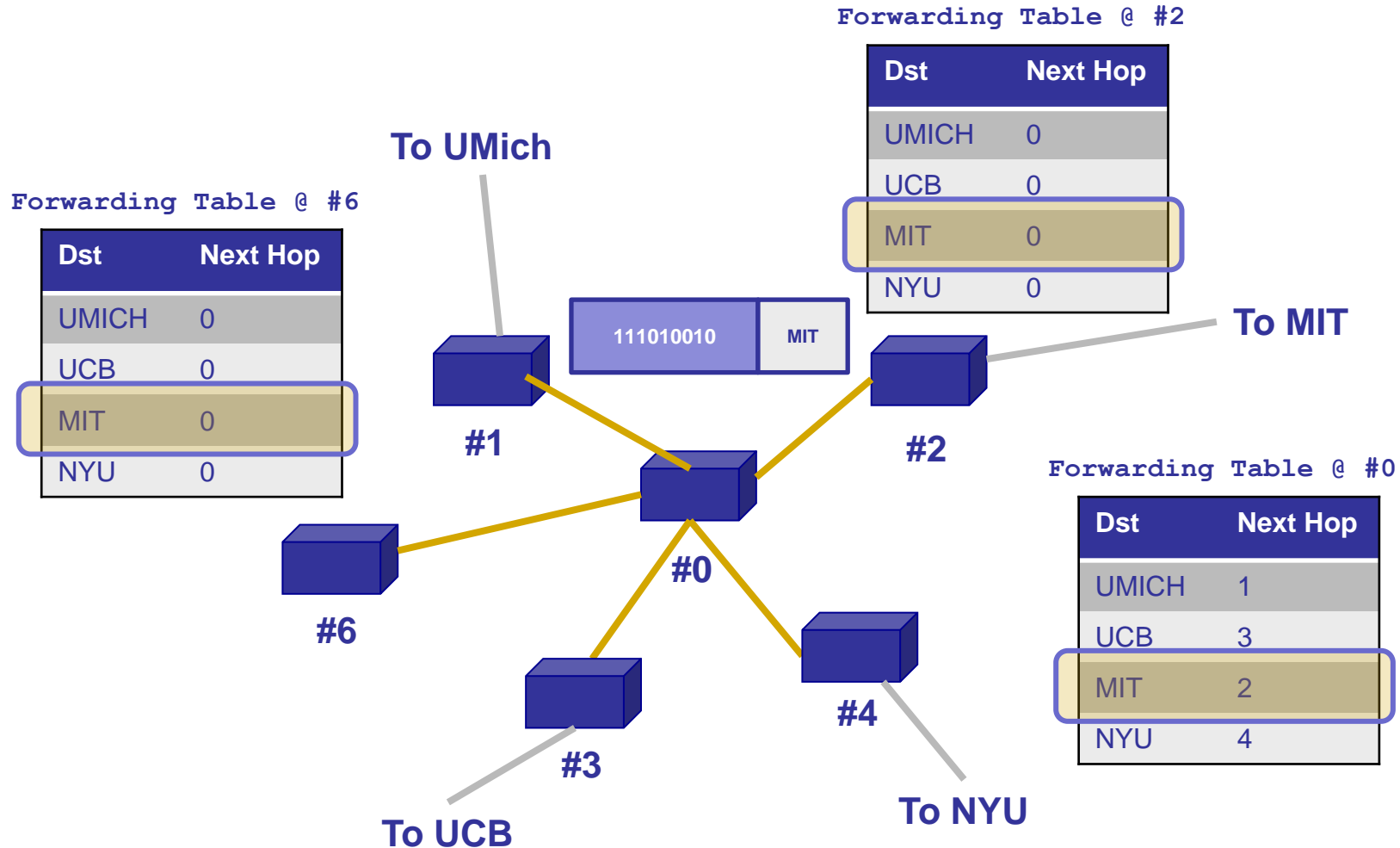
- Find a path to a given destination
- How do we know that the state contained in forwarding tables meets our goal?
 - This is what “**validity**” of routing state tells us
 - [This is non-standard terminology]

Local vs. global view of state

- *Local* routing state is the forwarding table in a single router
 - By itself, the state in a single router cannot be evaluated
 - It must be evaluated in terms of the global context

Example:

Local vs. global view of state



Local vs. global view of state

- *Local* routing state is the forwarding table in a single router
 - By itself, the state in a single router cannot be evaluated
 - It must be evaluated in terms of the global context
- *Global* state refers to the collection of forwarding tables in each of the routers
 - Global state determines which paths packets take
 - (Will discuss later where this routing state comes from)

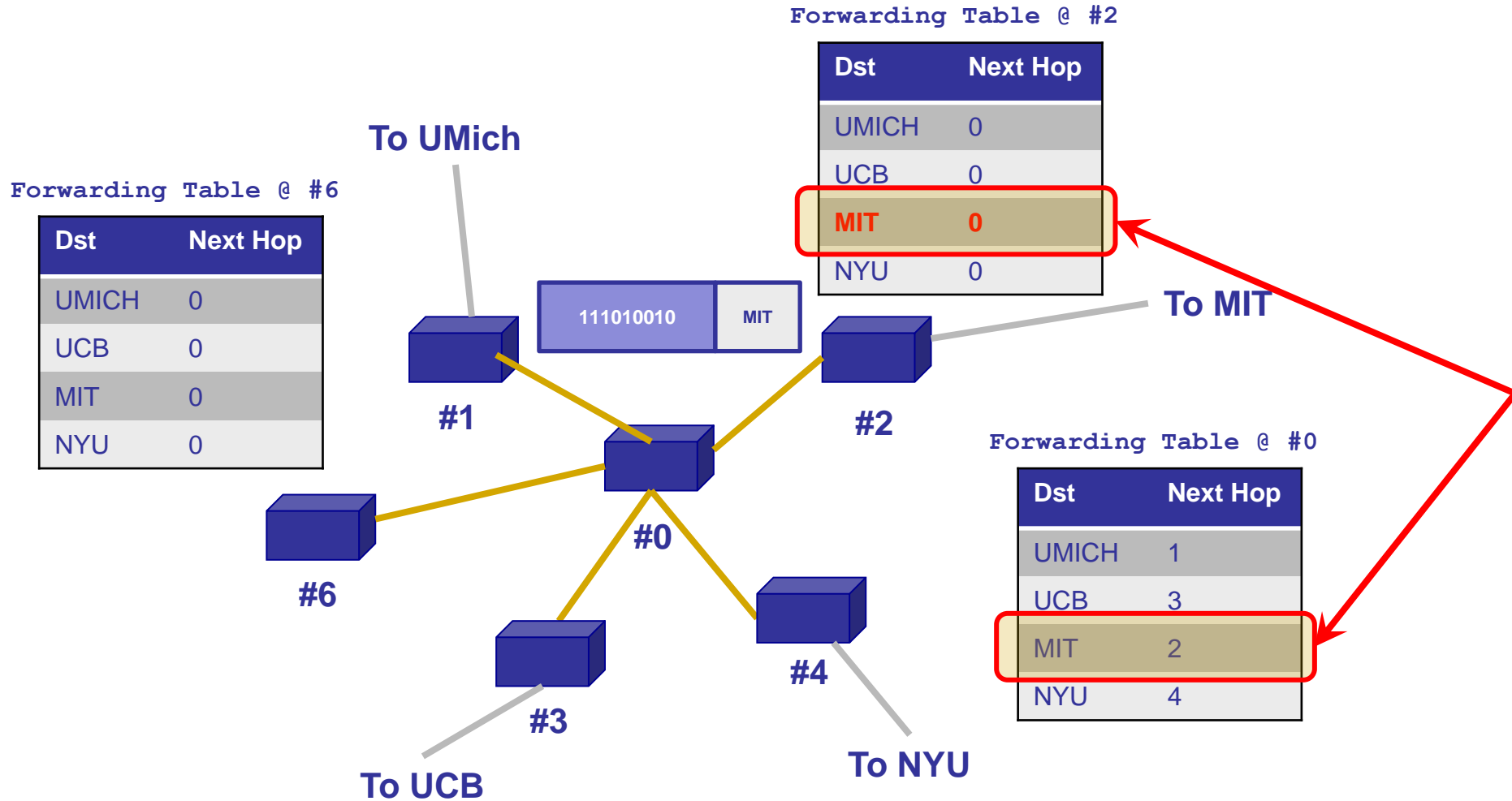
“Valid” routing state

- Global state is “valid” if it produces forwarding decisions that always deliver packets to their destinations
- Goal of routing protocols: compute valid state
 - How can we tell if routing state is valid?
- Need a succinct correctness condition for routing

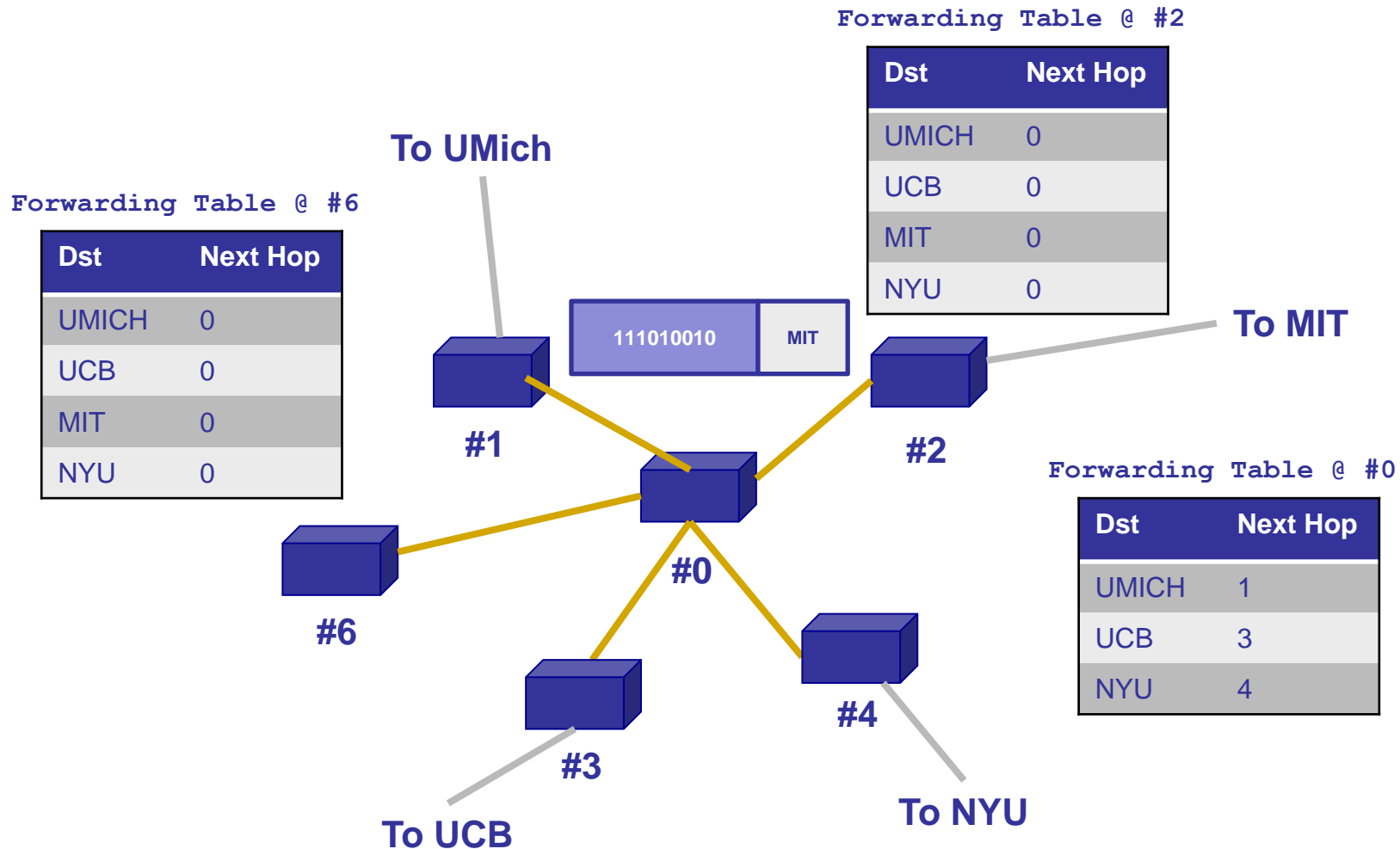
Necessary and sufficient condition

- Global routing state is valid *if and only if*:
 - There are no dead ends (other than destination)
 - There are no loops
- A **dead end** is when there is no outgoing link (next-hop)
 - A packet arrives, but the forwarding decision does not yield any outgoing link
- A **loop** is when a packet cycles around the same set of nodes forever

Loop!



Dead end to MIT @ #0



Necessary and sufficient condition

- Global routing state is valid *if and only if*:
 - There are no dead ends (other than destination)
 - There are no loops

Necessary (“only if”)

- If you run into a dead end before hitting destination,
 - you’ll never reach the destination
- If you run into a loop,
 - you’ll never reach destination

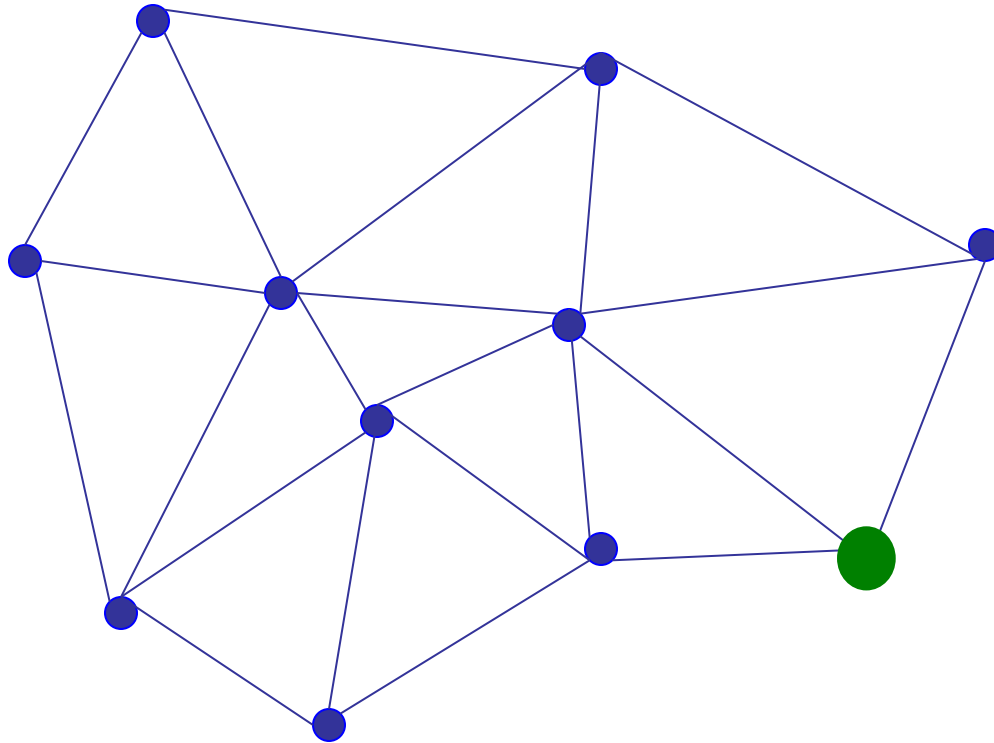
Sufficient (“if”)

- Assume there are no dead ends and no loops
- Packet must keep wandering, but without repeating
 - If ever enter same switch from same link, will loop
- Only a finite number of possible links for it to visit
 - It cannot keep wandering forever without looping
 - Must eventually hit destination

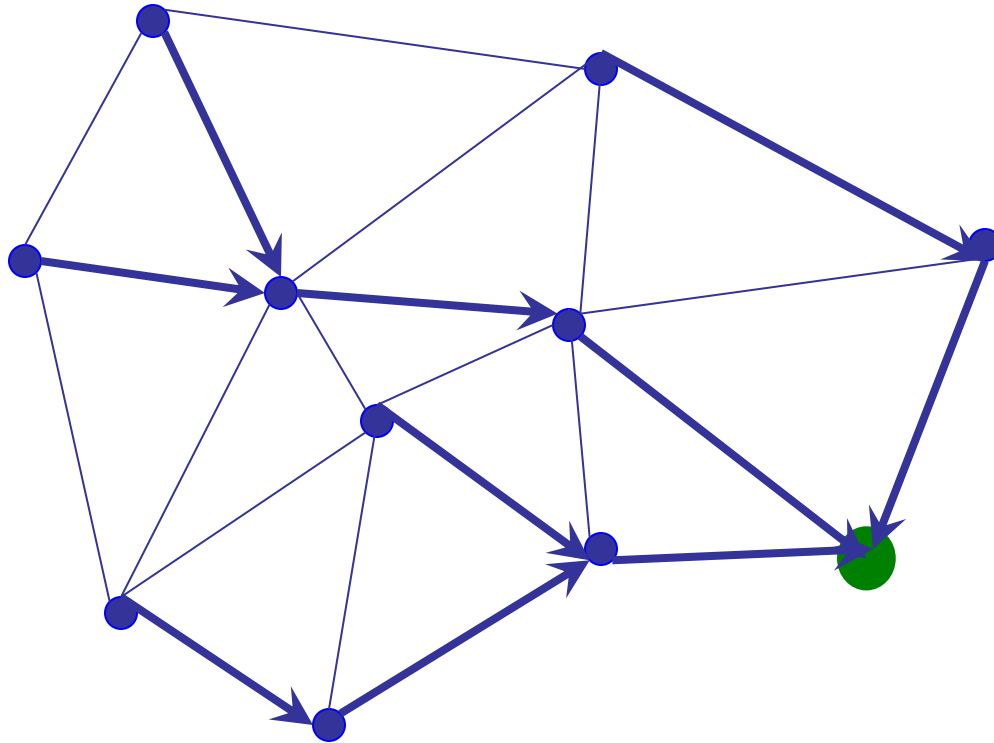
Checking validity of routing state

- Focus only on a single destination
 - Ignore all other routing state
- Mark outgoing link (“next hop”) with arrow
 - There is only one at each node
- Eliminate all links with no arrows
- Look at what’s left

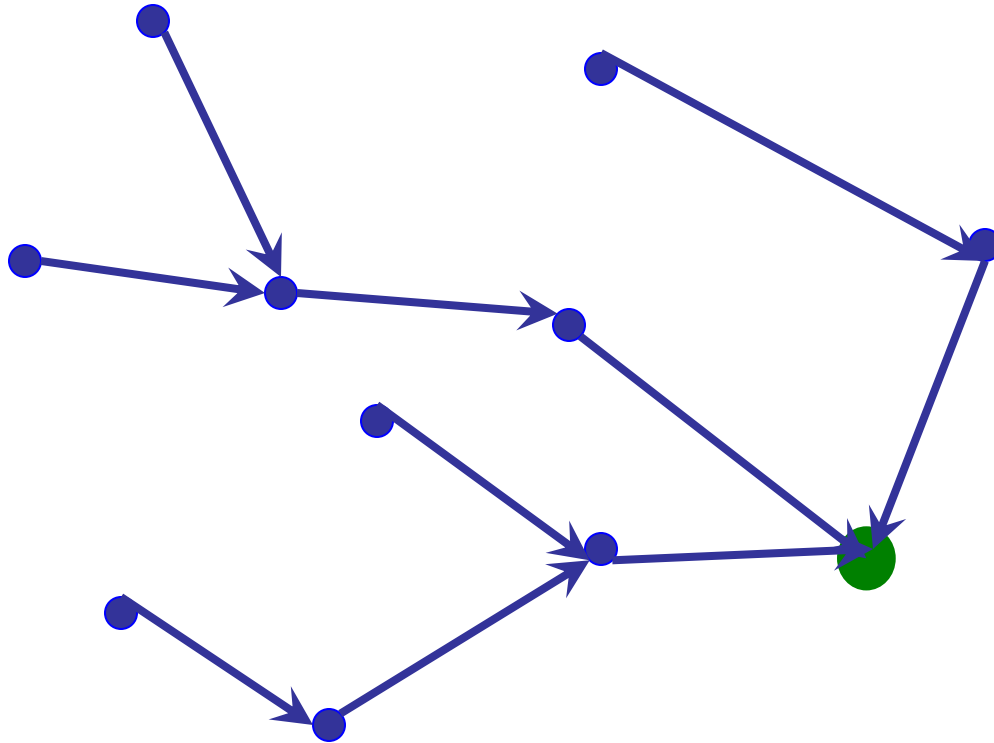
Pick destination



Put arrows on outgoing links (to green dot)

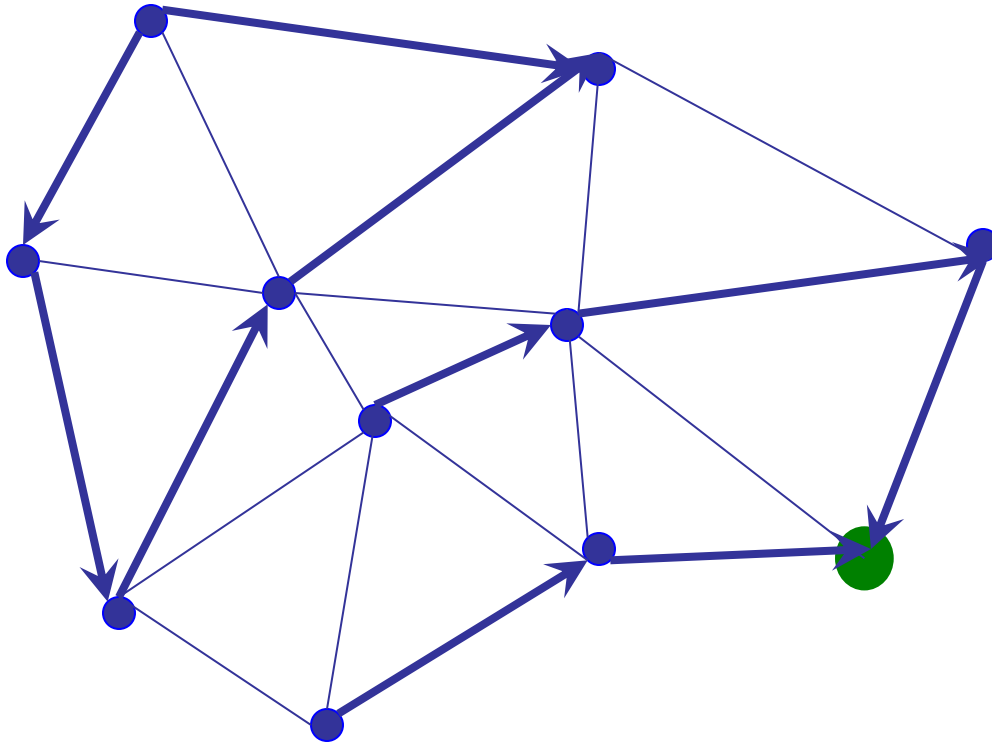


Remove unused links



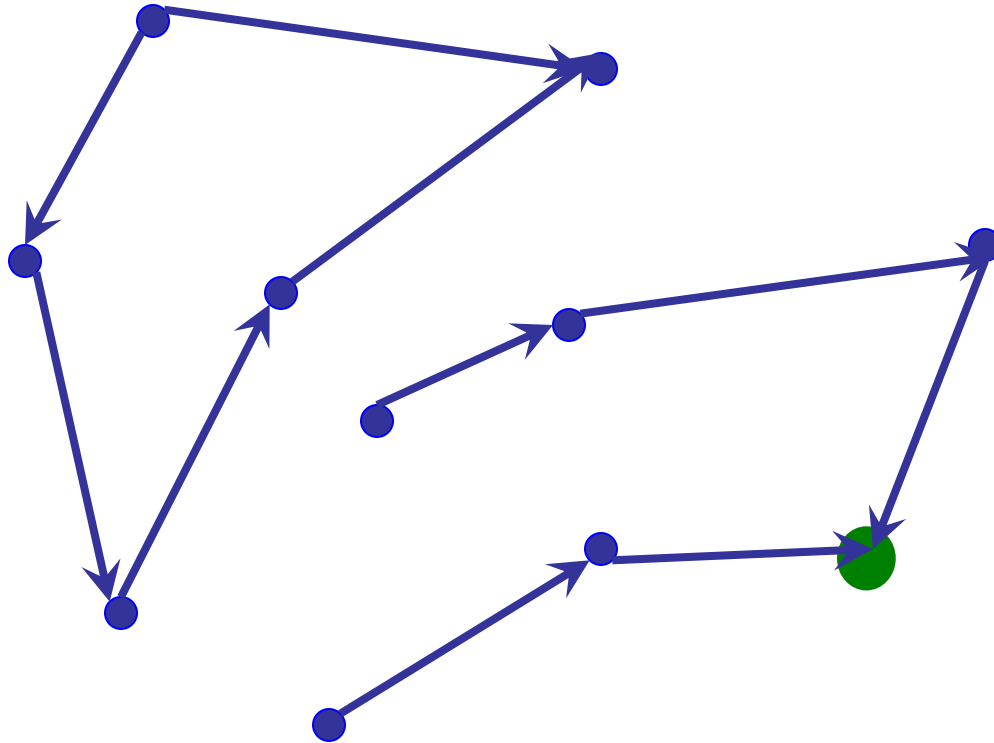
Leaves spanning tree: Valid

Example 2



Is this valid?

Not valid: Contains loop!



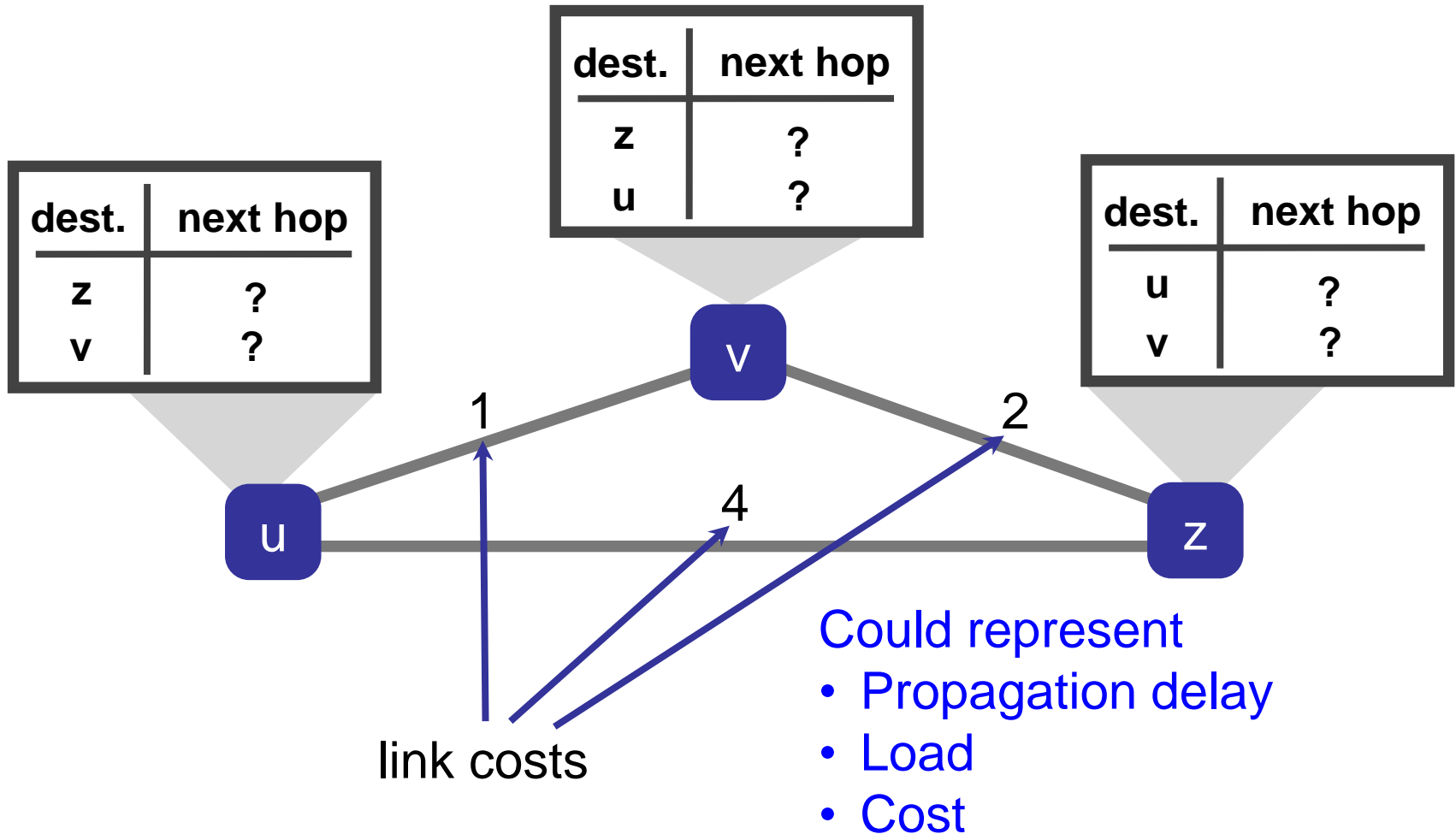
Routing validity

- Very easy to check validity of routing state for a particular destination
- Dead ends are nodes without outgoing arrow
- Loops are obvious too
 - Disconnected from rest of graph

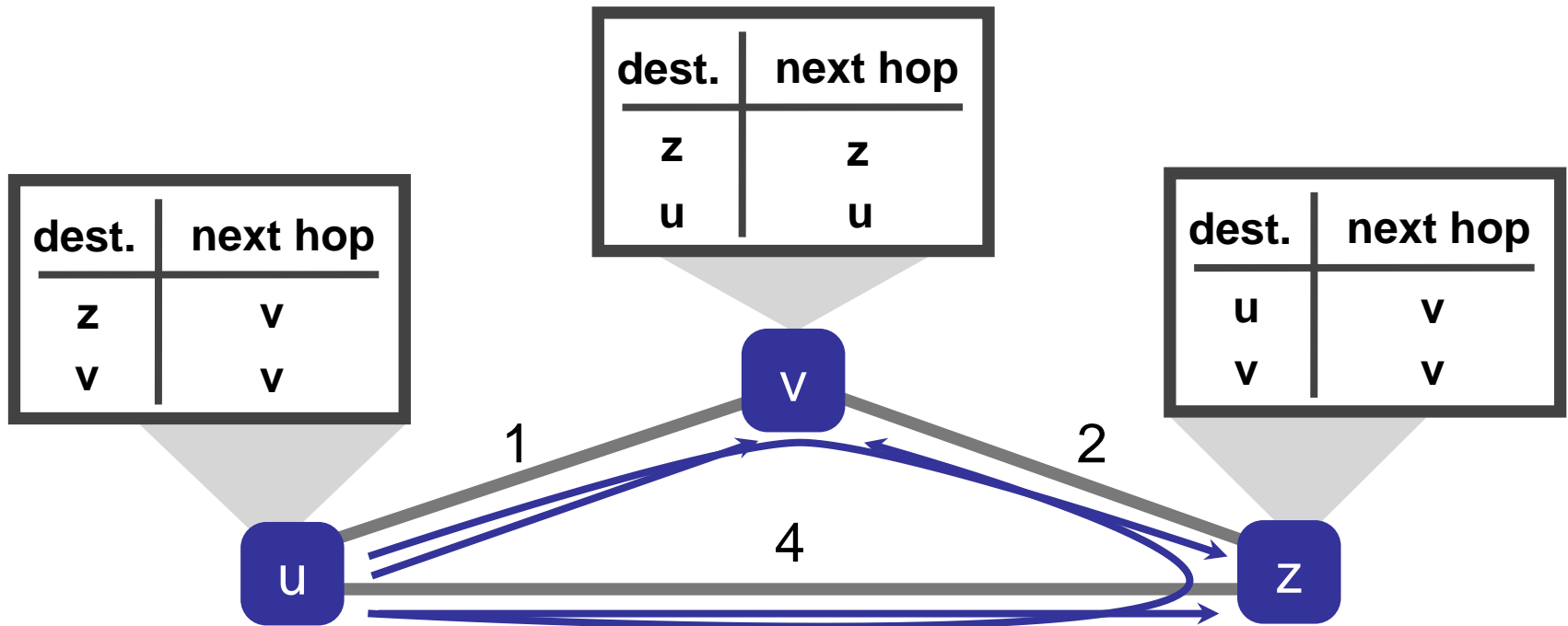
Goal of routing

- v1: Find a path to a given destination
- v2: Find a *least-cost path* to a given destination

Example



Example



least-cost path from u to z: u v z

least cost path from u to v: u v

Least-cost path routing

- **Given:** router graph & link costs
- **Goal:** find least-cost path
 - From each source router to each destination router

Least-cost routes

- Least-cost routes provide an easy way to avoid loops
 - No reasonable cost metric is minimized by traversing a loop
- Least-cost paths form a spanning tree for each destination rooted at that destination

Dijkstra's algorithm

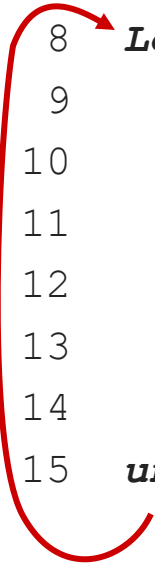
- Network topology, link costs known to all nodes
 - All nodes have same info
- Computes least-cost paths from one node (“src”) to all other nodes
 - After k iterations, know least-cost path to k destinations
- **Notations**
 - $c(x,y)$: link cost from x to y ;
 - ∞ if not direct neighbors
 - $D(v)$: current value of cost of path from src to dst v
 - $p(v)$: predecessor node along path from source to v
 - N' : set of nodes whose least-cost path definitively known

Dijkstra's algorithm

```
1  Initialization:  
2     $N' = \{u\}; D(u) = 0$   
3    for all nodes  $v$   
4      if  $v$  adjacent to  $u$   
5        then  $D(v) = c(u, v)$   
6      else  $D(v) = \infty$ 
```

Dijkstra's algorithm

```
1  Initialization:
2       $N' = \{u\}; D(u) = 0$ 
3      for all nodes  $v$ 
4          if  $v$  adjacent to  $u$ 
5              then  $D(v) = c(u, v)$ 
6              else  $D(v) = \infty$ 
7
8  Loop
9      find  $w$  not in  $N'$  such that  $D(w)$  is a minimum
10     add  $w$  to  $N'$ 
11     update  $D(v)$  for all  $v$  adjacent to  $w$  and not in  $N'$ :
12          $D(v) = \min( D(v), D(w) + c(w, v) )$ 
13         /* new cost to  $v$  is either old cost to  $v$  or known
14            least path cost to  $w$  plus cost from  $w$  to  $v$  */
15 until all nodes are in  $N'$ 
```

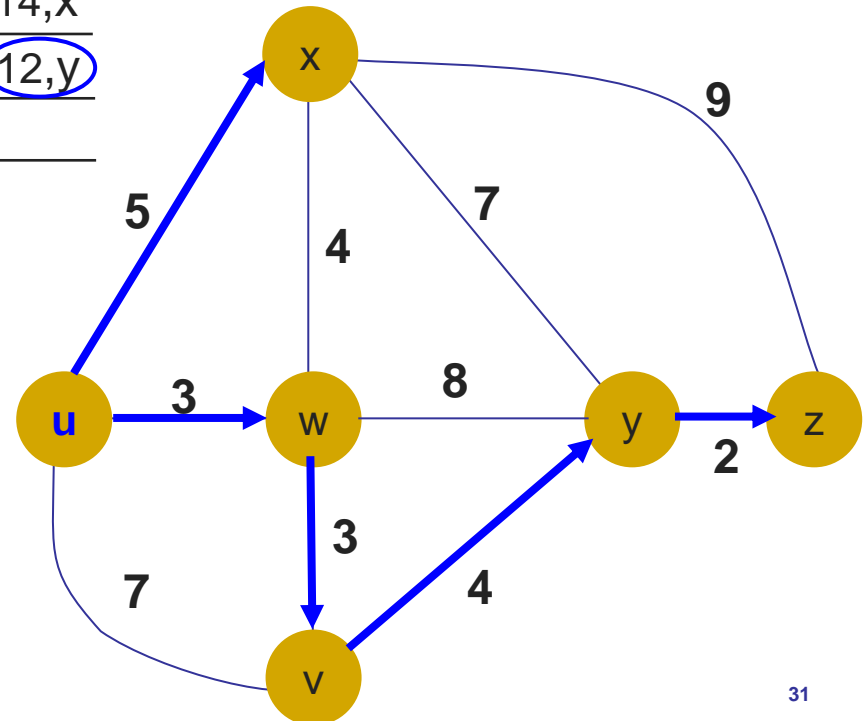


Dijkstra's algorithm: Example

Step	N'	D(v) p(v)	D(w) p(w)	D(x) p(x)	D(y) p(y)	D(z) p(z)
0	u	7,u	3,u	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞
2	uwx	6,w			11,w	14,x
3	uwxv				10,v	14,x
4	uwxvy					12,y
5	uwxvyz					

Notes:

- Construct shortest path tree by tracing predecessor nodes
- Ties can exist (can be broken arbitrarily)

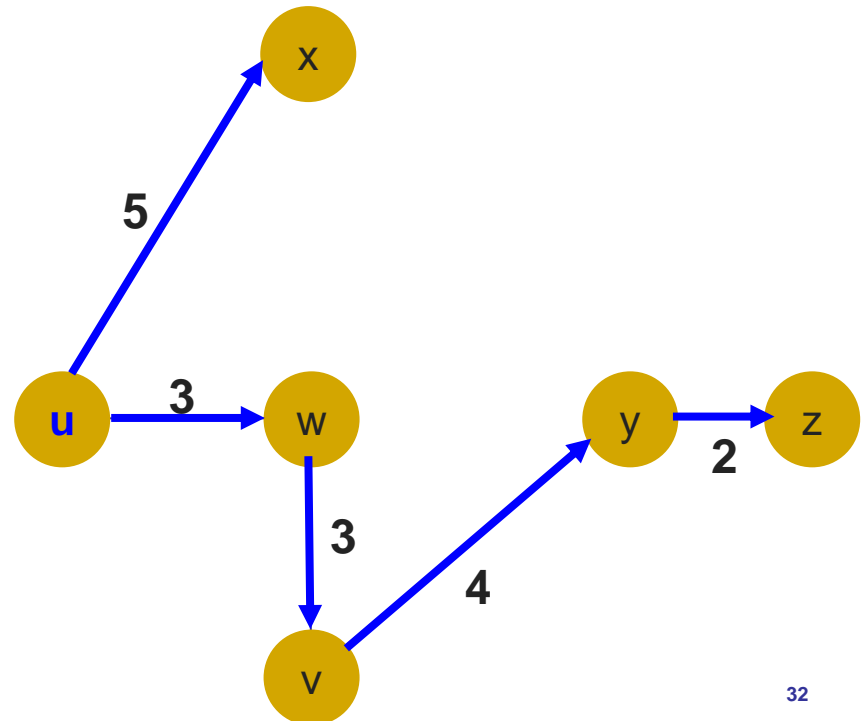


Dijkstra's algorithm: Example

*Resulting forwarding table
in **u***

destination	link
v	(u, w)
w	(u, w)
x	(u, x)
y	(u, w)
z	(u, w)

*Resulting least-cost tree
from **u***



Summary

- Network layer control plane calculates valid routes and sets up forwarding table
 - Avoiding loops and dead ends
- Least-cost routes can be calculated using Dijkstra's algorithm