CS-340 Introduction to Computer Networking

Lecture 10: Router Internals & Routing Algorithms

Steve Tarzia

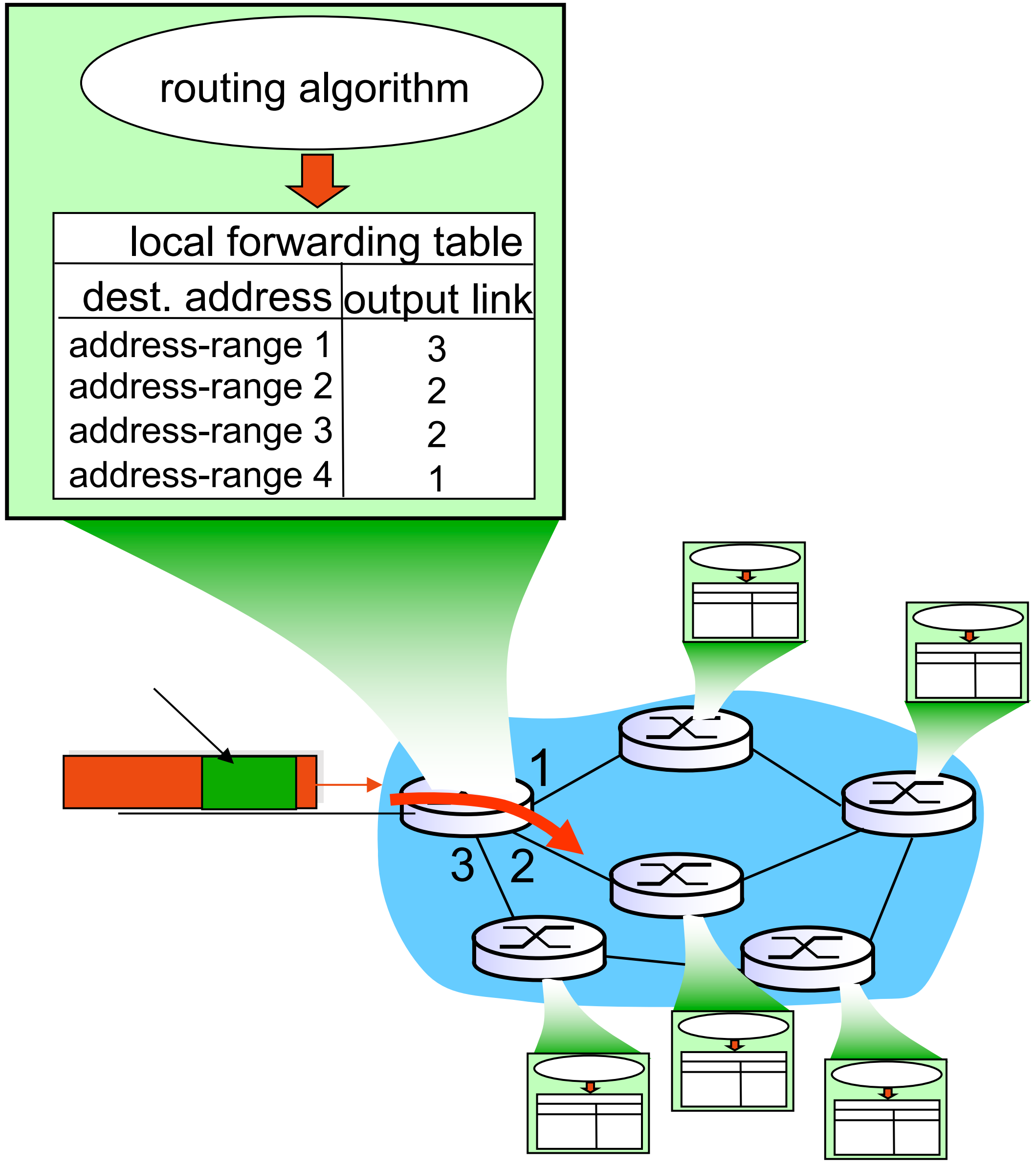
Many diagrams & slides are adapted from those by J.F Kurose and K.W. Ross

Last lecture: NAT & IPv6

- **Private networks** are isolated from the **public** Internet, but usually connected through a Network Address Translator (**NAT**).
  - **Port mapping** makes multiple machines on the private subnet look like multiple sockets (processes) on one big machine.
  - NAT requires no awareness or cooperation from hosts on either side.
  - NAT is also one way to implement a load balancer.
  - Besides NATs, **middleboxes** include *firewalls* and other security appliances.
- **IPv6** uses 128-bit addresses for practically unlimited public addresses.
  - IPv6 adds 20 bytes of header overhead.
  - Not directly compatible with IPv4. Adopted by ~30% of end hosts.
  - **Dual-stack** hosts have both IPv4 and IPv6 addresses to reach entire Internet.
  - Interoperates with IPv4 via **tunneling:** send IPv6 packet inside IPv4 packet.

Routing Review

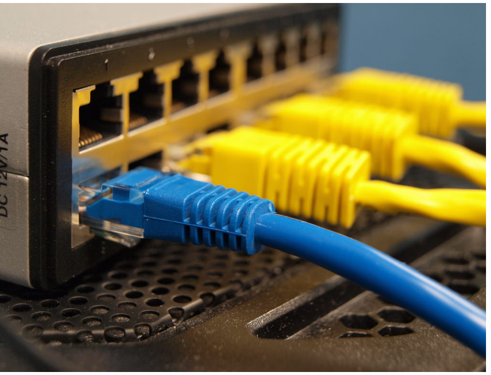
- Each packet has an IP header listing the source & destination *IP addresses* and the TTL.
- Routers use *forwarding tables* to direct IP packets to the next hop
- *Forwarding rules* associate ranges of addresses with the outbound links.
- Ranges are defined in *CIDR notation:*
    - 234.30.0.0/16



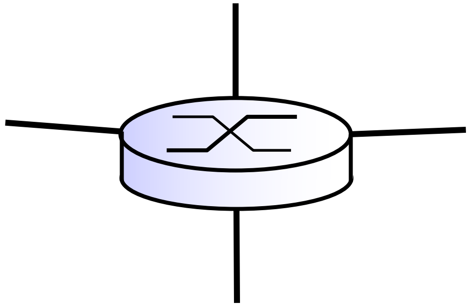
Input and output "**ports**" on routers

- The word ***port*** is overloaded in networking:
  - At the physical layer, the wired connections on on routers are called *ports*.
  - At the transport layer (TCP/UDP), ports numbers create logical connections.
- Most wired links are bidirectional, and we can think about each direction separately:

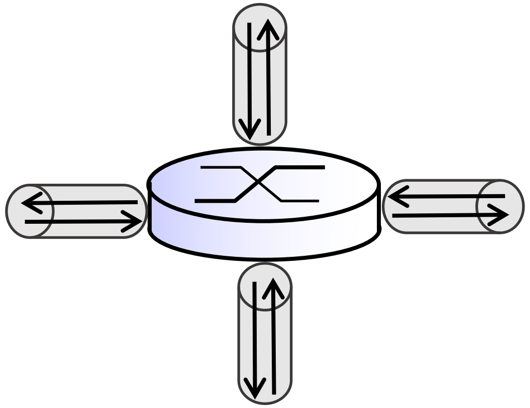
Physical view:



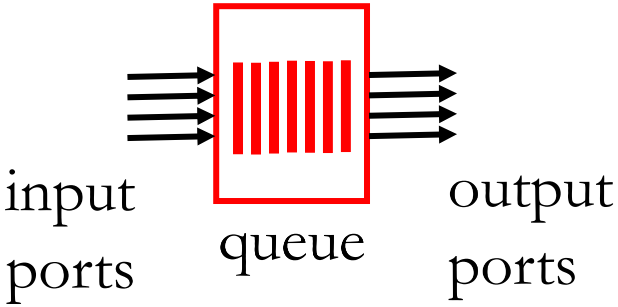
Schematic view:



In more detail:

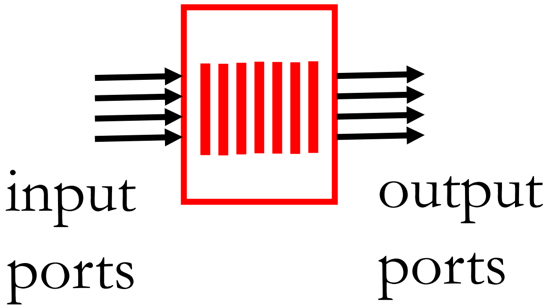


Abstractly:



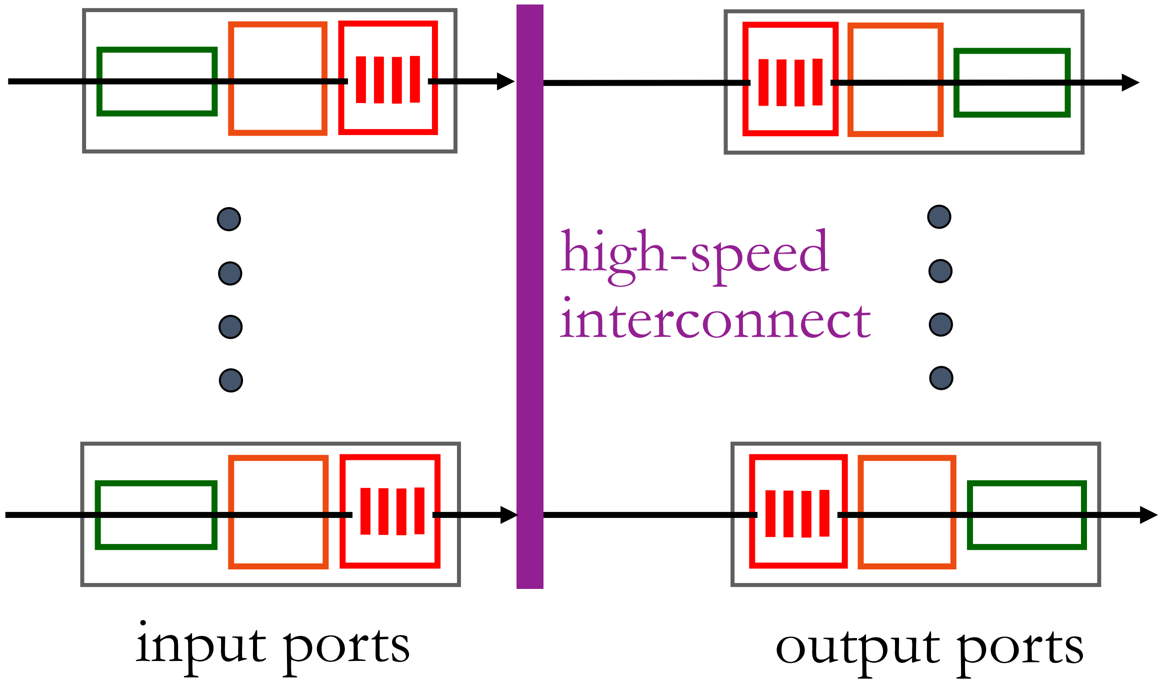
Simple router model

- One queue



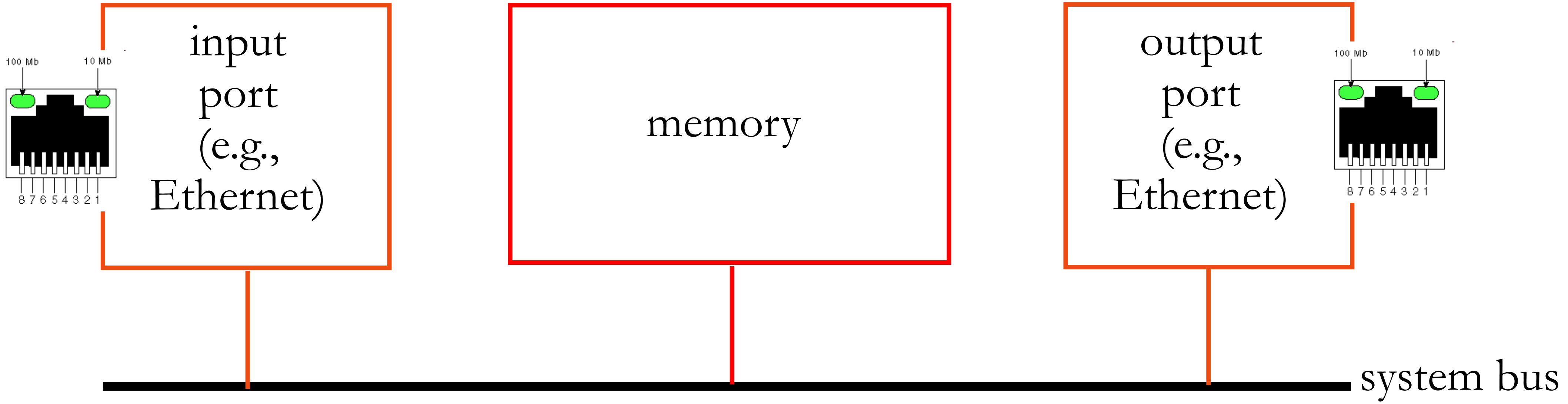
More realistic model

- A queue for each port



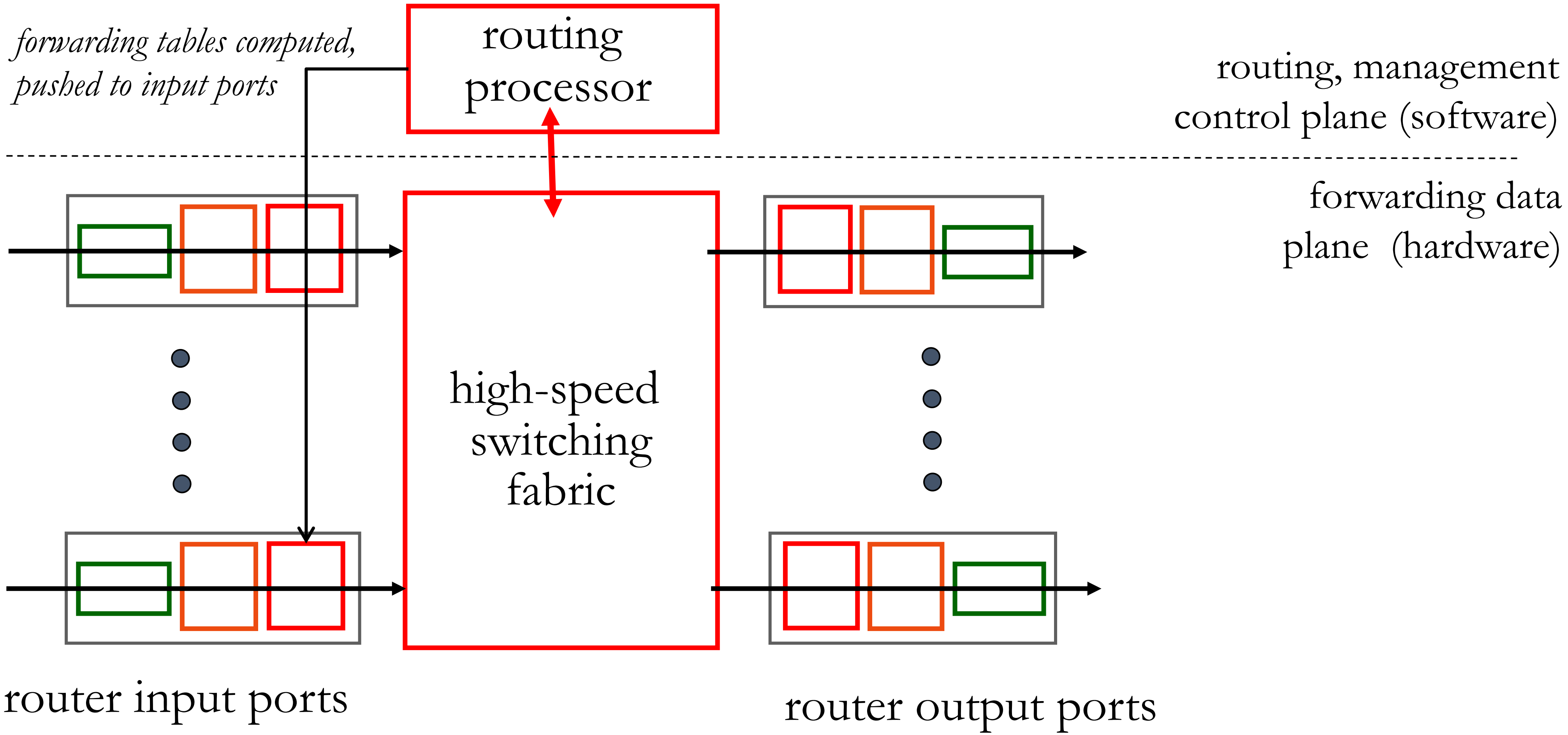
First generation routers

- General-purpose computers with several network cards.
- Routing ***software*** implements the forwarding logic.
  - Eg., ***iptables*** command configures Linux kernel's handling of packets
- However, memory and bus bandwidth become bottlenecks.
  - General-purpose computers are optimized for computing, not I/O

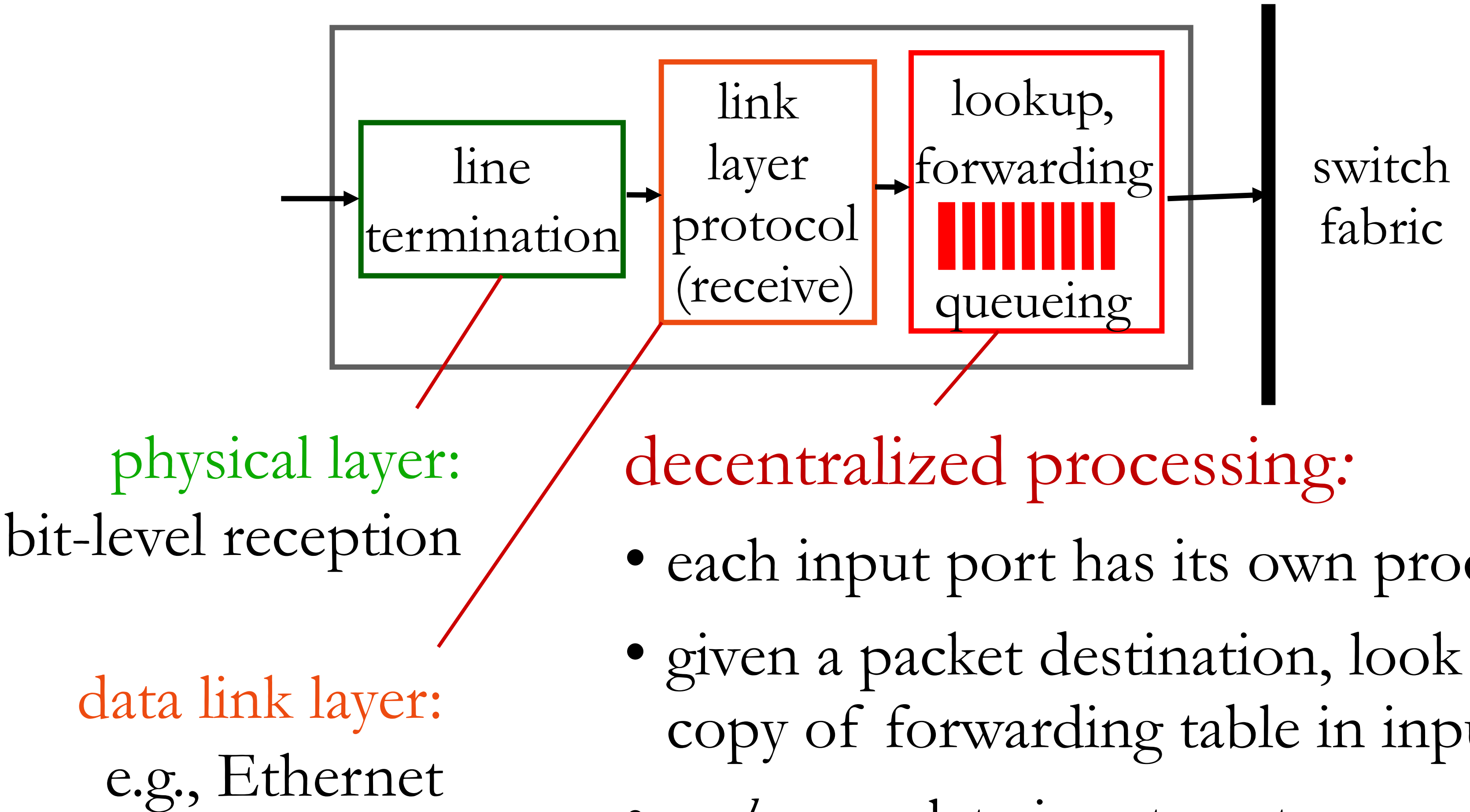


Modern router architecture

- **Control plane:** run routing algorithms (RIP, OSPF, BGP)
- **Data plane:** forward packets from incoming to outgoing links



Input ports process packets in parallel

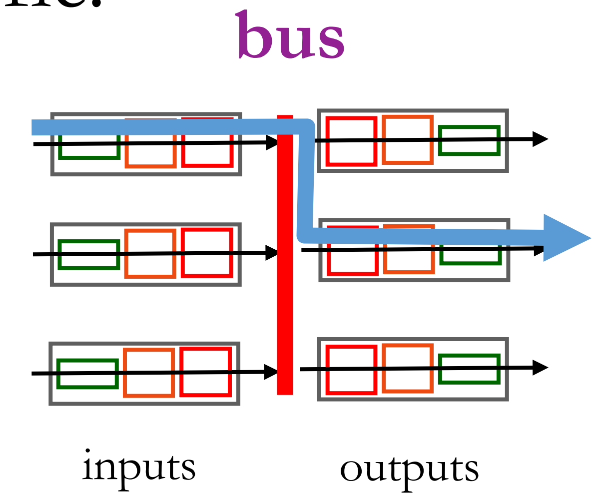


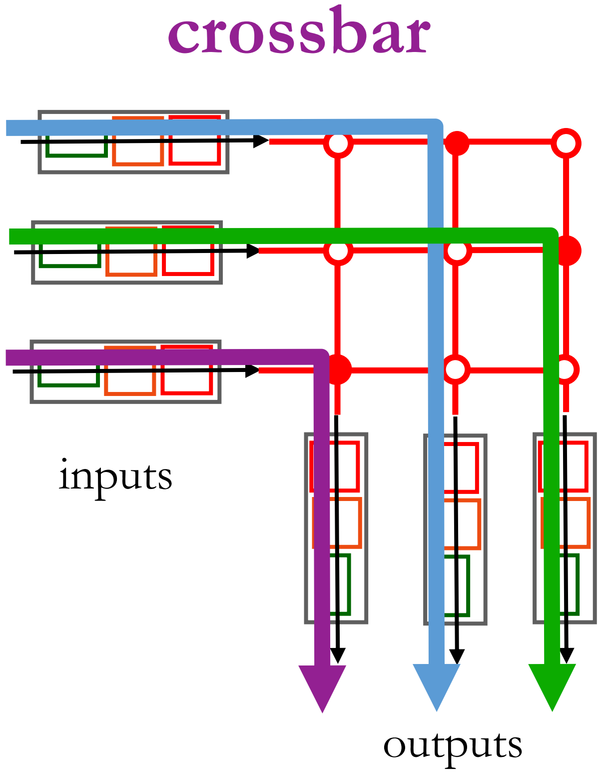
decentralized processing:

- each input port has its own processor and memory
- given a packet destination, look up output port using a copy of forwarding table in input port memory
- *goal*: complete input port processing at "line speed"
- queue packets if they arrive faster than forwarding rate into switch fabric

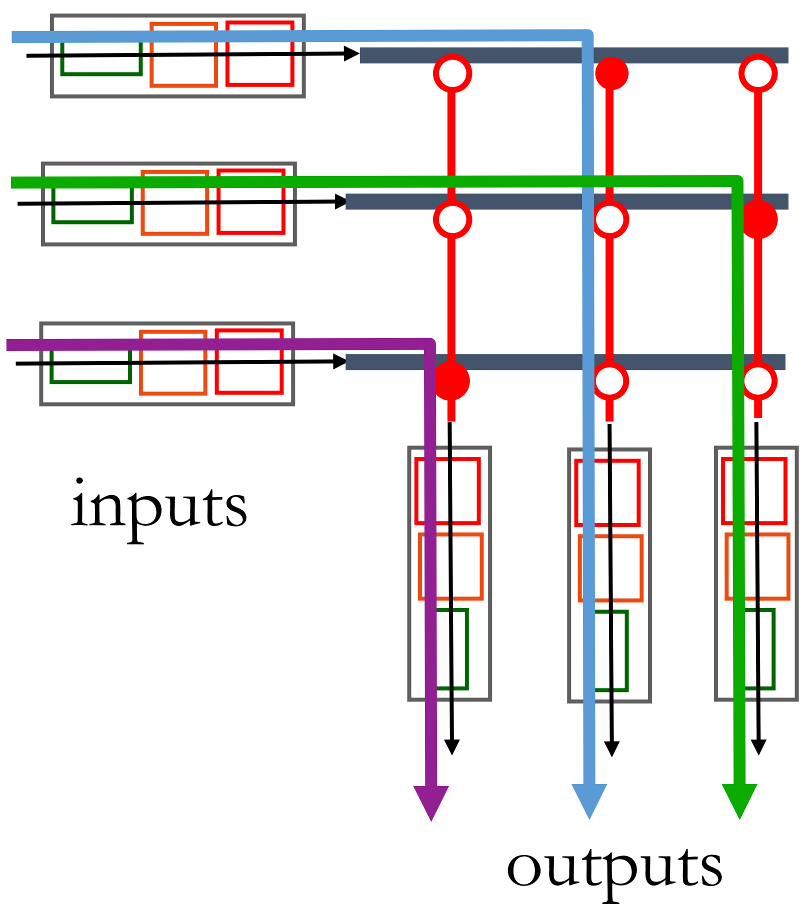
Switching fabric – connects input and output ports

- **Switching rate** – the maximum rate of data transfer from all input ports to all output ports. (A very important spec. for a router/switch.)
  - Ideally = # inputs × input line rate
  - In practice, switching rate is smaller.
- Two basic types of switching fabric:
  - **Bus**: simplest design. Can only be used by one in/out pair. Bus should be much faster than individual input line rate.
  - **Crossbar**: advanced design for core routers. Allows multiple simultaneous flows by opening and closing (*switching*) connections appropriately.





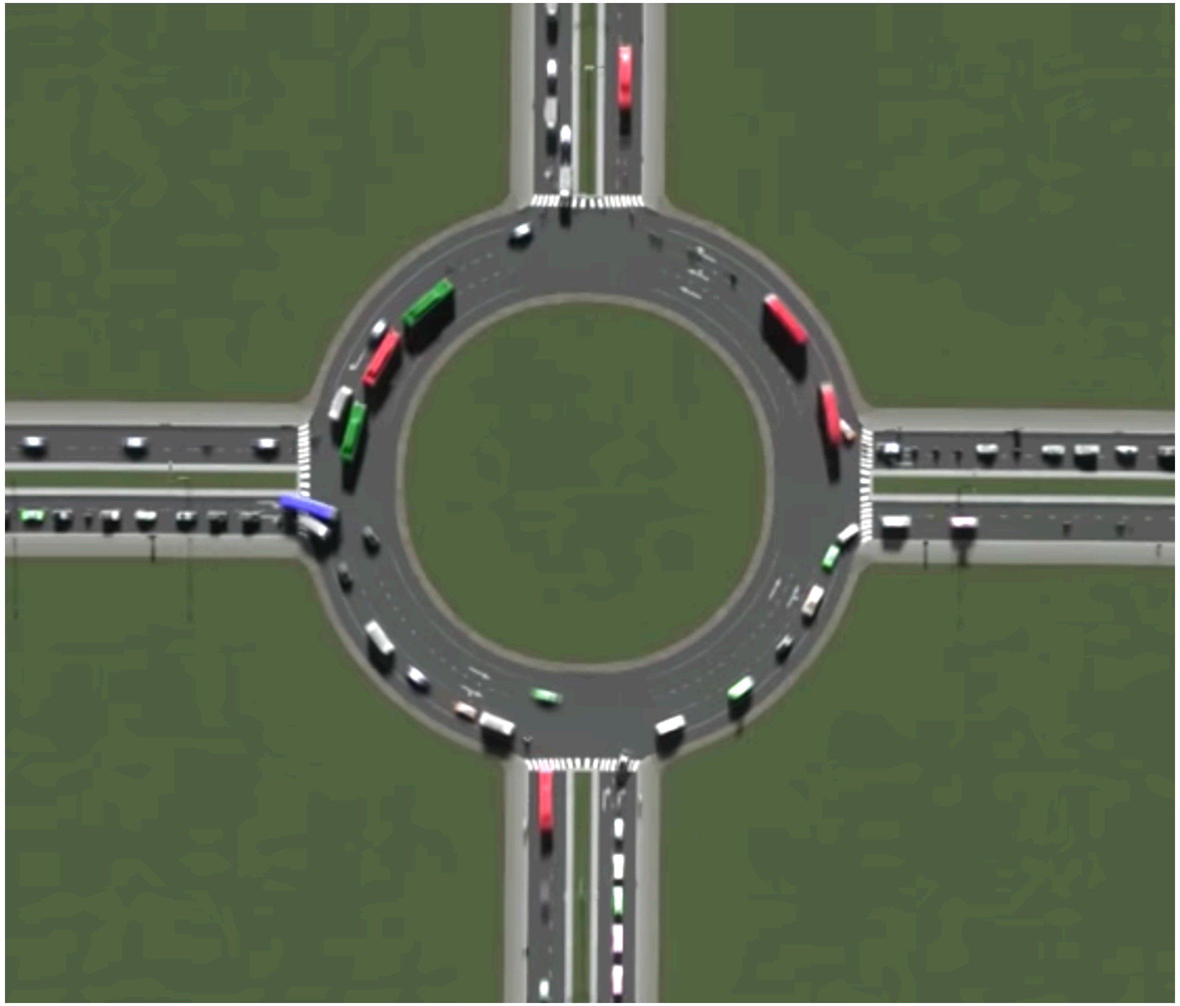
Crossbar switch



- Open circle means no connection between horizontal and vertical paths (input & output).
- Closed circle connects a vertical and a horizontal path, connecting an input to an output.
- Crossbar connections are changed as needed.
- Expensive to build, compared to bus:
  - For n inputs and outputs, requires n^2 switch points in the crossbar.

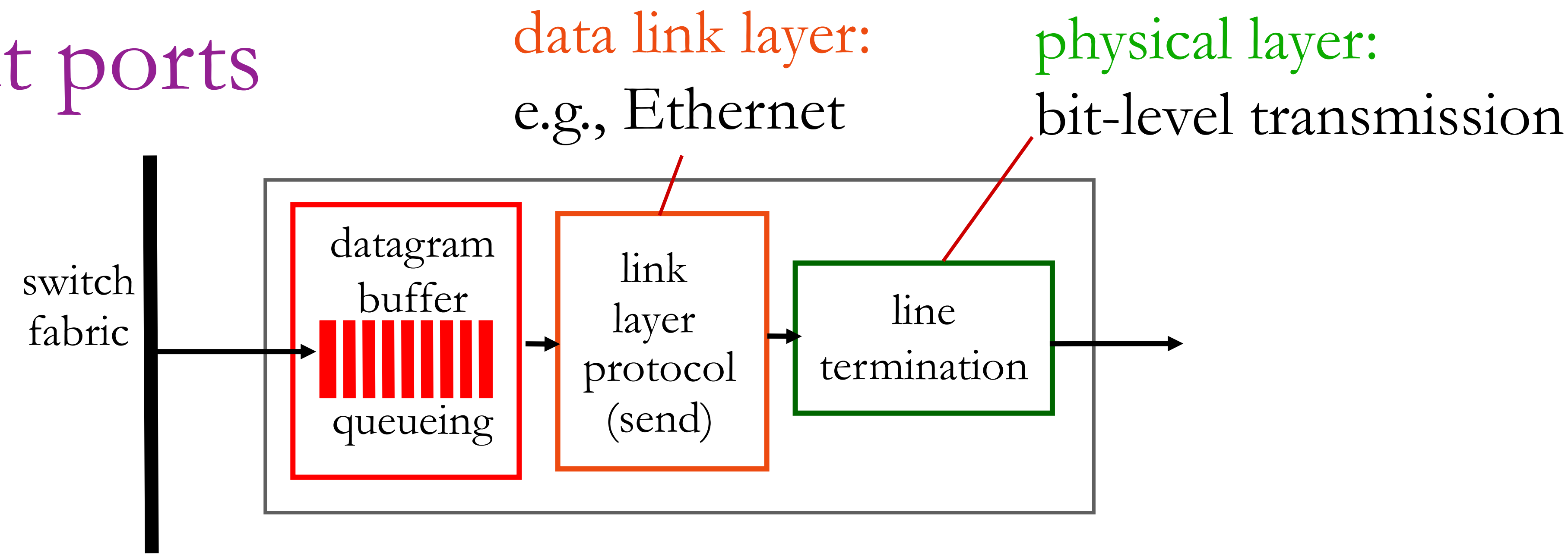
Road analogy for switching fabrics

Bus is like a **roundabout**



Crossbar is like a **stack exchange**

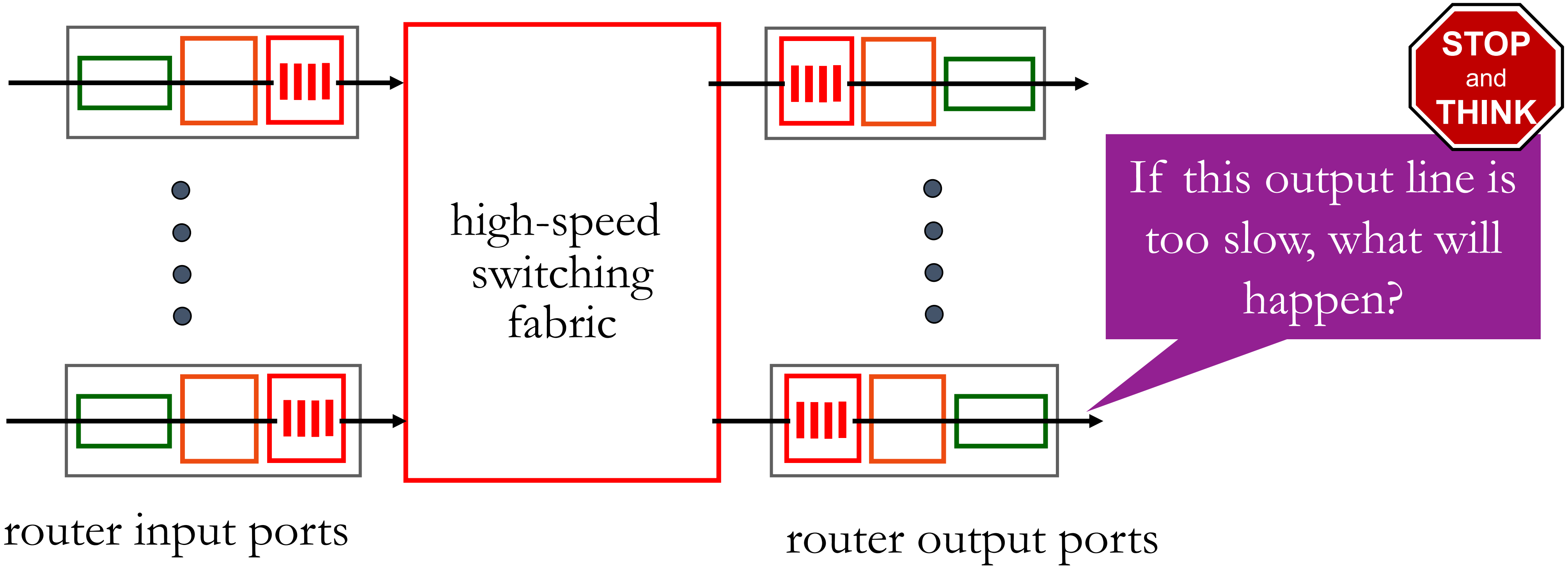
Output ports



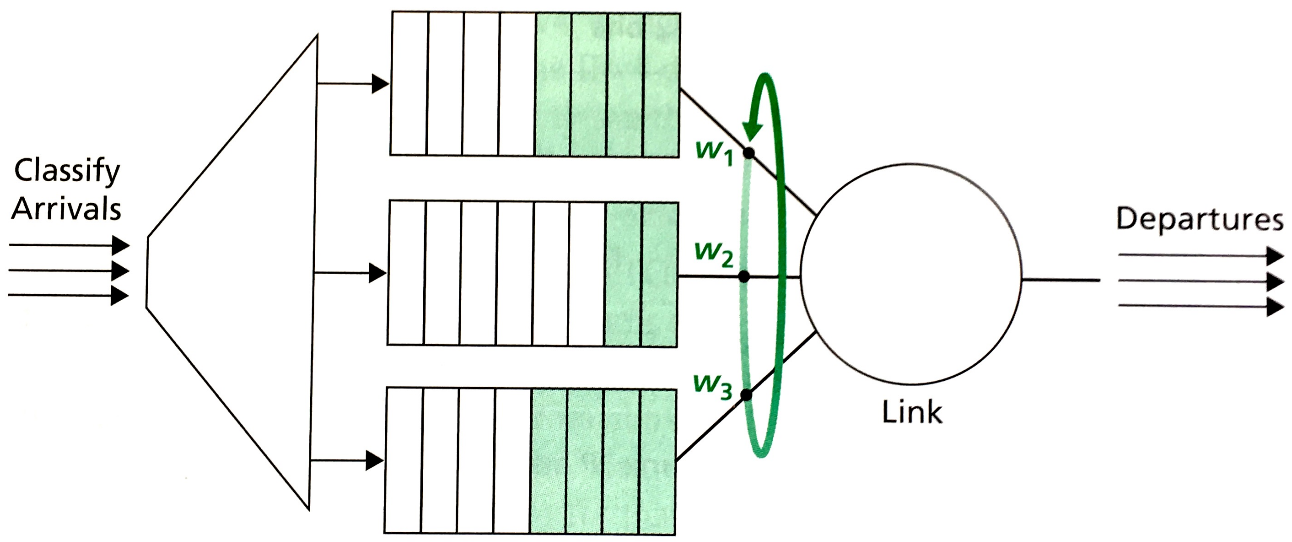
- ***Buffering*** is required because switch fabric may be faster than physical output link, link may be congested.
- Queued packet can be ***scheduled*** if desired.
  - Give higher priority to certain types of packets
  - Give higher priority to certain orgins/destination
- ***Net neutrality*** is the policy debate about whether ISPs can do this.

Queues and packet loss

- Packets can be dropped by router if any of the queues are full
  - *Switching fabric* may be overloaded, filling up the *input queues*
  - *Output ports* may be overloaded, filling up the *output queues*
- Routers silently drop packets when a queue is full.



Weighted fair queuing (WFQ)

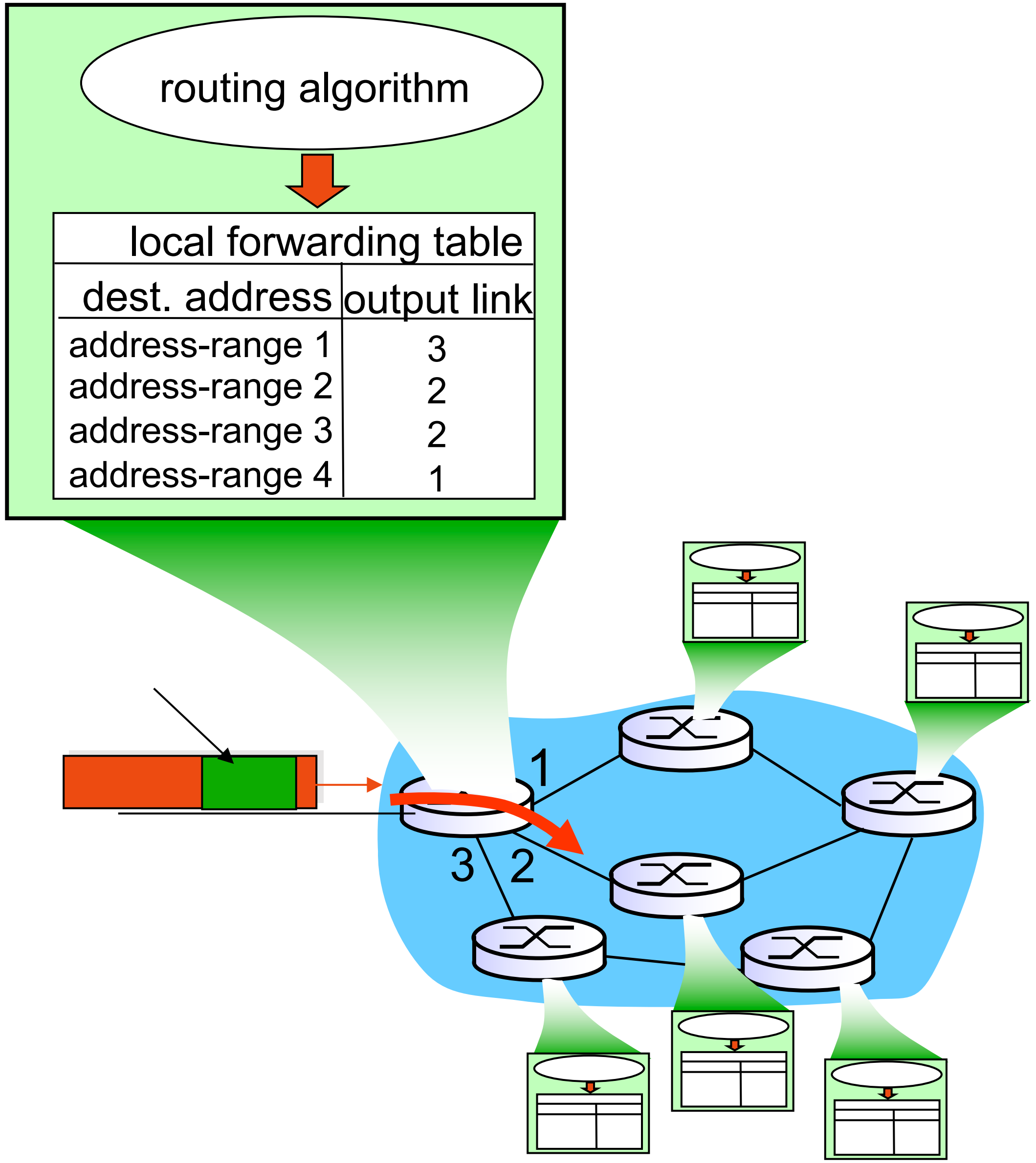


- A *discriminative* alternative to FIFO packet scheduling policy
  - Some traffic gets higher ***priority***. We have multiple queues instead of just one.
- Administrator creates rules to ***classify packets***, based on header fields:
  - Src./dest. IP address • Port number (service) • TCP vs UDP • QoS
- Each class is allocated a certain fraction of link capacity ($w_i/\sum w$)
  - Spend w_i time sending packets from queue i, then move to next queue.

Next Topic:
IP Control Plane

How to decide forwarding rules at each router?

(Chapter 5)

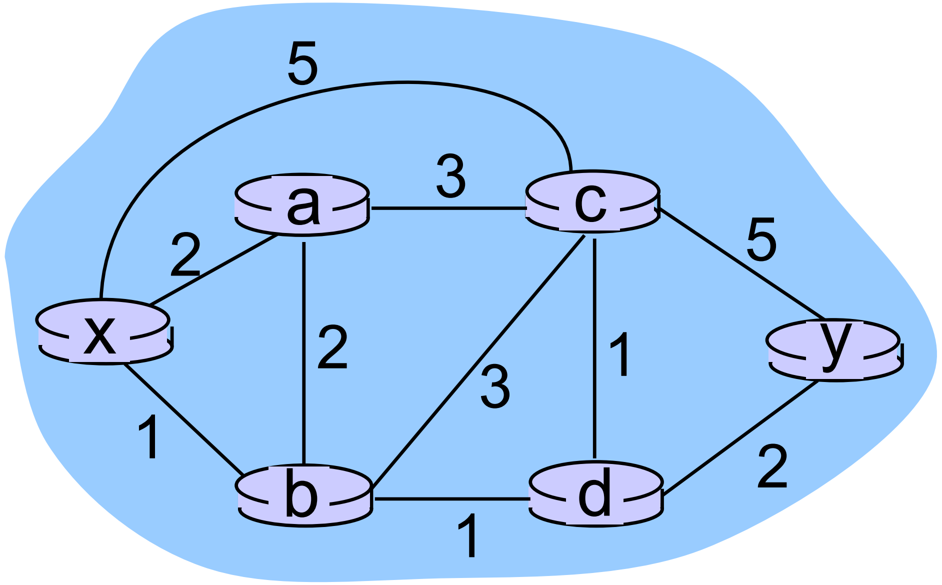


Centralized versus ***Distributed*** algorithms

- **Centralized/Global** routing:
  - Algorithm has full knowledge of the entire network.
  - Makes decisions that affect all routers
  - In routing, we call these **link state** algorithms.
  - Used within an organization (autonomous system) (eg., OSPF)
- **Distributed/Local** routing:
  - Each router must decide its own routing table using local observations.
  - Operates *iteratively.*
  - Routers continually share information with neighbors
  - Global information is gradually propagated across the network.
  - Used within and between autonomous systems (eg., RIP & BGP, respectively)
- Distributed algorithms are more difficult to design correctly.

Graph abstraction of computer networks

- A graph is a set of *vertexes* and *edges*
  G = (V, E)
- Vertexes represents routers:
  V = set of routers = {x, a, b, c, d, y}
- Edges represent links:
  E = set of edges = {(x,a), (x,b), (x,c), (a,b), (a,c), (b,c), (b,d), (c,d), (c,y), (d,y)}
- Edge labels/weights represent *distance* or *cost* to communicate:
  $C : E \rightarrow \{0, 1, 2, 3, \ldots\}$ *C maps edges to costs*
  c(x,a) = 2, c(a,c) = 3, c(x,b) = 1, …
  $c(x,y) = \infty$ *because the two vertices are not connected*

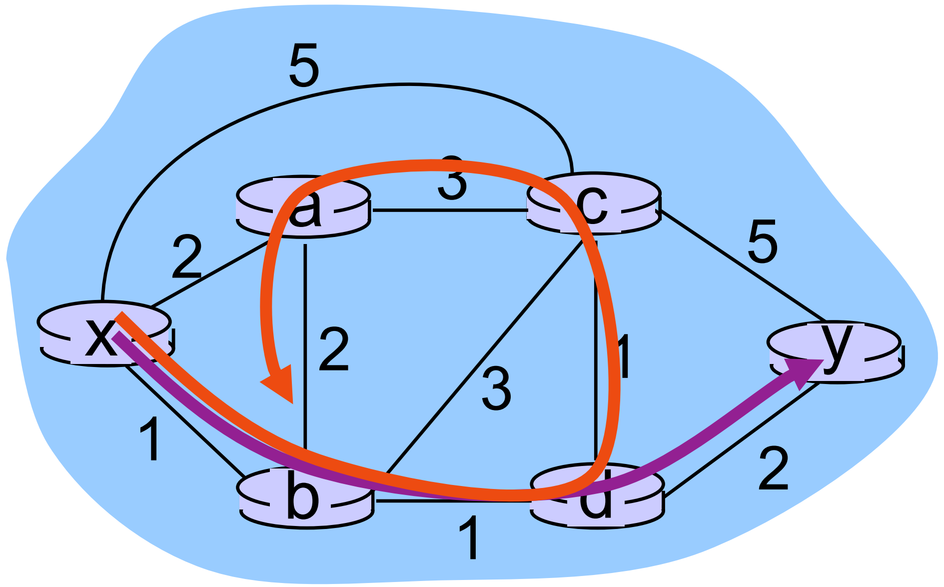


For now, **cost = delay**.
We'll ignore the limited capacity of links.

Shortest Path problem

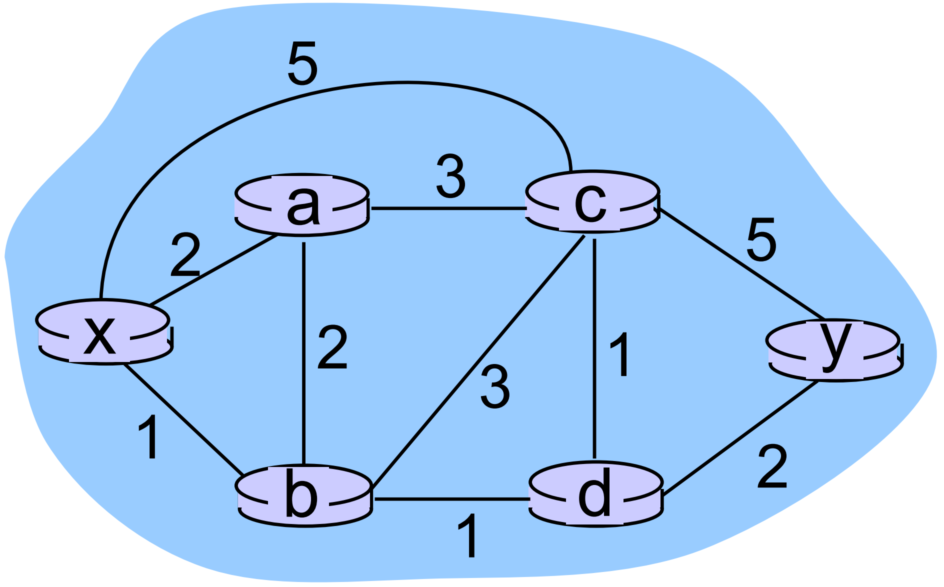
- What edges should I choose to construct a **path** from x → y with minimal total cost (delay)?
- You are given:
  - An edge-weighted graph
  - A starting vertex
  - A destination vertex
- Must output:
  - The path (a sequence of edges)
  - The total cost

Greedy path is disastrous



Shortest path has cost 1+1+2=4

Shortest path insight



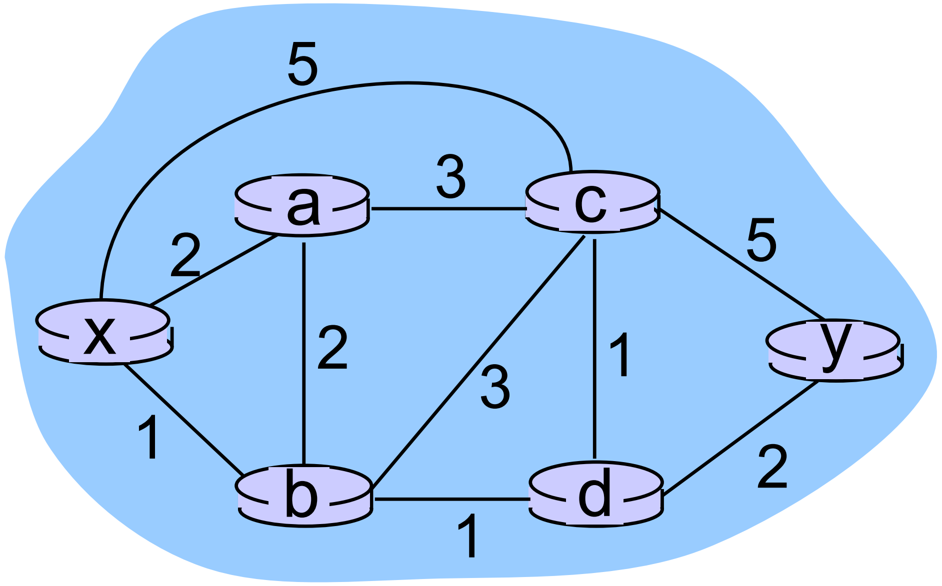
- Break down the problem into **subproblems**
- Let ***d(x,y)*** represent the cost of the shortest path between x and y. It must be true that:

$$d(x,y) = \min\{d(x,c) + c(c,y),\ d(x,d) + c(d,y)\}$$

Cost of path that *almost* gets there. Cost of the final step.

- Shortest path to ***y*** must pass through a neighbor, either vertex ***c*** or ***d***.
- The cost of the shortest (x,y) path *with c as the final stop* is the cost of the shortest (x,c) path plus the edge cost of the final step from c to y.
- Just choose the option with minimum total cost.

Bellman-Ford equation



- The equation works for any pair in the graph
- Let ***d(x,y)*** represent the cost of the shortest path between x and y. It must be true that:

$$d(x,y) = \begin{cases} 0 & \text{if } x=y \\ \min_v \{d(x,v) + c(v,y)\} & \text{if } x \neq y \end{cases}$$

- In other words,

  Except in the trivial case when x and y refer to the same vertex, the shortest path between x and y must pass through *some vertex* ***v*** that is adjacent to y before finally arriving at y.

  - The sub-path leading from x to v must be the *shortest* path from ***x*** to ***v***.
  - If we know the shortest path distances to all the vertices adjacent to ***y***, then we can easily choose which one of these creates the shortest path to ***y***.

Bellman-Ford recursion

$$d(x,y) = \begin{cases} 0 & \textit{if } x=y \\ \min_v \{d(x,v) + c(v,y)\} & \textit{if } x \neq y \end{cases}$$

Minimum taken over all neighbors v

- d(x,y) is the shortest path distance from x to y.
- c(v,y) is the cost of the edge directly connecting v and y.

- When calculating d(x,y), consider every possible **v** we could pass through.
- We know that one of those **v**'s is the right choice; the path has to pass through some other vertex before arriving at the finish.
- Assume we have already computed the minimum cost path to every vertex except y. This assumption leads to a ***recursive solution***.
  - Implement the recursive solution efficiently using **dynamic programing**.

Bellman-Ford algorithm


```
function BellmanFord(list vertices, list edges, vertex source)

    // Initialization
    for each vertex v in vertices:
        dist[v] := INFINITY        // Initially, vertices have infinite weight
        prev[v] := NULL            // and a null predecessor.
    dist[source] := 0              // Distance from source to itself is zero

    // Relax edges repeatedly.
    for i from 1 to size(vertices)-1:
        for each edge (u,v):
            alt := dist[u] + (u,v).cost()
            if alt < dist[v]:
                dist[v] := alt
                prev[v] := u

    // outputs are distance and predecessor arrays
    return dist[], prev[]
```

Invariant:
At round *i*, *distance[j]* is the shortest path from *source* to *j* having at most *i* hops.

Runtime complexity:
$\Theta(|V| \cdot |E|)$

Bellman-Ford demo

https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html

Dijkstra's algorithm

```
function Dijkstra(list vertices, list edges, source):
    // Initialization
    for each vertex v in Graph:
        dist[v] := INFINITY            // Unknown distance from source to v.
        prev[v] := NULL                // Previous node in optimal path from source.
    dist[source] := 0                  // Distance from source to itself is zero.
    Q := vertices.copy()               // The list of the "unvisited" vertices.

    // "visit" the closest unvisited vertex, u
    while Q.size() > 0:
        u := vertex in Q with minimum dist[u]
        Q.remove(u)
        // try using u to make shorter paths
        for each neighbor v of u:
            alt := dist[u] + (u,v).cost()
            if alt < dist[v]:
                dist[v] := alt
                prev[v] := u

    // outputs are distance and predecessor arrays
    return dist[], prev[]
```

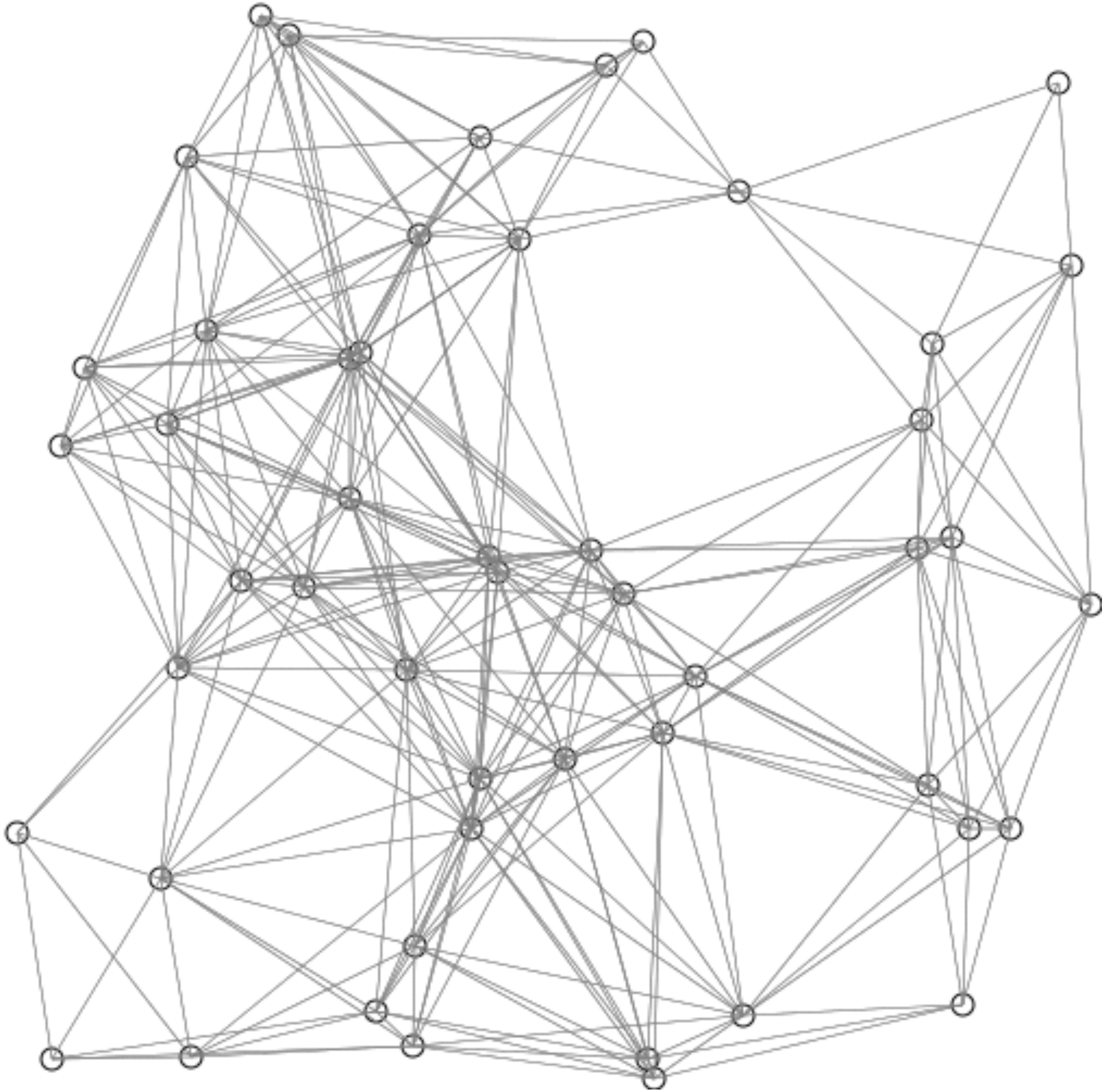
Invariant:
distance[j] is shortest path if j was visited, otherwise it's shortest using visited nodes

Runtime complexity:

$\Theta(|V|^2)$ or

$\Theta(|E| + |V| \log |V|)$

The faster version uses a priority queue.



Dijkstra’s algorithm demo

https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_en.html

Dijkstra's shortest path example

Bellman-Ford *vs* Dijkstra

Bellman-Ford

- Very simple
- **Slower**: $\Theta(|V| \cdot |E|)$
- Detects negative cycles
- Can be adapted to a *distributed* implementation in the *Distance-Vector* algorithm (used by BGP).

Dijkstra

- Slightly more complicated
- **Faster**: $\Theta(|E| + |V| \log |V|)$
- Cannot handle negative cycles
- Best choice for a *centralized* implementation.
- Book calls it the *Link-State (LS)* algorithm.
- Used by OSPF.

Distance-Vector algorithm

Each vertex/node/router x:

- Maintains a **distance vector (DV)**:
  - $d_x(y)$ = estimate of shortest path from x (myself) to y.
  - $\mathbf{D}_x$ is the distance vector at node x: $\mathbf{D}_x = [d_x(y) : \forall y\ ] = [d_x(0), d_x(1), \ldots]$
- Knows the cost to reach each neighbor v:
  - cost(x,v) = the cost of the link between x and y (infinity if no link exists).
- Initially *sends* its DV to all neighbors
- Keeps a copy of the latest DV received from all neighbors.
- After receiving a DV from a neighbor, *recalculate* its own DV
  - If its own DV has changed, send the updated DV to neighbors.

  iterate
- Eventually, the algorithm converges and each node knows the shortest path to every other node.

DV is recalculated using Bellman-Ford equation

- Initially, $d_x(y)$ = cost(x,y), or *infinity* if no direct (x,y) link exists.
- After receiving an updated $\mathbf{D}_u$ from neighbor *u*, or if we observe a change in local link costs, update $\mathbf{D}_x$:

  $d_x(y) \leftarrow \min_v\{cost(x,v) + d_v(y)\}$ for each node $y \in N$

Each node:

- *Waits* for changes to local link costs or updated DV from a neighbor.
- *Recalculates* estimates (DV).
- *Notifies* neighbors *if* DV has changed.

Note: the book's description of the DV algorithm is over-complicated. Please just learn DV from my two slides.

DV algorithm example

Recap

- *Weighted Fair Queueing* can prioritize classes of packets in router queue.
- *Routing algorithms* determine each router's forwarding table. It's a a *shortest path* problem on the *weighted graph* graph representing the network.
  - May be *centralized/global* or *distributed.*
- *Dijkstra's Algorithm* is a fast *centralized* (LS) algorithm for shortest path.
  - Used by *Open Shortest Path First (OSPF)* protocol within an AS.
  - Routers initially *flood/broadcast* local link information to entire network.
  - Each router then solves shortest path from itself to all other routers.
- *Distance Vector (DV)* algorithm is a *distributed* shortest path algorithm
  - Used by the *Border Gateway Protocol (BGP)* to route between AS's.
  - Initially, routers only knows distance to neighbors – broadcast to neighbors.
  - When receive a neighbor's DV, update own DV, & broadcast if DV changed.