Concurrent Computing (Computer Networks)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

April 12, 2016

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

Continued from last lecture ...

Notes:

- We now know enough to write a (rough) algorithm for processing IP packets:

Algorithm (IP packet processing)

Given a packet provided by the lower, link layer:

1. validate header (e.g., use checksum to check for errors),
2. process options in header,
3. check destination address: if the packet is for this host
   - 3.1 buffer fragments and apply reassembly process, then eventually
   - 3.2 provide payload to a higher layer (per protocol field)

   otherwise, assuming we want to forward the packet
   - 3.3 check TTL, and drop if exceeded,
   - 3.4 look-up next hop in forwarding table,
   - 3.5 apply fragmentation and header update processes, (e.g., decrement TTL, recompute checksum),
   - 3.6 transmit packet(s) via lower, link layer

and if/when errors occur, signal them appropriately.

Notes:

- We now know enough to write a (rough) algorithm for processing IP packets:

Algorithm (IP packet processing)

Given a packet provided by the lower, link layer:

1. validate header (e.g., use checksum to check for errors),
2. process options in header,
3. check destination address: if the packet is for this host
   - 3.1 buffer fragments and apply reassembly process, then eventually
   - 3.2 provide payload to a higher layer (per protocol field)

   otherwise, assuming we want to forward the packet
   - 3.3 check TTL, and drop if exceeded,
   - 3.4 look-up next hop in forwarding table,
   - 3.5 apply fragmentation and header update processes, (e.g., decrement TTL, recompute checksum),
   - 3.6 transmit packet(s) via lower, link layer

and if/when errors occur, signal them appropriately.

- Question: how does the forwarding table get populated with entries?
- Answer: **routing**, which is a *big* topic so we'll focus on uni-cast
  1. **distance vector routing**, and
  2. **link state routing**

  and hence ignore various alternatives.

Notes:

Concepts (1)

- Recall:
  - routing is the act of deciding a path used when forward packets from a given source to a given destination,
  - forwarding is a *local* process, routing is *global* in the sense it involves the whole (inter-)network,
  - goal is to make best use of connectivity and thus bandwidth: it can be viewed as a form of resource allocation.
- Solution(s):
  1. static (or fixed) routing, i.e., hard-code routing information by hand,
  2. source routing, i.e., let the source pre-determine routing decisions, or
  3. adaptive routing, e.g., i.e., use a distributed **routing algorithm** (or **routing protocol**).

Notes:

- In a sense, other mechanisms also influence the route taken by transmitted packets; when viewed alongside routing, they are applied at different time scales and hence for different purposes. For example, routing is able to make fairly fine-grained updates to the route taken, and can deal with the failure of equipment for example. Mechanisms such as traffic engineering (or load balancing) and network provisioning (i.e., equipment deployment) occur at more coarse-grained intervals to cater for demands such as network load (over a long period of time, rather than just instantaneously) and the number and usage patterns of users.
- The idea of source routing is that the source node *tells* the network how to route packets it sends; it does this via the IP options mechanism. There are two forms: **Strict Source Record Route (SSRR)** specifies the *exact* route that must be taken, whereas **Loose Source Record Route (LSRR)** just cites some hops that the packet must go through. The approach might seem useless: if we assume all nodes already know how to route their packets, there would be no need for routers! However, specific cases (such as network diagnostics) can make use of it even if it remains less useful in the general case.
- We already know that the forwarding table includes a next hop for every destination node (which are compressed using prefixes); in a sense, any static approach to routing is just fixing this table. When we use an adaptive approach, the routing algorithm might maintain some state, i.e., a routing table, that contains a super-set of such information. As such, we then periodically convert the routing table into the forwarding table.

Concepts (2)

- Good news: we can reason about routing by noting

$$\text{network} \equiv \text{graph} \implies \text{graph theory} \subset \text{data structures and algorithms},$$

and that

- a network graph will be weighted to capture the properties of each connection,
- we could use directed graphs (e.g., to capture uni-directional connection properties) ...
- ... but for simplicity we'll consider undirected graphs only

st. our network is modelled by

$$G = (V, E = \langle (u_0, v_0, d_0), (u_1, v_1, d_1), \ldots, (u_{m-1}, v_{m-1}, d_{m-1}) \rangle)$$

where $|V| = n$ and $|E| = m$.

Notes:

Concepts (3)

- Bad news: any algorithm (ideally) needs to be

correct	$\Rightarrow$	find paths that provide end-to-end connectivity
efficient	$\Rightarrow$	make good use of resources
fair	$\Rightarrow$	will not "stave" nodes of bandwidth
convergent	$\Rightarrow$	initialises/recovers quickly, e.g., after change to topology
scalable	$\Rightarrow$	remains efficient even with large n and/or m

and *must* be **decentralised**: the model is that

1. no (global) controller node exists,
2. nodes typically start with only local knowledge of topology,
3. nodes communicate (concurrently) with neighbours only, and
4. nodes *and* links can fail!

Notes:

Concepts (4)

- Recall: we have *already* assumed
  - routers make (global) routing decisions, whereas
  - hosts communicate locally, or forward to nearest router

  so we can route wrt. **routing units** (or regions), *not* per-node destinations.
- The strategy is to address scalablity using hierarchy, so
  1. route *to* region, then
  2. route *in* region

  e.g., by leveraging IP prefixes to coalesce multiple destinations into one region (or block) ...

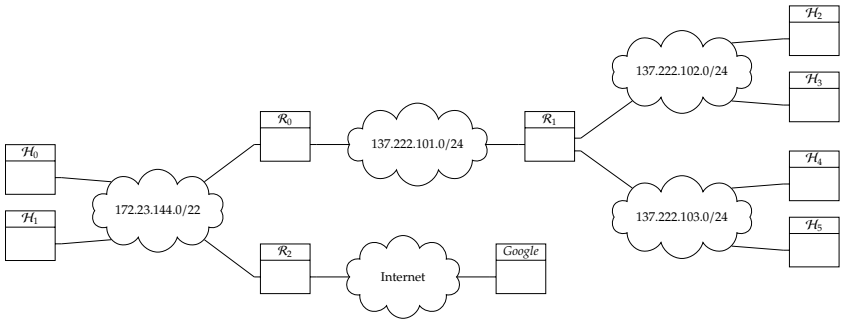
Notes:

- There are a number of obvious analogies for the hierarchical approach used. For example, imagine you want to travel to a location in an unfamiliar city: a sensible approach might be to first find directions to the city boundary, then from there to the location.
  This is also the goal with the strategy for routing: we first route packets to a boundary router, then to the destination. If we *didn't*, then
  1. routing and forwarding tables grow in size, plus
  2. there is a higher overhead to construct the tables (i.e., messages transmitted and computation performed).
- The use of regions implies that
  - nodes external to a region all have one route to nodes internal to it, i.e., the region is a coalescence of nodes within it, but
  - nodes internal to a region may have different routes to nodes external to it, i.e., the region itself is an abstraction, making no routing decisions.

  The advantage is that we tame an increase in scale, without complexity in the hosts (since the routers perform routing); a (potential) disadvantage is sub-optimal path quality. For example, the *routers* can decide what constitutes a routing unit; a good example might be an ISP network.

Concepts (5)

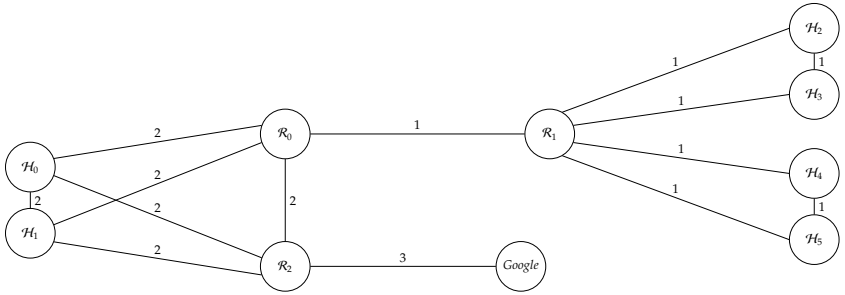
- ... so we *significantly* simplify the problem to:



Notes:

Concepts (5)

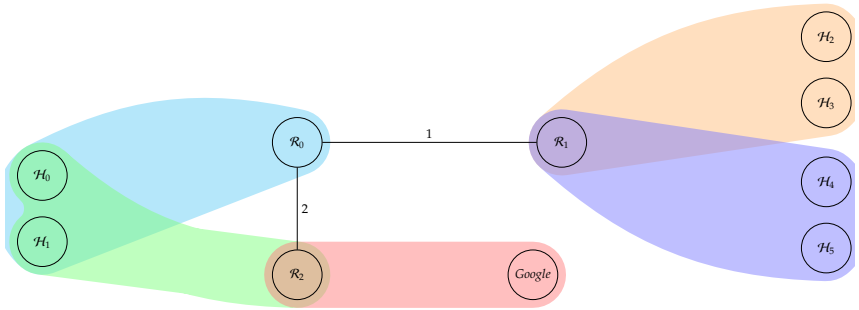
- ... so we *significantly* simplify the problem to:



Notes:

Concepts (5)

- ... so we *significantly* simplify the problem to:



Notes:

- Hopefully it's clear that

$$\text{shortest path algorithm} \Rightarrow \text{short path} \simeq \text{good route}$$

where "short" is defined wrt. a cost function, e.g.,

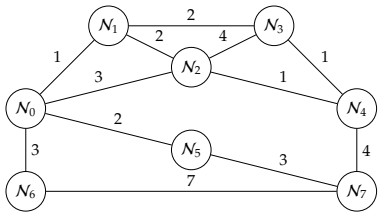
- latency,
- bandwidth,
- financial cost, or
- hop count,

but *typically* only captures features in topology (not load).

Notes:

- The reasons to select one cost function over another vary. For example, we might try to minimise latency simply to avoid circuitous routes (i.e., a route longer than the most direct one), bandwidth or cost to avoid slow or expansive links, or hop count to reduce the load on (local) switches in each sub-network. Clearly all nodes that engage in the same routing algorithm will need to use the same cost function (otherwise the global result will be unclear at best).

- Given an example graph, e.g.,



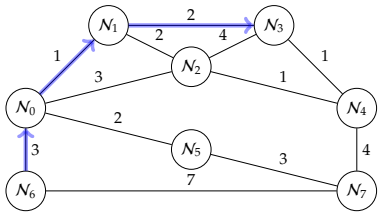
keep in mind some important facts

which mean we *only* need to consider the destination node.

Notes:

- In more detail, the routing algorithm does not need to consider the source of a given message to route it effectively: once it gets to node u (given the destination is node v), the next hop from u toward v is all we need (it doesn't matter what the original source was).

- Given an example graph, e.g.,



keep in mind some important facts

- in a given shortest path, any sub-path is also a shortest path (otherwise the super-path isn't the shortest),

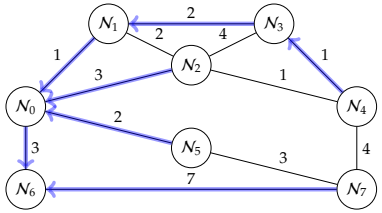
which mean we *only* need to consider the destination node.

Notes:

- In more detail, the routing algorithm does not need to consider the source of a given message to route it effectively: once it gets to node u (given the destination is node v), the next hop from u toward v is all we need (it doesn't matter what the original source was).

Routing Techniques (2) – Path Finding

- Given an example graph, e.g.,



keep in mind some important facts

- a sink tree (resp. source tree) for node u is the union of shortest paths ending at (resp. starting at) u,
- it has to be a tree, because if two paths meet then the next hop will be the same

which mean we *only* need to consider the destination node.

Notes:

- In more detail, the routing algorithm does not need to consider the source of a given message to route it effectively: once it gets to node u (given the destination is node v), the next hop from u toward v is all we need (it doesn't matter what the original source was).

- Given an example graph, e.g.,

keep in mind some important facts

- in a given shortest path, any sub-path is also a shortest path (otherwise the super-path isn't the shortest),
- a sink tree (resp. source tree) for node u is the union of shortest paths ending at (resp. starting at) u,
- it has to be a tree, because if two paths meet then the next hop will be the same

which mean we *only* need to consider the destination node.

Notes:

- In more detail, the routing algorithm does not need to consider the source of a given message to route it effectively: once it gets to node u (given the destination is node v), the next hop from u toward v is all we need (it doesn't matter what the original source was).

Algorithm (Dijkstra)

Input: A graph $G = (V, E)$ and source node $s \in V$
Output: A mapping $dist$ st. $dist(s, u)$ is the shortest path between nodes s and u

```
1  foreach u ∈ V do
2  |  dist(s, u) ← ∞
3  end
4  dist(s, s) ← 0, queue ← ∅
5  foreach u ∈ V do
6  |  Insert(queue, u, dist(s, u))
7  end
8  while |queue| > 0 do
9  |  u ← ExtractMin(queue)
10 |  foreach (u, v, d) ∈ E do
11 |  |  if dist(s, v) > dist(s, u) + d then
12 |  |  |  dist(s, v) ← dist(s, u) + d
13 |  |  |  DecreaseKey(queue, v, dist(s, v))
14 |  |  end
15 |  end
16 end
```

Algorithm (Bellman-Ford)

Input: A graph $G = (V, E)$ and source node $s \in V$
Output: A mapping $dist$ st. $dist(s, u)$ is the shortest path between nodes s and u

```
1  foreach u ∈ V do
2  |  dist(s, u) ← ∞
3  end
4  dist(s, s) ← 0
5  for i = 1 upto |V| do
6  |  foreach (u, v, d) ∈ E do
7  |  |  if dist(s, v) > dist(s, u) + d then
8  |  |  |  dist(s, v) ← dist(s, u) + d
9  |  |  end
10 |  end
11 end
```


Notes:

- Dijkstra's algorithm for computing shortest paths cannot cope with graphs that include negative weights. Within the context of routing, this issue can easily be avoided (most useful cost functions do so naturally) and so isn't necessarily a disadvantage versus alternatives. A *much* more important issue is that *both* algorithms need (global) information about the graph topology: in the model we are working in, this is almost certainly not available without extra steps.

Algorithm (Dijkstra)

Input: A graph $G = (V, E)$ and source node $s \in V$
Output: A mapping *dist* st. $dist(s, u)$ is the shortest path between nodes s and u, plus *hop* st. $hop(s, u)$ is the next hop from s toward u

```
1  foreach u ∈ V do
2  |  dist(s, u) ← ∞
3  end
4  dist(s, s) ← 0, queue ← ∅
5  foreach u ∈ V do
6  |  Insert(queue, u, dist(s, u))
7  end
8  while |queue| > 0 do
9  |  u ← ExtractMin(queue)
10 |  foreach (u, v, d) ∈ E do
11 |  |  if dist(s, v) > dist(s, u) + d then
12 |  |  |  dist(s, v) ← dist(s, u) + d, hop(s, v) ← u
13 |  |  |  DecreaseKey(queue, v, dist(s, v))
14 |  |  end
15 |  end
16 end
```

Algorithm (Bellman-Ford)

Input: A graph $G = (V, E)$ and source node $s \in V$
Output: A mapping *dist* st. $dist(s, u)$ is the shortest path between nodes s and u, plus *hop* st. $hop(s, u)$ is the next hop from s toward u

```
1  foreach u ∈ V do
2  |  dist(s, u) ← ∞
3  end
4  dist(s, s) ← 0
5  for i = 1 upto |V| do
6  |  foreach (u, v, d) ∈ E do
7  |  |  if dist(s, v) > dist(s, u) + d then
8  |  |  |  dist(s, v) ← dist(s, u) + d, hop(s, v) ← u
9  |  |  end
10 |  end
11 end
```

Notes:

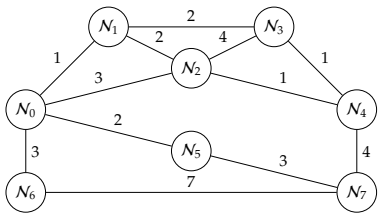
- Dijkstra's algorithm for computing shortest paths cannot cope with graphs that include negative weights. Within the context of routing, this issue can easily be avoided (most useful cost functions do so naturally) and so isn't necessarily a disadvantage versus alternatives. A *much* more important issue is that *both* algorithms need (global) information about the graph topology: in the model we are working in, this is almost certainly not available without extra steps.

- We can't do inter-network broadcast per se (since sub-networks may use different technologies) ...
- ... but **flooding** offers a high(er)-level analogy: goal is to transmit a message to *all* nodes.
- Flooding may be useful a building block, *or* as a routing "algorithm".
- Example:

Algorithm (**flood**)

Imagine node u wants to flood the network with some message m:

- u transmit m to all neighbours (bar any node that transmitted m to u),
- the neighbouring nodes do the same, remembering a sequence number to avoid cycles (i.e., repeat transmission of m),
- flooding terminates once all nodes receive m (resp. no nodes need to transmit m).



Notes:

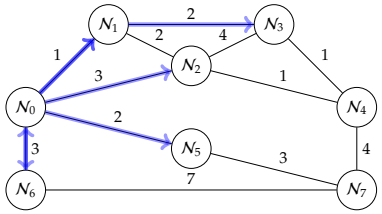
- It should be obvious that the flooding process is potentially *very* inefficient: a large amount of communication is needed, some computation and storage is required to maintain and check the sequence numbers, plus a given node could receive *many* copies of the message!

- We can't do inter-network broadcast per se (since sub-networks may use different technologies) ...
- ... but **flooding** offers a high(er)-level analogy: goal is to transmit a message to *all* nodes.
- Flooding may be useful a building block, *or* as a routing "algorithm".
- Example:

Algorithm (**flood**)

Imagine node u wants to flood the network with some message m:

- u transmit m to all neighbours (bar any node that transmitted m to u),
- the neighbouring nodes do the same, remembering a sequence number to avoid cycles (i.e., repeat transmission of m),
- flooding terminates once all nodes receive m (resp. no nodes need to transmit m).



Notes:

- It should be obvious that the flooding process is potentially *very* inefficient: a large amount of communication is needed, some computation and storage is required to maintain and check the sequence numbers, plus a given node could receive *many* copies of the message!

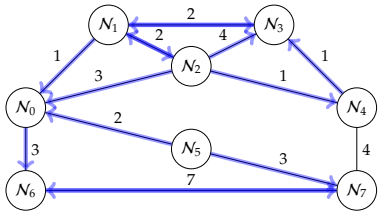
Routing Techniques (4) – Flooding

- We can't do inter-network broadcast per se (since sub-networks may use different technologies) ...
- ... but **flooding** offers a high(er)-level analogy: goal is to transmit a message to *all* nodes.
- Flooding may be useful a building block, *or* as a routing "algorithm".
- Example:

Algorithm (**flood**)

Imagine node u wants to flood the network with some message m:

- u transmit m to all neighbours (bar any node that transmitted m to u),
- the neighbouring nodes do the same, remembering a sequence number to avoid cycles (i.e., repeat transmission of m),
- flooding terminates once all nodes receive m (resp. no nodes need to transmit m).



Notes:

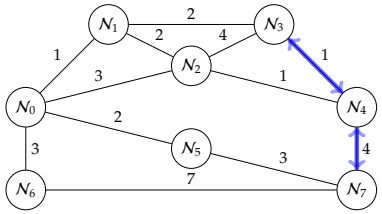
- It should be obvious that the flooding process is potentially *very* inefficient: a large amount of communication is needed, some computation and storage is required to maintain and check the sequence numbers, plus a given node could receive *many* copies of the message!

- We can't do inter-network broadcast per se (since sub-networks may use different technologies) ...
- ... but **flooding** offers a high(er)-level analogy: goal is to transmit a message to *all* nodes.
- Flooding may be useful a building block, *or* as a routing "algorithm".
- Example:

Algorithm (**flood**)

Imagine node u wants to flood the network with some message m:

- u transmit m to all neighbours (bar any node that transmitted m to u),
- the neighbouring nodes do the same, remembering a sequence number to avoid cycles (i.e., repeat transmission of m),
- flooding terminates once all nodes receive m (resp. no nodes need to transmit m).



Notes:

- It should be obvious that the flooding process is potentially *very* inefficient: a large amount of communication is needed, some computation and storage is required to maintain and check the sequence numbers, plus a given node could receive *many* copies of the message!

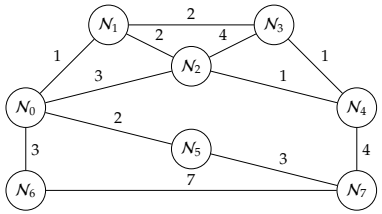
Routing Techniques (4) – Flooding

- We can't do inter-network broadcast per se (since sub-networks may use different technologies) ...
- ... but **flooding** offers a high(er)-level analogy: goal is to transmit a message to *all* nodes.
- Flooding may be useful a building block, *or* as a routing "algorithm".
- Example:

Algorithm (**flood**)

Imagine node u wants to flood the network with some message m:

- u transmit m to all neighbours (bar any node that transmitted m to u),
- the neighbouring nodes do the same, remembering a sequence number to avoid cycles (i.e., repeat transmission of m),
- flooding terminates once all nodes receive m (resp. no nodes need to transmit m).



Notes:

- It should be obvious that the flooding process is potentially *very* inefficient: a large amount of communication is needed, some computation and storage is required to maintain and check the sequence numbers, plus a given node could receive *many* copies of the message!

Routing Algorithms (1)

- Idea: in general, we'll have each routing algorithm
  1. maintain some state, i.e., a **routing table**,
  2. periodically transmit, receive and integrate information via (local) communication with neighbouring nodes,
  3. periodically translate the routing table into forwarding table, e.g., via some form of (local) computation

  keeping in mind that
  - periodically means at regular intervals, *plus* when a change is topology occurs, and
  - a cost of ∞ means a link does not exist *or* has failed: either way, avoid it!

Notes:

Routing Algorithms (2) – Distance Vector Routing

- Idea:

  distance vector routing $\simeq$ distributed Bellman-Ford,

  in the sense each node u
  1. maintains a **distance vector**

     $$\langle (v_0, d_0), (v_1, d_1), \ldots, (v_{l-1}, d_{l-1}) \rangle$$

     of next hop and cost tuples, initialised st.

     $$d_i = \begin{cases} 0 & \text{if } v_i = u \\ \infty & \text{otherwise} \end{cases},$$

  2. periodically transmits the distance vector to all neighbours,
  3. periodically updates the distance vector with new information, st.

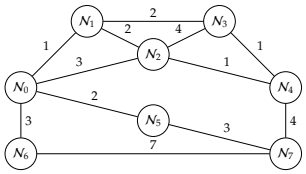
     $$dist(u, v) = \min_{\forall w,\ (u,w,d) \in E} \left[dist(w, v) + d \right].$$

- Example: **Routing Information Protocol (RIP)** [5].
  - uses hop count as a cost function, assuming $\infty \equiv 16$,
  - transmits distance vectors every $\sim$ 30s with $\sim$ 180s failure time-out.

Notes:

Routing Algorithms (3) – Distance Vector Routing

- Recalling that the graph in question is,



the algorithm proceeds as follows:

		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7
N_0	*dist* =	0	∞	∞	∞	∞	∞	∞	∞
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
N_1	*dist* =	∞	0	∞	∞	∞	∞	∞	∞
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
N_2	*dist* =	∞	∞	0	∞	∞	∞	∞	∞
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
N_3	*dist* =	∞	∞	∞	0	∞	∞	∞	∞
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
N_4	*dist* =	∞	∞	∞	∞	0	∞	∞	∞
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
N_5	*dist* =	∞	∞	∞	∞	∞	0	∞	∞
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
N_6	*dist* =	∞	∞	∞	∞	∞	∞	0	∞
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥
N_7	*dist* =	∞	∞	∞	∞	∞	∞	∞	0
	hop =	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

⇝

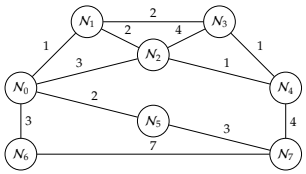
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7
N_0	*dist* =	0	1	3	∞	∞	2	3	∞
	hop =	⊥	N_1	N_2	⊥	⊥	N_5	N_6	⊥
N_1	*dist* =	1	0	2	2	∞	∞	∞	∞
	hop =	N_0	⊥	N_2	N_3	⊥	⊥	⊥	⊥
N_2	*dist* =	3	2	0	4	1	∞	∞	∞
	hop =	N_0	N_1	⊥	N_3	N_4	⊥	⊥	⊥
N_3	*dist* =	∞	2	4	0	1	∞	∞	∞
	hop =	⊥	N_1	N_2	⊥	N_4	⊥	⊥	⊥
N_4	*dist* =	∞	∞	1	1	0	∞	∞	4
	hop =	⊥	⊥	N_2	N_3	⊥	⊥	⊥	N_7
N_5	*dist* =	2	∞	∞	∞	∞	0	∞	3
	hop =	N_0	⊥	⊥	⊥	⊥	⊥	⊥	N_7
N_6	*dist* =	3	∞	∞	∞	∞	∞	0	7
	hop =	N_0	⊥	⊥	⊥	⊥	⊥	⊥	N_7
N_7	*dist* =	∞	∞	∞	∞	4	3	7	0
	hop =	⊥	⊥	⊥	⊥	N_4	N_5	N_6	⊥

Notes:

- If you look at what happens at each step, it becomes clear that after the i-th update node u learns about all i-hop routes (since that is how long it takes for the associated distance vector to be propagated to it). Eventually, the solution converges: there are no changes. However, keep in mind that the nodes continue to run the algorithm, since a) they cannot really know there are no changes pending (e.g., being propagated through the network), plus b) they may need to accommodate later changes to the topology (e.g., adding or removing nodes).
- Adding a node is simple. Once it is connected, it will start to exchange distance vectors with neighbouring nodes; this information with propagate through the network (at a rate of one hop per-exchange). Removing a node is more tricky, in the sense that node cannot inform the network: it has been removed, and/or is non-operational! Therefore, nodes need some way to "forget" or time-out entries in their distance vector.

Routing Algorithms (3) – Distance Vector Routing

- Recalling that the graph in question is,



the algorithm proceeds as follows:

		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7
N_0	*dist* =	0	1	3	∞	∞	2	3	∞
	hop =	⊥	N_1	N_2	⊥	⊥	N_5	N_6	⊥
N_1	*dist* =	1	0	2	2	∞	∞	∞	∞
	hop =	N_0	⊥	N_2	N_3	⊥	⊥	⊥	⊥
N_2	*dist* =	3	2	0	4	1	∞	∞	∞
	hop =	N_0	N_1	⊥	N_3	N_4	⊥	⊥	⊥
N_3	*dist* =	∞	2	4	0	1	∞	∞	∞
	hop =	⊥	N_1	N_2	⊥	N_4	⊥	⊥	⊥
N_4	*dist* =	∞	∞	1	1	0	∞	∞	4
	hop =	⊥	⊥	N_2	N_3	⊥	⊥	⊥	N_7
N_5	*dist* =	2	∞	∞	∞	∞	0	∞	3
	hop =	N_0	⊥	⊥	⊥	⊥	⊥	⊥	N_7
N_6	*dist* =	3	∞	∞	∞	∞	∞	0	7
	hop =	N_0	⊥	⊥	⊥	⊥	⊥	⊥	N_7
N_7	*dist* =	∞	∞	∞	∞	4	3	7	0
	hop =	⊥	⊥	⊥	⊥	N_4	N_5	N_6	⊥

⇝

		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7
N_0	*dist* =	0	1	3	3	4	2	3	5
	hop =	⊥	N_1	N_2	N_1	N_2	N_5	N_6	N_5
N_1	*dist* =	1	0	2	2	3	3	4	∞
	hop =	N_0	⊥	N_2	N_3	N_2	N_0	N_0	⊥
N_2	*dist* =	3	2	0	2	1	5	6	5
	hop =	N_0	N_1	⊥	N_4	N_4	N_0	N_0	N_4
N_3	*dist* =	3	2	2	0	1	∞	∞	5
	hop =	N_1	N_1	N_4	⊥	N_4	⊥	⊥	N_4
N_4	*dist* =	4	3	1	1	0	7	11	4
	hop =	N_2	N_2	N_2	N_3	⊥	N_7	N_7	N_7
N_5	*dist* =	2	3	5	∞	7	0	5	3
	hop =	N_0	N_0	N_0	⊥	N_7	⊥	N_0	N_7
N_6	*dist* =	3	4	6	∞	11	5	0	7
	hop =	N_0	N_0	N_0	⊥	N_7	N_0	⊥	N_7
N_7	*dist* =	5	∞	5	5	4	3	7	0
	hop =	N_5	⊥	N_4	N_4	N_4	N_5	N_6	⊥

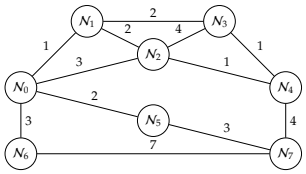
Notes:

- If you look at what happens at each step, it becomes clear that after the i-th update node u learns about all i-hop routes (since that is how long it takes for the associated distance vector to be propagated to it). Eventually, the solution converges: there are no changes. However, keep in mind that the nodes continue to run the algorithm, since a) they cannot really know there are no changes pending (e.g., being propagated through the network), plus b) they may need to accommodate later changes to the topology (e.g., adding or removing nodes).
- Adding a node is simple. Once it is connected, it will start to exchange distance vectors with neighbouring nodes; this information with propagate through the network (at a rate of one hop per-exchange). Removing a node is more tricky, in the sense that node cannot inform the network: it has been removed, and/or is non-operational! Therefore, nodes need some way to "forget" or time-out entries in their distance vector.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11
University of BRISTOL

Routing Algorithms (3) – Distance Vector Routing

- Recalling that the graph in question is,



the algorithm proceeds as follows:

		$\mathcal{N}_0$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_7$
$\mathcal{N}_0$	*dist* =	0	1	3	3	4	2	3	5
	hop =	$\perp$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_5$
$\mathcal{N}_1$	*dist* =	1	0	2	2	3	3	4	∞
	hop =	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_2$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$
$\mathcal{N}_2$	*dist* =	3	2	0	2	1	5	6	5
	hop =	$\mathcal{N}_0$	$\mathcal{N}_1$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_4$
$\mathcal{N}_3$	*dist* =	3	2	2	0	1	∞	∞	5
	hop =	$\mathcal{N}_1$	$\mathcal{N}_1$	$\mathcal{N}_4$	$\perp$	$\mathcal{N}_4$	$\perp$	$\perp$	$\mathcal{N}_4$
$\mathcal{N}_4$	*dist* =	4	3	1	1	0	7	11	4
	hop =	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\perp$	$\mathcal{N}_7$	$\mathcal{N}_7$	$\mathcal{N}_7$
$\mathcal{N}_5$	*dist* =	2	3	5	∞	7	0	5	3
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_7$	$\perp$	$\mathcal{N}_0$	$\mathcal{N}_7$
$\mathcal{N}_6$	*dist* =	3	4	6	∞	11	5	0	7
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_7$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_7$
$\mathcal{N}_7$	*dist* =	5	∞	5	5	4	3	7	0
	hop =	$\mathcal{N}_5$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\perp$

$\rightsquigarrow$

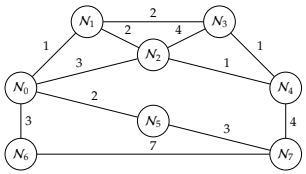
		$\mathcal{N}_0$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_7$
$\mathcal{N}_0$	*dist* =	0	1	3	3	4	2	3	5
	hop =	$\perp$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_5$
$\mathcal{N}_1$	*dist* =	1	0	2	2	3	3	4	6
	hop =	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_2$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$
$\mathcal{N}_2$	*dist* =	3	2	0	2	1	5	6	5
	hop =	$\mathcal{N}_0$	$\mathcal{N}_1$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_4$
$\mathcal{N}_3$	*dist* =	3	2	2	0	1	5	6	5
	hop =	$\mathcal{N}_1$	$\mathcal{N}_1$	$\mathcal{N}_4$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_1$	$\mathcal{N}_1$	$\mathcal{N}_4$
$\mathcal{N}_4$	*dist* =	4	3	1	1	0	6	7	4
	hop =	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\perp$	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_7$
$\mathcal{N}_5$	*dist* =	2	3	5	5	6	0	5	3
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_0$	$\mathcal{N}_7$
$\mathcal{N}_6$	*dist* =	3	4	6	6	7	5	0	7
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_7$
$\mathcal{N}_7$	*dist* =	5	6	5	5	4	3	7	0
	hop =	$\mathcal{N}_5$	$\mathcal{N}_5$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\perp$

Notes:

- If you look at what happens at each step, it becomes clear that after the i-th update node u learns about all i-hop routes (since that is how long it takes for the associated distance vector to be propagated to it). Eventually, the solution converges: there are no changes. However, keep in mind that the nodes continue to run the algorithm, since a) they cannot really know there are no changes pending (e.g., being propagated through the network), plus b) they may need to accommodate later changes to the topology (e.g., adding or removing nodes).
- Adding a node is simple. Once it is connected, it will start to exchange distance vectors with neighbouring nodes; this information with propagate through the network (at a rate of one hop per-exchange). Removing a node is more tricky, in the sense that node cannot inform the network: it has been removed, and/or is non-operational! Therefore, nodes need some way to "forget" or time-out entries in their distance vector.

Routing Algorithms (3) – Distance Vector Routing

- Recalling that the graph in question is,



the algorithm proceeds as follows:

		$\mathcal{N}_0$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_7$
$\mathcal{N}_0$	*dist* =	0	1	3	3	4	2	3	5
	hop =	$\perp$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_5$
$\mathcal{N}_1$	*dist* =	1	0	2	2	3	3	4	6
	hop =	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_2$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$
$\mathcal{N}_2$	*dist* =	3	2	0	2	1	5	6	5
	hop =	$\mathcal{N}_0$	$\mathcal{N}_1$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_4$
$\mathcal{N}_3$	*dist* =	3	2	2	0	1	5	6	5
	hop =	$\mathcal{N}_1$	$\mathcal{N}_1$	$\mathcal{N}_4$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_1$	$\mathcal{N}_1$	$\mathcal{N}_4$
$\mathcal{N}_4$	*dist* =	4	3	1	1	0	6	7	4
	hop =	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\perp$	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_7$
$\mathcal{N}_5$	*dist* =	2	3	5	5	6	0	5	3
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_0$	$\mathcal{N}_7$
$\mathcal{N}_6$	*dist* =	3	4	6	6	7	5	0	7
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_7$
$\mathcal{N}_7$	*dist* =	5	6	5	5	4	3	7	0
	hop =	$\mathcal{N}_5$	$\mathcal{N}_5$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\perp$

$\rightsquigarrow$

		$\mathcal{N}_0$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_7$
$\mathcal{N}_0$	*dist* =	0	1	3	3	4	2	3	5
	hop =	$\perp$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_1$	$\mathcal{N}_2$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\mathcal{N}_5$
$\mathcal{N}_1$	*dist* =	1	0	2	2	3	3	4	6
	hop =	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\mathcal{N}_2$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$
$\mathcal{N}_2$	*dist* =	3	2	0	2	1	5	6	5
	hop =	$\mathcal{N}_0$	$\mathcal{N}_1$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_4$
$\mathcal{N}_3$	*dist* =	3	2	2	0	1	5	6	5
	hop =	$\mathcal{N}_1$	$\mathcal{N}_1$	$\mathcal{N}_4$	$\perp$	$\mathcal{N}_4$	$\mathcal{N}_1$	$\mathcal{N}_1$	$\mathcal{N}_4$
$\mathcal{N}_4$	*dist* =	4	3	1	1	0	6	7	4
	hop =	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_3$	$\perp$	$\mathcal{N}_2$	$\mathcal{N}_2$	$\mathcal{N}_7$
$\mathcal{N}_5$	*dist* =	2	3	5	5	6	0	5	3
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_0$	$\mathcal{N}_7$
$\mathcal{N}_6$	*dist* =	3	4	6	6	7	5	0	7
	hop =	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\mathcal{N}_0$	$\perp$	$\mathcal{N}_7$
$\mathcal{N}_7$	*dist* =	5	6	5	5	4	3	7	0
	hop =	$\mathcal{N}_5$	$\mathcal{N}_5$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_4$	$\mathcal{N}_5$	$\mathcal{N}_6$	$\perp$

Notes:

- If you look at what happens at each step, it becomes clear that after the i-th update node u learns about all i-hop routes (since that is how long it takes for the associated distance vector to be propagated to it). Eventually, the solution converges: there are no changes. However, keep in mind that the nodes continue to run the algorithm, since a) they cannot really know there are no changes pending (e.g., being propagated through the network), plus b) they may need to accommodate later changes to the topology (e.g., adding or removing nodes).
- Adding a node is simple. Once it is connected, it will start to exchange distance vectors with neighbouring nodes; this information with propagate through the network (at a rate of one hop per-exchange). Removing a node is more tricky, in the sense that node cannot inform the network: it has been removed, and/or is non-operational! Therefore, nodes need some way to "forget" or time-out entries in their distance vector.

Routing Algorithms (1) – Link State Routing

- Idea:

  link state routing $\simeq$ flooding + Dijkstra,

  in the sense each node u

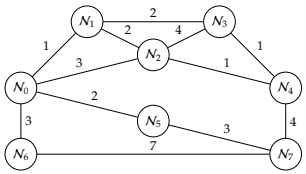
  1. floods network with **link state packets** (i.e., neighbouring nodes plus link costs) yielding global topology, then
  2. use Dijkstra to compute shortest paths.

- Examples:
  - **Open Shortest Path First (OSPF)** [7], and
  - **Intermediate System to Intermediate System (IS-IS)** [8].

Notes:

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



  the algorithm

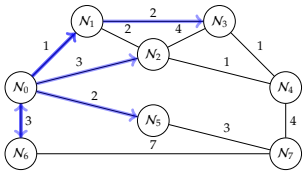
  1. uses flooding to communicate the topology to all nodes,
  2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

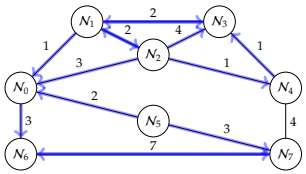
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

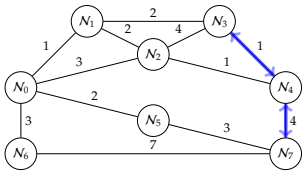
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

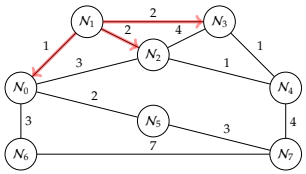
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

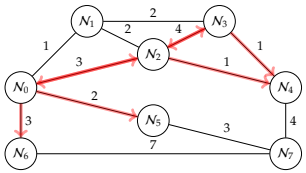
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

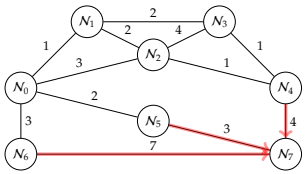
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

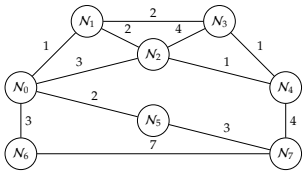
Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

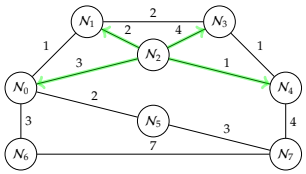
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

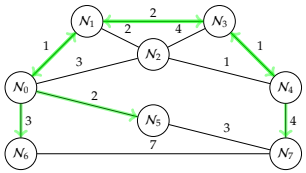
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

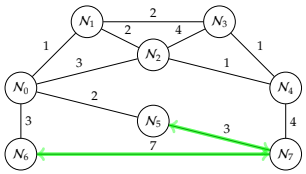
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

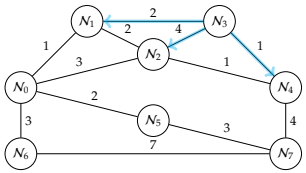
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

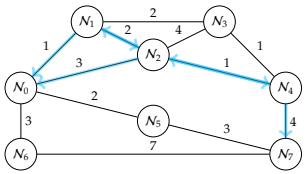
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

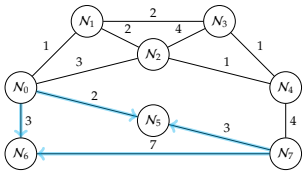
Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

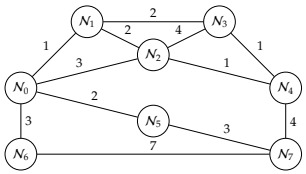
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

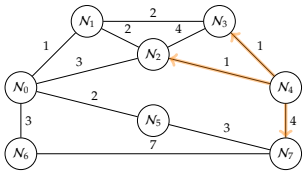
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

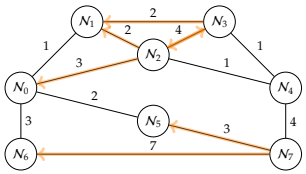
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

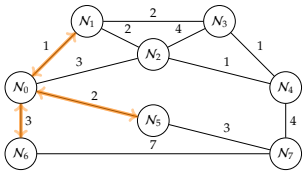
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

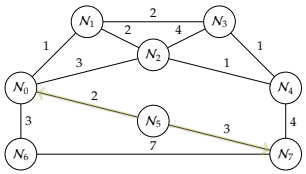
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

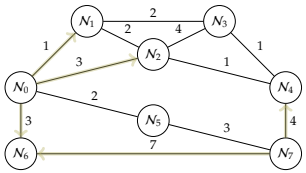
Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

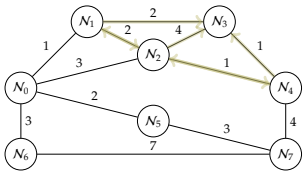
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

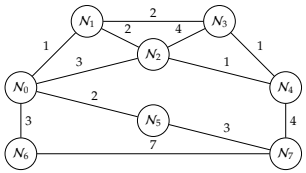
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

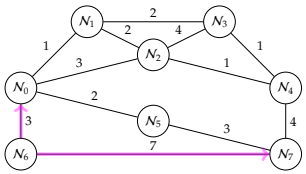
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

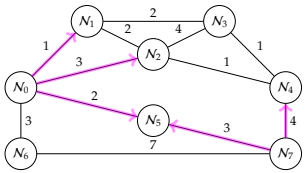
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

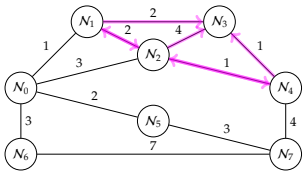
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

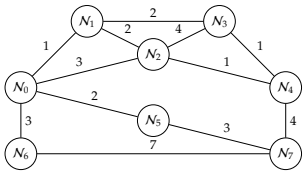
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

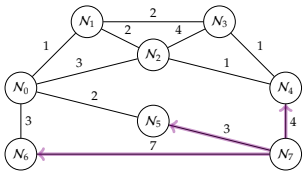
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

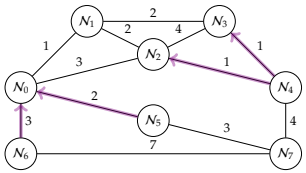
Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

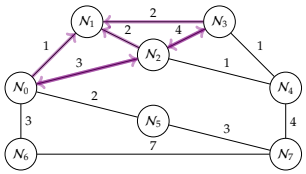
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

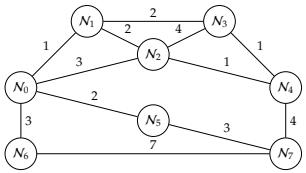
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

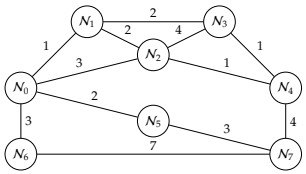
1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra.

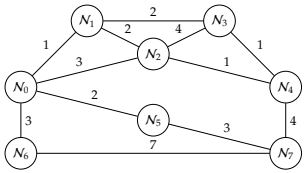
Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Computer Networks) git # d87e713 @ 2016-04-11 University of BRISTOL

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

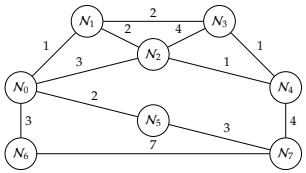
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	$dist =$	0	∞	∞	∞	∞	∞	∞	∞	$queue = \langle (N_0, 0), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, \infty), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	
N_1	$dist =$	∞	0	∞	∞	∞	∞	∞	∞	$queue = \langle (N_0, \infty), (N_1, 0), (N_2, \infty), (N_3, \infty), (N_4, \infty), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	
N_2	$dist =$	∞	∞	0	∞	∞	∞	∞	∞	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, 0), (N_3, \infty), (N_4, \infty), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	
N_3	$dist =$	∞	∞	∞	0	∞	∞	∞	∞	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, \infty), (N_3, 0), (N_4, \infty), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	
N_4	$dist =$	∞	∞	∞	∞	0	∞	∞	∞	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, 0), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	
N_5	$dist =$	∞	∞	∞	∞	∞	0	∞	∞	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, \infty), (N_5, 0), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	
N_6	$dist =$	∞	∞	∞	∞	∞	∞	0	∞	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, \infty), (N_5, \infty), (N_6, 0), (N_7, \infty) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	
N_7	$dist =$	∞	∞	∞	∞	∞	∞	∞	0	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, \infty), (N_5, \infty), (N_6, \infty), (N_7, 0) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

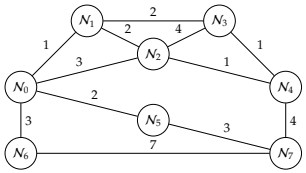
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	$dist =$	0	1	3	∞	∞	2	3	∞	$queue = \langle (N_1, 1), (N_2, 3), (N_3, \infty), (N_4, \infty), (N_5, 2), (N_6, 3), (N_7, \infty) \rangle$
	$hop =$	⊥	N_1	N_2	⊥	⊥	N_5	N_6	⊥	
N_1	$dist =$	1	0	2	2	∞	∞	∞	∞	$queue = \langle (N_0, 1), (N_2, 2), (N_3, 2), (N_4, \infty), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	N_0	⊥	N_2	N_3	⊥	⊥	⊥	⊥	
N_2	$dist =$	3	2	0	4	1	∞	∞	∞	$queue = \langle (N_0, 3), (N_1, 2), (N_3, 4), (N_4, 1), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	N_0	N_1	⊥	N_3	N_4	⊥	⊥	⊥	
N_3	$dist =$	∞	2	4	0	1	∞	∞	∞	$queue = \langle (N_0, \infty), (N_1, 2), (N_2, 4), (N_4, 1), (N_5, \infty), (N_6, \infty), (N_7, \infty) \rangle$
	$hop =$	⊥	N_1	N_2	⊥	N_4	⊥	⊥	⊥	
N_4	$dist =$	∞	∞	1	1	0	∞	∞	4	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, 1), (N_3, 1), (N_5, \infty), (N_6, \infty), (N_7, 4) \rangle$
	$hop =$	⊥	⊥	N_2	N_3	⊥	⊥	⊥	N_7	
N_5	$dist =$	2	∞	∞	∞	∞	0	∞	3	$queue = \langle (N_0, 2), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, \infty), (N_6, \infty), (N_7, 3) \rangle$
	$hop =$	N_0	⊥	⊥	⊥	⊥	⊥	⊥	N_7	
N_6	$dist =$	3	∞	∞	∞	∞	∞	0	7	$queue = \langle (N_0, 3), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, \infty), (N_5, \infty), (N_7, 7) \rangle$
	$hop =$	N_0	⊥	⊥	⊥	⊥	⊥	⊥	N_7	
N_7	$dist =$	∞	∞	∞	∞	4	3	7	0	$queue = \langle (N_0, \infty), (N_1, \infty), (N_2, \infty), (N_3, \infty), (N_4, 4), (N_5, 3), (N_6, 7) \rangle$
	$hop =$	⊥	⊥	⊥	⊥	N_4	N_5	N_6	⊥	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

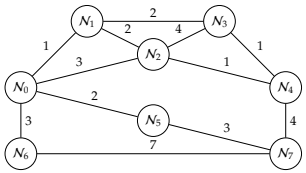
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	$dist =$	0	1	3	3	∞	2	3	∞	$queue = \langle (N_2,3),(N_5,3),(N_4,\infty),(N_5,2),(N_6,3),(N_7,\infty)\rangle$
	$hop =$	$\perp$	N_1	N_2	N_1	$\perp$	N_5	N_6	$\perp$	
N_1	$dist =$	1	0	2	2	∞	3	4	∞	$queue = \langle (N_2,2),(N_3,2),(N_4,\infty),(N_5,3),(N_6,4),(N_7,\infty)\rangle$
	$hop =$	N_0	$\perp$	N_2	N_3	$\perp$	N_0	N_0	$\perp$	
N_2	$dist =$	3	2	0	2	1	∞	∞	5	$queue = \langle (N_0,3),(N_1,2),(N_3,2),(N_5,\infty),(N_6,\infty),(N_7,5)\rangle$
	$hop =$	N_0	N_1	$\perp$	N_4	N_4	$\perp$	$\perp$	N_4	
N_3	$dist =$	∞	2	2	0	1	∞	∞	5	$queue = \langle (N_0,\infty),(N_1,2),(N_2,2),(N_5,\infty),(N_6,\infty),(N_7,5)\rangle$
	$hop =$	$\perp$	N_1	N_4	$\perp$	N_4	$\perp$	$\perp$	N_4	
N_4	$dist =$	4	3	1	1	0	∞	∞	4	$queue = \langle (N_0,4),(N_1,3),(N_3,1),(N_5,\infty),(N_6,\infty),(N_7,4)\rangle$
	$hop =$	N_2	N_2	N_2	N_3	$\perp$	$\perp$	$\perp$	N_7	
N_5	$dist =$	2	3	5	∞	∞	0	5	3	$queue = \langle (N_1,3),(N_2,5),(N_3,\infty),(N_4,\infty),(N_6,5),(N_7,3)\rangle$
	$hop =$	N_0	N_0	N_0	$\perp$	$\perp$	$\perp$	N_0	N_7	
N_6	$dist =$	3	4	6	∞	∞	5	0	7	$queue = \langle (N_1,4),(N_2,6),(N_3,\infty),(N_4,\infty),(N_5,5),(N_7,7)\rangle$
	$hop =$	N_0	N_0	N_0	$\perp$	$\perp$	N_0	$\perp$	N_7	
N_7	$dist =$	5	∞	∞	∞	4	3	7	0	$queue = \langle (N_0,5),(N_1,\infty),(N_2,\infty),(N_3,\infty),(N_4,4),(N_6,7)\rangle$
	$hop =$	N_5	$\perp$	$\perp$	$\perp$	N_4	N_5	N_6	$\perp$	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

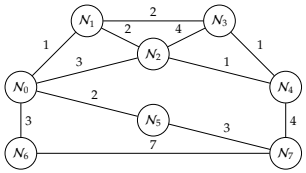
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	$dist =$	0	1	3	3	∞	2	3	5	$queue = \langle (N_2,3),(N_5,3),(N_4,\infty),(N_6,3),(N_7,5)\rangle$
	$hop =$	$\perp$	N_1	N_2	N_1	$\perp$	N_5	N_6	N_5	
N_1	$dist =$	1	0	2	2	3	3	4	∞	$queue = \langle (N_3,2),(N_4,3),(N_5,3),(N_6,4),(N_7,\infty)\rangle$
	$hop =$	N_0	$\perp$	N_2	N_3	N_2	N_0	N_0	$\perp$	
N_2	$dist =$	3	2	0	2	1	∞	∞	5	$queue = \langle (N_0,3),(N_3,2),(N_5,\infty),(N_6,\infty),(N_7,5)\rangle$
	$hop =$	N_0	N_1	$\perp$	N_4	N_4	$\perp$	$\perp$	N_4	
N_3	$dist =$	3	2	2	0	1	∞	∞	5	$queue = \langle (N_0,3),(N_2,2),(N_5,\infty),(N_6,\infty),(N_7,5)\rangle$
	$hop =$	N_1	N_1	N_4	$\perp$	N_4	$\perp$	$\perp$	N_4	
N_4	$dist =$	4	3	1	1	0	∞	∞	4	$queue = \langle (N_0,4),(N_1,3),(N_5,\infty),(N_6,\infty),(N_7,4)\rangle$
	$hop =$	N_2	N_2	N_2	N_3	$\perp$	$\perp$	$\perp$	N_7	
N_5	$dist =$	2	3	5	5	∞	0	5	3	$queue = \langle (N_2,5),(N_3,5),(N_4,\infty),(N_6,5),(N_7,3)\rangle$
	$hop =$	N_0	N_0	N_0	N_1	$\perp$	$\perp$	N_0	N_7	
N_6	$dist =$	3	4	6	6	∞	5	0	7	$queue = \langle (N_2,6),(N_3,6),(N_4,\infty),(N_5,5),(N_7,7)\rangle$
	$hop =$	N_0	N_0	N_0	N_1	$\perp$	N_0	$\perp$	N_7	
N_7	$dist =$	5	∞	5	5	4	3	7	0	$queue = \langle (N_0,5),(N_1,\infty),(N_2,5),(N_3,5),(N_6,7)\rangle$
	$hop =$	N_5	$\perp$	N_4	N_4	N_4	N_5	N_6	$\perp$	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

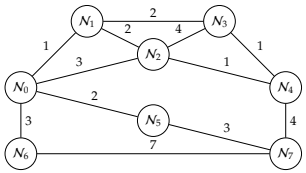
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	$dist =$	0	1	3	3	4	2	3	5	$queue = \langle (N_5, 3), (N_4, 4), (N_6, 3), (N_7, 5) \rangle$
	$hop =$	$\perp$	N_1	N_2	N_1	N_2	N_5	N_6	N_5	
N_1	$dist =$	1	0	2	2	3	3	4	∞	$queue = \langle (N_4, 3), (N_5, 3), (N_6, 4), (N_7, \infty) \rangle$
	$hop =$	N_0	$\perp$	N_2	N_3	N_2	N_0	N_0	$\perp$	
N_2	$dist =$	3	2	0	2	1	∞	∞	5	$queue = \langle (N_0, 3), (N_5, \infty), (N_6, \infty), (N_7, 5) \rangle$
	$hop =$	N_0	N_1	$\perp$	N_4	N_4	$\perp$	$\perp$	N_4	
N_3	$dist =$	3	2	2	0	1	∞	∞	5	$queue = \langle (N_0, 3), (N_5, \infty), (N_6, \infty), (N_7, 5) \rangle$
	$hop =$	N_1	N_1	N_4	$\perp$	N_4	$\perp$	$\perp$	N_4	
N_4	$dist =$	4	3	1	1	0	∞	∞	4	$queue = \langle (N_0, 4), (N_5, \infty), (N_6, \infty), (N_7, 4) \rangle$
	$hop =$	N_2	N_2	N_2	N_3	$\perp$	$\perp$	$\perp$	N_7	
N_5	$dist =$	2	3	5	5	7	0	5	3	$queue = \langle (N_2, 5), (N_3, 5), (N_4, 7), (N_6, 5) \rangle$
	$hop =$	N_0	N_0	N_0	N_1	N_7	$\perp$	N_0	N_7	
N_6	$dist =$	3	4	6	6	∞	5	0	7	$queue = \langle (N_2, 6), (N_3, 6), (N_4, \infty), (N_7, 7) \rangle$
	$hop =$	N_0	N_0	N_0	N_1	$\perp$	N_0	$\perp$	N_7	
N_7	$dist =$	5	6	5	5	4	3	7	0	$queue = \langle (N_1, 6), (N_2, 5), (N_3, 5), (N_6, 7) \rangle$
	$hop =$	N_5	N_0	N_4	N_4	N_4	N_5	N_6	$\perp$	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

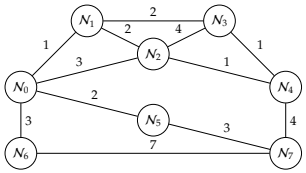
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	$dist =$	0	1	3	3	4	2	3	5	$queue = \langle (N_4, 4), (N_6, 3), (N_7, 5) \rangle$
	$hop =$	$\perp$	N_1	N_2	N_1	N_2	N_5	N_6	N_5	
N_1	$dist =$	1	0	2	2	3	3	4	7	$queue = \langle (N_5, 3), (N_6, 4), (N_7, 7) \rangle$
	$hop =$	N_0	$\perp$	N_2	N_3	N_2	N_0	N_0	N_4	
N_2	$dist =$	3	2	0	2	1	5	6	5	$queue = \langle (N_5, 5), (N_6, 6), (N_7, 5) \rangle$
	$hop =$	N_0	N_1	$\perp$	N_4	N_4	N_0	N_0	N_4	
N_3	$dist =$	3	2	2	0	1	5	6	5	$queue = \langle (N_5, 5), (N_6, 6), (N_7, 5) \rangle$
	$hop =$	N_1	N_1	N_4	$\perp$	N_4	N_0	N_0	N_4	
N_4	$dist =$	4	3	1	1	0	6	7	4	$queue = \langle (N_5, 6), (N_6, 7), (N_7, 4) \rangle$
	$hop =$	N_2	N_2	N_2	N_3	$\perp$	N_0	N_0	N_7	
N_5	$dist =$	2	3	5	5	6	0	5	3	$queue = \langle (N_3, 5), (N_4, 6), (N_6, 5) \rangle$
	$hop =$	N_0	N_0	N_0	N_1	N_2	$\perp$	N_0	N_7	
N_6	$dist =$	3	4	6	6	7	5	0	7	$queue = \langle (N_3, 6), (N_4, 7), (N_7, 7) \rangle$
	$hop =$	N_0	N_0	N_0	N_1	N_2	N_0	$\perp$	N_7	
N_7	$dist =$	5	6	5	5	4	3	7	0	$queue = \langle (N_1, 6), (N_3, 5), (N_6, 7) \rangle$
	$hop =$	N_5	N_0	N_4	N_4	N_4	N_5	N_6	$\perp$	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

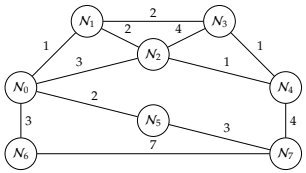
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	*dist* =	0	1	3	3	4	2	3	5	$queue = \langle (N_4, 4), (N_7, 5) \rangle$
	hop =	$\perp$	N_1	N_2	N_1	N_2	N_5	N_6	N_5	
N_1	*dist* =	1	0	2	2	3	3	4	6	$queue = \langle (N_6, 4), (N_7, 6) \rangle$
	hop =	N_0	$\perp$	N_2	N_3	N_2	N_0	N_0	N_5	
N_2	*dist* =	3	2	0	2	1	5	6	5	$queue = \langle (N_6, 6), (N_7, 5) \rangle$
	hop =	N_0	N_1	$\perp$	N_4	N_4	N_0	N_0	N_4	
N_3	*dist* =	3	2	2	0	1	5	6	5	$queue = \langle (N_6, 6), (N_7, 5) \rangle$
	hop =	N_1	N_1	N_4	$\perp$	N_4	N_0	N_0	N_4	
N_4	*dist* =	4	3	1	1	0	6	7	4	$queue = \langle (N_5, 6), (N_6, 7) \rangle$
	hop =	N_2	N_2	N_2	N_3	$\perp$	N_0	N_0	N_7	
N_5	*dist* =	2	3	5	5	6	0	5	3	$queue = \langle (N_4, 6), (N_6, 5) \rangle$
	hop =	N_0	N_0	N_0	N_1	N_2	$\perp$	N_0	N_7	
N_6	*dist* =	3	4	6	6	7	5	0	7	$queue = \langle (N_4, 7), (N_7, 7) \rangle$
	hop =	N_0	N_0	N_0	N_1	N_2	N_0	$\perp$	N_7	
N_7	*dist* =	5	6	5	5	4	3	7	0	$queue = \langle (N_1, 6), (N_6, 7) \rangle$
	hop =	N_5	N_0	N_4	N_4	N_4	N_5	N_6	$\perp$	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

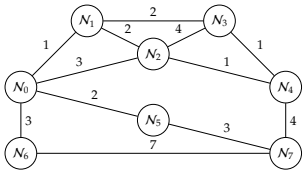
		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	*dist* =	0	1	3	3	4	2	3	5	$queue = \langle (N_7, 5) \rangle$
	hop =	$\perp$	N_1	N_2	N_1	N_2	N_5	N_6	N_5	
N_1	*dist* =	1	0	2	2	3	3	4	6	$queue = \langle (N_7, 6) \rangle$
	hop =	N_0	$\perp$	N_2	N_3	N_2	N_0	N_0	N_5	
N_2	*dist* =	3	2	0	2	1	5	6	5	$queue = \langle (N_6, 6) \rangle$
	hop =	N_0	N_1	$\perp$	N_4	N_4	N_0	N_0	N_4	
N_3	*dist* =	3	2	2	0	1	5	6	5	$queue = \langle (N_6, 6) \rangle$
	hop =	N_1	N_1	N_4	$\perp$	N_4	N_0	N_0	N_4	
N_4	*dist* =	4	3	1	1	0	6	7	4	$queue = \langle (N_6, 7) \rangle$
	hop =	N_2	N_2	N_2	N_3	$\perp$	N_0	N_0	N_7	
N_5	*dist* =	2	3	5	5	6	0	5	3	$queue = \langle (N_4, 6) \rangle$
	hop =	N_0	N_0	N_0	N_1	N_2	$\perp$	N_0	N_7	
N_6	*dist* =	3	4	6	6	7	5	0	7	$queue = \langle (N_7, 7) \rangle$
	hop =	N_0	N_0	N_0	N_1	N_2	N_0	$\perp$	N_7	
N_7	*dist* =	5	6	5	5	4	3	7	0	$queue = \langle (N_6, 7) \rangle$
	hop =	N_5	N_0	N_4	N_4	N_4	N_5	N_6	$\perp$	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Routing Algorithms (2) – Link State Routing

- Recalling that the graph in question is,



the algorithm

1. uses flooding to communicate the topology to all nodes,
2. has each node (in parallel) compute shortest paths using Dijkstra, i.e.,

		N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	
N_0	*dist* =	0	1	3	3	4	2	3	5	*queue* = ∅
	hop =	⊥	N_1	N_2	N_1	N_2	N_5	N_6	N_5	
N_1	*dist* =	1	0	2	2	3	3	4	6	*queue* = ∅
	hop =	N_0	⊥	N_2	N_3	N_2	N_0	N_0	N_5	
N_2	*dist* =	3	2	0	2	1	5	6	5	*queue* = ∅
	hop =	N_0	N_1	⊥	N_4	N_4	N_0	N_0	N_4	
N_3	*dist* =	3	2	2	0	1	5	6	5	*queue* = ∅
	hop =	N_1	N_1	N_4	⊥	N_4	N_0	N_0	N_4	
N_4	*dist* =	4	3	1	1	0	6	7	4	*queue* = ∅
	hop =	N_2	N_2	N_2	N_3	⊥	N_0	N_0	N_7	
N_5	*dist* =	2	3	5	5	6	0	5	3	*queue* = ∅
	hop =	N_0	N_0	N_0	N_1	N_2	⊥	N_0	N_7	
N_6	*dist* =	3	4	6	6	7	5	0	7	*queue* = ∅
	hop =	N_0	N_0	N_0	N_1	N_2	N_0	⊥	N_7	
N_7	*dist* =	5	6	5	5	4	3	7	0	*queue* = ∅
	hop =	N_5	N_0	N_4	N_4	N_4	N_5	N_6	⊥	

Notes:

- Changing the topology, e.g., adding or removing a link or node, demands we re-flood the network so the new topology is available. When dealing with a link, both end-points will notice the addition or removal and initiate a re-flood; when dealing with a node, it may be impossible for the node itself to initiate the re-flood (if it is removed), so this can be managed by each neighbour which notice the addition or removal.

Conclusions

- To summarise the two approaches covered, we can say

Metric	Distance Vector	Link State
correct	distributed Bellman-Ford	replicated Dijkstra
efficient	approximately	approximately
fair	approximately	approximately
convergent	slow: many exchanges	fast: flood then recompute
scalable	excellent	reasonable

i.e., selection is basically a trade-off between convergence and scalablity ...

- ... *plus* we need to cater for various corner cases:
  - distance vector routing can fail if
    - if network is partitioned (cf. graph cut, meaning "count to infinity" problem),

    while
  - link state routing can fail if
    - flooding sequence numbers can overflow or be corrupted,
    - nodes can fail and reset flooding sequence number,
    - if network is partitioned and then re-joined.

Notes:

Conclusions

- Take away points:
  - Using graph theory to study routing is advantageous in terms of designing and reasoning about solutions ...
  - ... even so, translating theory into practice is *still* a significant challenge wrt. function and efficiency.
  - Routing protocols are instances of more general **consensus protocols** whereby (distributed) parties need to agree on a shared state.
- Additional topics: a (non-exhaustive) list could include at least
  - the **Border Gateway Protocol (BGP)** [6], which is important for large(r) scale inter-networking, and
  - techniques such as **multi-path routing** [10] which allows packets to make use of *multiple* routes to a given destination.

Notes:

- It's fairly easy to find other instances of consensus protocols within other contexts; the way Bitcoin deals with (distributed) management of the block chain is a good example.

References

[1] Wikipedia: Distance vector routing protocol.
http://en.wikipedia.org/wiki/Distance-vector_routing_protocol.

[2] Wikipedia: Link-state routing protocol.
http://en.wikipedia.org/wiki/Link-state_routing_protocol.

[3] Wikipedia: Routing.
http://en.wikipedia.org/wiki/Routing.

[4] Wikipedia: Routing protocol.
http://en.wikipedia.org/wiki/Routing_protocol.

[5] C. Hedrick.
Routing Information Protocol.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 1058, 1988.
http://tools.ietf.org/html/rfc1058.

[6] K. Loughheed and Y. Rekhter.
A Border Gateway Protocol (BGP).
Internet Engineering Task Force (IETF) Request for Comments (RFC) 1105, 1989.
http://tools.ietf.org/html/rfc1105.

[7] J. Moy.
OSPF specification.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 1131, 1989.
http://tools.ietf.org/html/rfc1131.

Notes:

[8] D. Oran.
OSI IS-IS Intra-domain Routing Protocol.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 1142, 1990.
http://tools.ietf.org/html/rfc1142.

[9] W. Stallings.
Chapter 13: Routing in switched data networks.
In *Data and Computer Communications*. Pearson, 9th edition, 2010.

[10] D. Thaler and C. Hopps.
Multipath issues in unicast and multicast next-hop selection.
Internet Engineering Task Force (IETF) Request for Comments (RFC) 2991, 2000.
http://tools.ietf.org/html/rfc2991.

Notes: