The mathematics of packet routing

Richard G. Clegg (richard@richardclegg.org)

1 Reminder of the problem

The problem addressed in this lecture is the routing problem, that is the problem of finding out which links in the internet a packet must travel down to get from its origin to its destination. Remember that the internet is highly decentralised. It is not practical for any one router to keep a “map” of the whole internet.

You should always keep in mind that IPv4 (the current internet protocol) has no concept of route. That is to say, a packet of data does not specify its route and routers (unless they are running a special protocol) do not store the complete route from themselves to a destination but only the next “hop” that is on that route (that is only enough information to know where a packet should be sent next).

More information about the topics in this lecture can be found in [BG] (for a more mathematical viewpoint) and [Tan] (for an engineer's look at networks).

1.1 Interesting aside – traceroute, a splendid hack

Some of you may be aware of the traceroute command. Many networks block it these days... if yours does then try http://www.fifi.org/services/traceroute to see it in action. Here is an example from an address in Estonia via an ISP in Rochdale to a home user.

```
traceroute to host213-121-67-224: (213.121.67.224): 2-20 hops, 38 byte packets
  2 213.180.11.162  tondi-CR.online.ee  1.62 ms (ttl=127)
  3 213.180.25.1  liiva-CR.online.ee  1.82 ms (ttl=126)
  4 213.180.11.189 tix-CR.online.ee  2.16 ms (ttl=125)
  5 212.47.215.6  r1-Pa4-0-80-Tln-TIX.EE.KPNQwest.net  2.28 ms (ttl=251)
  6 134.222.224.5  r5-AT3-1.105.sthm-KPN1.SE.kpnquest.net  12.2 ms (ttl=250)
  7 134.222.119.226 r2-Ge0-2-0-0.Sthm-KQ1.SE.KPNQwest.net  34.3 ms (ttl=2461)
  8 134.222.230.157 r2-Se0-3-0.hmng-KQ2.DE.KPNQwest.net  33.4 ms (ttl=2471)
  9 134.222.230.117 r2-Se0-2-0.0.ffm-KQ1.DE.kpnquest.net  34.1 ms (ttl=2491)
 10 134.222.230.29 r2-Se0-3-0.0.1edn-KQ1.NL.kpnqwest.net  39.6 ms (ttl=2481)
 11 134.222.230.169 r1-Se0-0-0.1dn-KQ1.UK.kpnqwest.net  43.7 ms (ttl=2461)
 12 134.222.231.14  r1-Se0-0-0.0.Ldn-KQ4.UK.KPNQwest.net  44.9 ms (ttl=2451)
 13 134.222.109.241  r13-Gi5-0.200.1dn-KQ4.UK.kpnquest.net  45.4 ms (ttl=2451)
 14 195.66.225.10  linx-11.ukcore.bt.net  45.2 ms (ttl=2441)
 15 194.74.65.126  core2-pos14-0.ilford.ukcore.bt.net  45.3 ms (ttl=2431)
 16 194.74.65.222  core2-pos5-0.reading.ukcore.bt.net  46.7 ms (ttl=2421)
 17 62.6.196.109  core2-pos8-0.birmingham.ukcore.bt.net  54.3 ms (ttl=2411)
 18 194.74.16.194  core2-pos9-0.rochdale.ukcore.bt.net  51.0 ms (ttl=2401)
 19 217.32.168.5  vhaccess1-gig1-0.rochdale.fixed.bt.net  51.1 ms (ttl=2391)
 20 213.121.156.22  ugint0066-p.vhaccess1.rochdale.fixed-nte.bt.net  51.3 ms (ttl=2381)
```
Traceroute appears to be retrieving a route – but how does it do this? In fact it isn’t really getting a route. What it is doing is sending a series of packets with low TTL (time to live) to the same destination IP address. When the packets reach their expiry time then the receiving router sends back an ICMP “TTL exceeded” packet. This can be retrieved and the originating IP traced. A new packet is then sent with a TTL one higher. If your network administrator does not block ICMP then you can get an estimation of the route taken to the destination IP.

Question: under what circumstances will this not be a connected route?

2 Basic graph theory

Definition 2.1. A graph $G = (\mathcal{N}, \mathcal{A})$ is a finite set of $\mathcal{N}$ nodes and a set $\mathcal{A}$ of unordered pairs $(i, j)$ where $i, j \in \mathcal{N} : i \neq j$ (known as arcs). If $n_1$ and $n_2$ are nodes and $(n_1, n_2)$ (where $n_1 \neq n_2$) is an arc then this arc is said to be incident on $n_1$ and $n_2$.

Definition 2.2. A directed graph $G = (\mathcal{N}, \mathcal{A})$ is a finite set of $\mathcal{N}$ nodes and a set $\mathcal{A}$ of ordered pairs $(i, j)$ where $i, j \in \mathcal{N} : i \neq j$ (known as arcs).

Definition 2.3. A weighted graph is a graph (either directed or undirected) where every arc $(i, j)$ has a weight $w_{ij}$ associated with it.

These weights can be used in a variety of different ways. For example, they could represent the time taken for a packet to travel along a link, the delay the packet experiences, the level of congestion on the link, the cost a company would pay to use the link or even some combination of all of these.

Definition 2.4. A walk in a graph $G$ is a sequence of nodes in a graph $(n_1, n_2, \ldots, n_l)$ such that each adjacent pair $(n_1, n_2), (n_2, n_3), \ldots, (n_{l-1}, n_l)$ are arcs in $G$. A path is a walk with no repeated nodes. A cycle is a walk with $n_1 = n_l$ where $l > 3$ (for undirected graphs) or $l > 2$ (for directed graphs) and with no other repeated nodes. A connected graph is a graph where for any two nodes $i$ and $j$ we can find a walk which begins at $i$ and ends at $j$.

Assume from here on that we are dealing with weighted, directed connected graphs.

With the definitions we have here we can formulate the routing problem as follows. The shortest route from $i \in \mathcal{N}$ to $j \in \mathcal{N}$ is the walk $n_1, n_2, \ldots, n_l, n_{l+1}$ with $n_1 = i$ and $n_{l+1} = j$ which
minimises the sum \( \sum_{i=1}^{l} w_{n_i,n_{i+1}} \). In other words, the walk from \( i \) to \( j \) with the smallest weight over all arcs. It helps in formulating the problem to make assumption \( w_{ij} = \infty \) if \( (i,j) \notin N \). Note that shortest here need not mean distance but could be any metric chosen depending on what we pick as weights.

3 Dijkstra’s algorithm, a reminder

Dijkstra’s Algorithm was discovered by the pioneering mathematician and programmer E W Dijkstra (1930 – 2002). The algorithm finds the shortest path to all nodes from an origin node, on a graph \( G = (N, A) \). It requires that all arc weights are non-negative – that is for all \( (i,j) \in A : w_{ij} \geq 0 \).

Dijkstra’s Algorithm involves the labelling of a set of permanent nodes \( P \) and node distances \( D_j \) to each node \( j \in N \). Assume that we wish to find the shortest paths from node 1 to all nodes. Then we begin with: \( P = \{1\} \) and \( D_1 = 0 \) and \( D_j = w_{1j} \) where \( j \neq 1 \). Dijkstra’s algorithm then consists of following the procedure:

1. Find the next closest node. Find \( i \notin P \) such that:
   \[
   D_i = \min_{j \notin P} D_j
   \]
2. Update our set of permanently labelled nodes and our nodes distances: \( P := P \cup \{i\} \).
3. If all nodes in \( N \) are also in \( P \) then we have finished so stop here.
4. Update the temporary (distance) labels for the new node \( i \). For all \( j \notin P \)
   \[
   D_j := \min[D_j, w_{ij} + D_i]
   \]
5. Go to the beginning of the algorithm.

Note that the version of this algorithm is subtly different from that in Bertsekas & Gallager which finds the paths from all nodes to a single destination. (This is not an important detail and it should be obvious how to reverse the algorithm). To understand the algorithm we make the following claims:

**Proposition 3.1.** At the beginning of each iteration of Dijkstra’s algorithm then:

1. \( D_i \leq D_j \) for all \( i \in P \) and \( j \notin P \).
2. \( D_j \) is, for all \( j \), the shortest distance from 1 to \( j \) using paths with all nodes (except, possibly \( j \)) in \( P \).

If this proposition can be proved then we can see that, when \( P \) contains every node in \( N \) then all the \( D_j \) are shortest paths by the second part of this proposition. Therefore proving the above proposition is equivalent to proving that Dijkstra’s algorithm finds shortest paths.

The algorithm is proved in the appendices.
4 The Bellman–Ford algorithm

There is an obvious weakness with the Dijkstra algorithm as applied to the internet. It requires
a knowledge of all the nodes links and costs to get the answer. By contrast the Bellman–Ford
algorithm can be run in a distributed manner.

Notation. $D^h_i$ is the distance of the shortest walk from node 1 to node $i$ of $h$ steps or less.

Set initially $D^0_i = \infty$ for $i \neq 1$ and $D^h_1 = 0$ for all $h$.

The Bellman-Ford Algorithm is then simply, for all $i \neq 1$,

$$D^{h+1}_i = \min_j [D^h_j + w_{ji}]$$

The algorithm terminates after $h$ iterations if

$$D^h_i = D^{h-1}_i \text{ for all } i$$

Proposition 4.1. The scalars $D^h_i$ generate by the algorithm from the starting values given for
$D^0_i$ are equal to the shortest walk of length $\leq h$ from node 1 to node $i$.

Proposition 4.2. The algorithm terminates after a finite number of iterations if, and only if, all
cycles not containing node $i$ have non negative length. Furthermore, if the algorithm terminates,
it does so after at most $h \leq N$ iterations and, at termination, $D^h_i$ is the shortest path length
from 1 to $i$.

4.1 Bellman–Ford example

Here is an example network.

The algorithm then gives the answer.
The good thing about the Bellman–Ford algorithm is that it can be run in a distributed manner. Imagine each node starts off knowing only the distance to its neighbours. At each time step the router reports to its neighbours the minimum distance it knows so far to each destination. In this way the algorithm is run without at any point any node having to build up a complete map of the network. The algorithm can also be performed in an asynchronous manner (that is with router all updating at different times rather than simultaneously). This is perfect for a network (like the internet) with no central clock.

**4.2 Problems with routing**

Routing with the Bellman–Ford algorithm is subject to a few problems. One of the best known is the *count to infinity* problem or *bad news phenomenon*.

Consider the network shown below where we want to find the route to node 1 – $L$ is some large number.

<table>
<thead>
<tr>
<th>$h$</th>
<th>$D^b_1$</th>
<th>$D^b_2$</th>
<th>$D^b_3$</th>
<th>$D^b_4$</th>
<th>$D^b_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Imagine the situation where Bellman–Ford has converged. Node 2 can get to node 1 with a cost of 3 (via 3 and 4). Node 3 can get to node 1 with a cost of 2 (via 4) and node 4 can get to node 1 directly.

Then the link 4 → 1 fails. All nodes should route via arc (2,1) but how many iterations of the Bellman–Ford algorithm does this take? Let $D^b_n$ report the distance to node 1 reported at iteration $n$ and the failure occurs at node $n$.
References

[BG] Bersekas and Gallagher, *Data Networks (Second Edition)*

[Tan] Tanenbaum *Computer Networks (Third Edition)*

A Proof of Dijkstra’s algorithm

**Proposition A.1.** At the beginning of each iteration of Dijkstra’s algorithm then:

1. \( D_i \leq D_j \) for all \( i \in P \) and \( j \notin P \).

2. \( D_j \) is, for all \( j \), the shortest distance from 1 to \( j \) using paths with all nodes (except, possibly \( j \)) in \( P \).

If this proposition can be proved then we can see that, when \( P \) contains every node in \( \mathcal{N} \) then all the \( D_j \) are shortest paths by the second part of this proposition. Therefore proving the above proposition is equivalent to proving that Dijkstra’s algorithm finds shortest paths.

**Proof.** The proposition is trivially true at the first step since \( P \) consists only of the origin point (node 1) and \( D_j \) is 0 for \( j = 1 \), is \( w_{ij} \geq 0 \) for nodes reachable directly from node 1 and \( \infty \) otherwise.

The first condition is simply shown to be satisfied since it is preserved by the formula:

\[
D_j := \min[D_j, w_{ij} + D_i]
\]

which is applied to all \( j \notin P \) when node \( i \) is added to the set \( P \).

We show the second condition by induction. We have established already that it is true at the very start of the algorithm. Let us assume it is true for the beginning of some iteration of the algorithm and show that it must then be true at the beginning of the next iteration.

Let node \( i \) be the node we are adding to our set \( P \) and let \( D_k \) be the label of each node \( k \) at the beginning of the step. The second condition must, therefore, hold for node \( j = i \) (the new node we have added) by our induction hypothesis. It must also hold for all nodes \( j \in P \) by part one of the proposition which is already proven. It remains to prove that the second condition of the proposition is met for \( j \notin P \cup \{i\} \).

Consider a path from 1 to \( j \) which is shortest amongst those with all nodes except \( j \) in \( P \cup \{i\} \) and let \( D'_j \) be the corresponding shortest distance. Such a path must contain a path from 1 to
some node \( r \in P \cup \{i\} \) and an arc \((r, j)\). We have already established that the length of the path from 1 to \( r \) must be \( D_r \) and therefore we have:

\[
D'_j = \min_{r \in P \cup \{i\}} [D_r + w_{rj}] = \min \left[ \min_{r \in P} [D_r + w_{rj}], D_i + w_{ij} \right]
\]

However, by our hypothesis \( D_j = \min_{r \in P} [D_r + w_{rj}] \) therefore, \( D'_j = \min [D_j, D_i + w_{ij}] \) which is exactly what is set by the fourth step of the algorithm. Thus, after any iteration of the algorithm, the second part of the proposition is true if it was true at the beginning of the iteration. Thus the proof by induction is complete.

\[
\square
\]


c

## B Proof of the Bellman–Ford algorithm

### Proposition B.1. The scalars \( D^h_i \) generated by the algorithm from the starting values given for \( D_0^i \) are equal to the shortest walk of length \( \leq h \) from node 1 to node \( i \).

**Proof.** The first iteration will clearly give us \( D_1^i = w_{1i} \) for all \( i \) apart from \( i = 1 \) — which is, indeed, correct for the walk lengths of \( \leq 1 \) from 1 to \( i \). Let us suppose that \( D^k_i \) is the shortest walk of length \( \leq k \) (from 1 to \( i \)) then complete the proof by induction by showing that \( D^{k+1}_i \) is the shortest walk of length \( \leq k + 1 \).

One possibility is that the shortest walk of length \( \leq (k + 1) \) will be a walk of length \( k \) or less — in this case, \( D^{k+1}_i = D^k_i \). Otherwise, the walk will be a walk of length \( k + 1 \) with the final arc being \((j, i)\) added on to some walk of length \( k \) to node \( j \). Thus, we can conclude that our shortest walk of length \( k + 1 \) is given by:

\[
\text{Shortest walk to } i \text{ of length } \leq k + 1 = \min \left[ D^k_i, \min_j D^k_j + w_{ji} \right]
\]

However, since a walk of length \( \leq k + 1 \) must always be shorter or equal to a walk of length \( \leq k \) (since the former contains the latter) then this reduces to

\[
\text{Shortest walk to } i \text{ of length } \leq k + 1 = \min_j D^k_j + w_{ji}
\]

which is our original expression for \( D^{k+1}_j \) and completes our proof by induction. \( \square \)

### Proposition B.2. The algorithm terminates after a finite number of iterations if, and only if, all cycles not containing node \( i \) have non negative length. Furthermore, if the algorithm terminates, it does so after at most \( h \leq N \) iterations and, at termination, \( D^h_i \) is the shortest path length from 1 to \( i \).

**Proof.** If negative length cycles exist then such cycles could be repeated as many times as desired to reduce the shortest walk length and thus the algorithm could never converge. Conversely, if no negative length cycles exist then any walks containing cycles could be made shorter or kept the same length by deleting such a cycle. Thus, our shortest walks contain no cycles. The maximum length of a walk with no cycles is \( N - 1 \) (since such a walk will have covered every node). Thus, it trivially follows that \( D^N_i = D^{N-1}_i \) and the algorithm terminates after, at most, \( N \) iterations. \( \square \)