1 The Basics of Distance Vector Routing

Distance vector routing is an algorithm for routing that isolates nodes in terms of their knowledge of the network as a whole.[2] Routing is decentralized and local, meaning exchange of information only directly happens between neighbouring nodes. Full network information is never spread - in update messages to neighbours, a node only includes the distances of its own routing table entries.

Optimal routing can thus take some time to converge, but aside from a few specific issues (to be solved with proper implementation) the information that propagates through the network is fairly limited, simple and efficient. Likewise, local updates are easy to calculate.

For node x to determine the best route to a node y in the network, the Bellman-Ford equation is used, where

$D_x(y) = \min\{c(x, v) + D_v(y)\}$

for all x's neighbours v. Or in words: "The shortest distance from node x to node y is attained by the smallest sum of: the link cost between x and any node v and that node v's shortest distance to node y."

2 Testing During Implementation

Not many unexpected problems arose during the implementation that had to do with interpretation issues of the algorithm. Discussions amongst ourselves and with other lab groups led to the immediate addition of necessary help structures, as well as the very important distinction between the terms link cost, distance and route (distance coupled with a first hop).

- The `int`-array `costs` was renamed to `linkcosts`.
- A 2-dimensional `int`-array was used as a table for received neighbour information: "Best distance to node x of neighbour y".
- A boolean-array, `isNeighbour` was made part of the construction of a router node. Used to skip pointless calculations as well as determining the recipients of the `sendUpdate()` method.
- A `RouteEntry`-array, where the `RouteEntry` class is a pseudo-'struct' of two `ints` (`firsthop` and `distance`).

The `RouteEntry` class caused some very unexpected behaviour that expressed itself in weird ways. Ultimately, each element had to be assigned individually, rendering the choice of implementing the `RouteEntry` class fairly pointless. Lesson to be learned: Any non-primitive datatypes in Java need to not be loosely defined.

3 A Problem with Poisoned Reverse and the Basic Implementation

With the most simple implementation of a distance vector routing algorithm in a network, a few issues arise:

- Redundant routing update packets. Solved with *split horizon*[1] - if A informs B of a routing change, B should not need to inform A of a routing change of B as a direct result of that update.
- The *"count-to-infinity"* problem. Solved with *poisoned reverse* - while A is currently routing through B to C, A will in its updates to B say that A's distance to C is infinite, so that B won't atempt to route to C through A. Trying to route through each other will obviously lead to bouncing data and router update packets (especially the router update packets will propagate and potentially create a lot of unnecessary network exchange). The routing updates essentially "count

to infinity" by slowly incrementing the distance of the route for each update: "B's distance to C is now x? This means my, A's, distance to C is now $x + 1$. Better tell B quick!".

3.1 Where Poisoned Reverse Fails

Routing loops. Solved by *hold down*[1]—whenever a route gets worse, the local router should not change its route immediately, only update the distance (and consequently send out the update). This is to allow the news to propagate through the network before potentially rerouting. With *poisoned reverse* implemented, if we reroute immediately, we can't route to a node i through a node that is routing through us to that same node i. However, we could still "route through ourselves" by routing through a node that is indirectly routing through us to a node i. This creates a routing loop with symptoms similar to *counting to infinity*. See Figure 1.

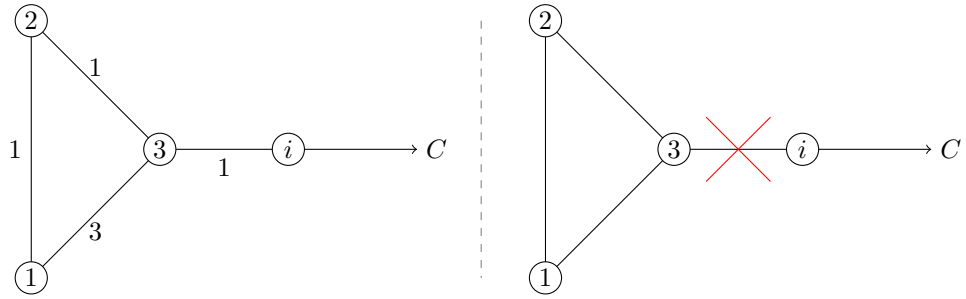

Figure 1: In case of a lost connection between nodes 3 and i, node 1 would still consider node 2 to be a valid route to i. Hold down *lets the news propagate through the system before rerouting, instead of allowing 3 to reroute through 1 to reach i.*

When the route gets worse, we can set a timer for a route which within its duration prohibits rerouting of that route, and/or possibly "schedules" the first allowed rerouting of that route. Letting the news propagate before any rerouting means we won't try and calculate a new best route with information that is outdated and indirectly routes through us, preventing loops. This temporary prohibition of rerouting when a route gets worse needs to be applied on worsened routes and not only on nodes next to the worsened link change—consider the pictures above and assume the actual link change happened somewhere in network C.

References

[1] Charles M. Kozierok. The TCP/IP Guide. http://www.tcpipguide.com/free/t_RIPSpecialFeaturesForResolvingRIPAlgorithmProblems.htm, 2016-03-16.

[2] James F. Kurose and Keith W. Ross. *Computer Networking, A Top-Down Approach.* Pearson Education Limited, 2013.