Theory of Network Communication

Dr. Christian Scheideler

Johns Hopkins University, Sept. 8 2003

1 Introduction

The goal of this lecture is to give an introduction to the state of the art in the theory of network communication. It is a widely accepted fact that algorithmic advances in the area of computer science are only useful to society if they are based on models that truthfully reflect the restrictions and requirements of the corresponding applications. This is certainly also true for network communication. For example, in general any infrastructure connecting processing units with each other may be called a network, but certain infrastructures such as a dedicated line between any pair of nodes are certainly unrealistic, because they are too expensive to build. Thus, messages may have to traverse several units to reach their destination, causing (among other problems) route selection and scheduling problems. Also, we will not just study network communication out of context, but we will also look at various topics that require or support efficient network communication such as distributed data management or design strategies for overlay networks.

In this section we start with an introduction to graph theory followed by simple algorithms for computing shortest paths in networks. It is partly based on [3].

1.1 Graph theory

A *graph* $G = (V, E)$ consists of a set of *nodes* (or *vertices*) V and a set of *edges* (or *arcs*) E. The nodes represent the processing units and the edges represent the communication links between the units. Often, we will set $n = |V|$ (the size of V) and $m = |E|$. The *size* of G is defined as the number of nodes it contains. For all $v, w \in V$, (v, w) denotes a *directed* edge from v to w, and $\{v, w\}$ denotes an *undirected* edge from v to w. G is called *undirected* if $E \subseteq \{\{v, w\} \mid v, w \in V\}$ and *directed* if $E \subseteq \{(v, w) \mid v, w \in V\}$. Unless explicitly mentioned, we assume for the rest of this lecture that G is undirected.

A sequence of contiguous edges in G is called a *path*. The *length* of the path is defined as the number of edges it contains. A path is called *node-simple* if it visits every node in G at most once. Similarly, it is called *edge-simple* (or *simple*) if it contains every edge in G at most once. G is called *connected* if, for any pair of nodes $v, w \in V$, there is a path in G from v to w. We call a simple path a *cycle* if it starts and ends at the same node. The *girth* of a graph G is defined as the length of the shortest cycle G contains. G is called a *tree* if it is connected and contains no cycle. A graph $T = (V', E')$ is called a *spanning tree* of G if $V' = V$, $E' \subseteq E$, and T is a tree. G is called *bipartite* if its node set can be partitioned into two node sets V_1 and V_2 such that $E \subseteq \{\{v, w\} \mid v \in V_1,\ w \in V_2\}$.

For any pair of nodes $v, w \in V$, let $\delta(v, w)$ denote the *distance* of v and w in G, that is, the length of a shortest path from v to w. The *diameter* D of G is defined as $\max\{\delta(v, w) \mid v, w \in V\}$. If $\{v, w\} \in E$ then v is called a *neighbor* of w. For any subset $U \subseteq V$, the *neighborhood* of U is defined as

$$\Gamma(U) = \{v \in V \setminus U \mid \exists u \in U : \{u, v\} \in E\} .$$

The number of neighbors of v is called the *degree* of v and denoted by d_v. The degree of G is defined as $d = \max\{d_v \mid v \in V\}$. If all nodes in G have the same degree, then G is called *regular*.

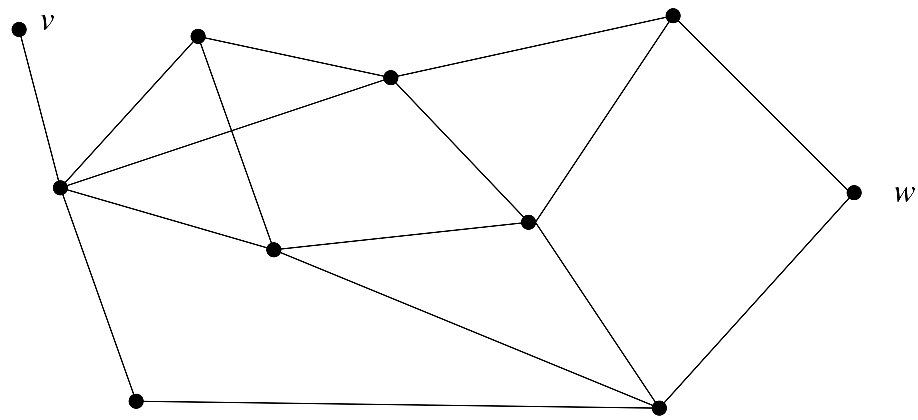


Figure 1: An example of an undirected graph with diameter 4.

A family of graphs $\mathcal{G} = \{G_n \mid n \in \mathbb{N}\}$ has degree $d(n)$ if for all $n \in \mathbb{N}$ the degree of G_n is $d(n)$. If it is clear to which family a graph belongs, we say that this graph has constant (or bounded) degree if and only if its family has constant degree.

A *network* is specified by a graph $G = (V, E)$ with edge capacities given by a function $c : E \to \mathbb{R}^+$. Given a graph G with capacities c, let the capacity of a node $v \in V$ be defined as

$$c(v) = \sum_{w \in V} c(v, w)$$

and the capacity of any node set or edge set U be defined as $c(U) = \sum_{u \in U} c(u)$.

2 Network flows I – Shortest Paths

In order to send information from a sending host to the destination host, the network must determine a *path* or *route* that the flow of information has to follow. This is the job of the *routing protocol*. At the heart of any routing protocol is an algorithm (the *routing algorithm*) that determines a "good" path from the source to the destination. In practice (for example, the Internet), a "good" path is one that has "least cost". Suppose that we have a network $G = (V, E)$ with edge costs $c : E \to \mathbb{R}_+$. For a source-destination pair $(s, t) \in V^2$, we want to find a path $p = (e_1, \ldots, e_\ell)$ from s to t in G that minimizes

$$c(p) = \sum_{i=1}^{\ell} c(e) .$$

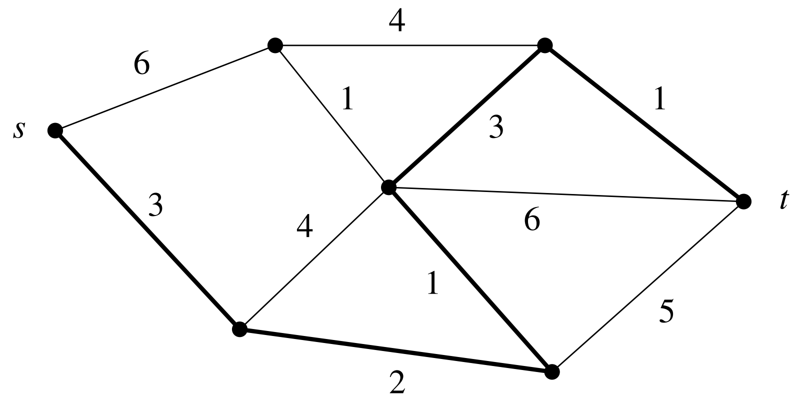


Figure 2: The least-cost path from s to t has a cost of 10.

If all edges have a cost of 1, this is simply a shortest path from s to t.

One way in which we can classify routing algorithms is according to whether they are global or local:

- A *global routing algorithm* computes the least-cost path between a source and destination using complete, global knowledge of the network. That is, the algorithm knows all link costs in the network. In practice, algorithms with global state information are often referred to as *link state algorithms*. We will study a global link state algorithm in Section 2.1.

- In a *local routing algorithm*, the calculation of the least-cost path is carried out in an iterative, distributed manner. No node has complete information about the costs of all network links. Instead, each node begins with only the knowledge of the costs of its own directly attached links. Then, through an iterative process of calculation and exchange of information with its neighboring nodes, a node gradually calculates the least-cost path to a destination or set of destinations. In practice, algorithms with local state information are also known as *distance vector* algorithms. We will study a distance vector algorithm in Section 2.2.

A second way of classifying routing algorithms is according to whether they are *static* or *dynamic*. In static routing algorithms, routes change very slowly over time (if at all) whereas dynamic routing algorithms change their routing paths as the network traffic loads or topology change. A dynamic algorithm can be run either periodically or in direct response to topology of link cost changes. While dynamic algorithms are more responsive to network changes, they are also more susceptible to problems such as routing loops and oscillation in routes.

Only two types of routing algorithms are typically used in the Internet: a dynamic global link state algorithm, and a dynamic local distance vector algorithm. We cover these algorithms in the next two sections.

2.1 A link state routing algorithm

Recall that in a link state algorithm the network topology and all link costs are known. In practice, this is accomplished by having each node broadcast the identities and costs of its attached links to all other nodes in the network. (This may either be done periodically or only if a cost changes.) The result of the nodes' link state broadcast is that all nodes have an identical and complete view of the network. Each node can then run the link state algorithm and compute the same set of least-cost paths as every other node.

The link state algorithm we present below is known as *Dijkstra's algorithm*, named after its inventor. A closely related algorithm is Prim's algorithm. See [2] for a general discussion of graph algorithms. Dijkstra's algorithm computes the least-cost paths from the node executing it to all other nodes in the network. Dijkstra's algorithm is iterative and has the property that after the kth iteration of the algorithm, the least-cost paths are known to k destination nodes, and among the least-cost paths to all destination nodes, these k paths have the k smallest costs.

In the following, s denotes the node executing Dijkstra's algorithm. For each node $u \in V$, let $D[u]$ denote the length of the best path we have found *so far* from s to u. Initially, $D[s] = 0$ and $D[u] = +\infty$ for each $u \neq s$. We define the set C (our "cloud" of visited vertices) to be initially empty. At each iteration of the algorithm, we select a node $u \neq C$ with smallest $D[u]$ and pull u into C. (I.e. in the very first iteration, we will pull s into C.) Once a new node u is pulled into C, we then update the label $D[v]$ for each node v that is adjacent to u and is outside of C, to reflect the fact that there may be a new and better way to get to v via u. In formal terms, we perform the operation

$$\textbf{if } D[u] + c(u,v) < D[v] \textbf{ then}$$
$$D[v] = D[u] + c(u,v)$$

We give the pseudo-code for Dijkstra's algorithm in Figure 3. We use a priority queue Q to store the nodes outside of the cloud C.

Algorithm Dijkstra(G, c):
  Input: A simple undirected graph G with nonnegative edge costs c and
    a distinguished node s
  Output: A label $D[u]$ for each node u in G such that $D[u]$ is the length of a
    least-cost path from s to u in G.
  $D[s] = 0$ and $D[u] = +\infty$ for each node $u \neq s$
  Let Q contain all nodes in G.
  while $Q \neq \emptyset$ **do**
    $u = Q$.RemoveMin() {*pulls node u from Q with minimum $D[u]$*}
    for each node $v \in Q$ adjacent to u **do**
      if $D[u] + c(u,v) < D[v]$ **then**
        $D[v] = D[u] + c(u,v)$
  return the label $D[u]$ for each node u

Figure 3: Dijkstra's algorithm for the single-source shortest path problem.

Theorem 2.1 *In Dijkstra's algorithm, whenever a node u is pulled into C, the label $D[u]$ is equal to $\delta(s,u)$, the length of the shortest path from s to u.*

Proof. We start with a simple lemma.

Lemma 2.2 *At any point during Dijkstra's algorithm, $D[u]$ is either ∞ or represents the cost of a path from s to u.*

Proof. The lemma can be shown by complete induction on the number of executions of $D[v] = D[u] + c(u, v)$. Certainly, the lemma is true before the first execution of this command. So suppose that it is true till some ith execution. Then it is also true after the $(i + 1)$th execution because from the induction hypothesis, $D[u]$ represents the cost of a path from s to u, and therefore $D[v] = D[u]+c(u, v)$ represents the cost of a path from s to v. □

Hence, it remains that at the end, $D[v]$ is the cost of a shortest path from s to v for all v.

Suppose that $D[v] > \delta(s, v)$ for some node $v \in V$, and let u be the *first* node the algorithm pulled into the cloud C such that $D[u] > \delta(s, u)$. There is a shortest path P from s to u (for otherwise $\delta(s, u) = +\infty = D[u]$). Let us therefore consider the moment when u is pulled into C, and let v be the first node of P (when going from s to u) that is not in C at this moment. (v may be equal to u.) Let w be the predecessor of v in P. (Note that we may have $w = s$.) We know, by our choice of v, that w is already in C at this point. Moreover, $D[w] = \delta(s, w)$, since u is the *first* incorrect node. When w is pulled into C, we tested (and possibly updated) $D[v]$ so that we had at that point

$$D[v] \leq D[w] + c(w, v) = \delta(s, w) + c(w, v) \ .$$

Since v is the next node on the shortest path from s to u and all edge costs are positive,

$$D[v] \leq \delta(s, u) \ .$$

Furthermore, since u is added to C before v (or with v if $v = u$), it must hold that

$$D[u] \leq D[v]$$

But that would mean that $D[u] \leq \delta(s, u)$, which is a contradiction to our assumption that $D[u] > \delta(s, u)$. □

Theorem 2.3 *The runtime of Dijkstra's algorithm is $O((n + m) \log n)$.*

Proof. The details of the runtime analysis are as follows:

- Inserting all the nodes in Q can be done in $O(n \log n)$ time by repeated insertions.
- At each iteration of the while loop, we spend $O(\log n)$ time to remove node u from Q and $O(d(u) \cdot \log n)$ time to update $D[v]$ for the nodes v adjacent to u. Hence, the overall running time of the while loop is

$$\sum_{u \in V} (1 + d(u)) \log n = O((n + m) \log n) \ .$$

□

Using a so-called Fibonacci Heap, the runtime can be improved to $O(m + n \log n)$.

2.2 A distance vector algorithm

While the link state algorithm is an algorithm using global information, the distance (DV) vector algorithm is iterative, asynchronous, and distributed. It is distributed in that each node receives some information from one or more of its directly attached neighbors, performs a calculation, and may then distribute the results of its calculations back to its neighbors. It is iterative in that this process continues on until no more information is exchanged between neighbors. (Interestingly, we will see that the algorithm is self terminating – there is no "signal" that the computation should stop; it just stops.) The algorithm is asynchronous in that it does not require all of the nodes to operate in lock step with each other.

The principle data structure in the DV algorithm is the *distance table* maintained at each node. Each node's distance table has a row for each destination in the network and a column for each of its directly attached neighbors. Consider a node u that is interested in routing to destination w via its directly attached neighbor v. Node u's distance table entry, $D_{u,v}(w)$, is the sum of the cost of the link (u, v) plus neighbor v's currently known minimum-cost path from v to w. That is,

$$D_{u,v}(w) = c(u, v) + \min_x D_{v,x}(w)$$

where the $\min_x$ expression is taken over all of v's direct neighbors (including u).

The distance vector algorithm we will study is also known as the Bellman-Ford algorithm, after its inventors. It is used in many routing protocols in practice, including Internet BGP, ISO IDRP, Novell IPX, and RIP. The pseudo-code for the Bellman-Ford algorithm can be found in Figure 4.

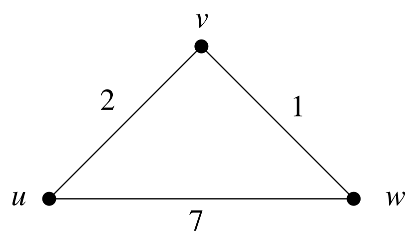
Algorithm Bellman-Ford:

At each node u:

    for all adjacent nodes v **do**
        $D_{u,v}(w) = +\infty$ for each $w \neq v$
        $D_{u,v}(v) = c(u, v)$
    for all destinations w **do**
        send $\min_v D_{u,v}(w)$ to each neighbor
    while true **do** $\{$*infinite loop*$\}$
        wait (until there is a link cost change to neighbor v or
                until there is an update from neighbor v)
        if $c(u, v)$ changes by γ
            for all destinations w **do** $D_{u,v}(w) = D_{u,v}(w) + \gamma$
        else if update received from v w.r.t. destination w
            $\{$*The shortest path from v to some w has changed and*$\}$
            $\{$*therefore v has sent a new value for* $\min_x D_{v,x}(w)$.$\}$
            $\{$*Call this value δ.*$\}$
            $D_{u,v}(w) = c(u, v) + \delta$
        if we have a new $\min_v D_{u,v}(w)$ for any destination w
            send the new value of $\min_v D_{u,v}(w)$ to all neighbors

Figure 4: The Bellman-Ford algorithm for the all pairs shortest path problem.

Figure 5 illustrates the operation of the DV algorithm for the simple three node network shown at the top of the figure. The operation of the algorithm is illustrated in a synchronous manner, where all nodes simultaneously receive messages from their neighbors, compute new distance table entries, and inform the neighbors of any changes in their new least-cost paths. However, after viewing this example, it should not be too difficult to see that the algorithm also works correctly in the completely asynchronous case.



1:

u (to \ via)	v	w
v	2	∞
w	∞	7

v (to \ via)	u	w
u	2	∞
w	∞	1

w (to \ via)	u	v
u	7	∞
v	∞	1

2:

u (to \ via)	v	w
v	2	8
w	3	7

v (to \ via)	u	w
u	2	8
w	9	1

w (to \ via)	u	v
u	7	3
v	9	1

3:

u (to \ via)	v	w
v	2	8
w	3	7

v (to \ via)	u	w
u	2	4
w	5	1

w (to \ via)	u	v
u	7	3
v	9	1

Figure 5: A DV algorithm example. Circled values represent values sent around by the nodes.

Theorem 2.4 *If all nodes work in a synchronous way, then the Bellman-Ford algorithm terminates after at most* $n-1$ *rounds, and when it terminates, then* $\min_v D_{u,v}(w) = \delta(u,w)$ *for all nodes* u *and* w.

Proof. For any two nodes u and w, let $\delta_\ell(u,w)$ denote the minimum cost of a path of length at most ℓ from u to w. Obviously, it holds for all $\ell \geq 0$ that

$$\delta_{\ell+1}(u,w) = \min_v \{c(u,v) + \delta_\ell(v,w)\}$$

for all u and w, where

$$\delta_0(u,w) = \begin{cases} 0 & u = w \\ +\infty & \text{else} \end{cases}$$

We want to prove by induction on the number of iterations of the while loop that at after the ℓth while loop,

$$\min_v D_{u,v}(w) = \min_v \{c(u,v) + \delta_\ell(v,w)\} \ .$$

For $\ell = 0$ (i.e. before executing the while loop) this is certainly true. So suppose that for $\ell \geq 0$ the hypothesis above has already been shown. Then it holds that after iteration $\ell + 1$,

$$\begin{aligned} \min_v D_{u,v}(w) &= \min_v \{c(u,v) + \min_x D_{v,x}(w)\} \\ &= \min_v \{c((u,v)) + \min_x \{c(v,x) + \delta_\ell(x,w)\}\} \\ &= \min_v \{c((u,v)) + \delta_{\ell+1}(v,w)\} \ . \end{aligned}$$

Hence, after $n-1$ iterations,

$$\min_v D_{u,v}(w) = \min_v \{c(u,v) + \delta_{n-1}(v,w)\} = \delta_n(u,w) \ .$$

Since we only have edges of nonnegative cost, any shortest path in a network can have a length of at most n. Thus, after $n-1$ iterations, we must have that

$$\min_v D_{u,v}(w) = \delta(u,w) \ .$$

□

The proof of this theorem implies that it takes at most n iterations for a network of size n to terminate with a minimum-cost path system. However, for this to hold it is crucial that the induction hypothesis is true for $\ell = 0$, i.e. $D_{u,v}(w) = +\infty$ for all adjacent nodes v with $v \neq w$ and $D_{u,v}(v) = c(u,v)$. What happens if the DV algorithms starts in a different state? In this case, it may take much more than n iterations to terminate. In fact, in the example in Figure 6, it would take 44 (!) iterations to terminate. Methods that can circumvent this problem have, for example, been proposed in [1].

2.3 Minimum spanning trees

So far, we only considered the situation of minimizing the cost for sending a message from a source to a destination. This communication mode is also known as *unicasting*. Another important communication primitive is *broadcasting*, i.e. sending a message from a source to *all* nodes of the network. This is usually done by sending the message along a spanning tree. At each point of the spanning tree where it branches off into different directions, the message is replicated so that one message can be sent in each direction. Hence, every edge of the spanning tree is only used once by the message. In order to minimize the cost of sending a broadcast message, we have to find a spanning tree of minimum cost. In formal terms, given a weighted undirected graph G, we are interested in finding a tree T that contains all the nodes in G and minimizes the sum

$$c(T) = \sum_{(v,w) \in T} c(v,w) \ .$$

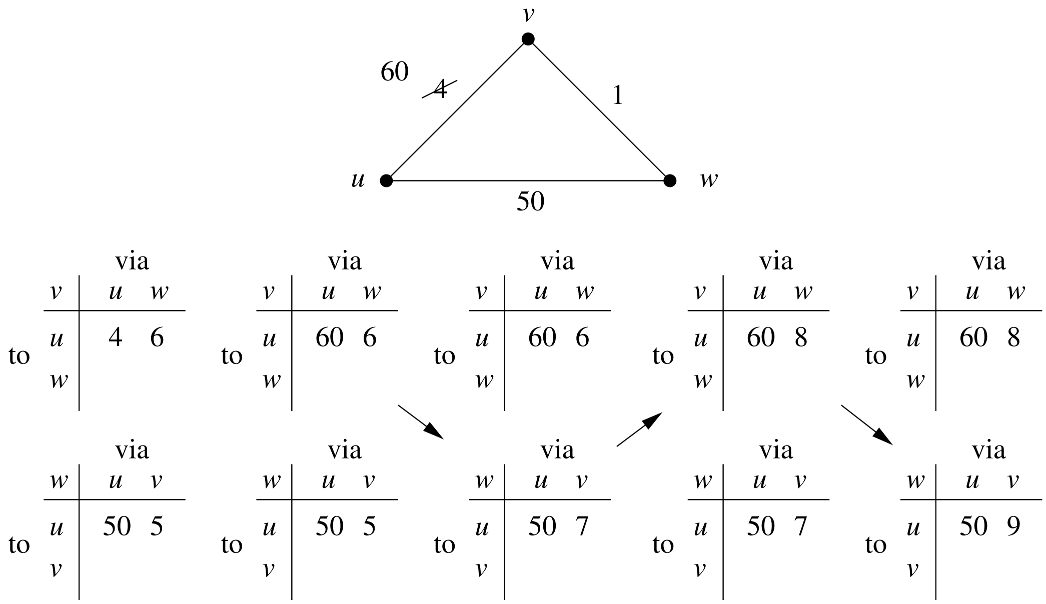


Figure 6: A bad example for the DV algorithm.

This problem is also known as the *minimum spanning tree* (or MST) problem.

Before we present an algorithm for the MST problem, we start with a crucial fact about minimum spanning trees.

Proposition 2.5 *Let G be a weighted connected graph and let V_1 and V_2 be a partition of the nodes of G into two disjoint nonempty sets. Furthermore, let e be an edge in G with minimum weight from among those with one endpoint in V_1 and the other in V_2. Then there is a minimum spanning tree T that has e as one of its edges.*

Proof. Let T be a minimum spanning tree of G. If T does not contain edge e, the addition of e to T must create a cycle. Therefore, there is some edge f of this cycle that has one endpoint in V_1 and the other in V_2. Moreover, by the choice of e, $c(e) \leq c(f)$. If we remove f from $T \cup \{e\}$, we obtain a spanning tree whose total cost is no more than before. Since T was a minimum spanning tree, this new tree must also be a minimum spanning tree. $\square$

In fact, if the costs in G are distinct, then the minimum spanning tree is unique. The proposition immediately yields two different ways of constructing a minimum spanning tree:

- Initially, each node is its own cluster. Then we consider the edges in the order of increasing cost. If an edge e connects two different clusters, then e is added to the minimum spanning tree, and the two clusters connected by e are merged into a single cluster. Otherwise, e is discarded. Once the algorithm has added enough edges to form a spanning tree, it terminates and outputs this tree as the minimum spanning tree. This algorithm is also known as Kruskal's MST algorithm.

- Initially, we only have a single cluster C consisting of a root node v. In each iteration, we choose the edge e of minimum cost that has one endpoint inside of C and the other outside of C. We add e to the minimum spanning tree and the outside node to C. Once C contains all the nodes of G, the algorithm terminates. This algorithm is also known as Prim's algorithm.

Now, recall our LS and DV algorithms above.

Using the LS algorithm

Since in a LS algorithm every node knows all the edge costs in the network, each node can simply run Kruskal's or Prim's algorithm in order to compute the MST.

Using the DV algorithm

Using the DV algorithm is a bit more difficult. Instead of computing minimum-cost paths between pairs of nodes, our aim is to compute paths with the minimum possible maximum-cost edge between pairs of nodes, i.e. for each pair (s, t) we want to find a path $p = (e_1, \ldots, e_\ell)$ from s to t so that $\gamma(p) = \max_i c(e_i)$ is minimized over all paths from s to t. This can simply be done by replacing the rule

$$D_{u,v}(w) = c(u, v) + \min_x D_{v,x}(w)$$

in the Bellman-Ford algorithm by the rule

$$D_{u,v}(w) = \max\{c(u, v), \min_x D_{v,x}(w)\} \ ,$$

i.e. $D_{u,v}(w)$ now represents the minimum possible maximum edge capacity along a path from u to w via v. In this case, we have after the ℓth iteration of the while loop that

$$\min_v D_{u,v}(w) = \min_v\{\max\{c(u, v), \gamma_\ell(v, w)\}\}$$

where $\gamma_\ell(v, w)$ is defined as the minimum $\gamma(p)$ over all paths p of length at most ℓ from v to w. Let $\gamma(v, w) = \gamma_n(v, w)$.

Furthermore, we have to decide now which edges participate in the minimum spanning tree, and which edges do not. For this we use the following rules:

- Suppose that $\{u, w\}$ still belongs to the spanning tree and u finds out that $\min_v D_{u,v}(w) < c(u, w)$. In this case, u removes $\{u, w\}$ from the spanning tree.

- Suppose that $\{u, w\}$ has been taken out of the spanning tree and u finds out that $\min_v D_{u,v}(w) \geq c(u, w)$. In this case, u adds $\{u, w\}$ to the spanning tree.

See Figure 7 for the algorithm that computes $\gamma(v, w)$ for all nodes v, w. If all edges have different costs (which is easy to guarantee with tricks like adding to the cost of each edge as least significant bits the addresses of its incident nodes), then this algorithm together with the rules above computes a minimum spanning tree.

Algorithm Modified-Bellman-Ford:

At each node u:

for all adjacent nodes v **do**
  $D_{u,v}(w) = +\infty$ for each $w \neq v$
  $D_{u,v}(v) = c(u,v)$
for all destinations w **do**
  send $\min_v D_{u,v}(w)$ to each neighbor
while true **do** $\{$*infinite loop*$\}$
  wait (until there is a link cost change to neighbor v or
    until there is an update from neighbor v)
  if $c(u,v)$ changes by γ
    $c(u,v) = c(u,v) + \gamma$
    for all destinations w **do** $D_{u,v}(w) = \max\{c(u,v), D_{u,v}(w)\}$
  else if update received from v w.r.t. destination w
    $\{$*v has sent a new value for* $\min_x D_{v,x}(w)$.$\}$
    $\{$*Call this value* γ.$\}$
    $D_{u,v}(w) = \max\{c(u,v), \gamma\}$
  if we have a new $\min_v D_{u,v}(w)$ for any destination w
    send the new value of $\min_v D_{u,v}(w)$ to all neighbors

Figure 7: The Bellman-Ford algorithm for the minimum spanning tree problem.

References

[1] B. Awerbuch, A. Bar-Noy, and M. Gopal. Approximate distributed Bellman-Ford algorithms. *IEEE Transactions on Communications*, 42(8):2515–2517, 1994.

[2] T. Corman, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 2001.

[3] J. Kurose and K. Ross. *Computer Networking – A Top-Down Approach Featuring the Internet*. Addison Wesley, 2001.