4

Routing Algorithms

A process (a node in a computer network) is in general not connected directly to every other process by a channel. A node can send packets of information directly only to a subset of the nodes called the *neighbors* of the node. Routing is the term used to describe the decision procedure by which a node selects one (or, sometimes, more) of its neighbors to forward a packet on its way to an ultimate destination. The objective in designing a routing algorithm is to generate (for each node) a decision-making procedure to perform this function and guarantee delivery of each packet.

It will be clear that some information about the topology of the network must be stored in each node as a working basis for the (local) decision procedure; we shall refer to this information as the *routing tables.* With the introduction of these tables the routing problem can be algorithmically divided into two parts; the definition of the table structure is of course related to the algorithmical design.

(1) *Table computation.* The routing tables must be computed when the network is initialized and must be brought up to date if the topology of the network changes.
(2) *Packet forwarding.* When a packet is to be sent through the network it must be forwarded using the routing tables.

Criteria for "good" routing methods include the following.

(1) *Correctness.* The algorithm must deliver every packet offered to the network to its ultimate destination.
(2) *Efficiency.* The algorithm must send packets through "good" paths, e.g., paths that suffer only a small delay and ensure high throughput of the entire network. An algorithm is called *optimal* if it uses the "best" paths.

(3) *Complexity.* The algorithm for the computation of the tables must use as few messages, time, and storage as possible. Other aspects of complexity are how fast a routing decision can be made, how fast a packet can be made ready for transmission, etc., but these aspects will receive less attention in this chapter.
(4) *Robustness.* In the case of a topological change (the addition or removal of a channel or node) the algorithm updates the routing tables in order to perform the routing function in the modified network.
(5) *Adaptiveness.* The algorithm balances the load of channels and nodes by adapting the tables in order to avoid paths through channels or nodes that are very busy, preferring channels and nodes with a currently light load.
(6) *Fairness.* The algorithm must provide service to every user in the same degree.

These criteria are sometimes conflicting, and most algorithms perform well only w.r.t. a subset of them.

As usual, a network is represented as a graph, where the nodes of the graph are the nodes of the network, and there is an edge between two nodes if they are neighbors (i.e., they have a communication channel between them). The optimality of an algorithm depends on what is called a "best" path in the graph; there are several notions of what is "best", each with its own class of routing algorithms:

(1) *Minimum hop.* The cost of using a path is measured as the number of hops (traversed channels or steps from node to node) of the path. A minimum-hop routing algorithm uses a path with the smallest possible number of hops.
(2) *Shortest path.* Each channel is statically assigned a (non-negative) *weight*, and the cost of a path is measured as the sum of the weights of the channels in the path. A shortest-path algorithm uses a path with lowest possible cost.
(3) *Minimum delay.* Each channel is dynamically assigned a weight, depending on the traffic on the channel. A minimum-delay algorithm repeatedly revises the tables in such a way that paths with a (near) minimal total delay are always chosen. As the delays encountered on the channels depend on the actual traffic, the various packets transmitted through the network influence each other; the impact on the required routing algorithm will be discussed at the end of Section 4.1.

Other notions of the optimality of paths may be useful in special applications, but will not be discussed here.

Chapter overview. The following material is treated in this chapter. In Section 4.1 it will be shown that, at least for minimum-hop and shortest-path routing, it is possible to route all packets for the same destination d optimally via a spanning tree rooted towards d. As a consequence, the source of a packet can be ignored when routing decisions are made.

Section 4.2 describes an algorithm to compute routing tables for a static network with weighted channels. The algorithm distributively computes a shortest path between each pair of nodes and stores in each source node the first neighbor on the path for each destination. A disadvantage of this algorithm is that the entire computation must be repeated after a topological change in the network: the algorithm is not robust.

The Netchange algorithm discussed in Section 4.3 does not suffer from this disadvantage: it can adapt to failing or recovering channels by a *partial* recomputation of the routing tables. To keep the analysis simple it is presented as a minimum-hop routing algorithm, that is, the number of hops is taken as the cost of a path. It is possible to modify the Netchange algorithm to deal with weighted channels that can fail or recover.

The routing algorithms of Sections 4.2 and 4.3 use routing tables (in each node) that contain an entry for each possible destination. This may be too heavy a storage demand for large networks of small nodes. In Section 4.4 some routing strategies will be discussed that code topological information in the address of a node, in order to use shorter routing tables or fewer table lookups. These so-called "compact" routing algorithms usually do not use optimal paths. A tree-based scheme, interval routing, and prefix routing are discussed.

Section 4.5 discusses hierarchical routing methods. In these methods, the network is partitioned into (connected) clusters, and a distinction is made between routing within a cluster and routing to another cluster. This paradigm can be used to reduce the number of routing decisions that must be made during a path, or to reduce the amount of space needed to store the routing table in each node.

4.1 Destination-based Routing

The routing decision made when forwarding a packet is usually based only on the *destination* of the packet (and the contents of the routing tables), and is *independent* of the original sender (the source) of the packet. Routing can

ignore the source and still use optimal paths, as implied by the results of this section. The results do not depend on the choice of a particular optimality criterion for paths, but the following assumptions must hold. (Recall that a path is *simple* if it contains each node at most once, and the path is a cycle if the first node equals the last node.)

(1) The cost of sending a packet via a path P is independent of the actual utilization of the path, in particular, the use of edges of P by other messages. This assumption allows us to regard the cost of using path P as a function of the path; thus denote the cost of P by $C(P) \in \mathbb{R}$.
(2) The cost of the concatenation of two paths equals the sum of the costs of the concatenated paths, i.e., for all $i = 0, \ldots, k$,

$$C(\langle u_0, u_1, \ldots, u_k \rangle) = C(\langle u_0, \ldots, u_i \rangle) + C(\langle u_i, \ldots, u_k \rangle).$$

Consequently, the cost of the empty path $\langle u_0 \rangle$ (this is a path from u_0 to u_0) satisfies $C(\langle u_0 \rangle) = 0$.
(3) The graph does not contain a cycle of negative cost.

(These criteria are satisfied by minimum-hop and shortest-path cost criteria.) A path from u to v is called *optimal* if there exists no path from u to v with lower cost. Observe that an optimal path is not always unique; there may exist different paths with the same (minimal) cost.

Lemma 4.1 *Let u, v be in V. If a path from u to v exists in G, then there exists a simple path that is optimal.*

Proof. As there are only a finite number of simple paths, there exists a simple path from u to v, say S_0, of lowest cost, i.e., for every *simple* path P' from u to v $C(S_0) \leq C(P')$. It remains to show that $C(S_0)$ is a lower bound for the cost of every (non-simple) path.

Write $V = \{v_1, \ldots, v_N\}$. By successively eliminating from P cycles that include v_1, v_2, etc., it will be shown that for each path P from u to v there exists a simple path P' with $C(P') \leq C(P)$. Let $P_0 = P$, and construct for $i = 1, \ldots, N$ the path P_i as follows. If v_i occurs at most once in P_{i-1} then $P_i = P_{i-1}$. Otherwise, write $P_{i-1} = \langle u_0, \ldots, u_k \rangle$, let u_{j_1} be the first and u_{j_2} be the last occurrence of v_i in P_{i-1}, and let

$$P_i = \langle u_0, \ldots, u_{j_1} (= u_{j_2}), u_{j_2+1}, \ldots, u_k \rangle.$$

By construction P_i is a path from u to v and contains all nodes of $\{v_1, \ldots, v_i\}$ at most once, hence P_N is a simple path from u to v. P_{i-1} consists of P_i and the cycle $Q = u_{j_1}, \ldots, u_{j_2}$, hence $C(P_{i-1}) = C(P_i) + C(Q)$. As there

are no cycles of negative weight, this implies $C(P_i) \leq C(P_{i-1})$, and hence $C(P_N) \leq C(P)$.

By the choice of S_0, $C(S_0) \leq C(P_N)$, from which $C(S_0) \leq C(P)$ follows. □

If G contains cycles of negative weight an optimal path does not necessarily exist; each path can be beaten by another path that runs through the negative cycle once more. For the next theorem, assume that G is connected (for disconnected graphs the theorem can be applied to each connected component separately).

Theorem 4.2 *For each $d \in V$ there exists a tree $T_d = (V, E_d)$ such that $E_d \subseteq E$ and such that for each node $v \in V$, the path from v to d in T_d is an optimal path from v to d in G.*

Proof. Let $V = \{v_1, \ldots, v_N\}$. We shall inductively construct a series of trees $T_i = (V_i, E_i)$ (for $i = 0, \ldots, N$) with the following properties.

(1) Each T_i is a subtree of G, i.e., $V_i \subseteq V$, $E_i \subseteq E$, and T_i is a tree.
(2) Each T_i (for $i < N$) is a subtree of T_{i+1}.
(3) For all $i > 0$, $v_i \in V_i$ and $d \in V_i$.
(4) For all $w \in V_i$, the simple path from w to d in T_i is an optimal path from w to d in G.

These properties imply that T_N satisfies the requirements for T_d.

To construct the sequence of trees, set $V_0 = \{d\}$ and $E_0 = \varnothing$. The tree T_{i+1} is constructed as follows. Choose an optimal simple path $P = \langle u_0, \ldots, u_k \rangle$ from v_{i+1} to d, and let l be the smallest index such that $u_l \in T_i$ (such an l exists because $u_k = d \in T_i$; possibly $l = 0$). Now set

$$V_{i+1} = V_i \cup \{u_j : j < l\} \quad \text{and} \quad E_{i+1} = E_i \cup \{(u_j, u_{j+1}) : j < l\}.$$

(The construction is pictorially represented in Figure 4.1.) It is easy to verify that T_i is a subtree of T_{i+1} and that $v_{i+1} \in V_{i+1}$. To see that T_{i+1} is a tree, observe that by construction T_{i+1} is connected, and the number of nodes exceeds the number of edges by one. (T_0 has the latter property, and in each stage as many nodes as edges are added.)

It remains to show that for all $w \in V_{i+1}$, the (unique) path from w to d in T_{i+1} is an optimal path from w to d in G. For the nodes $w \in V_i \subset V_{i+1}$ this follows because T_i is a subtree of T_{i+1}; the path from w to d in T_{i+1} is the same as the path in T_i, which is optimal. Now let $w = u_j$, $j < l$ be a node in $V_{i+1} \setminus V_i$. Write Q for the path from u_l to d in T_i, then in T_{i+1} u_j is connected to d by the path $\langle u_j, \ldots, u_l \rangle$ concatenated with Q, and it remains

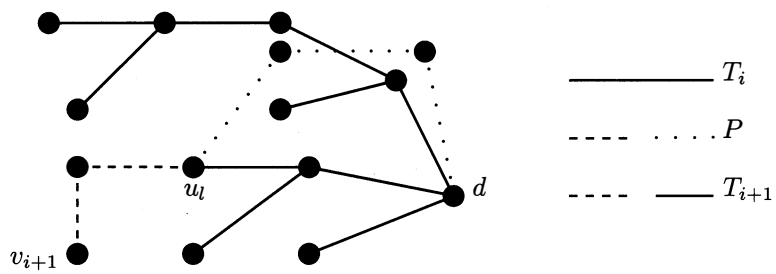


Figure 4.1 THE CONSTRUCTION OF T_{i+1}.

to show that this path is optimal in G. First, the suffix $P' = \langle u_l, \ldots, u_k \rangle$ of P is an optimal path from u_l to d, i.e., $C(P') = C(Q)$: the optimality of Q implies $C(P') \geq C(Q)$, and $C(Q) < C(P')$ implies (by the additivity of path costs) that the path $\langle u_0, \ldots, u_l \rangle$ concatenated with Q has lower cost than P, contradicting the optimality of P. Now assume that a path R from u_j to d has lower cost than the path $\langle u_j, \ldots, u_l \rangle$ concatenated with Q. Then, by the previous observation, R has a lower cost than the suffix $\langle u_j, \ldots, u_k \rangle$ of P, and this implies (again by the additivity of path costs) that the path $\langle u_0, \ldots, u_j \rangle$ concatenated with R has lower cost than P, contradicting the optimality of P. □

A spanning tree rooted towards d is called a *sink tree* for d, and a tree with the property given in Theorem 4.2 is called an *optimal sink tree.* The existence of optimal sink trees implies that it is no compromise to optimality if only routing algorithms are considered for which the forwarding mechanism is as in Algorithm 4.2. In that algorithm, $table\_lookup_u$ is a local procedure with one argument, returning a neighbor of u (after consulting the routing tables). Indeed, as all packets for destination d can be routed optimally over a spanning tree rooted at d, forwarding is optimal if, for all $u \neq d$, $table\_lookup_u(d)$ returns the father of u in the spanning tree T_d.

When the forwarding mechanism is of this form and no (further) topological changes occur, the correctness of routing tables can be certified using the following result. The routing tables are said to *contain a cycle* (for destination d) if there are nodes $u_1, \ldots, u_k$ such that for all i, $u_i \neq d$, for all $i < k$, $table\_lookup_{u_i}(d) = u_{i+1}$, and $table\_lookup_{u_k}(d) = u_1$. The tables are said to be *cycle-free* if they do not contain a cycle for any d.

(* A packet with destination d was received or generated at node u *)
if $d = u$
  then deliver the packet locally
  else send the packet to $table\_lookup_u(d)$

Algorithm 4.2 DESTINATION-BASED FORWARDING (FOR NODE u).

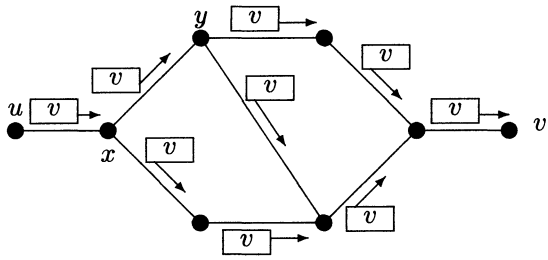
Lemma 4.3 *The forwarding mechanism delivers every packet at its destination if and only if the routing tables are cycle-free.*

Proof. If the tables contain a cycle for some destination d a packet for d is never delivered if its source is a node in the cycle.

Assume the tables are cycle-free and let a packet with destination d (and source u_0) be forwarded via $u_0, u_1, u_2, \ldots$. If the same node occurs twice in this sequence, say $u_i = u_j$, then the tables contain a cycle, namely $\langle u_i, \ldots, u_j \rangle$, contradicting the assumption that the tables are cycle-free. Thus, each node occurs at most once, which implies that this sequence is finite, ending, say, in node u_k $(k < N)$. According to the forwarding procedure the sequence can only end in d, i.e., $u_k = d$ and the packet has reached its destination in at most $N - 1$ hops. □

In some routing algorithms it is the case that the tables are not cycle-free during their computation, but only when the table computation phase has finished. When such an algorithm is used, a packet may traverse a cycle during computation of the tables, but reaches its destination in at most $N - 1$ hops after completion of the table computation if topological changes cease. If topological changes do not cease, i.e., the network is subject to an infinite sequence of topological changes, packets do not necessarily reach their destination even if tables are cycle-free during updates; see Exercise 4.1.

Bifurcated routing for minimum delay. If routing via minimum-delay paths is required, and the delay of a channel depends on its utilization (thus assumption (1) at the beginning of this section is not valid), the cost of using a path cannot simply be assessed as a function of this path alone. In addition, the traffic on the channel must be taken into account. To avoid congestion (and the resulting higher delay) on a path, it is usually necessary to send packets having the same source–destination pair via different paths; the traffic for this pair "splits" at one or more intermediate nodes as depicted in Figure 4.3. Routing methods that use different paths towards the same destination are called *multiple-path* or *bifurcated* routing methods.



Traffic for node v splits at nodes x and y.

Figure 4.3 EXAMPLE OF BIFURCATED ROUTING.

Because bifurcated routing methods are usually very intricate, they will not be treated in this chapter.

4.2 The All-pairs Shortest-path Problem

This section discusses an algorithm by Toueg [Tou80a] for computing simultaneously the routing tables for all nodes in a network. The algorithm computes for each pair (u, v) of nodes the length of a shortest path from u to v and stores the first channel of such a path in u. The problem of computing a shortest path between any two nodes of a graph is known as the all-pairs shortest-path problem. Toueg's distributed algorithm for this problem is based on the centralized Floyd–Warshall algorithm [CLR90, Section 26.4]. We shall discuss the Floyd–Warshall algorithm in Subsection 4.2.1, and subsequently Toueg's algorithm in Subsection 4.2.2. A brief discussion of some other algorithms for the all-pairs shortest-path problem follows in Subsection 4.2.3.

4.2.1 The Floyd–Warshall Algorithm

Let a weighted graph $G = (V, E)$ be given, where the weight of edge uv is given by ω_{uv}. It is not necessary to assume that $\omega_{uv} = \omega_{vu}$, but it will be assumed that the graph contains no cycles of total negative weight. The weight of a path $\langle u_0, \ldots, u_k \rangle$ is defined as $\sum_{i=0}^{k-1} \omega_{u_i u_{i+1}}$. The *distance* from u to v, denoted $d(u, v)$, is the lowest weight of any path from u to v (∞ if no such path exists). The all-pairs shortest-path problem is to compute $d(u, v)$ for each u and v. (In Section 4.2.2 the algorithm will be augmented to store the first edge of such a path as well.)

To compute all distances, the Floyd–Warshall algorithm uses the notion of

S-paths; these are paths in which all *intermediate* nodes belong to a subset S of V.

Definition 4.4 *Let S be a subset of V. A path $\langle u_0, \ldots, u_k \rangle$ is an S-path if for all i, $0 < i < k$, $u_i \in S$. The S-distance from u to v, denoted $d^S(u, v)$, is the lowest weight of any S-path from u to v (∞ if no such path exists).*

The algorithm starts by considering all $\varnothing$-paths, and incrementally computes S-paths for larger subsets S, until all V-paths have been considered. The following observations can be made.

Proposition 4.5 *For all u and S, $d^S(u, u) = 0$. Further, S-paths satisfy the following rules for $u \neq v$.*

(1) *There exists an $\varnothing$-path from u to v if and only if $uv \in E$.*
(2) *If $uv \in E$ then $d^{\varnothing}(u, v) = \omega_{uv}$, otherwise $d^{\varnothing}(u, v) = \infty$.*
(3) *If $S' = S \cup \{w\}$ then a simple S'-path from u to v is an S-path from u to v or an S-path from u to w concatenated with an S-path from w to v.*
(4) *If $S' = S \cup \{w\}$ then $d^{S'}(u, v) = \min(\, d^S(u, v),\, d^S(u, w) + d^S(w, v)\,)$.*
(5) *A path from u to v exists if and only if a V-path from u to v exists.*
(6) *$d(u, v) = d^V(u, v)$,*

Proof. For all u and S, $d^S(u, u) \leq 0$ because the empty path (consisting of zero edges) is an S-path from u to u of weight 0. No path has a smaller weight, because G contains no cycles of negative weight, so $d^S(u, u) = 0$.

For (1): an $\varnothing$-path has no intermediate nodes, so an $\varnothing$-path from u to v consists only of the channel uv.

For (2): this follows immediately from (1).

For (3): a simple S'-path from u to v contains node w either once, or zero times as an intermediate node. If it does not contain w as an intermediate node it is an S-path, otherwise it is the concatenation of two S-paths, one to w and one from w.

For (4): it can be shown by an application of Lemma 4.1 that (if an S'-path from u to v exists) there is a *simple* S' path of length $d^{S'}(u, v)$ from u to v, which implies $d^{S'}(u, v) = \min(\, d^S(u, v),\, d^S(u, w) + d^S(w, v)\,)$ by (3).

For (5): each V-path is a path, and vice versa.

For (6): each V-path is a path, and vice versa, hence an optimal V-path is also an optimal path.

□

```
begin (* Initialize S to ∅ and D to ∅-distance *)
      S := ∅ ;
      forall u, v do
            if u = v then D[u, v] := 0
            else if uv ∈ E then D[u, v] := ω_uv
                           else D[u, v] := ∞ ;
      (* Expand S by pivoting *)
      while S ≠ V do
         (* Loop invariant: ∀u, v : D[u, v] = d^S(u, v) *)
         begin pick w from V \ S ;
               (* Execute a global w-pivot *)
               forall u ∈ V do
                     (* Execute a local w-pivot at u *)
                     forall v ∈ V do
                           D[u, v] := min ( D[u, v], D[u, w] + D[w, v] ) ;
               S := S ∪ {w}
         end   (* ∀u, v : D[u, v] = d(u, v) *)
end
```

Algorithm 4.4 THE FLOYD–WARSHALL ALGORITHM.

Using Proposition 4.5 it is not difficult to design a "dynamic programming" algorithm to solve the all-pairs shortest-path problem; see Algorithm 4.4. The algorithm first considers $\varnothing$-paths, and incrementally computes S-paths for larger sets S (enlarging S by means of "pivot" rounds), until all paths have been considered.

Theorem 4.6 *Algorithm 4.4 computes the distance between each pair of nodes in $\Theta(N^3)$ steps.*

Proof. The algorithm starts with $D[u, v] = 0$ if $u = v$, $D[u, v] = \omega_{uv}$ if $uv \in E$ and $D[u, v] = \infty$ otherwise, and $S = \varnothing$. Hence by Proposition 4.5, parts (1) and (2), $\forall u, v : D[u, v] = d^S(u, v)$ holds. In a pivot round with pivot-node w the set S is expanded with w, and the assignment to $D[u, v]$ ensures (by parts (3) and (4) of the proposition) that the assertion $\forall u, v : D[u, v] = d^S(u, v)$ is preserved as a loop invariant. The program terminates when $S = V$, i.e., (by parts (5) and (6) of the proposition and the loop invariant) the S-distances equal the distances.

The main loop is executed N times, and contains N^2 operations (which can be executed in parallel or serially), which implies the time bound stated in the theorem. □

```
var S_u   : set of nodes ;
    D_u   : array of weights ;
    Nb_u  : array of nodes ;

begin S_u := ∅ ;
      forall v ∈ V do
            if v = u
              then begin D_u[v] := 0 ; Nb_u[v] := udef end
              else if v ∈ Neigh_u
                      then begin D_u[v] := ω_uv ; Nb_u[v] := v end
                      else begin D_u[v] := ∞ ; Nb_u[v] := udef end ;
      while S_u ≠ V do
         begin pick w from V \ S_u ;
               (* All nodes must pick the same node w here *)
               if u = w
                 then "broadcast the table D_w"
                 else "receive the table D_w" ;
               forall v ∈ V do
                      if D_u[w] + D_w[v] < D_u[v] then
                        begin D_u[v] := D_u[w] + D_w[v] ;
                              Nb_u[v] := Nb_u[w]
                        end ;
               S_u := S_u ∪ {w}
         end
end
```

Algorithm 4.5 THE SIMPLE ALGORITHM (FOR NODE u).

4.2.2 Toueg's Shortest-path Algorithm

A distributed algorithm for computing routing tables was given by Toueg [Tou80a], based on the Floyd–Warshall algorithm described in the previous subsection. It must be verified that the Floyd–Warshall algorithm is suitable for this purpose, i.e., that its assumptions are realistic in distributed systems. The most important assumption of the algorithm is that the graph does not contain cycles of negative weight. This assumption is indeed realistic for distributed systems, where it is usually the case that each individual channel is assigned a positive cost. An even stronger assumption will be made; see A1 below.

In this subsection the following assumptions are made.

A1. Each cycle in the network has a positive weight.
A2. Each node in the network initially knows the identities of all nodes (the set V).

A3. Each node knows which of the nodes are its neighbors (stored in $Neigh_u$ for node u) and the weights of its outgoing channels.

The correctness of Toueg's algorithm (Algorithm 4.6) will be more easily understood if we first consider a preliminary version of it, the "simple algorithm" (Algorithm 4.5).

The simple algorithm. To arrive at a distributed algorithm the variables and operations of the Floyd–Warshall algorithm are partitioned over the nodes of the network. The variable $D[u, v]$ is a variable belonging to node u; by convention, this will be expressed by subscripting: write $D_u[v]$ from now on. An operation that assigns a value to $D_u[v]$ must be executed by node u, and when the value of a variable of node w is needed in this operation, this value must be sent to u. In the Floyd–Warshall algorithm all nodes must use information from the pivot node (w in the loop body), which sends this information to all nodes simultaneously by a "broadcast" operation. Finally, the algorithm will be augmented with an operation to maintain not only the *lengths* of shortest S-paths (as in the variable $D_u[v]$), but also the first channel of such a path (in the variable $Nb_u[v]$).

The assumption that a cycle in the network has positive weight can be used to show that no cycles occur in the routing tables after each pivot round.

Lemma 4.7 *Let S and w be given and suppose that*

(1) *for all u $D_u[w] = d^S(u, w)$ and*
(2) *if $d^S(u, w) < \infty$ and $u \neq w$, then $Nb_u[w]$ is the first channel of a shortest S-path to w.*

Then the directed graph $T_w = (V_w, E_w)$, where

$$(u \in V_w \iff D_u[w] < \infty) \quad \text{and} \quad (ux \in E_w \iff (u \neq w \wedge Nb_u[w] = x))$$

is a tree rooted towards w.

Proof. First observe that if $D_u[w] < \infty$ for $u \neq w$, then $Nb_u[w] \neq udef$ and $D_{Nb_u[w]}[w] < \infty$. So for each node $u \in V_w$, $u \neq w$ there is a node x for which $Nb_u[w] = x$, and this node satisfies $x \in V_w$.

For each node $u \neq w$ in V_w there is one edge in E_w, so the number of nodes of T_w exceeds the number of edges by one and it suffices to show that T_w contains no cycle. As $ux \in E_w$ implies that $d^S(u, w) = \omega_{ux} + d^S(x, w)$, the existence of a cycle $\langle u_0, u_1, \ldots, u_k \rangle$ in T_w implies that

$$d^S(u_0, w) = \omega_{u_0u_1} + \omega_{u_1u_2} + \cdots + \omega u_{k-1}u_0 + d^S(u_0, w),$$

i.e.,

$$0 = \omega_{u_0u_1} + \omega_{u_1u_2} + \cdots + \omega u_{k-1}u_0,$$

which contradicts the assumption that each cycle has positive weight. □

The Floyd–Warshall algorithm can now be transformed in a straightforward way to obtain Algorithm 4.5. Each node initializes its own variables and executes N iterations of the main loop. The algorithm is not the ultimate solution, and it is not given completely because we have not specified how the broadcast of the table of the pivot node can be done (efficiently). For now it suffices to take for granted that because the operation "broadcast the table D_w" is executed by w and the operation "receive the table D_w" is executed by the other nodes, each node has access to the table D_w.

Some care must be given to the operation "pick w from $V \setminus S$", in order to guarantee that the nodes select the pivots in the same order. As it is assumed that all nodes know V in advance, we can simply assume that the nodes are selected in some prescribed order (e.g., the alphabetical order of the node names).

The correctness of the simple algorithm is expressed in the following theorem.

Theorem 4.8 *Algorithm 4.5 terminates in each node after N iterations of the main loop. When the algorithm terminates in node u, $D_u[v] = d(u, v)$, and if a path from u to v exists then $Nb_u[v]$ is the first channel of a shortest path from u to v, otherwise $Nb_u[v] = udef$.*

Proof. The termination and the correctness of $D_u[v]$ on termination follow from the correctness of the Floyd–Warshall algorithm (Theorem 4.6). The statement about the value of $Nb_u[v]$ follows because $Nb_u[v]$ is updated each time $D_u[v]$ is assigned. □

The improved algorithm. In order to do the broadcast in Algorithm 4.5 efficiently, Toueg observes that a node u for which $D_u[w] = \infty$ at the start of the w-pivot round does not change its tables during the w-pivot round. If $D_u[w] = \infty$, $D_u[w] + D_w[v] < D_u[v]$ is false for every v. Consequently, only the nodes that belong to T_w (at the beginning of the w-pivot round) need to receive w's table, and the broadcast operation can be done efficiently by sending the table D_w only via the channels that belong to the tree T_w. That is, w sends D_w to its sons in T_w, and each node of T_w that receives the table (from its father in T_w) forwards it to its sons in T_w.

At the beginning of the w-pivot round a node u with $D_u[w] < \infty$ knows

```
var S_u   : set of nodes ;
    D_u   : array of weights ;
    Nb_u  : array of nodes ;

begin S_u := ∅ ;
      forall v ∈ V do
            if v = u
              then begin D_u[v] := 0 ; Nb_u[v] := udef end
              else if v ∈ Neigh_u
                     then begin D_u[v] := ω_uv ; Nb_u[v] := v end
                     else begin D_u[v] := ∞ ; Nb_u[v] := udef end ;
      while S_u ≠ V do
         begin pick w from V \ S_u ;
               (* Construct the tree T_w *)
               forall x ∈ Neigh_u do
                     if Nb_u[w] = x then send ⟨ ys, w ⟩ to x
                                    else send ⟨ nys, w ⟩ to x ;
               num_rec_u := 0 ; (* u must receive |Neigh_u| messages *)
               while num_rec_u < |Neigh_u| do
                     begin receive ⟨ ys, w ⟩ or ⟨ nys, w ⟩ message ;
                           num_rec_u := num_rec_u + 1
                     end ;
               if D_u[w] < ∞ then (* participate in pivot round *)
                 begin if u ≠ w
                         then receive ⟨ dtab, w, D ⟩ fromthis Nb_u[w] ;
                       forall x ∈ Neigh_u do
                             if ⟨ ys, w ⟩ was received from x
                               then send ⟨ dtab, w, D ⟩ to x ;
                       forall v ∈ V do (* local w-pivot *)
                             if D_u[w] + D[v] < D_u[v] then
                               begin D_u[v] := D_u[w] + D[v] ;
                                     Nb_u[v] := Nb_u[w]
                               end
                 end ;
               S_u := S_u ∪ {w}
         end
end
```

Algorithm 4.6 TOUEG'S ALGORITHM (FOR NODE u).

who its father (in T_w) is, but not who its sons are. Therefore each node v must send a message to each of its neighbors u, telling u whether v is a son of u in T_w. The full algorithm is now given as Algorithm 4.6. A node can participate in the forwarding of w's table when it knows which of its neighbors are its sons in T_w. The algorithm uses three types of messages:

(1) A $\langle$ **ys**, w $\rangle$ message (**ys** stands for "your son") is sent by u to x at the beginning of the w-pivot round if x is the father of u in T_w.
(2) A $\langle$ **nys**, w $\rangle$ message (**nys** stands for "not your son") is sent by u to x at the beginning of the w-pivot round if x is not the father of u in T_w.
(3) A $\langle$ **dtab**, w, D $\rangle$ message is sent during the w-pivot round via each edge of T_w to transmit the value of D_w to each node that must use this value.

Assuming that a weight (of an edge or path) together with a node name can be represented by W bits, the complexity of the algorithm is expressed in the following theorem.

Theorem 4.9 *Algorithm 4.6 computes for each u and v the distance from u to v, and, if this distance is finite, the first channel of a path of this length. The algorithm exchanges $O(N)$ messages per channel, $O(N \cdot |E|)$ messages in total, $O(N^2W)$ bits per channel, $O(N^3W)$ bits in total, and requires $O(NW)$ bits of storage per node.*

Proof. Algorithm 4.6 is derived from Algorithm 4.5, which implies its correctness.

Each channel carries two $\langle$ **ys**, w $\rangle$ or $\langle$ **nys**, w $\rangle$ messages (one in each direction) and at most one $\langle$ **dtab**, w, D $\rangle$ message in the w-pivot round, which totals to at most $3N$ messages per channel. A $\langle$ **ys**, w $\rangle$ or $\langle$ **nys**, w $\rangle$ message contains $O(W)$ bits and a $\langle$ **dtab**, w, D $\rangle$ message contains $O(NW)$ bits, which gives the bound on the number of bits per channel. At most N^2 $\langle$ **dtab**, w, D $\rangle$ messages and $2N \cdot |E|$ $\langle$ **ys**, w $\rangle$ and $\langle$ **nys**, w $\rangle$ messages are exchanged, which totals to $O(N^2 \cdot NW + 2N \cdot |E| \cdot W) = O(N^3W)$ bits altogether. The D_u and Nb_u tables maintained in node u require $O(NW)$ bits. □

During the w-pivot round a node is allowed to receive and process only the messages of that round, i.e., those that carry the parameter w. If the channels satisfy the fifo property then the $\langle$ **ys**, w $\rangle$ and $\langle$ **nys**, w $\rangle$ messages arrive as the first messages after a node has started that round, one via each channel, and then the $\langle$ **dtab**, w, D $\rangle$ message is the next to arrive from $Nb_u[w]$ (if the node is in V_w). It is possible by careful programming to omit the parameter w from all messages if the channels are fifo. If the channels are not fifo it is possible that a message with parameter w' arrives while a node expects messages for round w, where w' is the pivot after w. In this case the parameter is used to distinguish the messages for each pivot round,

and local buffering (either in the channel or in the node) must be used to defer processing of the w'-message.

Toueg gives a further optimization of the algorithm, relying on the following result. (Node u_2 is a *descendant* of u_1 if u_2 belongs to the subtree of u_1.)

Lemma 4.10 *Let $u_1 \neq w$, and let u_2 be a descendant of u_1 in T_w at the beginning of the w-pivot round. If u_2 changes its distance to v in the w-pivot round, then u_1 changes its distance to v in the w-pivot round.*

Proof. As u_2 is a descendant of u_1 in T_w,

$$d^S(u_2, w) = d^S(u_2, u_1) + d^S(u_1, w). \tag{1}$$

Because $u_1 \in S$,

$$d^S(u_2, v) \leq d^S(u_2, u_1) + d^S(u_1, v). \tag{2}$$

Node u_2 changes $D_{u_2}[v]$ in this round iff

$$d^S(u_2, w) + d^S(w, v) < d^S(u_2, v). \tag{3}$$

By applying (2), and then (1), and subtracting $d^S(u_2, u_1)$, we obtain

$$d^S(u_1, w) + d^S(w, v) < d^S(u_1, v), \tag{4}$$

implying that u_1 changes $D_{u_1}[v]$ in this round. □

According to this lemma, Algorithm 4.6 can be modified as follows. After the receipt of the table D_w (message $\langle \mathbf{dtab}, w, D \rangle$) node u first executes the local w-pivot operation, and then forwards the table to its sons in T_w. When forwarding the table it suffices to send those entries $D[v]$ for which $D_u[v]$ has changed as a result of the local w-pivot operation. With this modification the routing tables are cycle-free not only between pivot rounds (as expressed in Lemma 4.7), but also during pivot rounds.

4.2.3 Discussion and More Algorithms

The presentation of Toueg's algorithm provides an example of how a distributed algorithm can be obtained in a straightforward manner from a sequential algorithm. To this end, the variables of the sequential algorithm are dispersed over the processes, and any assignment to variable x (in the sequential algorithm) is executed by the process holding x. Whenever the assigned expression contains references to variables held by other processes, communication between processes is required in order to pass the value of

this variable and to synchronize the processes. Specific properties of the sequential algorithm can be exploited to minimize the required amount of communication.

Toueg's algorithm is reasonably simple to understand, has low complexity, and routes via optimal paths; its main disadvantage is its bad robustness. When the topology of the network changes the entire computation must be performed anew. In addition, the algorithm has two properties that make it less attractive from the viewpoint of distributed algorithms engineering.

First, as already mentioned, the uniform selection by all nodes of the next pivot node (w) requires that the set of participating nodes is precisely known in advance. As this knowledge is in general not available a priori, the execution of an additional distributed algorithm to compute this set (e.g., Finn's algorithm, Algorithm 6.9) must precede execution of Toueg's algorithm.

Second, Toueg's algorithm is based on repeated application of the *triangle inequality* $d(u, v) \leq d(u, w) + d(w, v)$. Evaluating the right-hand side (by u) requires information about $d(w, v)$, and this information is in general *remote*, i.e., available neither in u nor in any of its neighbors. Dependence on remote data necessitates the transport of information to remote nodes, which can be observed in Toueg's algorithm (the broadcasting part).

Alternatively, the following defining equation for $d(u, v)$ can be used in algorithms for shortest-path problems:

$$d(u, v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{w \in Neigh_u} \omega_{uw} + d(w, v) & \text{otherwise} \end{cases} \tag{4.1}$$

Two properties of this equation make algorithms based upon it different from Toueg's algorithm.

(1) *Data locality.* In order to evaluate the right-hand side of Equation (4.1), node u only needs information available locally (namely, ω_{uw}) or at a neighbor (namely, $d(w, v)$). The transportation of data between remote nodes is avoided.

(2) *Destination independence.* Only distances *to* v (namely, $d(w, v)$ for neighbors w of u) are needed to compute the distance from u to v. Thus, the computation of all distances to a fixed destination v_0 can proceed independently of the computation of distances to other nodes, and also, can be studied in isolation.

In the remainder of this section two algorithms based on Equation (4.1) are discussed, namely, the Merlin–Segall and Chandy–Misra algorithms. Despite the advantage offered by data locality, the communication complexity

of these algorithms is no improvement over Toueg's algorithm. This is due to the destination independence introduced by Equation (4.1); apparently, using results for other destinations (as is done in Toueg's algorithm) is a more profitable technique than introducing data locality.

If it does not lead to a reduced communication complexity, then what is the importance of data locality? Reliance on remote data requires the latter's repeated broadcast if data can change due to topological changes in the network (channel and node failures and repairs). Achieving these broadcasts (under the possibility of new topological changes during the broadcast) turns out to be a non-trivial problem with expensive solutions (see, e.g., [Gaf87]). Therefore, algorithms based upon Equation 4.1 can be more easily adapted to handle topological changes. This is exemplified in Section 4.3, where such an algorithm is discussed in depth.

The Merlin–Segall algorithm. The algorithm proposed by Merlin and Segall [MS79] computes the routing tables for each destination completely separately; the computations for different destinations do not influence each other. For a destination v, the algorithm starts with a tree T_v rooted towards v, and repeatedly updates this tree so as to become an optimal sink tree for destination v.

For destination v, each node u maintains an estimate for the distance to v ($D_u[v]$) and the neighbor to which packets for u are forwarded ($Nb_u[v]$), which is also the father of u in T_v. In an update round each node u sends its estimated distance, $D_u[v]$, to all neighbors *except* $Nb_u[v]$ (in a $\langle$ **mydist**, $v, D_u[v] \rangle$ message). If node u receives from neighbor w a message $\langle$ **mydist**, $v, d \rangle$ and if $d + \omega_{uw} < D_u[v]$, u will change $Nb_u[v]$ to w and $D_u[v]$ to $d + \omega_{uw}$. The update round is controlled by v and requires the exchange of two messages of W bits on each channel.

It is shown in [MS79] that after i update rounds all shortest paths of at most i hops have been correctly computed, so that after at most N rounds all shortest paths to v are computed. Shortest paths to each destination are computed by executing the algorithm independently for each destination.

Theorem 4.11 *The algorithm of Merlin and Segall computes shortest-path routing tables by exchanging* $O(N^2)$ *messages per channel,* $O(N^2 \cdot W)$ *bits per channel,* $O(N^2 \cdot |E|)$ *messages in total, and* $O(N^2 \cdot |E|W)$ *bits in total.*

The algorithm can also adapt to changes in the topology and the weight of channels. An important property of the algorithm is that during update rounds also the routing tables are cycle-free.

```
var D_u[v_0]  : weight   init ∞ ;
    Nb_u[v_0] : node     init udef ;

For node v_0 only:
    begin D_{v_0}[v_0] := 0 ;
          forall w ∈ Neigh_{v_0} do send ⟨ mydist, v_0, 0 ⟩ to w
    end

Processing a ⟨ mydist, v_0, d ⟩ message from neighbor w by u:
    { ⟨ mydist, v_0, d ⟩ ∈ M_wu }
    begin receive ⟨ mydist, v_0, d ⟩ from w ;
          if d + ω_uw < D_u[v_0] then
            begin D_u[v_0] := d + ω_uw ; Nb_u[v_0] := w ;
                  forall x ∈ Neigh_u do send ⟨ mydist, v_0, D_u[v_0] ⟩ to x
            end
    end
```

Algorithm 4.7 THE CHANDY–MISRA ALGORITHM (FOR NODE u).

The Chandy–Misra algorithm. The algorithm proposed by Chandy and Misra [CM82] computes all shortest paths towards one destination using the paradigm of *diffusing computations*, that is, a distributed computation that is initiated by a single node, and joined by other nodes only after receiving a message.

To compute, for all nodes, the distance to node v_0 (and a preferred outgoing channel), each node u starts with $D_u[v_0] = \infty$ and waits for the receipt of messages. Node v_0 sends a $\langle$ **mydist**, $v_0, 0 \rangle$ message to all neighbors. Whenever node u receives a $\langle$ **mydist**, $v_0, d \rangle$ message from neighbor w, where $d + \omega_{uw} < D_u[v_0]$, u assigns $d + \omega_{uw}$ to $D_u[v_0]$ and sends a $\langle$ **mydist**, $v_0, D_u[v_0] \rangle$ message to all neighbors; see Algorithm 4.7.

It is not difficult to show that $D_u[v_0]$ is always an upper bound for $d(u, v_0)$, i.e., $d(u, v_0) \leq D_u[v_0]$ is implied by an invariant of the algorithm; see Exercise 4.3. To demonstrate that the algorithm computes the distances correctly, it must be shown that eventually a configuration is reached in which $D_u[v_0] \leq d(u, v_0)$ also holds for each u. We supply a proof of this property that uses an assumed weak fairness assumption, namely, that each message that is sent is eventually received in each computation. It is also possible to give a proof that does not rely on this assumption, but it is rather complicated.

Theorem 4.12 *In each computation of Algorithm 4.7 a configuration is reached in which, for each node u, $D_u[v_0] \leq d(u, v_0)$.*

Proof. Fix an optimal sink tree T for v_0 and number the nodes other than v_0 by v_1 through v_{N-1} in such a way that if v_i is the father of v_j, then $i < j$. Let C be a computation; it will be shown by induction on j that for each $j \leq N-1$ a configuration is reached in which, for each $i \leq j$, $D_{v_i}[v_0] \leq d(v_i, v_0)$. Observe that $D_{v_i}[v_0]$ never increases in the algorithm; so if $D_{v_i}[v_0] \leq d(v_i, v_0)$ holds in some configuration, it holds in all subsequent configurations as well.

The case $j = 0$: $d(v_0, v_0) = 0$, and $D_{v_0}[v_0] = 0$ after the execution of the initialization part by v_0, , so $D_{v_0}[v_0] \leq d(v_0, v_0)$ holds after this execution.

The case $j + 1$: Assume a configuration is reached in which for each $i \leq j$, $D_{v_i}[v_0] \leq d(v_i, v_0)$, and consider node v_{j+1}. There is a shortest path $v_{j+1}, v_i, \ldots, v_0$ of length $d(v_{j+1}, v_0)$ from v_{j+1} to v_0, where v_i is the father of v_{j+1} in T, hence $i \leq j$. Consequently, by the induction hypothesis, a configuration is reached in which $D_{v_i}[v_0] \leq d(v_i, v_0)$. Whenever $D_{v_i}[v_0]$ decreases, v_i sends $\langle\, \mathbf{mydist}, v_0, D_{v_i}[v_0] \,\rangle$ messages to its neighbors, hence a $\langle\, \mathbf{mydist}, v_0, d \,\rangle$ message is sent to v_{j+1} at least once with $d \leq d(v_i, v_0)$.

By assumption, this message is received in C by v_{j+1}. The algorithm implies that after receipt of this message $D_{v_{j+1}}[v_0] \leq d + \omega v_{j+1} v_i$ holds, and the choice of i implies that $d + \omega v_{j+1} v_i \leq d(v_{j+1}, v_0)$.

□

The full algorithm also includes a mechanism by which the nodes can detect that the computation has been completed; compare with the remark about the Netchange algorithm at the beginning of Subsection 4.3.3. The mechanism to detect this completion is a variation of the Dijkstra–Scholten algorithm discussed in Subsection 8.2.1.

The algorithm differs from the Merlin–Segall algorithm in two respects. First, there is no "father" of a node u that is excepted from being sent messages of the type $\langle\, \mathbf{mydist}, ., . \,\rangle$. This feature of the algorithm of Merlin and Segall ensures that the tables are always cycle-free, even during computation and in the presence of topological changes. Second, the exchange of $\langle\, \mathbf{mydist}, ., . \,\rangle$ messages is not coordinated in rounds, but takes place completely arbitrarily, which influences the complexity in a unfavorable way. The algorithm may require an exponential number of messages to compute the paths toward a single destination v_0. If all channel costs are assumed to be equal (i.e., minimum-hop routing is considered) all shortest paths towards v_0 are computed using $O(N \cdot |E|)$ messages (of $O(W)$ bits each), leading to the following result.

Theorem 4.13 *The algorithm of Chandy and Misra computes minimum-hop routing tables by exchanging* $O(N^2)$ *messages and* $O(N^2W)$ *bits per channel, and* $O(N^2 \cdot |E|)$ *messages and* $O(N^2 \cdot |E| \cdot W)$ *bits in total.*

An advantage of the algorithm of Chandy and Misra over that of Merlin and Segall is its simplicity, its smaller space complexity, and its lower time complexity.

4.3 The Netchange Algorithm

Tajibnapis' Netchange algorithm [Taj77] computes routing tables that are optimal according to the "minimum-hop" measure. The algorithm can be compared to the Chandy–Misra algorithm, but maintains additional information that allows the tables to be updated with only a *partial* recomputation after the failure or repair of a channel. The presentation of the algorithm in this section follows Lamport [Lam82]. The algorithm relies on the following assumptions.

N1. The nodes know the size of the network (N).
N2. The channels satisfy the fifo assumption.
N3. Nodes are notified of failures and repairs of their adjacent channels.
N4. The cost of a path equals the number of channels in the path.

The algorithm can handle the failure and repair or addition of channels, but it is assumed that a node is notified when an adjacent channel fails or recovers. The failure and recovery of nodes is not considered; instead it is assumed that the failure of a node is observed by its neighbors as the failure of the connecting channel. The algorithm maintains in each node u a table $Nb_u[v]$, giving for each destination v a neighbor of u to which packets for v will be forwarded. It cannot be required that the computation of these tables terminates within a finite number of steps in all cases because the repeated failure or repair of channels may ask for recomputation indefinitely. The requirements of the algorithm are as follows.

R1. If the topology of the network remains constant after a finite number of topological changes, then the algorithm terminates after a finite number of steps.
R2. When the algorithm terminates the tables $Nb_u[v]$ satisfy
    (a) if $v = u$ then $Nb_u[v] = \mathit{local}$;
    (b) if a path from u to $v \neq u$ exists then $Nb_u[v] = w$, where w is the first neighbor of u on a shortest path from u to v;
    (c) if no path from u to v exists then $Nb_u[v] = \mathit{udef}$.

```
var Neigh_u   : set of nodes ;   (* The neighbors of u *)
    D_u       : array of 0.. N ; (* D_u[v] estimates d(u, v) *)
    Nb_u      : array of nodes ; (* Nb_u[v] is preferred neighbor for v *)
    ndis_u    : array of 0.. N ; (* ndis_u[w, v] estimates d(w, v) *)

Initialization:
  begin forall w ∈ Neigh_u, v ∈ V do ndis_u[w, v] := N ;
        forall v ∈ V do
              begin D_u[v] := N ; Nb_u[v] := udef end ;
        D_u[u] := 0 ; Nb_u[u] := local ;
        forall w ∈ Neigh_u do send ⟨ mydist, u, 0 ⟩ to w
  end

Procedure Recompute (v):
  begin if v = u
          then begin D_u[v] := 0 ; Nb_u[v] := local end
          else begin (* Estimate distance to v *)
                     d := 1 + min{ndis_u[w, v] : w ∈ Neigh_u} ;
                     if d < N then
                       begin D_u[v] := d ;
                             Nb_u[v] := w with 1 + ndis_u[w, v] = d
                       end
                     else begin D_u[v] := N ; Nb_u[v] := udef end
                end ;
        if D_u[v] has changed then
          forall x ∈ Neigh_u do send ⟨ mydist, v, D_u[v] ⟩ to x
  end
```

Algorithm 4.8 THE NETCHANGE ALGORITHM (PART 1, FOR NODE u).

4.3.1 Description of the Algorithm

Tajibnapis' Netchange algorithm is given as Algorithms 4.8 and 4.9. The steps of the algorithm will first be motivated by an informal description of the operation of the algorithm, and subsequently the correctness of the algorithm will be proved formally. For sake of clear exposition the modeling of topological changes is simplified as compared to [Lam82] by assuming that the notification of the change is processed simultaneously in the two nodes affected by the change. It is indicated in Subsection 4.3.3 how asynchronous processing of these notifications is treated.

The selection of a neighbor to which packets for destination v will be forwarded is based on estimates of the distance of each node to v. The preferred neighbor is always the neighbor with the lowest estimate of this distance. Node u maintains an estimate $D_u[v]$ of $d(u, v)$ and estimates $ndis_u[w, v]$ of $d(w, v)$ for each neighbor w of u. The estimate $D_u[v]$ is

```
Processing a ⟨ mydist, v, d ⟩ message from neighbor w:
    { A ⟨ mydist, v, d ⟩ is at the head of Q_wv }
    begin receive ⟨ mydist, v, d ⟩ from w ;
          ndis_u[w, v] := d ; Recompute (v)
    end

Upon failure of channel uw:
    begin receive ⟨ fail, w ⟩ ; Neigh_u := Neigh_u \ {w} ;
          forall v ∈ V do Recompute (v)
    end

Upon repair of channel uw:
    begin receive ⟨ repair, w ⟩ ; Neigh_u := Neigh_u ∪ {w} ;
          forall v ∈ V do
                begin ndis_u[w, v] := N ;
                      send ⟨ mydist, v, D_u[v] ⟩ to w
                end
    end
```

Algorithm 4.9 THE NETCHANGE ALGORITHM (PART 2, FOR NODE u).

computed from the estimates $ndis_u[w, v]$, and the estimates $ndis_u[w, v]$ are obtained via communication with the neighbors.

The computation of the estimates $D_u[v]$ proceeds as follows. If $u = v$ then $d(u, v) = 0$ so $D_u[v]$ is set to 0 in this case. If $u \neq v$, a shortest path from u to v (if such a path exists) consists of a channel from u to a neighbor, concatenated with a shortest path from this neighbor to v, and consequently

$$d(u, v) = 1 + \min_{w \in Neigh_u} d(w, v).$$

Following this equation, node $u \neq v$ estimates $d(u, v)$ by applying this formula to the *estimated* values of $d(w, v)$, found in the tables as $ndis_u[w, v]$. As there are N nodes, a minimum-hop path has length at most $N - 1$. A node may suspect that no path exists if it computes an estimated distance of N or more; the value N is used in the table to represent this.

The algorithm requires a node to have an estimate of its neighbors' distances to v. These are obtained from these nodes because they communicate them in $\langle$ **mydist**, ., . $\rangle$ messages as follows. If node u computes the value d as an estimate of its distance to v ($D_u[v] = d$), this information is sent to all neighbors in a message $\langle$ **mydist**, $v, d \rangle$. Upon receipt of a message $\langle$ **mydist**, $v, d \rangle$ from neighbor w, u assigns $ndis_u[w, v]$ the value d. As a result of a change in $ndis_u[w, v]$ u's estimate of $d(u, v)$ can change and therefore the estimate is recomputed every time the $ndis_u$ table changes. If

the estimate indeed changes, to d' say, this is of course communicated to the neighbors using $\langle$ **mydist**, $v, d' \rangle$ messages.

The algorithm reacts to failures and repairs of channels by modifying the local tables, and sending a $\langle$ **mydist**, ., . $\rangle$ message if distance-estimates change. We assume that the notification that nodes receive about channel ups and downs (assumption N3) is in the form of $\langle$ **fail**, . $\rangle$ and $\langle$ **repair**, . $\rangle$ messages. The channel between nodes u_1 and u_2 is modeled by two queues, $Q_{u_1u_2}$ for the messages from u_1 to u_2 and $Q_{u_2u_1}$ for the messages from u_2 to u_1. When a channel fails these queues are removed from the configuration (effectively causing all messages in both queues to be lost) and the nodes at both ends of the channel receive a $\langle$ **fail**, . $\rangle$ message. If the channel between u_1 and u_2 fails, u_1 receives a $\langle$ **fail**, $u_2 \rangle$ message and u_2 receives a $\langle$ **fail**, $u_1 \rangle$ message. When a channel is repaired (or a new channel is added to the network) two empty queues are added to the configuration and the two nodes connected by the channel receive a $\langle$ **repair**, . $\rangle$ message. If the channel between u_1 and u_2 comes up u_1 receives a $\langle$ **repair**, $u_2 \rangle$ message and u_2 receives a $\langle$ **repair**, $u_1 \rangle$ message.

The reaction of the algorithm to the failures and repairs is as follows. When the channel between u and w fails, w is removed from $Neigh_u$ and vice versa. The distance estimate for each destination is recomputed and, of course, sent to all remaining neighbors if it has changed. This is the case if the best route previously was via the failed channel and there is no other neighbor w' with $ndis_u[w', v] = ndis_u[w, v]$. When the channel is repaired (or a new channel is added) w is added to $Neigh_u$, but u has as yet no estimate of the distance $d(w, v)$ (and vice versa). The new neighbor w is immediately informed about $D_u[v]$ for all destinations v (by sending $\langle$ **mydist**, $v, D_u[v] \rangle$ messages. Until u receives similar messages from w, u uses N as an estimate for $d(w, v)$, i.e., it sets $ndis_u[w, v]$ to N.

Invariants of the Netchange algorithm. We shall prove a number of assertions to be invariants; the assertions are given in Figure 4.10. The assertion $P(u, w, v)$ states that if u has finished processing $\langle$ **mydist**, $v, . \rangle$ messages from w then u's estimate of $d(w, v)$ equals w's estimate of $d(w, v)$. Let the predicate $up(u, w)$ be true if and only if a (bidirectional) channel between u and w exists and is operating. The assertion $L(u, v)$ states that u's estimate of $d(u, v)$ is always in agreement with u's local knowledge, and $Nb_u[v]$ is set accordingly.

The computation of the algorithm terminates when there are no more messages of the algorithm in transit in any channel. These configurations are not terminal for the whole system, because the system's computation

$$
\begin{aligned}
&P(u, w, v) \equiv \\
&\quad up(u, w) \iff w \in Neigh_u && (1)\\
&\quad \wedge\ up(u, w) \wedge Q_{wu} \text{ contains a } \langle\, \textbf{mydist}, v, d \,\rangle \text{ message} \\
&\qquad \Rightarrow \text{the last such message satisfies } d = D_w[v] && (2)\\
&\quad \wedge\ up(u, w) \wedge Q_{wu} \text{ contains no } \langle\, \textbf{mydist}, v, d \,\rangle \text{ message} \\
&\qquad \Rightarrow ndis_u[w, v] = D_w[v] && (3)
\end{aligned}
$$

$$
\begin{aligned}
&L(u, v) \equiv \\
&\quad u = v \Rightarrow (D_u[v] = 0 \wedge Nb_u[v] = local) && (4)\\
&\quad \wedge\ (u \neq v \wedge \exists w \in Neigh_u : ndis_u[w, v] < N - 1) \\
&\qquad \Rightarrow (D_u[v] = 1 + \min_{w \in Neigh_u} ndis_u[w, v] = 1 + ndis_u[Nb_u[v], v]) && (5)\\
&\quad \wedge\ (u \neq v \wedge \forall w \in Neigh_u : ndis_u[w, v] \geq N - 1) \\
&\qquad \Rightarrow (D_u[v] = N \wedge Nb_u[v] = udef) && (6)
\end{aligned}
$$

Figure 4.10 THE INVARIANTS $P(u, w, v)$ AND $L(u, v)$.

may later continue, starting with a channel failure or repair (to which the algorithm must react). We shall call message-less configurations *stable*, and define the predicate **stable** by

$$\textbf{stable} \equiv \forall u, w : up(u, w) \Rightarrow Q_{wu} \text{ contains no } \langle\, \textbf{mydist}, ., . \,\rangle \text{ message.}$$

It must be assumed that initially the variables $Neigh_u$ correctly reflect the existence of working communication channels, i.e., that (1) holds initially. To prove the invariance of the assertions three types of transition must be considered.

(1) The receipt of a $\langle\, \textbf{mydist}, ., . \,\rangle$ message. The entire execution of the resulting code fragment is assumed to occur atomically and is considered a single transition. Note that in this transition a message is received and possibly a number of messages is sent.
(2) The failure of a channel and the processing of a $\langle\, \textbf{fail}, . \,\rangle$ message by the nodes at both ends of the channel.
(3) The repair of a channel and the processing of a $\langle\, \textbf{repair}, . \,\rangle$ message by the two connected nodes.

Lemma 4.14 *For all u_0, w_0, and v_0, $P(u_0, w_0, v_0)$ is an invariant.*

Proof. Initially, i.e., after the execution of the initialization procedure by each node, (1) holds by assumption. If initially we have $\neg up(u_0, w_0)$, (2) and (3) trivially hold. If initially we have $up(u_0, w_0)$, then $ndis_{u_0}[w_0, v_0] = N$. If $w_0 = v_0$ then $D_{w_0}[w_0] = 0$ but a message $\langle\, \textbf{mydist}, v_0, 0 \,\rangle$ is in $Q_{w_0u_0}$, so (2) and (3) are true. If $w_0 \neq v_0$ then $D_{w_0}[v_0] = N$ and no message is in

the queue, which also implies that (2) and (3) hold. We consider the three types of state transition mentioned above in turn.

Type (1). Assume that u receives a $\langle \mathbf{mydist}, v, d \rangle$ message from w.

This causes no topological change and no change in the *Neigh* sets, hence (1) remains true. If $v \neq v_0$ this receipt does not change anything in $P(u_0, w_0, v_0)$.

If $v = v_0$, $u = u_0$, and $w = w_0$ the value of $ndis_{u_0}[w_0, v_0]$ may change. However, if another $\langle \mathbf{mydist}, v_0, . \rangle$ message is still in the channel then the value of this message continues to satisfy (2), so (2) is preserved and (3) also because its premise is false. If the received message was the last one in the channel of this type then $d = D_{w_0}[v_0]$ by (2), which implies that the conclusion of (3) becomes true and (3) is preserved. The premise of (2) becomes false, so (2) is preserved.

If $v = v_0$, $u = w_0$ (and u_0 is a neighbor of u) the conclusion of (2) or (3) may be falsified if the value $D_{w_0}[v_0]$ changes as a result of the execution of *Recompute*(v) in w_0. In this case, however, a message $\langle \mathbf{mydist}, v_0, . \rangle$ with the new value is sent to u_0, which implies that the premise of (3) is falsified, and the conclusion of (2) becomes true, so both (2) and (3) are preserved. This is also the only case in which a $\langle \mathbf{mydist}, v_0, . \rangle$ message is added to $Q_{w_0 u_0}$, and it always satisfies $d = D_{w_0}[v_0]$.

If $v = v_0$ and $u \neq u_0$, w_0 nothing changes in $P(u_0, w_0, v_0)$.

Type (2). Assume that channel uw fails.

If $u = u_0$ and $w = w_0$ this failure falsifies the premise of (2) and (3) so these clauses are preserved. (1) is preserved because w_0 is removed from $Neigh_{u_0}$ and vice versa. The same happens if $u = w_0$ and $w = u_0$.

If $u = w_0$ but $w \neq u_0$ the conclusion of (2) or (3) may be falsified because the value $D_{w_0}[v_0]$ changes. In this case the sending of a $\langle \mathbf{mydist}, v_0, . \rangle$ message by w_0 again falsifies the premise of (3) and makes the conclusion of (2) true, hence (2) and (3) are preserved.

In all other cases nothing changes in $P(u_0, w_0, v_0)$.

Type (3). Assume that channel uw is added.

If $u = u_0$ and $w = w_0$ this makes $up(u_0, w_0)$ true, but by the addition of w_0 to $Neigh_{u_0}$ (and vice versa) this preserves (1).

The sending of $\langle \mathbf{mydist}, v_0, D_{w_0}[v_0] \rangle$ by w_0 makes the conclusion of (2) true and the premise of (3) false, so $P(u_0, w_0, v_0)$ is preserved.

In all other cases nothing changes in $P(u_0, w_0, v_0)$.

□

Lemma 4.15 *For each u_0 and v_0, $L(u_0, v_0)$ is an invariant.*

Proof. Initially $D_{u_0}[u_0] = 0$ and $Nb_{u_0}[u_0] = local$. For $v_0 \neq u_0$, initially $ndis_{u_0}[w, v_0] = N$ for all $w \in Neigh_u$, and $D_{u_0}[v_0] = N$ and $Nb_{u_0}[v_0] = udef$.

Type (1). Assume that u receives a $\langle$ **mydist**, $v, d \rangle$ message from w.
If $u \neq u_0$ or $v \neq v_0$ no variable mentioned in $L(u_0, v_0)$ changes.
If $u = u_0$ and $v = v_0$ the value of $ndis_{u_0}[w, v_0]$ changes, but $D_{u_0}[v_0]$ and $Nb_{u_0}[v_0]$ are recomputed exactly so as to satisfy $L(u_0, v_0)$.

Type (2). Assume that channel uw fails.
If $u = u_0$ or $w = u_0$ then $Neigh_{u_0}$ changes, but again $D_{u_0}[v_0]$ and $Nb_{u_0}[v_0]$ are recomputed exactly so as to satisfy $L(u_0, v_0)$.

Type (3). Assume that channel uw is added.
If $u = u_0$ then $Neigh_{u_0}$ changes by the addition of w, but as u sets $ndis_{u_0}[w, v_0]$ to N this preserves $L(u_0, v_0)$.

□

4.3.2 Correctness of the Netchange Algorithm

The two correctness requirements for the algorithm will now be proved.

Theorem 4.16 *When a stable configuration is reached, the tables $Nb_u[v]$ satisfy*

(1) *if $u = v$ then $Nb_u[v] = local$;*
(2) *if a path from u to $v \neq u$ exists then $Nb_u[v] = w$, where w is the first neighbor of u on a shortest path from u to v;*
(3) *if no path from u to v exists then $Nb_u[v] = udef$.*

Proof. When the algorithm terminates, the predicate **stable** holds in addition to $P(u, w, v)$ for all u, v, and w, and this implies that for all u, v, and w

$$up(u, w) \Rightarrow ndis_u[w, v] = D_w[v]. \tag{4.2}$$

Applying also $L(u, v)$ for all u and v we obtain

$$D_u[v] = \begin{cases} 0 & \text{if } u = v \\ 1 + \min\limits_{w \in Neigh_u} D_w[v] & \text{if } u \neq v \wedge \exists w \in Neigh_u : D_w[v] < N - 1 \\ N & \text{if } u \neq v \wedge \forall w \in Neigh_u : D_w[v] \geq N - 1 \end{cases} \tag{4.3}$$

which is sufficient to prove that $D_u[v] = d(u, v)$ if u and v are in the same connected component of the network, and $D_u[v] = N$ if u and v are in different connected components.

First it is shown by induction on $d(u, v)$ that if u and v are in the same connected component then $D_u[v] \leq d(u, v)$.

Case $d(u, v) = 0$**:** this implies $u = v$ and hence $D_u[v] = 0$.

Case $d(u, v) = k + 1$**:** this implies that there exists a node $w \in Neigh_u$ with $d(w, v) = k$. By induction $D_w[v] \leq k$, which by (4.3) implies $D_u[v] \leq k + 1$.

Now it will be shown by induction on $D_u[v]$ that if $D_u[v] < N$ then there is a path between u and v and $d(u, v) \leq D_u[v]$.

Case $D_u[v] = 0$**:** Formula (4.3) implies that $D_u[v] = 0$ only for $u = v$, which gives the empty path between u and v, and $d(u, v) = 0$.

Case $D_u[v] = k + 1 < N$**:** Formula (4.3) implies that there is a node $w \in Neigh_u$ with $D_w[v] = k$. By induction there is a path between w and v and $d(w, v) \leq k$, which implies there is a path between u and v and $d(u, v) \leq k + 1$.

It follows that if u and v are in the same connected component then $D_u[v] = d(u, v)$, otherwise $D_u[v] = N$. This, Formula (4.2), and $\forall u, v : L(u, v)$ imply the stated result about $Nb_u[v]$. □

To prove that a stable situation is eventually reached if topological changes cease, a norm function with respect to **stable** will be defined. Define, for a configuration γ of the algorithm,

$$\begin{aligned} t_i = \quad & (\text{the number of } \langle\, \textbf{mydist}, ., i \,\rangle \text{ messages}) \\ + \; & (\text{the number of ordered pairs } u, v \text{ s.t. } D_u[v] = i) \end{aligned}$$

and the function f by

$$f(\gamma) = (t_0, t_1, \ldots, t_N).$$

$f(\gamma)$ is an $(N + 1)$-tuple of natural numbers, on which a lexicographic order (denoted $\leq_l$) is assumed. Recall that $(\mathbb{N}^{N+1}, \leq_l)$ is a well-founded set (Exercise 2.5).

Lemma 4.17 *The processing of a* $\langle\, \textbf{mydist}, ., . \,\rangle$ *message decreases* f.

Proof. Assume node u with $D_u[v] = d_1$ receives a $\langle\, \textbf{mydist}, v, d_2 \,\rangle$ message, and after recomputation the new value of $D_u[v]$ is d. The algorithm implies that $d \leq d_2 + 1$.

Case $d < d_1$**:** Now $d = d_2 + 1$ which implies that t_{d_2} is decreased by one (and t_{d_1} as well), and only t_d with $d > d_2$ is increased. This implies that the value of f is decreased.

Case $d = d_1$**:** No new $\langle\, \textbf{mydist}, ., . \,\rangle$ messages are sent by u, and the only effect on f is that t_{d_2} is decreased by one, so the value of f is decreased.

Case $d > d_1$: Now t_{d_1} is decreased by one (and t_{d_2} as well), and only t_d with $d > d_1$ is increased. This implies that the value of f is decreased.

□

Theorem 4.18 *If the topology of the network remains constant after a finite number of topological changes, then the algorithm reaches a stable configuration after a finite number of steps.*

Proof. If the network topology remains constant only further processing of $\langle$ **mydist**, ., . $\rangle$ messages takes place, and, by the previous lemma, the value of f decreases with every such transition. It follows from the well-foundedness of the domain of f that only a finite number of these transitions can take place; hence the algorithm reaches a configuration satisfying **stable** after a finite number of steps. □

4.3.3 Discussion of the Algorithm

The formal correctness results of the algorithm, guaranteeing the convergence to correct tables within finite time after the last topological change, are not very indicative about the actual behavior of the algorithm. The predicate **stable** may in practice be false most of the time (namely, if topological changes are frequent) and when **stable** is false nothing is known about the routing tables. They may contain cycles or even give erroneous information about the reachability of a destination node. The algorithm can therefore only be used in applications where topological changes are so infrequent that the convergence time of the algorithm is small compared with the average time between (bursts of) topological changes. This is all the more the case because **stable** is a global property, and stable configurations of the algorithm are indistinguishable from non-stable ones for the nodes. This means that a node never knows whether its routing table correctly reflects the network topology, and cannot defer forwarding data packets until a stable configuration is reached.

Asynchronous processing of notifications. It has been assumed in this section that the notifications of topological changes are processed atomically together with the change in a single transition. The processing takes place at both sides of the removed or added channel simultaneously. Lamport [Lam82] has carried out the analysis in a little more detail to allow a delay in processing these notifications. The communication channel from w to u is modeled as the concatenation of three queues.

(1) OQ_{wu}, the output queue of w;
(2) TQ_{wu}, the queue of messages (and data packets) currently being transmitted;
(3) IQ_{wu}, the input queue of u.

Under the normal operation of a channel w sends a message to u by appending it to OQ_{wu}, messages move from OQ_{wu} to TQ_{wu} and from TQ_{wu} to IQ_{wu}, and u receives them by deleting them from IQ_{wu}. When the channel fails the messages in TQ_{wu} are thrown away and messages in OQ_{wu} are thereafter also thrown away rather than appended to TQ_{wu}. The $\langle$ **fail**, w $\rangle$ message is placed at the end of IQ_{wu}, and when normal operation is resumed the $\langle$ **repair**, w $\rangle$ message is also placed at the end of IQ_{wu}. The predicates $P(u, w, v)$ take a slightly more complicated form, but the algorithm remains the same.

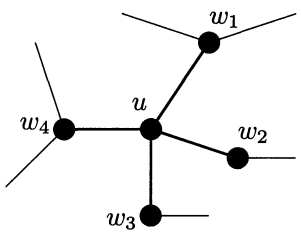
Shortest-path routing. It is possible to assign a weight to each channel and modify the algorithm so as to compute shortest paths rather than minimum-hop paths. The procedure *Recompute* of the Netchange algorithm takes the weight of channel uw into account when estimating the length of the shortest path via w if the constant 1 is replaced by ω_{uw}. The constant N in the algorithm must be replaced by an upper bound on the diameter of the network.

It is fairly easy to show that when the modified algorithm reaches a stable configuration the routing tables are indeed correct and give optimal paths (all cycles in the network must have positive weight). The proof that the algorithm eventually reaches such a configuration requires a more complicated norm function.

It is even possible to extend the algorithm to deal with varying channel weights; the reaction of node u to a change in a channel weight is the recomputation of $D_u[v]$ for all v. The algorithm would be practical, however, only in situations where the average time between channel-cost changes is large compared to the convergence time, which is a quite unrealistic assumption. In these situations an algorithm should be preferred that guarantees cycle-freedom also during convergence, for example the Merlin–Segall algorithm.

4.4 Routing with Compact Routing Tables

The routing algorithms discussed so far all require that each node maintains a routing table with a separate entry for each possible destination. When a packet is forwarded through the network these tables are accessed in each



dest.	chan.
v_1	w_2
u	-
v_j	w_3
v_N	w_1

chan.	destinations
w_1	..., v_N
w_2	v_1, ...
w_3	..., v_j, ...
w_4	

Figure 4.11 REDUCING THE SIZE OF ROUTING TABLES.

node of the path (except the destination). In this section some routing table organizations that decrease the storage and table lookup overheads of routing mechanisms will be studied. How these routing tables can be computed by a distributed algorithm will not be considered here. For simplicity of presentation it is assumed throughout this section that the network is connected.

The strategy for obtaining smaller tables in each of the three routing mechanisms discussed in this section is easily explained as follows. If the routing tables of a node store the outgoing channel for each destination separately, the routing table necessarily has length N; hence the tables require $\Omega(N)$ bits, no matter how compactly the outgoing channel is encoded for each destination. Now consider a reorganization of the table, in which the table contains for each *channel* of the node an entry telling which destinations must be routed via this channel; see Figure 4.11. The table now has "length" *deg* for a node with *deg* channels; the actual saving in storage of course depends on how compactly the set of destinations for each channel can be represented. In order to keep a table-lookup efficient the table must be organized in such a way that the outgoing channel for a given destination can be retrieved quickly from the table.

4.4.1 The Tree-labeling Scheme

The first compact routing method was proposed by Santoro and Khatib [SK85]. The method is based on a labeling of the nodes with integers from 0 to $N-1$, in such a way that the set of destinations for each channel is an interval. Let $\mathbb{Z}_N$ denote the set $\{0, 1, \ldots, N-1\}$. In this section all arithmetic in this set is done modulo N, i.e., $(N-1)+1 \equiv 0$, but the order is as in $\mathbb{Z}$.

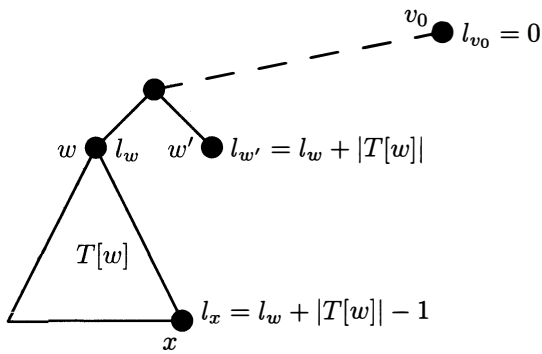


Figure 4.12 PREORDER TREE TRAVERSAL.

Definition 4.19 *The cyclic interval* $[a, b)$ *in* $\mathbb{Z}_N$ *is the set of integers defined by*

$$[a, b) = \begin{cases} \{a, a+1, \ldots, b-1\} & \text{if } a < b \\ \{0, \ldots, b-1, a, \ldots, N-1\} & \text{if } a \geq b \end{cases}$$

Observe that $[a, a) = \mathbb{Z}_N$, and for $a \neq b$ the complement of $[a, b)$ is $[b, a)$. The cyclic interval $[a, b)$ is called *linear* if $a < b$.

Theorem 4.20 *The nodes of a tree* T *can be numbered in such a way that for each outgoing channel of each node the set of destinations that must be routed via that channel is a cyclic interval.*

Proof. Pick an arbitrary node v_0 as the root of the tree and for each w let $T[w]$ denote the subtree of T rooted at w. It is possible to number the nodes in such a way that for each w the numbers assigned to the nodes in $T[w]$ form a linear interval, for example, by a preorder traversal of the tree as in Figure 4.12. In this order, w is the first node of $T[w]$ to be visited and after w all nodes of $T[w]$ are visited before a node not in $T[w]$ is visited; hence the nodes in $T[w]$ are numbered by the linear interval $[l_w, l_w + |T[w]|)$ (l_w is the label of w).

Let $[a_w, b_w)$ denote the interval of numbers assigned to the nodes in $T[w]$. A neighbor of w is either a son or the father of w. Node w forwards to a son u the packets with destinations in $T[u]$, i.e., the nodes with numbers in $[a_u, b_u)$. Node w forwards to its father the packets with destinations not in $T[w]$, i.e., the nodes with numbers in $\mathbb{Z}_N \setminus [a_w, b_w) = [b_w, a_w)$. □

```
(* A packet with address d was received or generated at node u *)
if d = l_u
  then deliver the packet locally
  else begin select α_i s.t. d ∈ [α_i, α_{i+1}) ;
             send packet via the channel labeled with α_i
       end
```

Algorithm 4.13 INTERVAL FORWARDING (FOR NODE u).

A single cyclic interval can be represented using only $2 \log N$ bits by giving the start point and end point. Because in this application a collection of disjoint intervals with union $\mathbb{Z}_N$ must be stored (by adding u to one of the intervals in node u), $\log N$ bits per interval are sufficient. Only the start point of the interval corresponding to each channel is stored; the corresponding end point is equal to the next begin point of an interval in the same node. The begin point of the interval corresponding with channel uw at node u is given by

$$\alpha_{uw} = \begin{cases} l_w & \text{if } w \text{ is a son of } u, \\ l_u + |T[u]| & \text{if } w \text{ is the father of } u. \end{cases}$$

Assuming that the channels of node u of degree deg_u are labeled with $\alpha_1, \ldots, \alpha_{deg_u}$, where $\alpha_1 < \cdots < \alpha_{deg_u}$, the forwarding procedure is as given in Algorithm 4.13. The channel-labels partition the set $\mathbb{Z}_N$ into deg_u segments, each corresponding to one channel; see Figure 4.14. Observe that there is (at most) one interval that is not linear. If the labels are sorted at the node, the correct label is found in $O(\log deg_u)$ steps using binary search. The index i is counted modulo deg_u, i.e., $\alpha_{deg_u+1} = \alpha_1$.

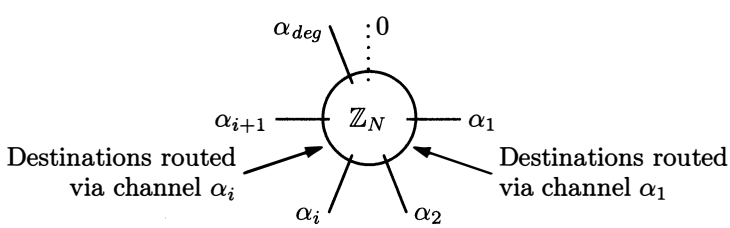


Figure 4.14 THE PARTITION OF $\mathbb{Z}_N$ IN A NODE.

The tree labeling scheme routes optimally on trees, because in a tree there exists only one simple path between each two nodes. The scheme can also be used if the network is not a tree. A fixed spanning tree T of the network is chosen, and the scheme is applied to this tree. Channels not belonging to the spanning tree are never used; each is labeled with a special symbol in the routing table to indicate that no packet is routed via this channel.

To compare the path lengths chosen by this scheme with the optimal paths, let $d_T(u, v)$ denote the distance from u to v in T and $d_G(u, v)$ the distance from u to v in G. Let D_G denote the *diameter* of G, defined as the maximum over u and v of $d_G(u, v)$.

Lemma 4.21 *There is no uniform bound on the ratio between $d_T(u, v)$ and $d_G(u, v)$. This holds already in the special case of the hop measure for paths.*

Proof. Choose G to be the ring on N nodes, and observe that a spanning tree of G is obtained by removing one channel, say xy, from G. Now $d_G(x, y) = 1$ and $d_T(x, y) = N-1$, so the ratio is $N-1$. The ratio can be made arbitrarily large by choosing a large ring. □

The following lemma relies on the symmetry of channel costs, i.e., it is assumed that $\omega_{uw} = \omega_{wu}$. This implies that $d_G(u, v) = d_G(v, u)$ for all u and v.

Lemma 4.22 *T can be chosen in such a way that for all u and v, $d_T(u, v) \leq 2D_G$.*

Proof. Choose T to be the optimal sink tree for a node w_0 (as in Theorem 4.2). Then

$$\begin{aligned} d_T(u, v) &\leq d_T(u, w_0) + d_T(w_0, v) \\ &= d_T(u, w_0) + d_T(v, w_0) && \text{by symmetry of } \omega \\ &= d_G(u, w_0) + d_G(v, w_0) && \text{by the choice of } T \\ &\leq D_G + D_G && \text{by definition of } D_G. \end{aligned}$$

□

Concluding, a path chosen by this scheme can be arbitrarily bad if compared with an optimal path between the same two nodes (Lemma 4.21), but if a suitable spanning tree is chosen, it is at most twice as bad as the path between two other nodes in the system (Lemma 4.22). This implies that the scheme is good if most communication is between nodes at distance $\Theta(D_G)$, but should not be used if most communication is between nodes at a short distance (in G) from each other.

Besides the factor concerning the lengths of the chosen paths, the tree routing scheme has the following disadvantages.

(1) Channels not belonging to T are not used, which is a waste of network resources.
(2) Traffic is concentrated on a tree, which may lead to congestion.
(3) Each single failure of a channel of T partitions the network.

4.4.2 Interval Routing

Van Leeuwen and Tan [LT87] extended the tree labeling scheme to non-tree networks in such a way that (almost) every channel is used for packet traffic.

Definition 4.23 *An interval labeling scheme (ILS) for a network is*

(1) *an assignment of different labels from $\mathbb{Z}_N$ to the nodes of the network, and,*
(2) *for each node, an assignment of different labels from $\mathbb{Z}_N$ to the channels of that node.*

The interval routing algorithm assumes that an ILS is given, and forwards packets as in Algorithm 4.13.

Definition 4.24 *An interval labeling scheme is valid if all packets forwarded in this way eventually reach their destination.*

It will be shown that a valid interval labeling scheme exists for every connected network G (Theorem 4.25); for arbitrary connected networks, however, the scheme is usually not very efficient. The optimality of the paths chosen by interval routing schemes will be studied after the existence proof.

Theorem 4.25 *For each connected network G a valid interval labeling scheme exists.*

Proof. A valid interval labeling scheme is constructed by an extension of the tree labeling scheme of Santoro and Khatib, applied to a spanning tree T of the network. Given a spanning tree, a *frond edge* is an edge that does not belong to this spanning tree. Furthermore, v is an *ancestor* of u iff $u \in T[v]$. As the main problem of the construction is how to assign labels to frond edges (the tree edges will be labeled as in the tree labeling scheme), a spanning tree is chosen in such a way that all frond edges take a restricted form, as we will show.

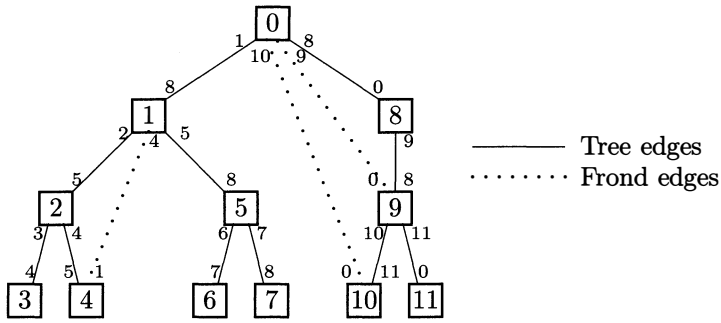


Figure 4.15 A DEPTH-FIRST SEARCH ILS.

Lemma 4.26 *There exists a spanning tree such that all frond edges are between a node and an ancestor of that node.*

Proof. Each spanning tree that is obtained by a depth-first search through the network has this property; see [Tar72] and Section 6.4. □

In the sequel, let T be a fixed depth-first search spanning tree of G.

Definition 4.27 *A depth-first search ILS for G (with respect to T) is a labeling scheme for which the following rules are satisfied.*

(1) *The node labels are assigned as in a preorder traversal of T, i.e., the nodes in subtree $T[w]$ are labeled with the numbers in $[l_w, l_w + |T[w]|)$. Write $k_w = l_w + |T[w]|$.*

(2) *The label of edge uw at node u is called α_{uw}.*

   (a) *If uw is a frond edge then $\alpha_{uw} = l_w$.*

   (b) *If w is a son of u (in T) then $\alpha_{uw} = l_w$.*

   (c) *If w is the father of u then $\alpha_{uw} = k_u$ unless $k_u = N$ and u has a frond to the root.*
   (In the latter situation, the frond edge is labeled 0 at u by the rule (a), so assigning the label k_u would violate the requirement that all edge labels at u are different. Labels are regarded as modulo N, so $N \equiv 0$.)

   (d) *If w is the father of u, u has a frond to the root, and $k_u = N$, then $\alpha_{uw} = l_w$.*

An example of a depth-first search ILS is given in Figure 4.15. Observe that all frond edges are labeled according to rule (2a), the father-edges of nodes

4, 8, and 10 are labeled according to rule (2c), and the father-edge of node 9 is labeled according to rule (2d).

It will now be shown that a depth-first search ILS is a valid scheme. Observe that $v \in T[u] \iff l_v \in [l_u, k_u)$. The following three lemmas concern the situation where node u forwards a packet with destination v to node w (a neighbor of u) using Algorithm 4.13. This implies that $l_v \in [\alpha_{uw}, \alpha)$ for some label α in u, and that there is no label $\alpha' \neq \alpha_{uw}$ in node u such that $\alpha' \in [\alpha_{uw}, l_v)$.

Lemma 4.28 *If $l_u > l_v$ then $l_w < l_u$.*

Proof. First, consider the case where $\alpha_{uw} \leq l_v$. Node w is not a son of u because in that case $\alpha_{uw} = l_w > l_u > l_v$. If uw is a frond then also $l_w = \alpha_{uw} < l_v < l_u$. If w is the father of u then $l_w < l_u$ holds in any case.
Second, consider the case where α_{uw} is the largest edge label in u, and there is no label $\alpha' \leq l_v$ (i.e., l_v is in the lower part of a non-linear interval). In this case the edge to u's father is not labeled with 0, but with k_u (because $0 \leq l_v$, and there is no label $\alpha' \leq l_v$). The label k_u is the largest label in this case; an edge to a son or downward frond w' has $\alpha_{uw'} = l_{w'} < k_u$, and a frond to an ancestor w' has $\alpha_{uw'} = l_{w'} < l_u$. So w is the father of u in this case, which implies $l_w < l_u$. □

The next two lemmas concern the case where $l_u < l_v$. We deduce that either $v \in T[u]$ or $l_v \geq k_u$, and in the latter case $k_u < N$ holds so that the edge to the father of u is labeled with k_u.

Lemma 4.29 *If $l_u < l_v$ then $l_w \leq l_v$.*

Proof. First consider the case where $v \in T[u]$; let w' be the son of u such that $v \in T[w']$. We have $\alpha_{uw'} = l_{w'} \leq l_v$ and this implies that $\alpha_{uw'} \leq \alpha_{uw} \leq l_v < k_{w'}$. We deduce that w is not the father of u, hence $l_w = \alpha_{uw}$, which implies $l_w \leq l_v$.
Second, consider the case where $l_v \geq k_u$. In this case w is the father of u; this can be seen as follows. The edge to the father is labeled with k_u, and $k_u \leq l_v$. The edge to a son w' of u is labeled with $l_{w'} < k_u$, the edge to a downward frond w' is labeled with $l_{w'} < k_u$, and the edge to an upward frond w' is labeled with $l_{w'} < l_u$. Because w is the father of u, $l_w < l_u < l_v$. □

A norm function with respect to delivery at v can be defined as follows. The *lowest common ancestor* of two nodes u and v is the lowest node in the tree that is an ancestor of both u and v. Let $\text{lca}(u, v)$ denote the label of the

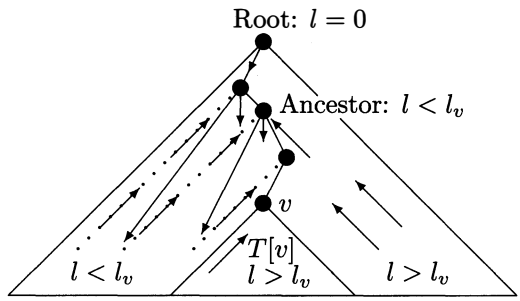


Figure 4.16 ROUTING OF PACKETS FOR v IN THE DEPTH-FIRST SEARCH ILS.

lowest common ancestor of u and v, and define

$$f_v(u) = (-\text{lca}(u, v),\ l_u).$$

Lemma 4.30 *If $l_u < l_v$ then $f_v(w) < f_v(u)$.*

Proof. First consider the case where $v \in T[u]$, which implies $\text{lca}(u, v) = l_u$. If w' is the son of u such that $v \in T[w']$, we have (as in the previous lemma) that $l_{w'} \leq l_w < k_{w'}$, hence $w \in T[w']$, which implies $\text{lca}(w, v) \geq l_{w'} > l_u$. So $f_v(w) < f_v(u)$.
Second, consider the case that $l_v \geq k_u$. As in the previous lemma, w is the father of u, and because $v \notin T[u]$, $\text{lca}(w, v) = \text{lca}(u, v)$. But now $l_w < l_u$, so $f_v(w) < f_v(u)$. □

It can now be shown that each packet reaches its destination. The flow of packets towards v is indicated in Figure 4.16. Let a packet for v be generated in node u. By Lemma 4.28, the node label decreases at every hop, until, within a finite number of hops, the packet is received by a node w with $l_w \leq l_v$. Every node to which the packet is forwarded after w also has a label $\leq l_v$, by Lemma 4.29. Within a finite number of hops the packet is received by v, because in each hop f_v decreases or the packet arrives at v, by Lemma 4.30. This completes the proof of Theorem 4.25. □

Efficiency of interval routing: the general case. Theorem 4.25 states that a valid ILS exists for each network, but does not imply anything about the efficiency of the paths chosen by the scheme. It should be clear that depth-first search ILSs are used to demonstrate the *existence* of a scheme for each network, but that they are not necessarily the best possible schemes. For example, if the depth-first search scheme is applied to a ring of N nodes,

there are nodes u and v with $d(u, v) = 2$, and the scheme uses $N-2$ hops to deliver a packet from u to v (Exercise 4.8). There exists an ILS for the same ring that delivers each packet via a minimum-hop path (Theorem 4.34).

In order to analyze the quality of the routing method in this respect, the following definitions are first made.

Definition 4.31 *An ILS is optimal if it forwards all packets via optimal paths.*
An ILS is neighborly if it delivers a packet from one node to a neighbor of that node in one hop.
An ILS is linear if the interval corresponding with each edge is linear.

We shall call an ILS *minimum-hop* (or *shortest-path*) if it is optimal with respect to the minimum-hop (or shortest-path, respectively) cost measure. It is easily seen that if a scheme is minimum hop then it is neighborly. It is also easy to verify that an ILS is linear if and only if in each node u with $l_u \neq 0$ there is an edge labeled 0, and in the node with label 0 there is an edge with label 0 or 1. It turns out that for general networks the quality of the routing method is poor, but for several classes of special network topology the quality of the scheme is very good. This makes the method suitable for processor networks with a regular structure, such as those used for the implementation of parallel computers with a virtual global shared memory.

It is not known exactly how, for an arbitrary network, the best interval labeling scheme compares with an optimal routing algorithm. Some lower bounds for the path lengths, implying that an optimal ILS does not always exist, were given by Ružička.

Theorem 4.32 [Ruž88] *There exists a network G such that for each valid ILS of G there exist nodes u and v such that a packet from u to v is delivered only after at least $\frac{3}{2}D_G$ hops.*

It is also not known how the best depth-first search ILS for a network compares with the overall best ILS for the same network. Exercise 4.7 gives a very bad depth-first search ILS for a network that actually admits an optimal ILS (by Theorem 4.37), but there may be a better depth-first search ILS for the same network.

In situations where most of the communication is between neighbors, being neighborly is a sufficient requirement for the ILS. As can be seen from Figure 4.15 a depth-first search ILS is not necessarily neighborly; node 4 forwards packets for node 2 via node 1.

Multiple interval routing schemes. The efficiency of the routing method can be improved by allowing more than one label to be assigned to each edge; we speak of *multiple interval routing* in this case. Indeed, this defines the set of destinations for this edge to be the union of several intervals and by increasing the number of intervals even optimal routing can be achieved for arbitrary networks. To see this, first consider optimal routing tables, such as computed by the Netchange algorithm for example, and observe that the set of destinations routed through any particular edge can be written as the union of cyclic intervals. With this elementary approach one finds at most $N/2$ labels per edge, and at most N labels altogether in any node; the storage of such a table takes as much space as the storage of classical, full routing tables.

It is now possible to trade storage complexity against routing efficiency; the natural question arises how many labels are really necessary in any network to achieve optimal routing. Pioneering work of Flammini *et al.* [FLMS95] developed a method for finding schemes and proving lower bounds, showing that in the general case $\Theta(N)$ labels per link may be required for optimality. But allowing just a small compromise in the length of the paths, a large reduction in table size can be achieved; results regarding this tradeoff are summarized by Ružička [Ruž98].

Linear interval routing schemes. It is essential for the applicability of the interval routing method that cyclic intervals are considered. Although some networks do have valid, and even optimal, linear-interval labeling schemes, it is not possible to label every network validly with linear intervals. The applicability of linear-interval labeling schemes was investigated by Bakker, Van Leeuwen, and Tan [BLT91].

Theorem 4.33 *There exists a network for which no valid linear-interval labeling scheme exists.*

Proof. Consider a spider graph with three legs of length two, as depicted in Figure 4.17. The smallest label (0) and largest label (6) are assigned to two nodes, and as there are three legs, there is (at least) one leg that contains neither the smallest nor the largest label. Let x be the first node from the center in this leg. Node x forwards packets addressed to 0 and 6 to the center, and the only linear interval that contains both 0 and 6 is the entire set $\mathbb{Z}_N$. Consequently, x also forwards packets for its other neighbor towards the center, and these packets never reach their destination. □

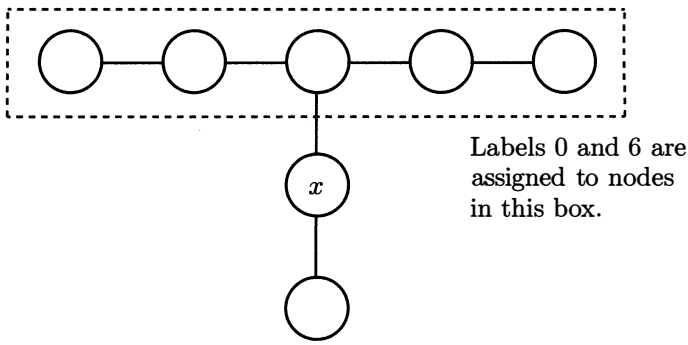


Figure 4.17 THE SPIDER GRAPH WITH THREE LEGS.

Bakker, Van Leeuwen and Tan completely characterize the class of network topologies that admit a shortest-path linear ILS, and present a number of results concerning the classes of graph topologies that admit adaptive and minimum-hop linear ILSs. Networks that allow a linear scheme at all were characterized by Fragniaud and Gavoille [FG94]; Eilam *et al.* [EMZ96] showed that the path lengths obtained with linear schemes in these networks can be as big as $O(D^2)$.

Optimality of interval routing: special topologies. It turns out that there are optimal interval labeling schemes for several classes of networks having a regular structure. Networks of these structures are used, for example, in the implementation of parallel computers.

Theorem 4.34 [LT87] *There exists a minimum-hop ILS for a ring of N nodes.*

Proof. The node labels are assigned from 0 to $N-1$ in clockwise order. For node i the clockwise channel is assigned the label $i+1$ and the anticlockwise channel is assigned $(i + \lceil N/2 \rceil) \bmod N$; see Figure 4.18. With this labeling scheme the node with label i sends packets for nodes $i+1, \ldots, (i+\lceil N/2 \rceil)-1$ via the clockwise channel and packets for nodes $(i + \lceil N/2 \rceil), \ldots, i-1$ via the anticlockwise channel, which is optimal. □

As the ILS in the proof of Theorem 4.34 is optimal, it is neighborly; it is not linear.

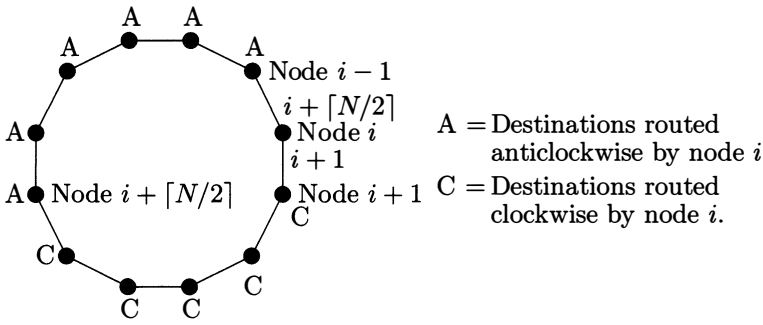


Figure 4.18 OPTIMAL ILS FOR A RING.

Theorem 4.35 [LT87] *There exists a minimum-hop ILS for an $n \times n$ grid.*

Proof. The node labels are assigned in row major order, i.e., the ith node on the jth row is labeled $(j-1)n+(i-1)$. The up-channel of this node is labeled 0, the left-channel of this node is labeled $(j-1)n$, the right-channel of this node is labeled $(j-1)n+i$, and the down-channel of this node is labeled $j \cdot n$; see Figure 4.19.

It is now easy to verify that when node u forwards a packet to node v,

Case 1: if v is in a row higher than u, then u sends the packet via its up-channel;

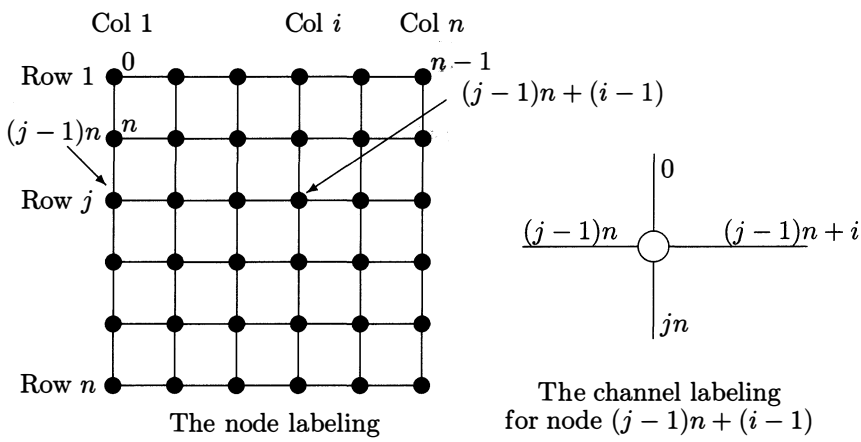


Figure 4.19 OPTIMAL ILS FOR AN $n \times n$ GRID.

Case 2: if v is in a row lower than u, then u sends the packet via its down-channel;

Case 3: if v is in the same row as u but to the left, then u sends the packet via its left-channel; and

Case 4: if v is in the same row as u but to the right, then u sends the packet via its right-channel.

In all cases, u sends the packet to a node closer to v, which implies that the chosen path is optimal. □

As the ILS in the proof of Theorem 4.35 is optimal, it is neighborly; the scheme is also linear.

We state the following two results without proof; the construction of labeling schemes as claimed in the theorems is left as an exercise for the reader.

Theorem 4.36 *There exists a minimum-hop linear ILS for the Hypercube.*

Theorem 4.37 [FJ88] *There exists a shortest-path ILS for outerplanar networks with arbitrary channel weights.*

Interval routing has a number of attractive advantages, as follows, over classical routing mechanisms based on storing a preferred outgoing channel for each destination separately.

(1) *Low space complexity.* The routing tables can be stored in $O(deg \cdot \log N)$ bits for a node of degree *deg*.
(2) *Efficient computation of routing tables.* Routing Tables for a depth-first search ILS can be computed using a distributed depth-first search traversal of the network, which can be done using $O(E)$ messages in $O(N)$ time; see Section 6.4.
(3) *Optimality.* The routing method is able to choose optimal paths for several classes of network, cf. Theorems 4.34 through 4.37.

These advantages make the method suitable for processor networks with a regular topology. As Transputers are often used to construct such topologies, the Inmos C104 routing chips (see Subsection 1.1.5) are designed to use interval routing.

Unfortunately, for applications in networks of arbitrary topology, when the method uses a depth-first search ILS there are a number of disadvantages, as follows.

(1) *Poor robustness.* It is not possible to adapt a depth-first search ILS slightly if a channel or node is added to or removed from the network.

The depth-first search tree on which the ILS is based may no longer satisfy the requirement that there are fronds only between a node and its ancestor. As a result, a minor modification of the topology may require a complete recomputation of the routing tables, including the assignment of new addresses (labels) to each node.

(2) *Non-optimality.* A depth-first search ILS may route packets via paths of length $\Omega(N)$, even in cases of networks of small diameter; see Exercise 4.7.

Many variants of interval routing were discussed in the literature. Multiple interval routing, already discussed earlier, appears to be a quite practical method because the cost of having extra labels is not prohibitive with modern memory techniques. A multi-dimensional variant was described by Flammini *et al.* [FGNT98]. The node names and labels are points and intervals in a multi-dimensional space rather than $\mathbb{Z}_N$ and the method brings advantages in networks that have a multi-dimensional structure, such as product graphs.

A more flexible way to exploit structure of a network is *boolean routing* introduced by Flammini *et al.* [FGS93]. Here node names are bitstrings, and node labels are predicates; a message with destination label λ can be sent over a link labeled with $\mathcal{L}$ iff $\mathcal{L}(\lambda)$ holds. The method allows fairly complicated network structures to be used efficiently, but of course there is considerable complexity in the interpretation of the labels.

4.4.3 Prefix Routing

To overcome the disadvantages of interval routing, Bakker, Van Leeuwen, and Tan [BLT93] designed a routing method for which the tables can be computed using arbitrary spanning trees. The use of non-restricted spanning trees can increase both the robustness and the efficiency. If a channel is added between two existing nodes, the spanning tree remains a spanning tree and the new channel is a frond. If a new node is added together with a number of channels connecting it to existing nodes, the spanning tree is extended using one of these channels and the new node, and the other channels are fronds. The optimality can be improved by choosing a small-depth spanning tree as in Lemma 4.22.

The node and channel labels used in prefix routing are strings rather than the integers used in interval routing. Let Σ be an alphabet; in the sequel a label will be a string over Σ, ϵ denotes the empty string, and Σ^* the set of strings over Σ. To select a channel in which to forward a packet, the

```
(* A packet with address d was received or generated at node u *)
if d = l_u
  then deliver the packet locally
  else begin α_i := the longest channel-label s.t. α_i ◁ d ;
             send packet via the channel labeled with α_i
       end
```

Algorithm 4.20 PREFIX FORWARDING (FOR NODE u).

algorithm considers all channel labels that are a *prefix* of the destination address. The longest of these labels is selected, and the corresponding channel is used to forward the packet. For example, assume a node has channels labeled with **aabb**, **abba**, **aab**, **aabc**, and **aa**, and must forward a packet with address **aabbc**. The channel labels **aabb**, **aab**, and **aa** are prefixes of **aabbc**, and the longest of these three labels is **aabb**, hence the node forwards the packet via the channel labeled **aabb**. The forwarding algorithm is given as Algorithm 4.20. We write $\alpha \triangleleft \beta$ to denote that α is a prefix of β.

Definition 4.38 *A prefix labeling scheme (over Σ) for a network G is*

(1) *an assignment of different strings from Σ^* to the nodes of G; and*
(2) *for each node, an assignment of different strings to the channels of that node.*

The prefix routing algorithm assumes that a prefix labeling scheme (PLS) is given, and forwards packets as in Algorithm 4.20.

Definition 4.39 *A prefix labeling scheme is valid if all packets forwarded in this way eventually reach their destination.*

Theorem 4.40 *For each connected network G a valid PLS exists.*

Proof. We shall define a class of prefix labeling schemes and prove, as in Theorem 4.25, that the schemes in this class are valid. Let T denote an arbitrary rooted spanning tree of G.

Definition 4.41 *A tree PLS for G (with respect to T) is a prefix labeling scheme in which the following rules are satisfied.*

(1) *The node label of the root is ϵ.*
(2) *If w is a son of u then l_w extends l_u by one letter; i.e., if $u_1, \ldots, u_k$ are the sons of u in T then $l_{u_i} = l_u.a_i$, where $a_1, \ldots, a_k$ are k different letters from Σ.*

(3) *If uw is a frond then $\alpha_{uw} = l_w$.*
(4) *If w is a son of u then $\alpha_{uw} = l_w$.*
(5) *If w is the father of u then $\alpha_{uw} = \epsilon$ unless u has a frond to the root; in that case, $\alpha_{uw} = l_w$.*

In a tree PLS each node except the root has a channel labeled ϵ, and this channel connects the node to an ancestor (the father of the node or the root of the tree). Observe that for every channel uw, $\alpha_{uw} = l_w$ or $\alpha_{uw} = \epsilon$. For all u and v, v is an ancestor of u if and only if $l_v \triangleleft l_u$.

It must first be shown that a packet never gets "stuck" in a node different from its destination, that is, each node different from the destination can forward the packet using Algorithm 4.20.

Lemma 4.42 *For all nodes u and v such that $u \neq v$ there is a channel in u labeled with a prefix of l_v.*

Proof. If u is not the root of T then u has a channel labeled ϵ, which is a prefix of l_v. If u is the root then v is not the root, and $v \in T[u]$ holds. If w is the son of u such that $v \in T[w]$ then by construction $\alpha_{uw} \triangleleft l_v$. □

The following three lemmas concern the situation where node u forwards a packet for node v to a node w (a neighbor of u) using Algorithm 4.20.

Lemma 4.43 *If $u \in T[v]$ then w is an ancestor of u.*

Proof. If $\alpha_{uw} = \epsilon$ then w is an ancestor of u as mentioned above. If $\alpha_{uw} = l_w$ then, since $\alpha_{uw} \triangleleft l_v$, also $l_w \triangleleft l_v$. This implies that w is an ancestor of v, and also of u. □

Lemma 4.44 *If u is an ancestor of v then w is an ancestor of v, closer to v than u.*

Proof. Let w' be the son of u such that $v \in T[w']$ then $\alpha_{uw'} = l_{w'}$ is a non-empty prefix of l_v. As α_{uw} is the longest prefix (in u) of l_v, it follows that $\alpha_{uw'} \triangleleft \alpha_{uw} \triangleleft l_v$, so w is an ancestor of v below u. □

Lemma 4.45 *If $u \notin T[v]$, then w is an ancestor of v or $d_T(w, v) < d_T(u, v)$.*

Proof. If $\alpha_{uw} = \epsilon$ then w is the father of u or the root; the father of u is closer to v than u because $u \notin T[v]$, and the root is an ancestor of v. If $\alpha_{uw} = l_w$ then, as $\alpha_{uw} \triangleleft l_v$, w is an ancestor of v. □

Let the value *depth* be the depth of T, i.e., the number of hops of the longest simple path from the root to any leaf. It can be seen that each packet with destination v arrives at its destination in at most $2 \cdot \mathit{depth}$ hops. If the

packet is generated in an ancestor of v then v is reached within *depth* hops by Lemma 4.44. If the packet is generated in the subtree $T[v]$ then an ancestor of v is reached within *depth* hops by Lemma 4.43, after which v is reached within another *depth* hops by the previous observation. (Because the path contains only ancestors of the source in this case, its length is actually bounded by *depth* also.) In all other cases an ancestor of v is reached within *depth* hops by Lemma 4.45, after which v is reached within another *depth* hops. (Thus, in this case the length of the path is bounded by $2 \cdot \mathit{depth}$.) This concludes the proof of Theorem 4.40 □

Corollary 4.46 *For each network G with diameter D_G (measured in hops) there exists a prefix labeling scheme that delivers all packets in at most $2D_G$ hops.*

Proof. Use a tree PLS with respect to a tree chosen as in Lemma 4.22. □

We conclude the discussion of the tree labeling scheme with a brief analysis of its space requirement. As before, let *depth* be the depth of T, and let k be the maximal number of sons of any node of T. Then the longest label consists of *depth* letters, and as Σ must contain (at least) k letters, a label can be stored in $\mathit{depth} \cdot \log k$ bits. The routing table of a node with *deg* channels is stored in $O(\mathit{deg} \cdot \mathit{depth} \cdot \log k)$ bits.

Several other prefix labeling schemes have been proposed by Bakker *et al.* [BLT93]. Their paper also characterizes the class of topologies that allows *optimal* prefix labeling schemes when the weight of links may change dynamically.

4.5 Hierarchical Routing

A way of reducing the various cost parameters of a routing method is the use of a *hierarchical division* of the network and an associated hierarchical routing method. The goal in most cases is to exploit the fact that much communication in computer networks is local, i.e., between nodes at relatively small distances from each other. Some of the cost parameters of a routing method depend on the size of the entire network rather than the length of the chosen path, as we now explain.

(1) *The length of addresses.* As each of the N nodes has a different address, each address consists of at least $\log N$ bits; more bits may even be required if information is encoded in addresses, such as in prefix routing.

(2) *The size of the routing table.* In the routing methods described in Sections 4.2 and 4.3, the table contains an entry for each node, and thus has a linear size.
(3) *The cost of table lookups.* The cost of a single table lookup is likely to be larger for a large routing table or for larger addresses. The total table-lookup time for the delivery of a single message also depends on the number of times the tables must be accessed.

In a hierarchical routing method, the network is divided into clusters, each cluster being a connected subset of nodes. If the source and destination of a packet are in the same cluster, the cost of forwarding the message is low, because to route within the cluster, the cluster is treated as a smaller isolated network. For the method described in Subsection 4.5.1, in each cluster there is a fixed single node (the cluster *center*) that can make the more complicated routing decisions necessary to send packets to other clusters. Thus, longer routing tables and the manipulation of long addresses are only necessary in the centers. Each cluster itself can be divided into subclusters in order to obtain a multi-level division of the nodes.

It is not necessarily desirable that each communication between clusters must take place via a cluster center; this type of design has the disadvantage that the entire cluster becomes vulnerable to failure of the center. Lentfert *et al.* [LUST89] describe a hierarchical routing method in which each node is equally able to send messages outside the cluster. Yet the method uses only small tables, because entire clusters to which a node does not belong are treated as a single node. Awerbuch *et al.* [ABNLP90] use the paradigm of hierarchical routing to construct a class of routing schemes that allows a trade-off between the efficiency and space requirements.

4.5.1 Reducing the Number of Routing Decisions

All the routing methods discussed so far require that routing decisions are made at each intermediate node, which means that for a route of length l the routing tables must be accessed l times. For minimum-hop strategies l is bounded by the diameter of the network, but for general, cycle-free routing strategies (such as interval routing) $N-1$ is the best bound one can give. In this subsection we shall discuss a method by which the number of table-lookups can be decreased.

We make use of the following lemma, which concerns the existence of a suitable division of the network into connected clusters.

Lemma 4.47 *For each $s \leq N$ there exists a division of the network into clusters $C_1, \ldots, C_m$ such that*

(1) *each cluster is a connected subgraph,*
(2) *each cluster contains at least s nodes, and*
(3) *each cluster has radius at most $2s$.*

Proof. Let $D_1, \ldots, D_m$ be a maximal collection of disjoint connected subgraphs such that each D_i has radius $\leq s$ and contains at least s nodes. Every node not in $\cup_{i=1}^{m} D_i$ is connected to one of the subsets by a path of length at most s, otherwise the path could be added as a separate cluster. Form the clusters C_i by including every node not in $\cup_{i=1}^{m} D_i$ in a cluster closest to it. The extended clusters still contain at least s nodes each, they are still connected and disjoint, and they have radius at most $2s$. □

The routing method assigns a color to each packet, and it is assumed that only a few colors are used. Nodes now act as follows. Depending on its color, a packet is either forwarded immediately over a fixed channel (corresponding to the color) or a more complex routing decision is called for. It is allowed that nodes have different protocols for handling packets.

Theorem 4.48 [LT86] *For every network of N nodes there is a routing method that requires at most $O(\sqrt{N})$ routing decisions for each packet, and uses three colors.*

Proof. Assume that a division as implied by Lemma 4.47 is given and observe that each C_i contains a node c_i such that $d(v, c_i) \leq 2s$ for each $v \in C_i$, because C_i has radius at most $2s$. Let T be a minimum-size subtree of G connecting all the c_i. Because T is minimal it contains at most m leaves, hence it contains at most $m-2$ branch points (nodes of degree larger than 2); see Exercise 4.9. We refer to the nodes of T as *centers* (the c_i), branch points, and path nodes (the remaining nodes).

The routing method first sends a packet to the center c_i of the cluster of its source node (green phase), then via T to the center c_j of the cluster of its destination node (blue phase), and finally within C_j to the destination itself (phase red). The green phase uses a fixed sink tree for the center of each cluster, and requires no routing decisions. The path nodes of T have two incident tree channels, and forward each blue packet via the tree channel through which they did not receive the packet. Branch points and centers in T must make routing decisions. For the red phase a shortest-path routing strategy within the cluster can be used, which bounds the number of decisions in this phase to $2s$. This bounds the number of routing decisions

to $2m-2+2s$, which is at most $2N/s-2+2s$. Choosing $s \approx \sqrt{N}$ gives the bound $O(\sqrt{N})$. □

Theorem 4.48 establishes a bound on the overall number of routing decisions necessary to deliver each packet, but does not rely on any particular algorithm by which these decisions are taken. The routing method used in T can be the tree routing scheme of Santoro and Khatib, but it is also possible to apply the principle of clustering to T itself to reduce the number of routing decisions even further.

Theorem 4.49 [LT86] *For every network of N nodes and every positive integer $f \leq \log N$ there is a routing method that requires at most $O(f \cdot N^{1/f})$ routing decisions for each packet, and uses $2f+1$ colors.*

Proof. The argument is similar to the proof of Theorem 4.48, but instead of choosing $s \approx \sqrt{N}$ the construction is applied recursively to the tree T (with the same cluster size s). The tree is a connected network, essentially of $< 2m$ nodes because the path nodes of T only pass on packets from one fixed channel to the other, and can be ignored.

The clustering is repeated f times. The network G has N nodes. The tree obtained after one level of clustering has at most N/s centers and N/s branch points, i.e., $N(2/s)$ essential nodes. If the tree obtained after i levels of clustering has m_i essential nodes, then the tree obtained after $i+1$ levels of clustering has at most m_i/s centers and m_i/s branch points, i.e., $m_i(2/s)$ essential nodes. The tree obtained after f levels of clustering has at most $m_f = N(2/s)^f$ essential nodes.

Each level of clustering increases the number of colors by two, hence with f levels of clustering $2f+1$ colors are used. At most $2m_f$ decisions are needed in the highest level, and s decisions are needed at each level of clustering in the destination cluster, which brings the number of routing decisions to $2m_f + fs$. Choosing $s \approx 2N^{1/f}$ gives $m_f = O(1)$, hence the number of routing decisions is bounded by $f \cdot s = O(f \cdot N^{1/f})$. □

The use of approximately $\log N$ colors leads to a routing method that requires $O(\log N)$ routing decisions. The inspection of the color of a packet also becomes a kind of routing decision in this case, but it involves small tables (of length $O(\log N)$ at most) and is actually required only in a small fraction of the nodes.

Exercises to Chapter 4

Section 4.1

Exercise 4.1 *Assume that routing tables are updated after each topological change in such a way that they are cycle-free even during updates. Does this guarantee that packets are always delivered even when the network is subject to a possibly infinite number of topological changes?*
Prove that no routing algorithm can guarantee delivery of packets under continuing topological changes.

Section 4.2

Exercise 4.2 *A student proposes to omit the sending of* $\langle\, \mathbf{nys}, w \,\rangle$ *messages from Algorithm 4.6; he argues that a node knows that a neighbor is* not *a son in* T_w *if no* $\langle\, \mathbf{ys}, w \,\rangle$ *message is received from that neighbor.*
Is it possible to modify the algorithm in this way? What happens to the complexity of the algorithm?

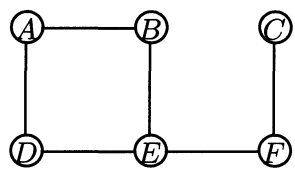
Exercise 4.3 *Prove that the following assertion is an invariant of the Chandy–Misra algorithm for computing paths towards* v_0 *(Algorithm 4.7).*

$$\begin{aligned} &\forall u,\, w : \langle\, \mathbf{mydist}, v_0, d \,\rangle \in M_{wu} \quad \Rightarrow \quad d(w,\, v_0) \leq d \\ \wedge\ &\forall u : d(u,\, v_0) \leq D_u[v_0] \end{aligned}$$

Give an example of an execution for which the number of messages is exponential in the number of channels of the network.

Section 4.3

Exercise 4.4 *Give the values of all variables in a terminal configuration of the Netchange algorithm when the algorithm is applied to a network of the following topology:*



After a terminal configuration has been reached, a channel between A *and* F *is added. What messages does* F *send to* A *when processing the* $\langle\, \mathbf{repair}, A \,\rangle$

notification? What messages does A send upon receipt of these messages from F?

Section 4.4

Exercise 4.5 *Give an example to demonstrate that Lemma 4.22 does not hold for networks with asymmetric channel cost.*

Exercise 4.6 *Does there exist an ILS that does not use all channels for routing? Does there exist a valid one? An optimal one?*

Exercise 4.7 *Give a graph G and a depth-first search tree T of G such that G has $N = n^2$ nodes, the diameter of G and the depth of T are $O(n)$, and there are nodes u and v such that a packet from u to v is delivered after $N - 1$ hops with the depth-first search ILS.*
(The graph can be chosen in such a way that G is outerplanar, which implies (by Theorem 4.37) that G actually has an optimal ILS.)

Exercise 4.8 *Give the depth-first search ILS for a ring of N nodes. Find nodes u and v such that $d(u, v) = 2$, and the scheme uses $N - 2$ hops to deliver a packet from u to v.*

Section 4.5

Exercise 4.9 *Prove that the minimality of the tree T in the proof of Theorem 4.48 implies that it has at most m leaves. Prove that any tree with m leaves has at most $m - 2$ branch points.*