Design and Analysis of Distributed Algorithms

Chapter 4:

ROUTING

©2005, Nicola Santoro.

Contents

1 Routing **1**

2 Shortest-Path Routing **1**

2.1 Gossiping the Network Maps . . . 2
2.2 Iterative Construction of Routing Tables . . . 4
2.3 Constructing Shortest-Path Spanning Tree . . . 6
2.3.1 Protocol Design . . . 7
2.3.2 Analysis . . . 10
2.3.3 Constructing All Routing Tables . . . 14
2.4 Min-Hop Routing . . . 15
2.4.1 Breadth-First Spanning Tree Construction . . . 16
2.4.2 Multiple Layers: An Improved Protocol . . . 18
2.4.3 Reducing Time with More Messages . . . 23
2.5 Suboptimal Solutions: Routing Trees . . . 25

3 Coping with Changes **28**

3.1 Adaptive Routing . . . 28
3.1.1 Map Update . . . 29
3.1.2 Vector Update . . . 29
3.1.3 Oscillation . . . 30
3.2 Fault-Tolerant Tables . . . 31
3.2.1 Point-of-failure Rerouting . . . 31
3.2.2 Point-of-failure Shortest-path Rerouting . . . 32
3.3 On Correctness and Guarantees . . . 35
3.3.1 Adaptive Routing . . . 35
3.3.2 Fault-Tolerant Tables . . . 36

4 Routing in Static Systems: Compact Tables **37**

4.1 The Size of Routing Tables . . . 37
4.2 Interval Routing . . . 38
4.2.1 An Example: Ring Networks . . . 38
4.2.2 Routing with Intervals . . . 40

5 Bibliographical Notes **42**

6 Exercises, Problems, and Answers **43**

6.1 Exercises 43

6.2 Problems 48

6.3 Answers to Exercises 49

1 Routing

Communication is at the basis of computing in a distributed environment but to achieve it efficiently is not a simple nor trivial task.

Consider an entity x that wants to communicate some information to another entity y; e.g., x has a message that it wants to be delivered to y. In general, x does not know where y is or how to reach it (i.e., which paths lead to it); actually, it might not even know if y is a neighbour or not.

Still, the communication is always possible if the network $\vec{G}$ is strongly connected. In fact, it is sufficient for x to *broadcast* the information: every entity, including y will receive it. This simple solution, called *broadcast routing*, is obviously not efficient; on the contrary, it is impractical, expensive in terms of cost, and not very secure (too many other nodes receive the message), even if its performed only on a spanning-tree of the network.

A more efficient approach is to choose a single path in $\vec{G}$ from x to y: the message sent by x will travel only along this path, relayed by the entities in the path, until it reaches its destination y. The process of determining a path between a *source* x and a *destination* y is known as *routing*.

If there is more than one path from x to y, we would like obviously to choose the "best" one, i.e., the least expensive one. The *cost* $\theta(a, b) \geq 0$ of a link (a, b), traditionally called *length*, is a value that depends on the system (reflecting, e.g., time delay, transmission cost, link reliability, etc), and the cost of a path is the sum of the costs of the links composing it. The path of minimum cost is called *shortest path*; clearly the objective is to use this path for sending the message. The process of determining the most economic path between a source and a destination is known as *shortest-path routing*.

The (shortest-path) routing problem is commonly solved by storing at each entity x information that will allow to address a message to its destination though a (shortest) path. This information is called *routing table*.

In this Chapter we will discuss several aspects of the routing problem. First of all, we will consider the construction of the routing tables. We will then address the problem of maintaining the information of the tables up-to-date should changes occurr in the system. Finally we will discuss how to represent routing information in a compact way, suitable for systems where space is a problem. In the following, and unless otherwise specified, we will assume the standard set of restrictions **IR**: *Bidirectional Links* (BL), *Connectivity* (CN), *Total Reliability* (TR), and *Initial Distinct Values* (ID).

2 Shortest-Path Routing

The *routing table* of an entity contains information on how to reach any possible destination. In this section we examine how this information can be acquired, and the table constructed. As we will see, this problem is related to the construction of particular spanning-trees of the

Routing Destination	**Shortest Path**	**Cost**
h	(s,h)	1
k	$(s,h)(h,k)$	4
c	(s,c)	10
d	$(s,c)(c,d)$	12
e	(s,e)	5
f	$(s,e)(e,f)$	8

Table 1: Full routing table for node s

network. In the following, and unless otherwise specified, we will focus on shortest-paths routing.

Different types of routing tables can be defined, depending on the amount of information contained in them. We will consider for now the *full* routing table: for each destination, there is stored a shortest path to reach it; if there is more than one shortest path, only the lexicographically smallest[1] will be stored. For example, in the network of Figure 1, the routing table $RT(s)$ for s is shown in Table 1.

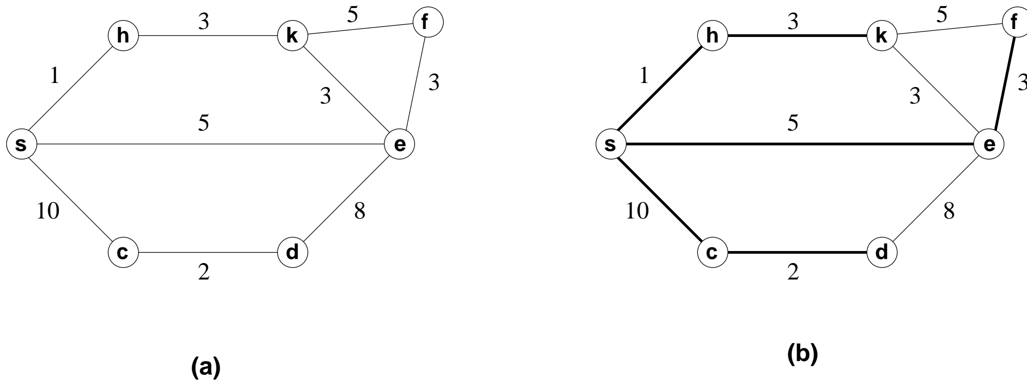

Figure 1: Determining the shortest paths from s to the other entities.

We will see different approaches to construct routing tables, some depending on the amount of local storage an entity has available.

2.1 Gossiping the Network Maps

A first obvious solution would be to construct at every entity the entire *map* of the network with all the costs; then, each entity can locally and directly compute its shortest-path routing

[1] The lexicographic order will be over the strings of the names of the nodes in the paths.

table. This solution obviously requires that the local memory available to an entity is large enough to store the entire map of the network.

The map of the network can be viewed as a $n \times n$ array $MAP(G)$, one row and one column per entity, where for any two entities x and y, the entry $MAP[x, y]$ contains information on whether link (x, y) exists, and if so on its cost. In a sense, each entity x knows initially only its own row $MAP[x, \star]$. To know the entire map, every entity needs to know the initial information of all the other entities.

This is a particular instance of a general problem called *input collection* or *gossip*: every entity has a (possibly different) piece of information; the goal is to reach a final configuration where every entity has all the pieces of information. The solution of the gossiping problem using normal messages is simple:

every entity broadcasts its initial information.

Since it relies solely on broadcast, this operation is more efficiently performed in a tree. Thus, the protocol will be as follows:

Map_Gossip:

1. An arbitrary spanning-tree of the network is created, if not already available; this tree will be used for all communication.
2. Each entity acquires full information about its neighbourhood (e.g., names of the neighbours, cost of the incident links, etc.), if not already available.
3. Each entity broadcasts its neighbourhood information along the tree.

At the end of the execution, each entity has a complete map of the network with all the link costs; it can then locally construct its shortest-path routing table.

The construction of the initial spanning-tree can be done using $O(m + n \log n)$ messages, e.g. using protocol *MegaMerger*. The acquisition of neighbourhood information requires a single exchange of messages between neighbours, requiring in total just $2m$ messages. Each entity x then broadcasts on the tree $deg(x)$ items of information. Hence the total number of messages will be at most

$$\sum_x deg(x)(n-1) \; = \; 2m(n-1)$$

Thus, we have

$$\mathbf{M}[Map_Gossip] = \; 2\; m\; n + \; l.o.t. \tag{1}$$

This means that, in *sparse* networks, all the routing tables can be constructed with at most $O(n^2)$ normal messages. Such is the case of meshes, tori, butterflies, etc.

Routing Destination	**Shortest Path**	**Cost**
h	(s, h)	1
k	?	∞
c	(s, c)	10
d	?	∞
e	(s, e)	5
f	?	∞

Table 2: Initial approximation of $RT(s)$

In systems that allow *very long messages*, not-surprisingly the gossip problem, and thus the routing table construction problem, can be solved with substantially fewer messages (Exercises 6.3 and 6.4).

The time costs of gossiping on a tree depend on many factors, including the diameter of the tree and the number of initial items an entity initially has (Exercise 6.2).

2.2 Iterative Construction of Routing Tables

The solution we have just seen requires that each entity has locally available enough storage to store the entire map of the network. If this is not the case, the problem of constructing the routing tables is more difficult to resolve.

Several traditional *sequential* methods are based on an *iterative* approach. Initially, each entity x knows only its neighbouring information: for each neighbour y, the entity knows the cost $\theta(x, y)$ of reaching it using the direct link (x, y). Based on this initial information, x can construct an *approximation* of its routing table. This imperfect table is usually called *distance vector*, and in it the cost for those destinations x knows nothing about will be set to ∞. For example, the initial distance vector for node s in the network of Figure 1 is shown in Table 2.

This approximation of the routing table will be refined, and eventually corrected, through a sequence of iterations. In each iteration, every entity communicates its current distance vector with all its neighbours. Based on the received information, each entity updates its current information, replacing paths in its own routing table if the neighbours have found better routes.

How can an entity x determine if a route is better ? The answer is very simple: when, in an iteration, x is told by a neighbour y that there exists a path π_2 from y to z with cost g_2, x checks in its current table the path π_1 to z and its cost g_1, as well as the cost $\theta(x, y)$. If $\theta(x, y) + g_2 < g_1$, then going directly to y and then using π_2 to reach z is less expensive than going to z through the path π_1 currently in the table. Among several better choices, obviously x will select the best one.

Specifically: let $V_y^i[z]$ denote the cost of the "best" path from y to z known to y in iteration

	s	h	k	c	d	e	f
s	-	1	∞	10	∞	5	∞
h	1	-	3	∞	∞	∞	∞
k	∞	3	-	∞	∞	3	5
c	10	∞	∞	-	2	∞	∞
d	∞	∞	∞	2	-	8	∞
e	5	∞	3	∞	8	-	3
f	∞	∞	5	∞	∞	3	-

Table 3: Initial distance vectors.

	s	h	k	c	d	e	f
s	-	1	4	10	12	5	8
h	1	-	3	11	∞	6	8
k	4	3	-	∞	11	3	5
c	10	11	∞	-	2	10	∞
d	12	∞	11	2	-	8	11
e	5	6	3	10	8	-	3
f	8	8	5	∞	11	3	-

Table 4: Distance vectors after 1st iteration.

i; this information is contained in the distance vector sent by y to all its neighbours at the beginning of iteration $i+1$. After sending its own distance vector and upon receiving the distance vectors of all its neighbours, entity x computes

$$w[z] = Min_{y \in N(x)}(\theta(x,y) + V_y^i[z])$$

for each destination z. If $w[z] < V_x^i[z]$ then the new cost and the corresponding path to z is chosen, replacing the current selection.

Why should interaction just with the neighbours be sufficient follows from the fact that the cost $\gamma_a(b)$ of the shortest-path from a to b has the following defining property:

	s	h	k	c	d	e	f
s	-	1	4	10	12	5	8
h	1	-	3	11	13	6	8
k	4	3	-	13	11	3	5
c	10	11	13	-	2	10	13
d	12	13	11	2	-	8	11
e	5	6	3	10	8	-	3
f	8	8	5	13	11	3	-

Table 5: Distance vectors after 2nd iteration.

Property 2.1 $\gamma_a(b) = \begin{cases} 0 & if \quad a = b \\ Min_{w \in N(a)} \; \{ \; \theta(a,w) + \gamma_w(b) \; \} & otherwise. \end{cases}$

The Protocol *Iterated_Construction* based on this strategy converges to the correct information, and will do so after at most $n-1$ iterations (Exercise 6.8). For example, in the graph of Figure 1, the process converges to the correct routing tables after only two iterations; see Tables 3-5: for each entity, only the cost information for every destination is displayed.

The main advantage of this process is that the amount of *storage* required at an entity is proportional to the size of the routing table and *not* to the map of the entire system.

Let us analyze the message and time costs of the associated protocol.

In each iteration, an entity sends its distance vector containing costs and path information; actually, it is not necessary to send the entire path but only the first hop in it (see discussion in Section 4). In other words, in each iteration, an entity x needs to send n items of information to its $deg(x)$ neighbours. Thus, in total, an iteration requires $2nm$ messages. Since this process terminates after at most $n-1$ iterations, we have

$$\mathbf{M}[Iterated_Construction] = 2 \; (n-1) \; n \; m \tag{2}$$

That is, this approach is more expensive than the one based on constructing all the maps; it does however require less local storage.

As for the time complexity, let $\tau(n)$ denote the amount of ideal time required to transmit n items of information to the same neighbour; then

$$\mathbf{T}[Iterated_Construction] = (n-1) \; \tau(n) \tag{3}$$

Clearly, if the system allows *very long messages*, the protocol can be executed with fewer messages. In particular, if messages containing $O(n)$ items of information (instead of $O(1)$) are possible, then in each iteration an entity can transmit its entire distance vector to a neighbour with just *one* message and $\tau(n) = 1$. The entire process can thus be accomplished with $O(n \; m)$ messages and the time complexity would then be just $n-1$.

2.3 Constructing Shortest-Path Spanning Tree

The first solution we have seen, protocol *Map_Gossip*, requires that each entity has locally available enough storage to store the entire map of the network. The second solution, protocol *Iterative_Construction*, avoids this problem but it does so at the expenses of a substantially increased amount of messages.

Our goal is to design a protocol that, without increasing the local storage requirements constructs the routing tables with a smaller amount of communication. Fortunately, there is an important property that will help us in achieving this goal.

Consider the paths contained in the full routing table $RT(s)$ of an entity s, e.g., the ones in Table 1. These paths define a sub-graph of the network (since not every link is included).

This sub-graph is special: it is connected, contains all the nodes, and does not have cycles (see Figure 1 where the subgraph links are in bold); in other words,

it is a spanning tree !

It is called the *shortest-path spanning-tree rooted in* s ($PT(s)$), sometimes known also as the *sink tree* of s.

This fact is important because it tell us that, to construct the routing table $RT(s)$ of s, we just need to construct the shortest-path spanning-tree $PT(s)$.

2.3.1 Protocol Design

To construct the shortest-path spanning-tree $PT(s)$, we can adapt a classical *serial* strategy for constructing $PT(s)$ starting from the source s:

Serial Strategy

- We are given a connected fragment T of $PT(s)$, containing s (initially, T will be composed of just s).
- Consider now all the links going outside of T (i.e., to nodes not yet in T). To each such link (x, y) associate the value $v(x, y) = \gamma_s(x) + \theta(x, y)$; i.e., $v(x, y)$ is the cost of reaching y from the source s by first going to x (through a shortest path) and then using the link (x, y) to reach y.
- Add to T the link (a, b) for which $v(a, b)$ is minimum; in case of a tie, choose the one leading to the node with the lexicographically smallest name.

The reason this strategy works is because of the following property:

Property 2.2 *Let T and (a, b) be as defined in the Serial Strategy. Then $T \cup (a, b)$ is a connected fragment T of $PT(s)$.*

That is, the new tree, obtained by adding the chosen (a, b) to T, is also a connected fragment of $PT(s)$, containing s; and it is clearly larger than T. In other words, using this strategy, the shortest-path spanning-tree $PT(s)$ will be constructed, starting from s, by adding the appropriate links, one at the time.

The algorithm based on this strategy will be a sequence of *iterations* started from the root. In each iteration, the outgoing link (a, b) with minimum cost $v(a, b)$ is chosen; the link (a, b) and the node b are added to the fragment, and a new iteration is started. The process terminates when the fragment includes all the nodes.

Our goal is now to implement this algorithm efficiently in a *distributed* way.

First of all, let us consider what a node y in the fragment T knows. Definitely y knows which of its links are part of the current fragment; it also knows the length $\gamma_s(y)$ of the shortest path from the source s to it.

IMPORTANT. Let us assume for the moment that y also knows which of its links are *outgoing* (i.e., lead to nodes outside of the current fragment), and which are *internal*.

In this case, to find the outgoing link (a, b) with minimum cost $v(a, b)$ is rather simple, and the entire iteration is composed of four easy steps:

Iteration

1. The root s broadcasts in T the start of the new iteration.
2. Upon receiving the start, each entity x in the current fragment T computes locally $v(x, y) = \gamma_s(x) + \theta(x, y)$ for each of its outgoing incident links (x, y); it then selects among them the link $e = (x, y')$ for which $v(x, y')$ is minimized.
3. The overall minimum $v(a, b)$ among all the locally selected $v(e)$'s is computed at s, using a minimum-finding for (rooted) trees (e.g., see Sect. **??**), and the corresponding link (a, b) is chosen as the one to be added to the fragment.
4. The root s notifies b of the selection; the link (a, b) is added to the spanning-tree; b computes $\gamma_s(b)$, and s is notified of the end of the iteration.

Each iteration can be performed efficiently, in $O(n)$ messages, since each operation (broadcast, min-finding, notifications) are performed on a tree of at most n nodes.

There are a couple of problems that need to be addressed. A small problem is how can b compute $\gamma_s(b)$. This value is actually determined at s by the algorithm in this iteration; hence, s can communicate it to b when notifying it of its selection.

A more difficult problem regards the knowledge of which links are *outgoing* (i.e., they lead to nodes outside of the current fragment); we have assumed that an entity in T has such a knowledge about its links. But how can such a knowledge be ensured ?

As described, during an iteration, messages are sent only on the links of T and on the link selected in that iteration. This means that the outgoing links are all *unexplored* (i.e., no message has been sent or received on them). Since we do not know which are outgoing, an entity could perform the computation of Step 2 for each of its *unexplored* incident links and select the minimum among those. Consider for example the graph of Figure 2(a) and assume that we have already constructed the fragment shown in Figure 2(b). There are four unexplored links incident to the fragment (shown as leading to square boxes), each with its value (shown in the corresponding square box); the link (s, e) among them has minimum value and is chosen; it is outgoing and it is added to the segment. The new segment is shown in Figure 2(c) together with the unexplored links incident on it.

However, not all unexplored links are outgoing: an unexplored link might be *internal* (i.e., leading to a node already in the fragment), and selecting such a link would be an error. For

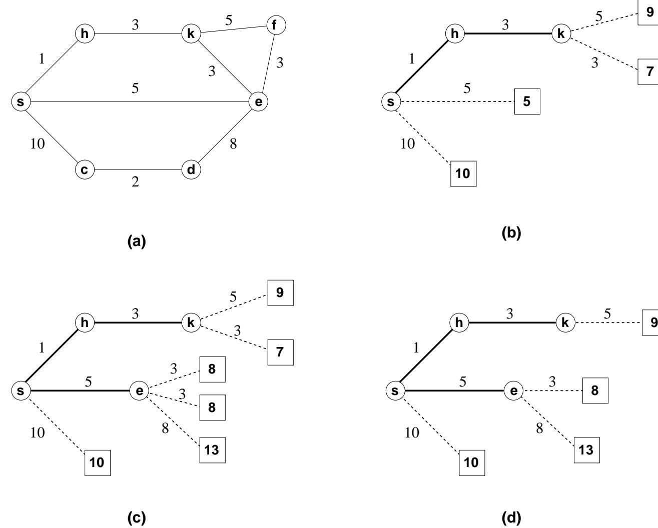

Figure 2: Determining the next link to be added to the fragment.

example, in Figure 2(c), the unexplored link (e, k) has value $v(e, k) = 7$ which is minimum among the unexplored edges incident on the fragment, and hence would be chosen; however, node e is already in the fragment.

We could allow for errors: we choose among the unexplored links and, if the link (in our example: (e, k)) selected by the root s in step 3 turns out to be internal (k would find out in step 4 when the notification arrives), we eliminate that link from consideration and select another one. The drawback of this approach is its overall cost. In fact, since initially all links are unexplored, we might have to perform the entire selection process for every link. This means that the cost will be $O(nm)$, which in the worst case is $O(n^3)$: a high price to construct a single routing table.

A more efficient approach is to add a mechanism so no error will occur. Fortunately, this can be achieved simply and efficiently as follows.

When a node b becomes part of the tree, it sends a message to all its neighbours notifying them that it is now part of the tree. Upon receiving such a message, a neighbour c knows that this link must no longer be used when performing shortest path calculations for the tree. As a side effect, in our example, when the link (s, e) is chosen in Figure 2(b), node e knows already that the link (e, k) leads to a node already in the fragment; thus such a link is thus not considered, as shown in Figure 2(d).

RECALL. We have used a similar strategy with the protocol for Depth First Traversal, to decrease its time complexity.

IMPORTANT. It is necessary for b to ensure that all its neighbours have received its message before a new iteration is started. Otherwise, due to time delays, a neighbour c

might receive the request to compute the minimum for the next iteration *before* the message from b has even arrived; thus, it is possible that c (not knowing yet that b is part of the tree) chooses its link to b as its minimum, and such a choice is selected as the overall minimum by the root s. In other words, it is still possible that an internal link is selected during an iteration.

Summarizing, to avoid mistakes, it is sufficient to modify rule 4 as follows:

- 4'. The root s sends an *Expand* message to b and the link (a, b) is added to the spanning-tree; b computes $\gamma_s(b)$, sends a notification to its neighbours, waits for their acknowledgment, and then notifies s of the end of the iteration.

This ensures that there will be only $n-1$ iterations, each adding a new node to the spanning-tree, with a total cost of $O(n^2)$ messages. Clearly we must also consider the cost of each node notifying its neighbours (and them sending acknowledgments), but this adds only $O(m)$ messages in total.

The protocol, called *PT_Construction*, is shown in Figures 3-6.

2.3.2 Analysis

Let us now analyze the cost of protocol *PT_Construction* in details. There are two basic activities being performed: the expansion of the current fragment of the tree, and the announcement (with acknowledgments) of the addition of the new node to the fragment.

Let us consider the expansion first. It consists of a "start-up" (the root broadcasting the *Start_Iteration* message), a "convergecast" (the minimum value is collected at the root using the *MinValue* messages), two "notifications" (the root notifies the new node using the *Expansion* message, and the new node notifies the root using the *Iteration_Completed* message). Each of these operations are performed on the current fragment which is a tree, rooted in the source. In particular, the start-up and the convergecast operations each cost only one message on every link; in the notifications, messages are sent only on the links in path from the source to the new node, and there will be only one message in each direction. Thus, in total, on each link of the tree constructed so far, there will be at most four messages due to the expansion; two messages will also be sent on the new link added in this expansion. Thus in the expansion at iteration i, at most $4(n_i - 1) + 2$ messages will be sent, where n_i is the size of the current tree. Since the tree is expanded by one node at the time, $n_i = i$: initially there is only the source; then the fragment is composed of the source and a neighbour, and so on. Thus, the total number of messages due to the expansion is

$$\sum_{i=1}^{n-1}(4(n_i - 1) + 2) = \sum_{i=1}^{n-1}(4i - 2) = 2n(n-1) - 2(n-1) = 2n^2 - 4n + 2$$

The cost due to announcements and acknowledgments is simple to calculate: each node will send a *Notify* message to all its neighbours when it becomes part of the tree, and receives an *Ack* from each of them. Thus, the total number of messages due to the notifications is

$$2\sum_{x \in V} |N(x)| = 2\sum_{x \in V} deg(x) = 4m$$

PROTOCOL PT_Construction.

- States: $\mathcal{S}$ = { INITIATOR, IDLE, AWAKE, ACTIVE, WAITING_FOR_ACK, COMPUTING, DONE };
  $\mathcal{S}_{INIT}$ = { INITIATOR,IDLE }; $\mathcal{S}_{TERM}$ = { DONE }.
- Restrictions: **IR** ; UI.

```
INITIATOR
          Spontaneously
          begin
               source:= true;
               my_distance:= 0;
               ackcount:= |N(x)|;
               send(Notify) to N(x);
          end

          Receiving(Ack)
          begin
               ackcount:= ackcount - 1;
               if ackcount = 0 then
                  iteration:= 1;
                  v(x,y) := MIN{v(x,z) : z ∈ N(x)};
                  path_length:= v(x,y);
                  Children:={y};
                  send(Expand, iteration, path_length) to y;
                  Unvisited:= N(x) − {y};
                  become ACTIVE;
               endif
          end


IDLE
          Receiving(Notify)
          begin
               Unvisited:= N(x) − {sender};
               send(Ack) to sender;
               become AWAKE;
          end


AWAKE
          Receiving(Expand, iteration, path_value)
          begin
               my_distance:= path_value;
               parent:= sender;
               Children:= ∅;
               if |N(x)| > 1 then
                  send(Notify) to N(x) − {sender};
                  ackcounter:= |N(x)| − 1;
                  become WAITING_FOR_ACK;
               else
                  send(IterationCompleted) to parent;
                  become ACTIVE;
               endif
          end
```

Figure 3: Protocol *PT-Construction* 1 ...

```
AWAKE
        Receiving(Notify)
        begin
             Unvisited:= Unvisited−{sender};
             send(Ack) to sender;
        end

WAITING_FOR_ACK
           Receiving(Ack)
           begin
                ackcount:= ackcount - 1;
                if ackcount = 0 then
                   send(IterationCompleted) to parent;
                   become ACTIVE;
                endif
           end


ACTIVE
           Receiving(Iteration_Completed)
           begin
                if not(source) then
                   send(Iteration_Completed) to parent;
                else
                   iteration:= iteration + 1;
                   send(Start_Iteration, iteration) to children;
                   Compute_Local_Minimum;
                   childcount:= 0;
                   become COMPUTING;
                endif
           end

           Receiving(Start_Iteration, counter)
           begin
                iteration:= counter;
                Compute_Local_Minimum;
                if children = ∅ then
                   send(MinValue, minpath) to parent;
                else
                   send(Start_Iteration, iteration) to children;
                   childcount:=0;
                   become COMPUTING;
                endif
           end
```

Figure 4: Protocol *PT-Construction* 2...

```
ACTIVE
        Receiving(Expand, iteration, path_value)
        begin
             send(Expand, iteration, path_value) to exit;
             if exit = mychoice then
               Children := Children ∪ {mychoice};
               Unvisited := Unvisited − {mychoice};
             endif
        end

        Receiving(Notify)
        begin
             Unvisited:= Unvisited −{sender};
             send(Ack) to sender;
        end

        Receiving(Terminate)
        begin
             send(Terminate) to children;
             become DONE;
        end


COMPUTING
        Receiving(MinValue, path_value)
        begin
             if path_value < minpath then
               minpath:= path_value;
               exit:= sender;
             endif
             childcount :=childcount + 1;
             if childcount = |Children| then
               if not(source) then
                 send(MinValue, minpath) to parent;
                 become ACTIVE;
               else
                 Check_for_Termination;
               endif
             endif
        end
```

Figure 5: Protocol *PT_Construction* 3

```
Procedure Check_for_Termination
begin
     if minpath= inf then
        send(Terminate) to Children;
        become DONE;
     else
        send(Expand, iteration, minpath) to exit;
        become ACTIVE;
     endif
end


Procedure Compute_Local_Minimum
begin
     if Unvisited = ∅ then
        minpath:= inf;
     else
        link_length:= v(x,y) = MIN{v(x,z) : z ∈ Unvisited};
        minpath:= my_distance + link_length;
        mychoice:= exit:= y;
     endif
end
```

Figure 6: Procedures used by protocol *PT_Construction*

To complete the analysis, we need to consider the final broadcast of the *Termination* message which is performed on the constructed tree; this will add $n-1$ messages to the total, yielding the following:

$$\mathbf{M}[PT_Construction] \leq 2n^2 - 4n + 2 + 4m + n - 1 = 2n^2 + 4m - 3n + 1 \qquad (4)$$

By adding a little bookkeeping, the protocol can be used to construct the routing table $RT(s)$ of the source (Exercise 6.13). Hence, we have a protocol which constructs the routing table of a node using $O(n^2)$ messages.

We will see later how more efficient solutions can be derived for the special case when all the links have the same cost (or, alternatively, there is no cost on the links).

Note that we have made no assumptions other that the costs are not-negative; in particular, we did *not* assume FIFO channels (i.e., message ordering).

2.3.3 Constructing All Routing Tables

Protocol *PT_Construction* allows us to construct the shortest-path tree of a node, and thus to construct the routing table of that entity. To solve the original problem of constructing all the routing table, also known as *all-pairs shortest-paths construction*, this process must be repeated for all nodes. The complexity of resulting protocol *PT_All* follows immediately from equation 4:

Algorithm	**Cost**	restrictions
Map_Gossip	$O(n\ m)$	$\Omega(m)$ local storage
Iterative_Construction	$O(n^2\ m)$	
PT	$O(n^3)$	

Table 6: Summary: Costs of constructing all shortest-path routing tables.

$$\mathbf{M}[PT_All] \leq 2n^3 \ - 3n^2 + 4(m-1)n \tag{5}$$

It is clear that some information computed when constructing $PT(x)$ can be re-used in the construction of $PT(y)$. For example, the shortest path from x to y is just the reverse of the one from y to x (under the bidirectional links assumption we are using); hence we just need to determine one of them. Even stronger is the so-called *optimality principle*:

Property 2.3 *If a node x is in the shortest path π from a to b, then π is also a fragment of $PT(x)$*

Hence, once a shortest path π has been computed for the shortest path tree of an entity, this path can be added to the shortest path tree of all the entities in the path. So, in the example of Figure 1, the path $(s,e)(e,f)$ in $PT(s)$ will also be part of $PT(e)$ and $PT(f)$. However, to date, it is not clear how this fact can be used to derive a more efficient protocol for constructing all the routing tables.

A summary of the results we have discussed is shown in Table 6

In systems that allow *very long messages*, not-surprisingly the problem can be solved with fewer messages. For example, if messages can contain $O(n)$ items of information (instead of $O(1)$), *all* the shortest-path trees can be constructed with just $O(n^2)$ messages (Exercise 6.15). If messages can contain $O(n^2)$ items then, as discussed in Section **??**, *any graph problem* including the construction of all shortest-path trees can be solved using $O(n)$ messages once a leader has been elected (requiring at least $O(m + mlogn)$ normal messages).

2.4 Min-Hop Routing

Consider the case when all links have the same cost (or alternatively, there are no costs associated to the links); that is, $\theta(a,b) = \theta$ for all $(a,b) \in E$.

This case is special in several respects. In particular, observe that the shortest path from a to b will have cost $\gamma_a(b) = \theta\ d_G(a,b)$, where $d_G(a,b)$ is the distance (in number of hops) of a from b in G; in other words, the cost of a path will depend solely on the number of hops (i.e., the number of links) in that path. Hence, the shortest-path between two nodes will be the one of minimum hops. For these reasons, routing in this situation is called *min-hop routing.*

An interesting consequence is that the shortest-path spanning-tree of a node coincides with its *breadth-first spanning-tree*! In other words, a breadth-first spanning-tree rooted in a node is the shortest-path spanning-tree of that node when all links have the same cost.

Protocol *PT_Construction* works for any choice of the costs, provided they are non-negative, so it constructs a breadth-first spanning-tree if all the costs are the same. However, we can take advantage of the fact that all links have the same costs to obtain a more efficient protocol. Let us see how.

2.4.1 Breadth-First Spanning Tree Construction

Without any loss of generality, let us assume that $\theta = 1$; thus, $\gamma_s(a) = d_G(s, a)$.

We can use the same strategy of protocol *PT_Construction* of starting from s and successively expanding the fragment; only, instead of choosing one link (and thus one node) at the time, we can choose several simultaneously: in the first step, s chooses all the nodes at distance 1 (its neighbours); in the second step, s chooses simultaneously all the nodes at distance 2; in general, in step i, s chooses simultaneously all the nodes at distance i; notice that, before step i, none of the nodes at distance i were part of the fragment. Clearly, the problem is to determine, in step i, which nodes are at distance i from s.

Observe this very interesting property: all the neighbours of s are at distance 1 from s; all their neighbours (not at distance 1 from s) are at distance 2 from s; in general

Property 2.4 *If a node is at distance i from s, then its neighbours are at distance either $i-1$ or i or $i+1$ from s.*

This means that, once the nodes at distance i from s have been chosen (and become part of the fragment), we need to consider only their neighbours to determine which nodes are at distance $i + 1$.

So the protocol, which we shall call *BF*, is rather simple. Initially, the root s sends a “Start iteration 1”message to each neighbor indicating the first iteration of the algorithm, and considers them its children. Each recipient marks its distance as 1, marks the sender as its *parent*, and sends an acknowledgment back to the parent. The tree is now composed of the root s and its neighbours, which are all at distance 1 from s.

In general, after iteration i all the nodes at distance up to i are part of the tree. Furthermore, each node at distance i knows which of its neighbours are at distance $i - 1$ (Exercise 6.16).

In iteration $i + 1$, the root *broadcasts* on the current tree a “Start iteration $i + 1$” message. Once this message reaches a node x at distance i, it will send a ”Explore $i + 1$” message to its neighbours that are *not* at distance $i - 1$ (recall, x knows which they are), and waits for a reply from each of them. These neighbours are either at distance i like x itself, or $i + 1$; those at distance i are already in the tree so do not need to be included. Those at distance $i+1$ must be attached to the tree; however, each must be attached only once (otherwise we create a cycle and do not form a tree). See Figure 7.
When a neighbour y receives the ”Explore” message, the content of its reply will depend on whether or not y is already part of the tree. If y is not part of the tree, it now knows that it is at distance $i + 1$ from s; it then marks the sender as its *parent*, sends a positive acknowledgment to it, and becomes part of the tree. If y is part of the tree (even if it just

happened in this iteration), it will reply with a negative acknowledgment.
When x receives the reply from y: if the reply is positive, it will mark y as a *child*; otherwise, it will mark y as already in the tree. Once all the replies have been received, it participates in a *convergecast* notifying the root that the iteration has been completed.

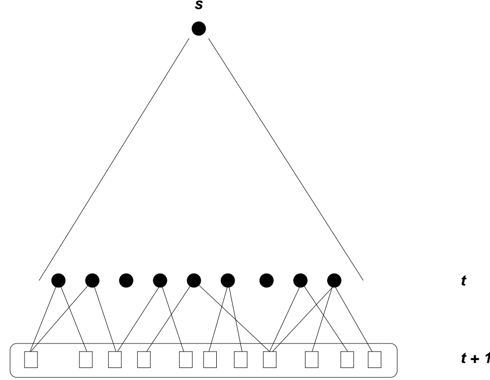

Figure 7: Protocol *BH* expands an entire level in each iteration.

Cost
Let us now examine the cost of protocol *BF*. Denote by n_i the number of nodes at distance at most i from s. In each iteration, there are three operations involving communication: (1) the broadcast of "Start"on the tree constructed so far; (2) the sending of "Explore" messages sent by the nodes at distance i, and the corresponding replies; and (3) the convergecast to notify the root of the termination of the iteration.

Consider first the cost of operation (2), that is the cost of the "Explore" messages and the corresponding replies. Consider a node x at distance i. As already mentioned, its neighbours are at distance either $i-1$ or i or $i+1$. The neighbours at distance $i-1$ sent an "Explore" message to x in stage $i-1$, so x sent a reply to each of them. In stage i x sent an "Explore" message to all its other neighbours. Hence, in total, x sent just one message (either "Explore" or reply) to each of its neighbours. This means that, in total, the number of "Explore" and "Reply" messages is

$$\sum_{x\in V} \mid N(x)\| = 2m$$

We will consider now the overall cost of operations (1) and (3). In iteration $i+1$, both the broadcast and convergecast are performed on the tree constructed in iteration i, thus costing $n_i - 1$ messages each, for a total of $2n_i - 2$ messages. Therefore, the total cost will be

$$\sum_{1\leq i<r(s)} 2(n_i - 1)$$

where $r(s)$ denotes the eccentricity of s (i.e., the hight of the breadth-first spanning tree of s).

Summarizing

$$\mathbf{M}[BF] \leq 2m + \sum_{1 \leq i < r(s)} 2(n_i - 1) \leq \; 2m \; + \; 2(n-1) \; d(G) \tag{6}$$

where $d(G)$ is the diameter of the graph. We know that $n_i < n_{i+1}$ and that $n_{r(s)} = n$ in any network G and for any root s, but the actual values depend on the nature of G and on the position of s. For example, in the *complete graph*, $r(s) = 1$ for any s, so the entire construction is completed in the first iteration; however, $m = n(n-1)/2$; hence the cost will be

$$n(n-1) + 2(n-1) = n^2 + n - 2$$

On the other hand, if G is a *line* and s is an endpoint of the line, $r(s) = n - 1$ and in each iteration we only add one node (i.e., $n_i = i$); thus $\sum_{1 \leq i < r(s)} 2(n_i - 1) = n^2 - 4n + 3$; however $m = n - 1$ hence the cost will be

$$2(n-1) + n^2 - 4n + 3 = n^2 - 2n + 1$$

As for the time complexity, in iteration i, the "Start" messages travel from the root s to the nodes at distance $i - 1$, hence arriving there after $i - 1$ time units; therefore, the nodes at distance i will receive the "Explore i" message after i time units. At that time, they will start the convergecast to notify the root of the termination of the iteration; this process requires exactly i time units. In other words, iteration i will cost exactly $2i$ time units. Summarizing

$$\mathbf{T}[BF] = 2 \sum_{1 \leq i \leq r(s)} i = r(s)(r(s) + 1) \leq d(G)^2 + d(G) \tag{7}$$

2.4.2 Multiple Layers: An Improved Protocol

To improve the costs, we must understand the structure of protocol *BF*. We know that its execution of protocol *BF* is a sequence of iterations, started by the root.

Each iteration $i + 1$ of protocol *BF* can be thought of as composed of three different phases:

1. *Initialization*: the root node broadcasts the "Start iteration $i + 1$" along the already constructed tree, which will reach the leaves (i.e., the nodes at distance i from the root).

2. *Expansion*: in this phase, which is started by the leaves, new nodes (i.e., all those of level $i + 1$) are added to the tree forming a larger fragment.

3. *Termination*: the root is notified of the end of this iteration using a convergecast on the new tree.

Initialization and termination are bookkeeping operations that allow the root to somehow synchronize the execution of the algorithm, iteration by iteration. For this reason, the two of them, together, are also called *synchronization*. Each synchronization costs $O(n)$ messages (since it is done on a tree). Hence, this activity alone costs

$$O(nL)$$

messages where L is the number of iterations.

In the original protocol *BF*, we expand the tree one level at the time; hence $L = deg(G)$ and the total cost for synchronization alone is $O(n\ deg(G))$ messages (see expression 6). This means that, to reduce the cost of synchronization, we need to decrease the number of iterations. To do so, we need each iteration to grow the current tree by more than a single level; that is, we need each expansion phase to add several levels to the current fragment.

Let us see how to expand the current tree by $l \geq 1$ levels, in a single iteration, efficiently (see Figure 8). Assume that initially each node $x \neq r$ has a variable $level_x = \infty$, while $level_r = 0$.

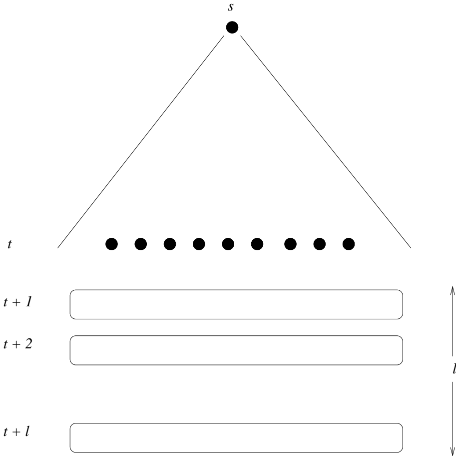

Figure 8: Protocol *BF_Levels* expands l levels in each iteration.

Let t be the current level of the leaves; each leaf will start the exploration by sending *Explore*$(t + 1, l)$ to its still unexplored neighbours. In general, the expansion messages will be of the form *Explore*$(level, counter)$, where $level$ is the next level to be assigned and $counter$ denotes how many more levels should be expanded by the node receiving the message.

When a node x not yet in the tree receives its first expansion message, say *Explore*(j, k) from neighbour y, it will *accept* the message, consider the sender y as its parent in the tree, and set its own level to be j. It then considers the number k of levels still to be expanded. If $k = 0$, x sends immediately a *Positive*(j) reply to its parent y. Instead, if $k > 0$, x will send *Explore*$(j + 1, k - 1)$ to all its other neighbours, and wait for their reply: those that reply

Positive$(j+1)$ are considered its children, those that reply *Negative*$(j+1)$ are considered not-children; if/when all have sent a reply with level $j+1$, x sends a *Positive*(j) reply to its parent y.

Note that this first "Explore" message will not necessarily determine x's parent or level in the final tree; in fact, it is possible that x will receive later an *Explore*(j', k') message with a smaller level $j' < j$ from a neighbour z. (Note: it might even be possible that $y = z$). What we will do in this case is to have x "trow away" the work already done and "start from scratch" with the new information: x will accept the message, consider z its parent, set its level to j', send *Explore*$(j'+1, k'-1)$ to all its other neighbours (assuming $k' > 0$), and wait for their reply. Note that x might have to "trow away" work already done more than once during an iteration. How many times ? It is not difficult to figure out that it can happen at most $t - j + 1$ times, where j is the first level it receives in this iteration (Exercise 6.19).

We still have to specify under what conditions will a node x send a negative reply to a received message *Explore*(j, k); the rule is simple: x will reply *Negative*(j) if no shorter path is found from the root s to x, i.e. if $j \geq level_x$.

A more detailed description of the expansion phase of the protocol, which we will call *BF_Levels*, is shown in Figure 9, describing the behaviour of a node x not part of the current fragment. As mentioned, the expansion phase is started by the leaves of the current fragment, which we will call *sources* of this phase, upon receiving the *Start Iteration* message from the root. Each source will then send *Explore*$(t+1, l)$ to their unexplored neighbours, where t is the level of the leaves and l (a design parameter) is the number of levels that will be added to the current fragment in this iteration. The *terminating phase* also is started by the sources (i.e., the leaves of the already existing fragment), upon receiving a reply to all their expansion messages.

Correctness

During the extension phase all the nodes at distance at most $t + l$ from the root are indeed reached, as can be easily verified (Exercise 6.20). Thus, to prove the correctness of the protocol we need just to prove that those nodes will be attached to the existing fragment at the proper level.

We will prove this by induction on the levels. First of all, all the nodes at level $t+1$ are neighbours of the sources and thus each will receive at least one *Explore*$(t+1, l)$ message; when this happens, regardless of whatever has happened before, each will set its level to $t+1$; since this is the smallest level that they can ever receive, their level will not change during the rest of the iteration.

Let it be true for the nodes up to level $t+k$, $1 \leq k \leq l-1$; we will show that it holds also for the nodes in level $t+k+1$. Let π be the path of length $t+k+1$ from s to x and let u be the neighbour of x in this path; by definition, u is at level $t+k$ and, by inductive hypothesis, it has correctly set $(level_u) = t+k$. When this happened, u sent a message *Explore*$(t+k+1, l-k-1)$ to all its neighbours, except its parent. Since x is clearly not u's parent, it will eventually receive this message; when this happens, x will correctly set $(level_x) = t+k+1$. So we must show that the expansion phase will not terminate before x

When x receivs *Explore*(j, k) from its neighbour y:

1. If $j < level_x$, a shorter path from the root s to x has been found.
   (a) If x already has a parent, then x disregards all previous information (including the identity of its parent).
   (b) x considers y to be its parent, and sets $level_x = j$.
   (c) If $k > 0$, x sends *Explore*$(j + 1, k - 1)$ to all its neighbours except its parent. If $k = 0$, then a positive reply *Positive*(j) is sent to the parent y.
2. Let $j > level_x$. In this case, this is not a shorter path to x; x replies with a negative acknowledgment *Negative*(j).

When x receives a reply from its neighbour z:

1. If the level of the reply is $(level_x + 1)$ then:
   (a) if the reply is *Negative*$(level_x + 1)$, then x considers z a *non-child*.
   (b) if the reply is *Positive*$(level_x + 1)$ then x considers z a *child*.
   (c) If, with this message, x has now received a reply with level $(level_x + 1)$ from all its neighbours except its parent, then it sends *Positive*$(level_x)$ to its parent.
2. If the level of the reply is not $(level_x + 1)$ then the message is discarded.

Figure 9: Exploration phase of *BF_Levels*: x is not part of the current fragment

receives this message. Focus again on node u; it will not send a positive acknowledgment to its parent (and thus the phase can not terminate) until it receives a reply from all its other neighbours, including x. Since, to reply, x must first receive the message, x will correctly set its level during the phase.

Cost

To determine the cost of protocol *BF_Levels*, we need to analyze the cost of the synchronization and of the expansion phases.

The cost of a synchronization, as we discussed earlier, is at most $2(n-1)$ messages, since both the initialization broadcast and the termination convergecast are performed on the currently available tree. Hence, the total cost of all synchronizations activities depends on the number of *iterations*. this quantity is easily determined. Since there are $radius(r) < d(G)$ levels, and we add l levels in every iterations, except in the last where we add the rest, the number of iterations is at most $\lceil d(G)/l \rceil$. This means that the total amount of messages due to synchronization is at most

$$2(n-1) \lceil \frac{d(G)}{l} \rceil \leq 2 \frac{(n-1)^2}{l} \quad (8)$$

Let us now analyze the cost of the expansion phase in iteration i, $1 \leq i \leq \lceil d(G)/l \rceil$. Observe that, in this phase, only the nodes in the levels $L(i) = \{(i-1)l+1, (i-1)l+2, \ldots, il-1, il\}$ as well as the sources (i.e., the nodes at level $(i-1)l$) will be involved, and messages will only be sent on the m_i links between them. The messages sent during this phase will be just *Explore*$(t+1, l)$, *Explore*$(t+2, l-1)$, *Explore*$(t+3, l-2)$,…,*Explore*$(t+l, 0)$, and the corresponding replies, *Positive*(j) or *Negative*(j), $t+1 \leq j \leq t+l$.

A node in one of the levels in $L(i)$ sends to its neighbours at most one of each of those *Explore* messages; hence there will be on each of edge at most $2l$ *Explore* messages (l in each direction), for a total of $2lm_i$. Since for each *Explore* there is at most one reply, the total number of messages sent in this phase will be no more than $4lm_i$ This fact, observing that the set of links involved in each iteration are disjoint, yields less than

$$\sum_{i=1}^{\lceil d(G)/l \rceil} 4 \, l \, m_i = 4 \, l \, m \quad (9)$$

messages for all the explorations of all iterations. Combining (8) and (9), we obtain

$$\mathbf{M}[BF_Levels] \leq \frac{2(n-1)d(G)}{l} + 4 \, l \, m \quad (10)$$

If we choose $l = O(n/\sqrt{m})$, expression (10) becomes

$$\mathbf{M}[BF_Levels] = O(n \sqrt{m})$$

Network	**Algorithm**	**Messages**	Time
General	*BF*	$O(m + nd)$	$O(d^2)$
General	*BF_Levels*	$O(n\sqrt{m})$	$O(d^2\sqrt{m}/n + d)$
Planar	*BF_Levels*	$O(n^{1.5})$	$O(d^2/\sqrt{n} + d)$

Table 7: Summary: Costs of constructing a breadth-first tree.

This formula is quite interesting. In fact, it depends not only on n but also on the *square root* of the number m of links.

If the network is *sparse* (i.e., it has $O(n)$ links), then the protocol uses only

$$O(n^{1.5})$$

messages; note that this occurs in any *planar* network.

The worst case will be with very *dense* networks (i.e., $m = O(n^2)$). However in this case the protocol will use at most

$$O(n^2)$$

messages, which is no more than protocol BF.

In other words, protocol *BF_Levels* will have the same cost as protocol BF only for very dense networks, and will be much better in all other systems; in particular, whenever $m = o(n^2)$, it uses a subquadratic number of messages.

Let us consider now the *ideal time* costs of the protocol. Iteration i consists of reaching levels $L(i)$ and returning to the root; hence the ideal time will be exactly $2il$ if $1 \leq i < \lceil d(G)/l \rceil$, and time $2d(G)$ in the last iteration. Thus, without considering the roundup, in total we have

$$\mathbf{T}[BF_Levels] = \sum_{i=1}^{d(G)/l} 2\ l\ i\ =\ \frac{d(G)^2}{l} + d(G) \tag{11}$$

The choice $l = O(n/\sqrt{m})$ we considered when counting the messages will give

$$\mathbf{T}[BF_Levels] = O(d(G)^2\sqrt{m}/n)$$

which, again, is the same ideal time as protocol BF only for very dense networks, and less in all other systems.

2.4.3 Reducing Time with More Messages

If time is of paramount importance, better results can be obtained at the cost of more messages. For example, if in protocol *BF_Levels* we were to choose $l = d(G)$, we would obtain an *optimal time costs*.

$$\mathbf{T}[BF_Levels] = 2d(G)$$

IMPORTANT. We measure ideal time considering a synchronous execution where the communication delays are just one unit of time. In such an execution, when $l = d(G)$, the number of messages will be exactly $2m + n - 1$ (Exercise 6.22). In other words, in this synchronous execution, the protocol has *optimal message costs*. However, this is *not* the message complexity of the protocol, just the cost of that particular execution. To measure the message complexity we must consider all possible executions. Remember: to measure ideal time we consider only synchronous executions, while to measure message costs we must look at all possible executions, both synchronous and asynchronous (and choose the worst one).

The cost in messages choosing $l = d(G)$ is given by expression (10) that becomes

$$O(m\ d(G))$$

This quantity is reasonable only for networks of small degree. By the way, a priori knowledge of $d(G)$ is not necessary to obtain these bounds (either time or messages): Exercise 6.21.

If we are willing to settle for a low but *suboptimal* time, it is possible to achieve it with a better message complexity. Let us how.

In protocol *BF_Levels* the network (and thus the tree) is viewed as divided into "strips", each containing l levels of the tree. See Figure 10.

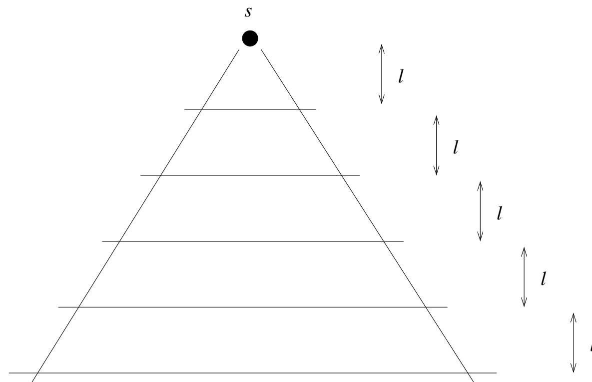

Figure 10: We need more efficient expansion of l levels in each iteration.

The way the protocol works right now, in the expansion phase, each source (i.e., each leaf of the existing tree) constructs its own bf-tree over the nodes in the next l levels. These bf-trees have differential growth rates, some growing quickly, some slowly. Thus, it is possible for a quickly growing bf-tree to have processed many more levels than a slower bf-tree. Whenever there are conflicts due to transmission delays (e.g., the arrival of a message with a better

level) or to concurrency (e.g., the arrival of another message with the same level), these conflicts are resolved, either by "trowing away" everything already done and joining the new tree, or sending a negative reply. It is the amount of work performed to take care of these conflicts that drives the costs of the protocol up. For example, when a node joins a bf-tree and has a (new) parent, it must send out messages to all its other neighbours; thus, if a node has a high degree and frequently changes trees, these adjacent edge messages dominate the communication complexity. Clearly the problem is how to perform these operations efficiently.

Conflicts and overlap occurring during the constructions of those different bf-trees in the l levels can be reduced by organizing the sources into *clusters* and coordinating the actions of the sources that are in the same cluster, as well as coordinating the different clusters.

This in turn requires that the sources in the same cluster must be connected so to minimize the communication costs among them. The connection through a tree is the obvious option, and is called a *cover tree*. To avoid conflicts, we want that for different clusters the corresponding cover trees have no edges in common. So we will have a *forest* of cover trees, which we will call the *cover* of all the sources. To coordinate the different clusters in the cover we must be able to reach all sources; this however can already be done using the current fragment (recall, the sources are the leaves of the fragment).

The message costs of the expansion phase will grow with the number of different clusters competing for the same node (the so-called *load factor*); on the other hand, the time costs will grow with the depth of the cover trees (the so-called *depth factor*). Notice that it is possible to obtain tradeoffs between the load factor and the depth factor by varying the size of the cover (i.e., the number of trees in the forest); e.g., increasing the size of the forest reduces the depth factor while increasing the load factor.

We are thus faced with the problem of constructing clusters with small amount of competition and shallow cover trees. Achieving this goal yields a *time* cost of $O(d^{1+\epsilon})$ and a *message* cost of $O(m^{1+\epsilon})$ for any fixed $\epsilon > 0$. See Exercise 6.23.

2.5 Suboptimal Solutions: Routing Trees

Up to now, we have considered only *shortest-path routing*; that is, we have been looking at systems that *always* route a message to its destination through the shortest-path. We will call such mechanisms *optimal*. To construct optimal routing mechanisms, we had to construct n shortest-path trees, one for each node in the network, a task that we have seen is quite communication expensive.

In some cases, the *shortest path* requirement is important but not crucial; actually, in many systems, guarantee of delivery with few communication activities is the only requirement.

If the shortest-path requirement is relaxed or even dropped, the problem of constructing a routing mechanism (tables and forwarding scheme) becomes simpler and can be achieved quite efficiently. Because they do not guarantee shortest-paths, such solutions are called *sub-optimal*. Clearly there are many possibilities depending on what (sub optimal) requirements the routing mechanism must satisfy.

A particular class of solutions is the one using a *single spanning-tree* of the network for all the routing, which we shall call *routing tree*. The advantages of such an approach is obvious: we need to construct just one tree. Delivery is guaranteed and no more that $diam(T)$ messages will be used on the tree T. Depending on which tree is used, we have different solutions. Let us examine a few.

- *Center-Based Routing.* Since the maximum number of messages used to deliver a message is at most $diam(T)$, a natural choice for a routing tree is the spanning tree with the smallest diameter. Clearly, for any spanning tree T we have $diam(T) \geq diam(G)$, so the best we can hope for is to select a tree that has the same diameter as the network. One such a tree is shortest-path tree rooted in a center of the network. In fact, let c a center of G (i.e., a node where the *maximum* distance is minimized) and let $PT(c)$ be the shortest-path tree of c. Then (Exercise 6.24):

$$diam(G) = diam(PT(c))$$

  To construct such a tree, we need first of all to determine a center c and then construct $PT(c)$, e.g., using protocol *PT_Construction*.

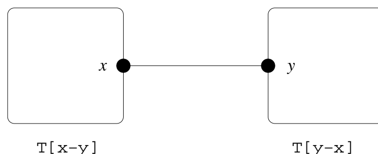

Figure 11: The message traffic between the two subtrees passes through edge $e = (x, y)$.

- *Median-Based Routing.* Once we choose a tree T, an edge $e = (x, y)$ of T linking the subtree $T[x-y]$ to the subtree $T[y-x]$ will be used every time a node in $T[x-y]$ wants to send a message to a node in $T[y-x]$, and viceversa (see Figure 11), where each use costs $\theta(e)$. Thus, assuming that overall every node generates the same amount of messages for every other node and all nodes overall generate the same amount of messages, the cost of using T for routing all this traffic is

$$Traffic(T) = \sum_{(x,y)\in T} \ |T[x-y]| \ |T[y-x]| \ \theta(x,y)$$

  It is not difficult to see that such a measure is exactly the sum of all distances between nodes (Exercise 6.25). Hence, the best tree T to use is one that minimizes the sum of all distances between nodes. Unfortunately, to construct the minimum-sum-distance spanning-tree of a network is not simple. In fact, the problem is NP-hard. Fortunately, it is not difficult to construct a near-optimal solution. In fact, let z be a *median* of the network (i.e., a node for which the *sum* of distances $SumDist(z) = \sum_{v\in V} d_G(x,z)$ to all other nodes is minimized) and let $PT(z)$ be the shortest-path tree of z. If $T\star$ is the spanning tree that minimizes traffic, then traffic (Exercise 6.26):

$$Traffic(PT(z)) \leq 2 \ \ Traffic(T\star)$$

Thus, to construct such a tree, we need first of all to determine a median z and then construct $PT(z)$, e.g., using protocol *PT_Construction*.

- *Minimum-Cost Spanning-Tree Routing.* A natural choice for routing tree is a minimum-cost spanning-tree (MST) of the network. The construction of such a tree can be done e.g., using protocol *MegaMerger* discussed in Chapter **??**.

All the solutions above have different advantages; for example, the center-based one offers the best worst-case cost, while the median-based one has the best average cost. Depending on the nature of the systems and of the applications, each might be preferable to the others.

There are also other measures that can be used to evaluate a routing tree. For example, a common measure is the so called *stretch factor* $\sigma_G(T)$ of a spanning-tree T of G defined as

$$\sigma_G(T) = \ Max_{x,y\in V} \ \frac{d_T(x,y)}{d_G(x,y)} \tag{12}$$

In other words, if a spanning-tree T has a *stretch factor* α, then for each pair of nodes x and y the cost of the path from x to y in T is at most α times the cost of the shortest path between x and y in G. A design goal could thus be to determine spanning trees with small stretch factors (see Exercises 6.27 and 6.28). These ratio are sometimes be difficult to calculate.

Alternate, easier to compute, measures are obtained by taking into account only pairs of *neighbours* (instead of pairs of arbitrary nodes). One such measure is the so called *dilation*, that is the lenght of the longest path in the spanning-tree T corresponding to an edge of G, defined as

$$dilation_G(T) = \ Max_{(x,y)\in E} \ \ d_T(x,y) \tag{13}$$

We also can define the *edge-stretch* factor $\epsilon_G(T)$ (or *dilation factor*) of a spanning-tree T of G as

$$\epsilon_G(T) = \ Max_{(x,y)\in E} \ \ \frac{d_T(x,y)}{\theta(x,y)} \tag{14}$$

As an example, consider the spanning-tree $PT(c)$ used in the center-based solution; if all the link costs are the same, we have that for every two nodes x and y

$$1 \leq d_G(x,y) \leq d_{PT(c)}(x,y) \leq d_{PT(c)} = d_G$$

This means that in $PT(c)$ (unweighted) stretch factor $\sigma_G(T)$, dilation $dilation_G(T)$, and edge -stretch factor $\epsilon_G(T)$ are all bounded by the same quantity, the diameter d_G of G.

For a given spanning tree T, the stretch factor and the dilation factor measure the *worst ratio* between the distance in T and in G for the same pair of nodes and the same edge,

respectively. Another important cost measures are the *average stretch* factor describing the average ratio:

$$\overline{\sigma}_G(T) = \ Average_{x,y \in V} \ \frac{d_T(x,y)}{d_G(x,y)} \tag{15}$$

and the *average edge-stretch* factor (or average dilation factor) $\overline{\epsilon}_G(T)$ of a spanning-tree T of G as

$$\overline{\epsilon}_G(T) = \ Average_{(x,y) \in E} \ \frac{d_T(x,y)}{\theta(x,y)} \tag{16}$$

Construction of spanning trees with low average edge-stretch can be done effectively (Exercises 6.32 and 6.33).

Summarizing, the main disadvantage of using a *routing tree* for all routing tasks is the fact that the routing path offered by such mechanisms is not optimal. If this is not a problem, these solutions are clearly a useful and viable alternative to shortest-path routing.

The choice of which spanning tree, among the many, should be used depends on the nature of the system and of the application. Natural choices include the ones described above, as well as those minimizing some of the cost measures we have introduced (see Exercises 6.28, 6.29, 6.30).

3 Coping with Changes

In some systems, it might be possible that the cost associated to the links change over time; think for example of having a tariff (i.e., cost) for using a link during weekdays different from the one charged in the weekend. If such a change occurs, the shortest path between several pairs of node might change, rendering the information stored in the tables obsolete and possibly incorrect. Thus, the routing tables need to be adjusted.

In this section we will consider the problem of dealing with such events. We will assume that, when the cost of a link (x, y) changes, both x and y are aware of the change and of the new cost of the link. In other words, we will replace the *Total Reliability* restriction with *Total Component Reliability* (thus, the only changes are in the costs) in addition to the *Cost Change Detection* restriction.

Note that costs that change in time can also describe the occurrence of some link failures in the system: the *crash failure* of an edge can be described by having its cost becoming exceedingly large. Hence, in the following, we will talk of link crash failures and of cost changes as the same types of events,

3.1 Adaptive Routing

In these *dynamical* networks where cost changes in time, the construction of the routing tables is only the first step for ensuring (shortest-path) routing: there must be a mechanism to deal with the changes in the network status, adjusting the routing tables accordingly.

3.1.1 Map Update

A simple, albeit expensive solution is the *Map_Update* protocol.

It requires first of all that each table contains the *complete map* of the entire network; the next "hop" for a message to reach its destination is computed based on this map. The construction of the maps can be done e.g. using protocol *Map_Gossip* discussed in Section 2.1. Clearly, any change will render the map inaccurate. Thus, integral part of this protocol is the update mechanism:

Maintenance

- as soon as an entity x detects a local change (either in the cost or in the status of an incident link), x will update its map accordingly and inform all its neighbours of the change through an "update" message;
- as soon as an entity y receives an "update" from a neighbour, it will will update its map accordingly and inform all its neighbours of the change through an "update" message.

NOTE. In several existing systems, an even more expensive *periodic* maintenance mechanism is used: Step 1 of the Maintenance Mechanism is replaced by having each node, periodically and even if there are no detected changes, send its entire map to all its neighbours. This is for example the case with the second Internet routing protocol: the complete map is being sent to all neighbours every 10 to 60 seconds (10 seconds if there is a cost change; 60 seconds otherwise).

The great advantage of this approach is that it is fully adaptive and can cope with any amount and type of changes. The clear disadvantage is the amount of information required locally and the volume of transmitted information.

3.1.2 Vector Update

To alleviate some of the disadvantages of the *Map_Update* protocol, an alternative solution consists in using protocol *Iterative_Construction*, that we designed to construct the routing tables, to keep them up-to-date should faults or changes occur. Every entity will just keep its routing table.

Note that a single change might make all the routing tables incorrect. To complicate things, changes are detected only locally, where they occur, and without a full map it might be impossible to detect if it has any impact on a remote site; furthermore, if more several changes occur concurrently, their cumulative effect is unpredictable: a change might "undo" the damage inflicted to the routing tables by another change.

Whenever an entity x detects a local change (either in the cost or in the status of an incident link), the update mechanism is invoked which will trigger an execution of possibly several iterations of protocol *Iterative_Construction*.

In regards to the update mechanism, we have two possible choices:

- Recompute the routing tables: everybody starts a new *execution* of the algorithm, trowing away the current tables, or
- Update current information: everybody start a new *iteration* of the algorithm with x using the new data, and continuing until the tables converge.

The first choice is very costly since, as we know, the construction of the routing tables is an expensive process. For these reasons, one might want to recompute only what and when is necessary; hence the second choice.

The second choice was used as the original Internet routing protocol; unfortunately, it has some problems.

A well known problem is the so called *count-to-infinity* problem. Consider the simple network shown in Figure 12. Initially all links have cost 1. Then the cost of link (z, w) becomes a *large* integer $K >> 1$. Both nodes z and w will then start an iteration that will be performed by all entities. During this iteration, z it told by y that there is a path from y to w of cost 2; hence, at the end of the iteration, z sets its distance to w to 3. In the next iteration, y sets its distance to w to 4 since the best path to w (according to the vectors it receives from x and z) is through x. In general, after the $2i + 1$-th iteration, x and z will set their cost for reaching w to $2(i + 1) + 1$, while z will set it to $2(i + 1)$. This process will continue until z sets its cost for w to the actual value K. Since K can be arbitrarily large, the number of iterations can be arbitrarily large.

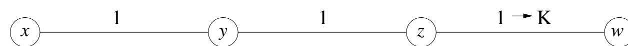

Figure 12: The count-to-infinity problem.

Solving this problem is not easy. See Exercises 6.35 and 6.36.

3.1.3 Oscillation

We have seen some approaches to maintain routing information in spite of failures and changes in the system.

A problem common to all the approaches is called *oscillation*. It occurs if the cost of a link is proportional to the amount of traffic on the link. Consider for example two disjoint paths π_1 and π_2 between x and y, where initially π_1 is the “best” path. Thus, the traffic is initially sent to π_1; this will have the effect of increasing its cost until π_2 becomes the best path. At this point the traffic will be diverted on π_2 increasing its cost, etc. This oscillation between the two paths will continue forever, requiring continuous execution of the update mechanism.

3.2 Fault-Tolerant Tables

To continue to deliver a message through a shortest path to its destination in presence of cost changes or link crash failures, an entity must have up-to-date information on the status of the system (e.g., which links are up, their current cost, etc.). As we have seen, maintaining the routing tables correct when the topology of the network or the edge values may change is a very costly operation. This is true even if faults are very limited.

Consider for example a system where at any time there is at most one link down (not necessarily the same one at all times), and no other changes will ever occur in the system; this situation is called *single link crash failure* (SLF).

Even in this restricted case, the *amount of information* that must be kept in addition to the shortest-paths is formidable (practically the entire map !). This is because the crash failure of a single edge can dramatically change *all* the shortest-path information. Since the tables must be able to cope with every possible choice of the failed link, even in such a limited case, the memory requirements become soon unfeasible.

Furthermore when a link fails, *every* node must be notified so it can route messages along the new shortest paths; the subsequent recovery of that node also will require such a notification. Such an notification process needs to be repeated at each crash failure and recovery, for the entire lifetime of the system. Hence, the *amount of communication* is rather high and never ending as long as there are changes.

Summarizing, the service of delivering a message through a *shortest path* in presence of cost changes or link crash failures, called *shortest-path rerouting* (SR), is expensive (sometimes to the point of being unfeasible) both in terms of storage and of communication.

The natural question is whether there exists a less expensive alternative. Fortunately, the answer is positive. In fact, if we relax the shortest-path rerouting requirement and settle for lower-quality services, then the situation changes drastically; for example, as we will see, if the requirement is just message delivery (i.e., not necessarily through a shortest path), this service be achieved in our SLF system with very simple routing tables and *without any maintenance mechanism ! !*

In the rest of this section, we will concentrate on the *single-link crash failure* case.

3.2.1 Point-of-failure Rerouting

To reduce the amount of communication and of storage, a simple and convenient alternative is to offer, after the crash failure of an arbitrary *single* link, a lower-quality service called *point-of-failure rerouting* (PR):

Point-of-failure (Shortest-path) Rerouting :

1. if the shortest path is not affected by the failed link, then the message will be delivered through that path;

2. otherwise, when the message reaches the node where the crash failure has occurred (the "point of failure"), the message will then be re-routed through a (shortest) path to its destination if no other failure occurs.

This type of service has clearly the advantage that there is no need to notify the entities of a link crash failure and its subsequent reactivation (if any): the message is forwarded as there are no crash failures and if, by chance, the next link it must take has failed, it will be just then provided with an alternative route. This means that, once constructed with the appropriate information for rerouting,

the routing tables do not need to be maintained or updated.

For this reason, the routing tables supporting such a service are called *fault-tolerant* tables.

The amount of information that a fault-tolerant table must contain (in addition to the shortest-paths) to provide such a service will depend on what type of information is being kept at the nodes to do the rerouting, and on whether or not the rerouting is guaranteed to be through a shortest path.

A solution consists in every node x knowing two (or more) *edge-disjoint paths* for each destination : the shortest path, and a secondary one to be used only if the link to the next "hop" in the shortest path has failed. So the routing mechanism is simple: when a message for destination r arrives at x, x determines the neighbour y in the shortest path to r. If (x, y) is up, x will send the message to y, otherwise, it will determine the neighbour z in the secondary path to r and forward the message to z.

The storage requirements of this solution are minimal: for each destination, a node needs to store in its routing table only one link in addition to the one in the fault-free shortest-path. Since we already know how to determine the shortest-path trees, the problem is reduced to the one of computing the secondary-paths. See Exercise 6.34.

NOTE. The secondary paths of a node do not necessarily form a tree.

A major drawback of this solution is that rerouting is not through a shortest path: if the crash failure occurs, the system does not provide any service other than message delivery. Although acceptable in some contexts, this level of service might not be tolerable in general. Surprisingly, it is actually possible to offer *shortest-path rerouting* storing at each node only one link for each destination in addition to the one in the fault-free shortest-path.

We are now going to see how to design such a service.

3.2.2 Point-of-failure Shortest-path Rerouting

Consider a message originated by x and whose destination is s; its routing in the system will be according to the information contained in the shortest-path spanning-tree $PT(s)$. The tree $PT(s)$ is rooted in s; so every node $x \neq s$ has a parent $p_s(x)$, and every edge in $PT(s)$ links a node to its parent.

When the link $e_s[x] = (p_s(x), x)$ fails, it disconnects the tree into two subtrees, one containing s and the other x; call them $T[s-x]$ and $T[x-s]$; see Figure 13.

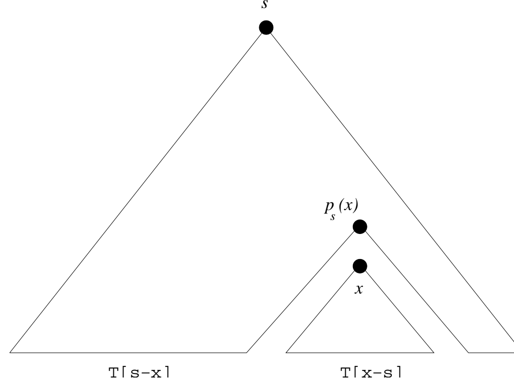

Figure 13: The crash failure of $e_s[x] = (p_s(x), x)$ disconnects the tree $PT(s)$.

When e_x fails, a new path from x to s must be found. It can not be *any*: it must be the shortest path possible between x and s in the network without $e_s[x]$.

Consider a link $e = (u, v) \in G \setminus PT(s)$, not part of the tree, that can reconnect the two subtrees created by the crash failure of $e_s[x]$; i.e., $u \in T[s-x]$ and $v \in T[x-s]$. We will call such a link a *swap edge* for $e_s[x]$.

Using e we can create a new path from x to s: the path will consist of three parts: the path from x to v in $T[x/e_x]$, the edge (u, v), and the path from u to s; see Figure 14. The cost of going from x to s using this path will then be

$$d_{PT(s)}(s, u) + \theta(u, v) + d_{PT(s)}(v, x) \;=\; d(s, u) + \theta(u, v) + d(v, x)$$

This is the cost of using e as a swap for $e_s[x]$. For each $e_s[x]$ there are several edges that can be used as swaps, each with a different cost. If we want to offer shortest-path rerouting from x to s when $e_s[x]$ fails, we must use the *optimal swap*, that is the swap edge for $e_s[x]$ of minimum cost.

So the first task that must be solved is to how find the optimal swap for each edge $e_s[x]$ in $PT(s)$. This computation can be done efficiently (Exercises 6.37 and 6.38); its result is that every node x knows the optimal swap edge for its incident link $e_s[x]$. To be used to construct the PSR routing tables, this process must be repeated n times, one for each destination s (i.e., for each shortest-path spanning tree $PT(s)$).

Once the information about the optimal swap edges has been determined, it needs to be integrated in the routing tables so to provide point-of-failure shortest-path rerouting.

The routing table of a node x must contain information about (1) the shortest paths as well as about (2) the alternative paths using the optimal swaps:

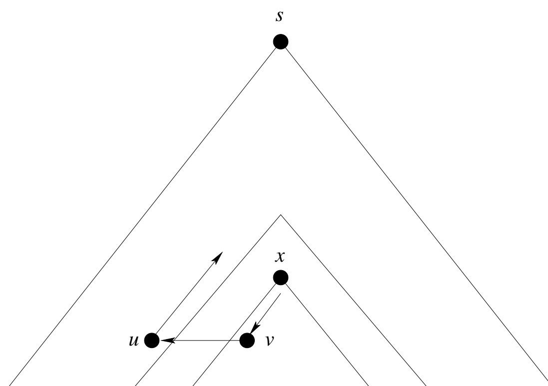

Figure 14: Point-of-failure rerouting using the swap edge $e = (u, v)$ of $e_s[x]$.

Final Destination	Normal Link	Rerouting Link	Swap Destination	Swap Link
s	$(p_s(x), x)$	$(p_v(x), x)$	v	(u, v)

Table 8: Entry in the routing table of x; $e = (u, v)$ is the optimal swap edge for $e_s[x]$.

1. **Shortest path information.** First and foremost, the routing table of x contains for each destination s the link to the neighbour in the shortest path to s if there are no failures. Denote by $p_s(x)$ this neighbour; the choice of symbol is not accidental: this neighbour is the *parent* of x in $PT(s)$ and the link is really $e_s[x] = (p_s(x), x)$.

2. **Alternative path information.** In the entry for the destination s, the routing table of x must also contain the information needed to reroute the message if $e_s[x] = (p_s(x), x)$ is down. Let us see what this information is.

   Let $e = (u, v)$ be the optimal swap edge that x has computed for $e_s[x]$; this means that the shortest-path from x to s if $e_s[x]$ fails is by first going from x to v, then over the link (u, v), and finally from u to s. In other words, if $e_s[x]$ fails, x must reroute the message for s to v; i.e., x must send it to its neighbour in the shortest path to v. The shortest-paths to v are described by the tree $PT(v)$; in fact, this neighbour is just $p_v(x)$ and the link over which the message to s must be sent when rerouting is precisely $e_v[x] = (p_v(x), x)$ (see Exercise 6.39).

   Concluding, the additional information x must keep in the entry for destination s are the rerouting link $e_v[x] = (p_v(x), x)$ and the closest node v on the optimal swap edge for $e_s[x]$; this information will be used only if $e_s[x]$ is down.

Any message must thus contain, in addition to the *final destination* (node s in our example), also a field indicating the *swap destination* (node v in our example), the *swap link* (link (u, v)

in our example) and a bit to explain which of the two must be considered (see Table 8). The *routing mechanism* is rather simple. Consider a message originating from r for node s.

PSR routing mechanism

1. Initially, r sets the final destination to s, the swap destination and the swap link to empty, and the bit to 0; it then sends the message towards the final destination using the normal link indicated in its routing table.
2. If a node x receives the message with final destination s and bit set to 0, then
   (a) if $x = s$ the message has reached its destination: s processes the message.
   (b) if $e_s[x] = (p_s(x), x)$ is up, x forwards the unchanged message on that link.
   (c) if $e_s[x] = (p_s(x), x)$ is down, then x
      i. copies to the swap destination and swap link fields of the message the swap destination and swap link entries for s in its routing table;
      ii. sets the bit to 1;
      iii. sends the message on the rerouting link indicated in its table
3. If a node x receives the message with final destination s and bit set to 1, and swap destination set to v, then
   (a) if $x = v$ then
      i. it sets the bit to 0;
      ii. it sends the message on the swap link.
   (b) otherwise, it forwards the unchanged message on the link $e_v[x] = (p_v(x), x)$.

Destination	Mode	SwapDest	SwapLink	Content
s	1	v	(u, v)	INFO

Figure 15: Message rerouted by x using the swap edge $e = (u, v)$ of $e_s[x]$.

3.3 On Correctness and Guarantees

3.3.1 Adaptive Routing

In all *adaptive routing* approaches, maintenance of the tables is carried out by broadcasting information about the status of the network; this can be done periodically or just when

changes do occur. In all cases, news of changes detected by a node will eventually reach any node (still connected to it). However, due to time delays, while an update is being disseminated, nodes still unaware will be routing messages based on incorrect information. In other words, as long as there are changes occurring in the system (and for some time afterwards), the information in the tables is unreliable and might be incorrect. In particular, it is likely that routing will *not* be done through a shortest path; it is actually possible that messages might not be delivered as long as there are changes. This sad status of affairs is not due to the individual solutions but solely to the fact that time delays are unpredictable. As a result,

> *it is impossible to make any guarantee on correctness and in particular on shortest-path delivery for adaptive routing mechanisms.*

This situation occurs even if the changes at any time are few and their nature limited, as the *single link crash failure* (SLF). It would appear that we should be able to operate correctly in such a system; unfortunately this is not true:

> *it is impossible to provide shortest-path routing even in the single link crash failure case*

This is because the crash failure of a single edge can dramatically change *all* the shortest-path information; thus, when the link fails, every node must be notified so it can route messages along the new shortest paths; the subsequent recovery of that node also will require such a notification. Such an notification process needs to be repeated at each crash failure and recovery, and again the unpredictable time delays will make it impossible to guarantee correctness of the information available at the entities, and thus of the routing decision they make based on that information.

Questions. What, if anything, can be guaranteed ?

The only think that we can say is that, *if* the changes stop (or there are no changes for a long period of time) then the updates to the routing information converge to the correct state, and routing will proceed according to the existing shortest paths. In other words, if the "noise" caused by changes stops, eventually the entities get the correct result.

3.3.2 Fault-Tolerant Tables

In the *fault-tolerant tables* approach, no maintenance of the routing tables is needed once they have been constructed. Therefore, there are no broadcasts or notifications of changes that, due to delays, might affect the correctness of the routing.

However, also faut-tolerant tables suffer because of the unpredictability of time delays. For example, even with the *single link crash failure*, point-of failure shortest-path rerouting can not be guaranteed to be correct: while the message for s is being rerouted from x towards the swap edge $e_s[x]$, the link $e_s[x]$ might recover (i.e., come up again) and another link on the may go down. Thus the message will again be rerouted, and might continue to do so if a "bad" sequence of recovery-failure occurs. In other words, not only the message will not

reach s through a shortest-path from the first point-of-failure, but it will not reacg s at all as long as there is a change. It might be argued that such a sequence of events is highly unlikely; but it is possible. Thus, again,

Questions. What, if anything, can be guaranteed ?

As in the case of adaptive routing, the only guarantee that *if* the changes stop (or there are no changes for a long period of time) then messages will be (during that time) correctly delivered through point-of-failure shortest paths.

4 Routing in Static Systems: Compact Tables

There are systems that are static in nature; for example, if *Total Reliability* holds, no changes will occur in the network topology. We will consider *static* also any system where the routing table, once constructed, can not be modified (e.g., because they are hardcoded/hardwired). Such is, for example, any system etched on a chip; should faults occur, the entire chip will be replaced.

In these systems, an additional concern in the design of shortest-path routing tables is their *size*; that is, an additional design goal is to construct table that are as small as possible.

4.1 The Size of Routing Tables

The full routing table can be quite large. In fact, for each of its $n-1$ destinations, it contains the specification (and the cost) of the shortest path to that destination. This means that each entry possibly contains $O(n \log w)$ bits, where $w \geq n$ is the range of the entities' names, for a total table size of $O(n^2 \log w)$ bits. Assuming the best possible case, i.e., $w = n$, the number of bits required to store all the n full routing tables is

$$S_{FULL} = O(n^3 \log n)$$

For n large, this is a formidable amount of space just to store the routing tables.

Observe that, for any destination, the first entry in the shortest path will always be a link to a neighbour. Thus, it is possible to simplify the routing table by specifying for each destination y only the neighbour of x on the shortest path to it. Such a table is called *short*. For example, the short routing table for s in the network of Figure 1 is shown in Table 9.

In its short representation, each entry of the table of an entity x will contain $\log w$ bits to represent the destination's name and another $\log w$ bits, to represent the neighbour's name. In other words, the table contains $2(n-1)\log w$ bits. Assuming the best possible case, i.e., $w = n$, the number of bits required to store *all* the routing tables is

$$2n(n-1)\log n$$

This amount of space can be further reduced if, instead of the neighbours' names we use the local port numbers leading to them. In this case, the size will be $(n-1)(\log w + \log p_x)$ bits,

Destination	**Neighbour**
h	h
k	h
c	c
d	c
e	e
f	e

Table 9: Short representation of $RT(s)$

Port	**Destinations**
$port_s(h)$	h, k
$port_s(c)$	c, d
$port_s(e)$	e, f

Table 10: Alternative short representation of $RT(s)$

where $p_x \geq deg(x)$ is the range of the local port numbers of x. Assuming the best possible case, i.e., $w = n$ and $p_x = deg(x)$ for all x, this implies that the number of bits required to store all the routing tables is *at least*

$$S_{SHORT} = \sum_x (n-1) \log deg(x) = (n-1) \log \Pi_x deg(x)$$

which can be still rather large.

Notice that the same information can be represented by listing for each port the destinations reached via shortest-path through that port; e.g., see Table 10. This alternative representation of $RT(x)$ uses only $deg(x) + (n-1)\log(n)$ bits for a total of

$$S_{ALT} = \sum_x (deg(x) + (n-1)\log n) = 2m + n(n-1)\log n \tag{17}$$

It appears that there is not much more that it can be done to reduce the size of the table. This is however not the case if we, as designers of the system, had the power to choose the *names* of the nodes and of the links.

4.2 Interval Routing

The question we are going to ask is whether it is possible to drastically reduce this amount of storage if we know the network topology and we have the power of choosing the names of the nodes and the port labels.

4.2.1 An Example: Ring Networks

Consider for example a *ring* network, and assume for the moment that all links have the same cost.

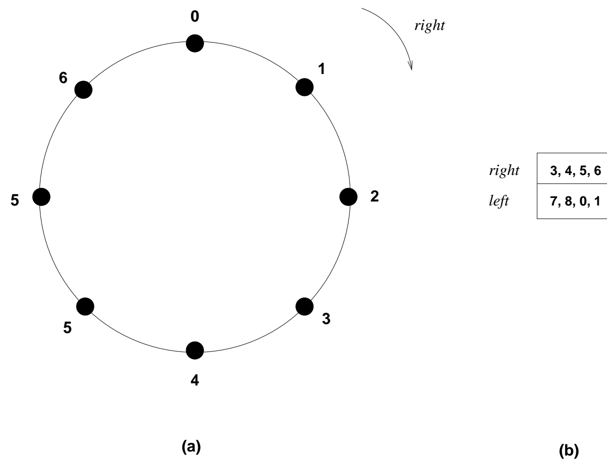

Figure 16: (a) Assigning names and labels. (b) Routing table of node 2

Suppose that we assign as names to the nodes consecutive integers, starting from 0 and continuing clockwise, and we label the ports *right* or *left* depending on whether or not they are in the clockwise direction, respectively. See Figure 16(a).

Concentrate on node 0. This node, like all the others, has only two links. Thus, whenever 0 has to route a message for $z > 0$, it must just decide whether to send it to *right* or to *left*. Observe that the choice will be *right* for $1 \leq z \leq \lfloor n/2 \rfloor$, and *left* for $\lfloor n/2 \rfloor + 1 \leq z \leq n-1$. In other words, the destinations are consecutive integers (modulo n). This is true not just for node 0: if x has to route a message for $z \neq x$, the choice will be *right* if z is in the interval $< x+1, x+2, \ldots x + \lfloor n/2 \rfloor >$, and *left* if z is in the interval $< x + \lfloor n/2 \rfloor + 1, \ldots, x-1 >$, where the operations are *modulo n*. See Figure 16(b).

In other words, in all these routing tables, the set of destinations associated to a port is an *interval* of consecutive integers, and, in each table, the intervals are *disjoint*. This is very important for our purpose of reducing the space.

In fact, an interval has a very short *representation*: it is sufficient to store the two *end* values; that is just $2 \log n$ bits. We can actually do it with just $\log n$ bits; see Exercise 6.40. Since a table consists just of two intervals, we have routing tables of $4 \log n$ bits each, for a grand total of just

$$4n \log n$$

This amount should be contrasted with the one of Expression 17 that, in the case of rings becomes $n^2 \log n + l.o.t.$. In other words, we are able to go from quadratic to just linear space requirements. Note that is is true even if the costs of the links are not all the same: Exercise 6.41.

The phenomenon we have just described is not isolated, as we will discuss next.

4.2.2 Routing with Intervals

Consider the names of the nodes in in a network G. Without any loss of generality, we can always assume that the names are consecutive positive integers, starting from 0; that is, the set of names is $Z_n = \{0, 1, \ldots, n-1\}$.

Given two integers $j, k \in Z_n$, we denote by (j, k) the sequence

$$(j,k) = \langle j, j+1, j+2, \ldots, k\rangle \text{ if } j < k$$
$$(j,k) = \langle j, j+1, j+2, \ldots n-1, 0, 1, \ldots, k\rangle \text{ if } j \geq k$$

Such a sequence (j, k) is called a circular *interval* of Z_n; the empty interval $\emptyset$ is also an interval of Z_n.

Suppose that we are able to assign names to the nodes so that the shortest-path routing tables for G have the following two properties. At every node x:

1. **interval:** For each link incident to x, the (names of the) destinations associated to that link form a circular interval of Z_n.
2. **disjointness:** Each destination is associated to only one link incident to x.

If this is the case, then we can have for G a very *compact representation* of the routing tables, like in the example of the ring network. In fact, for each link the set of destinations is an interval of consecutive integers, and, like in the ring, the intervals associated to the links of a given nodes are all disjoint.

In other words, each table consists of a set of intervals (some of them may be empty), one for each incident link. From the storage point of view, this is very good news because we can represent such intervals by just their *start* values (or, alternatively, by their end values).

In other words, the routing table of x will consist of just one entry for each of its links. This means that the amount of storage for its table is only $deg(x) \log n$ bits. In turn, this means that the number of bits used in total to represent all the routing tables will be just

$$S_{INTERVAL} = \sum_x \deg(x) \log n \;=\; 2\; m\; \log n \tag{18}$$

How will the *routing mechanism* then work with such tables? Suppose x has a message whose destination is y. Then x checks in its table which interval y is part of (since the intervals are disjoint, y will belong to exactly one), and sends the message to the corresponding link.

Because of its nature, this approach is called *interval routing*. If it can be done, as we have just seen, it allows for efficient shortest-path routing with a minimal amount of storage requirements.

It however requires that we, as designers, find an appropriate way to assign names to nodes so that the interval and disjointness properties holds. Given a network G, it is not so obvious how to do it or whether it can be done at all.

Tree Networks

First of all we will consider *tree* networks. As we will see, in a tree it is always possible to achieve our goal and can actually be done in several different ways.

Given a tree T, we first of all choose a node s as the source, transforming T into the tree $T(s)$ rooted in s; in this tree, each node x has a parent and some children (possibly none). We then assign as names to the nodes consecutive integers, starting from 0, according to the *post-order traversal* of $T(s)$; e.g., using procedure

```
Post_Order_Naming(x, k)
begin
     Unnamed_Children(x):= Children(x);
     while Unnamed_Children(x) ≠ ∅ do
          y ← Unnamed_Children(x);
          Post_Order_Naming(y, k)
     endwhile
     myname:= k;
     k:= k + 1;
end
```

started by calling *Post_Order_Naming*$(s, 0)$. This assignment of names has several properties. For example, any node has a larger name than all its descendents. More importantly, it has the *interval* and *disjointness* properties (Exercise 6.45).

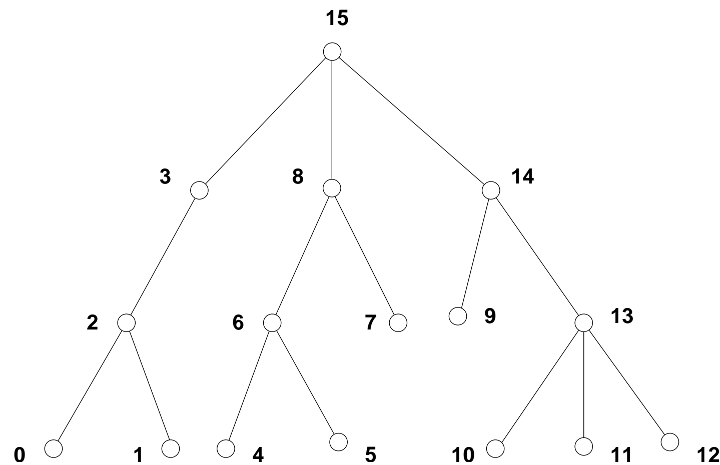

Figure 17: Naming for interval routing in trees

Informally, the interval property follows is because, when executing *Post_Order_Naming* with input (x, k), x and its descendents will be given as names consecutive integers starting from k. See for example Figure 18.

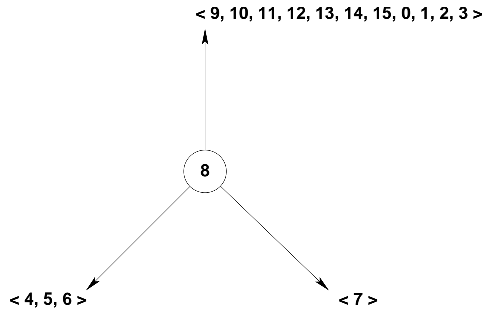

Figure 18: Disjoint intervals

Other Networks

Most regular network topologies we have considered in the past can be assigned names so that *interval routing* is possible. This is for example the case of the $p \times q$ *mesh* and *torus*, *hypercube*, *butterfly*, and *cube-connected-cycles*; see Exercises 6.48 and 6.49. For these networks the construction is rather simple.

Using a more complex construction, names can be assigned so that interval routing can be done also in any *outerplanar* graph (Exercise 6.50); recall that a graph is outerplanar if it can be drawn in the plane with all the nodes lying on a ring and all edges lying in the interior of the ring without crossings.

Question: Can interval routing be done in every network ?

The answer is unfortunately: *No.* In fact there exist rather simple networks, the so called *globe* graphs (one is shown in Figure 19), for which interval routing is *impossible* (Exercise 6.52).

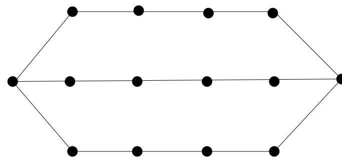

Figure 19: A *globe* graph: interval routing is not possible.

5 Bibliographical Notes

[12, 14, 15]

Shortest-paths: The $O(n^3)$ construction for a single tree was obtained independently by Bob Gallager [15] and To-Yat Cheung [9]. Chandi and J. Misra [19], C.C. Chen [8], T.Y. Cheung [9], J. Park and N. Tokura and T. Masuzawa and K. Hagihara [24], B. Awerbuch [2]

Long Messages (Exercise 6.15): Haldar [16]

Planar networks: Greg N. Frederickson [12]

BFS:
Protocol *BF* (known as the "Coordinated Minimum Hop Algorithm") is due to Bob Gallager [15]. Also to Gallager [15] is due the idea of reducing time by partitioning the layers of the breadth-first tree into groups (Section 2.4.3) and a series of time-messages tradeoffs (" Modified Coordinated Minimum Hop Algorithm"). Protocol *BF_Layers* has been designed by Greg N. Frederickson [12]. The problem of reducing time while mantaining a reasonable message complexity has been investigated by B. Awerbuch [3], B. Awerbuch and R. G. Gallager [6], Zhu and T. Cheung [27]. The near-optimal bounds (Exercise 6.23) have been obtained by B. Awerbuch [4].
cover: B. Awerbuch and B. Berger and L. Cowen and D. Peleg [5]

Center-based and median-based routing was first dicussed in details by David W. Wall and Susanna Owicki [26]. The lower-bound on average edge-stretch and the construction of spanning trees with low average edge-stretch (Exercises 6.31,6.32 and 6.33) are due to Noga Alon, Richard Karp, David Peleg and Doug West [1].

Edge-disjoint: Mohanty and Bhattacharjee [21]
Storing, in addition to the shortest-path routing tables, k other spanning trees for each destination suggested by Alon Itai and M. Rodeh[17].
Swap-edges: H. Ito and K. Iwama and Y. Okabe and T. Yoshihiro [18]. Enrico Nardelli, Guido Proietti and Peter Widmayer[23]. Paola Flocchini, Linda Pagli, Tony Mesa, Giuseppe Prencipe, and Nicola Santoro [11].

Compact: Nicola Santoro and Ramez Kathib [25], Jan van Leeuwen and Richard Tan [20].
Outerplanar graphs (Exercise 6.50) is due to Greg Frederickson and Ravi Janardan[13].
Linear interval routing [7] by E.M. Bakker and Jan van Leeuwen and Richard Tan (Exercise 6.51 and Problem 6.1)
Characterization Lata Narayanan and Sunil Shende [22] and T. Eilam and Shlomo Moran and Shmuel Zaks [10].

6 Exercises, Problems, and Answers

6.1 Exercises

Exercise 6.1 *Write the set of rules corresponding to Protocol* Map_Gossip *described in Section 2.1.*

Exercise 6.2 $\star\star\star$
Consider a tree network where each entity has a single item of information. Determine the time costs of gossiping. What would the time costs be if each entity x initially has $deg(x)$ items ?

Exercise 6.3 *Consider a tree network where each entity has $f(n)$ items of information. Assume that messages can contain $g(n)$ items of information (instead of $O(1)$); with how many messages can gossiping be performed ?*

Exercise 6.4 *Using your answer to question 6.3, with how many messages can all routing table be constructed if $g(n) = O(n)$?*

Exercise 6.5 *Consider a tree network where each entity has $f(n)$ items of information. Assume that messages can contain $g(n)$ items of information (instead of $O(1)$); with how many messages can all items of information be collected at a* single *entity ?*

Exercise 6.6 *Using your answer to question 6.5, with how many messages can all routing table be constructed at that single entity if $g(n) = O(n)$?*

Exercise 6.7 *Write the set of rules corresponding to Protocol* Iterated_Construction *described in Section 2.2. Implement and properly test your implementation.*

Exercise 6.8 *Prove that Protocol* Iterated_Construction *converges to the correct routing tables, and will do so after at most $n-1$ iterations.* Hint. Use induction to prove that $V_x^i[z]$ is the cost of the shortest-path from x to z using at most i hops.

Exercise 6.9 *We have assumed that the cost of a link is the same in both directions; i.e., $\theta(x,y) = \theta(y,x)$. However, there are cases when $\theta(x,y)$ can be different from $\theta(y,x)$. What modifications have to be made so that Protocol* Iterated_Construction *works correctly also in those cases ?*

Exercise 6.10 *In protocol* PT_Construction*, no action is provided for an* idle *entity receiving an* Expand *message. Prove that such a message will never be received in such a state.*

Exercise 6.11 *In procedure* Compute_Local_Minimum *of protocol* PT_Construction*, an entity might set path_length to infinity. Show that, if this happens, this entity will set path_length to infinity in all subsequent iterations.*

Exercise 6.12 *In protocol* PT_Construction*, each entity will eventually set path_length to infinity. Show that, when this happens to a leaf of the constructed tree, that entity can be removed from further computations.*

Exercise 6.13 *Modify protocol* PT_Construction *so it constructs the routing table $RT(s)$ of the source s.*

Exercise 6.14 *We have assumed that the cost of a link is the same in both directions; i.e.,* $\theta(x, y) = \theta(y, x)$*. However, there are cases when* $\theta(x, y)$ *can be different from* $\theta(y, x)$*. What modifications have to be made so that Protocol* PT_Construction *works correctly also in those cases ?*

Exercise 6.15 *Assume that messages can contain* $O(n)$ *items of information (instead of* $O(1)$*). Show how to construct* all *the shortest-path trees with just* $O(n^2)$ *messages.*

Exercise 6.16 *Prove that, after iteration* $i - 1$ *of Protocol* BF_Construction*,*
(a) all the nodes at distance up to $i - 1$ *are part of the tree;*
(c) each node at distance $i - 1$ *knows which of its neighbours are at distance* $i - 1$*.*

Exercise 6.17 *Write the set of rules corresponding to Protocol* BF *described in Section 2.2. Implement and properly test your implementation.*

Exercise 6.18 *Write the set of rules corresponding to Protocol* BF_Levels*. Implement and properly test your implementation.*

Exercise 6.19 *Let* Explore(j, k) *be the first message* x *accepts in the expansion phase of protocol* BF_Levels*. Prove that the number of times* x *will change its level in this phase is at most* $j - t + 1 < l$*.*

Exercise 6.20 *Prove that, in the* expansion *phase of an iteration of Protocol* BF_Levels*, all nodes in levels* $t + 1$ *to* $t + l$ *are reached and attached to the existing fragment, where* t *is the level of the sources (i.e., the leaves in the current fragment).*

Exercise 6.21 *Consider protocol* BF_Levels *when* $l = d(G)$*. Show how to obtain the same message and time complexity without any a priori knowledge of* $d(G)$*.*

Exercise 6.22 *Prove that, if we choose* $l = d(G)$ *in protocol* BF_Levels*, then in any synchronous execution the number of messages will be exactly* $2m + n - 1$*.*

Exercise 6.23 $\star\star$
Show how to construct a breadth-first spanning-tree in time $O(d(G)^{1+\epsilon})$ *using no more than* $O(m^{1+\epsilon})$ *messages, for any* $\epsilon > 0$*.*

Exercise 6.24 *Let* c *be a center of* G *and let* $SPT(c)$ *be the shortest-path tree of* c*. Prove that* $diam(G) = diam(SPT(c))$*.*

Exercise 6.25 *Let* T *be a spanning-tree of* G*. Prove that* $\sum_{(x,y)\in T} |T[x-y]||T[y-x]|w(x, y) = \sum_{u,v\in T} d_T(u, v)$*.*

Exercise 6.26 (median-based routing)
Let z *be a* median *of* G *(i.e., a node for which the sum of distances to all other nodes is minimized) and let* $PT(z)$ *be the shortest-path tree of* z*. Prove that* Traffic$(PT(z)) \leq 2$ Traffic$(T\star)$*, where* $T\star$ *is the spanning-tree of* G *for which* $Traffic$ *is minimized.*

Exercise 6.27 *Consider a ring network R_n with weighted edges. Prove or disprove that $PT(c) = MSP(R_n)$, where c is a center of R_n and $MSP(R_n)$ is the minimum-cost spanning-tree of R_n.*

Exercise 6.28 *Consider a ring network R_n with weighted edges. Let c and z be a center and a median of R_n, respectively.*

1. *For each of the following spanning trees of R_n, compare the* stretch *factor and the* edge-stretch *factor: $PT(c)$, $PT(z)$, and the minimum-cost spanning-tree $MSP(R_n)$.*
2. *Determine bounds on the average edge-stretch factor of $PT(c)$, $PT(z)$, and $MSP(R_n)$.*

Exercise 6.29 $\star$
Consider a $a \times a$ square mesh $M_{a,a}$ where all costs are the same.

1. Is it possible to construct two spanning-trees T' and T'' such that $\sigma(T') < \sigma(T'')$ but $\epsilon(T') > \epsilon(T'')$? Explain.

2. Is it possible to construct two spanning-trees T' and T'' such that $\overline{\sigma}(T') < \overline{\sigma}(T'')$ but $\overline{\epsilon}(T') > \overline{\epsilon}(T'')$? Explain.

Exercise 6.30 *Consider a square mesh $M_{a,a}$ where all costs are the same. Construct two spanning-trees T' and T'' such that $\sigma(T') < \sigma(T'')$ but $\overline{\epsilon}(T') > \overline{\epsilon}(T'')$.*

Exercise 6.31 $\star$
Show that there are graphs G with unweighted edges where $\overline{\epsilon}_G(T) = \Omega(\log n)$ for every spanning tree T of G

Exercise 6.32 $\star\star$
Design an efficient protocol for computing a spanning tree with low average edge-stretch of a network G with unweighted *edges.*

Exercise 6.33 $\star\star$
Design an efficient protocol for computing a spanning tree with low average edge-stretch of a network G with weighted *edges.*

Exercise 6.34 $\star$
Design a protocol for computing the secondary-paths of a node x. You may assume that the shortest path tree $PT(x)$ has already been constructed and that each node knows its and its neighbours' distance from x. Your protocol should use no more messages than that required to construct $PT(x)$.

Exercise 6.35 (split horizon) $\star\star$
Consider the following technique, called split horizon*, for solving the* count-to-infinity *problem discussed in Section 3.1.2: during an iteration, a node a does* not *send its cost for destination c to its neighbour b if b is the next node in the "best" path (so far) from a to c. In the example of Figure 12, in the first iteration y does not send its cost for w to z, and thus z will correctly set its cost for w to K. In the next two iterations y and x will correctly set their cost for w to $K+1$ and $K+2$, respectively. Prove or disprove that split horizon solves the count-to-infinity problem.*

Exercise 6.36 (split horizon with poison reverse) $\star\star$
Consider the following technique, called split horizon with poison reverse, *for solving the* count-to-infinity *problem discussed in Section 3.1.2: during an iteration, a node a sends its cost for destination c set to ∞ to its neighbour b if b is on the "best" path (so far) from a to c. Prove or disprove that split horizon with poison reverse solves the count-to-infinity problem.*

Exercise 6.37 $\star$
Design an efficient protocol that, given a shortest-path spanning tree $PT(s)$, determines an optimal swap for every edge in $PT(s)$: at the end of the execution, every node x knows the optimal swap edge for its incident link $e_s[x]$. Your protocol should use no more than $O(nh(s))$ messages, where $h(s)$ is the height of $PT(x)$.

Exercise 6.38 $\star$
Show how to answer Exercise 6.37 using no more than $O(n^*(s))$ messages, where $n^*(s)$ is the number of edges in the transitive closure of $PT(x)$.

Exercise 6.39 *Let $e = (u, v)$ be the optimal swap edge that x has computed for $e_s[x]$. Prove that, if $e_s[x]$ fails, to achieve point-of-failure shortest-path rerouting, x must send the message for s to the incident link $(p_v(x), x)$.*

Exercise 6.40 *Show how to represent the intervals of a* ring *with just* $\log n$ *bits per interval.*

Exercise 6.41 *Show how that the intervals of a* ring *can be represented with just* $\log n$ *bits per interval, even if the costs of the links are not all the same.*

Exercise 6.42 *Let G be a network and assume that we can assign names to the nodes so that, in each routing table, the destinations for each link form an interval. Determine what conditions the intervals must satisfy so that they can be represented with just* $\log n$ *bits each.*

Exercise 6.43 *Redefine properties* interval *and* disjointness *in the case the n integers used as names are not consecutive; i.e., they are chosen from a larger set Z_w, $w > n$.*

Exercise 6.44 *Show an assignment of names in a tree that does* not *have the* interval *property. Does there exists an assignment of distinct names in a tree that has the* interval *property but not the* disjointness *one ? Explain your answer.*

Exercise 6.45 *Prove that, in a tree, the assignment of names by Post-Order traversal has both* interval *and* disjointness *properties.*

Exercise 6.46 *Prove that, in a tree, also the assignment of names by* Pre-Order *traversal has both* interval *and* disjointness *properties.*

Exercise 6.47 *Determine whether interval routing is possible in the regular graph shown in Figure 20. If so, show the routing table; otherwise explain why.*

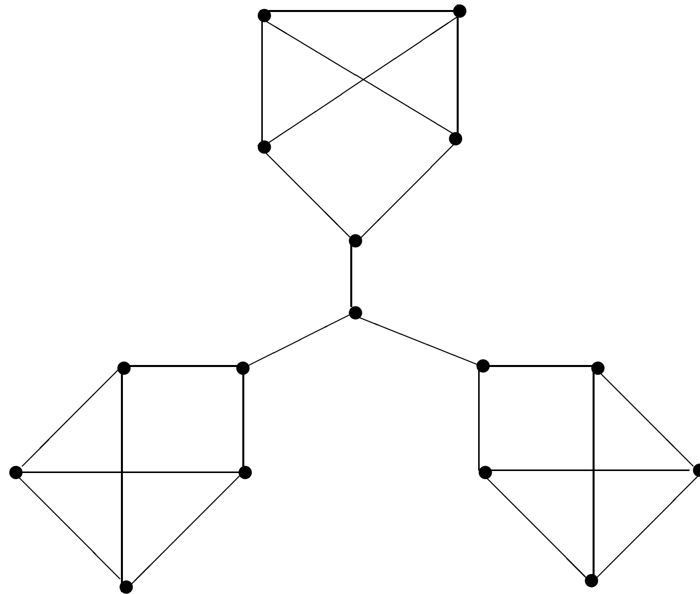

Figure 20: A regular graph.

Exercise 6.48 *Design an optimal interval routing scheme for $p \times q$* mesh *and* torus*. How many bits of storage will it require ?*

Exercise 6.49 *Design an optimal interval routing scheme for a d-dimensional*
(a) hypercube*; (b)* butterfly*; (c)* cube-connected cycles*. How many bits of total storage will each require ?*

Exercise 6.50 $\star\star$ *Show how to assign names to the nodes of an* outerplanar *graph so that interval routing is possible.*

Exercise 6.51 $\star\star$ *If for every x all the intervals in its routing table are strictly increasing (i.e., there is no "wraparound" node "0), the interval routing is called* linear*. Prove that there are networks for which there exists interval routing but linear interval routing is impossible.*

Exercise 6.52 *Prove that in the globe graph of Figure 19 interval routing is not possible.*

6.2 Problems

Problem 6.1 Linear Interval Routing. $\star\star$ *If for every x all the intervals in its routing table are strictly increasing (i.e., there is no "wraparound" node "0), the interval routing is called* linear*. Characterize the class of graphs for which there exists a linear interval routing.*

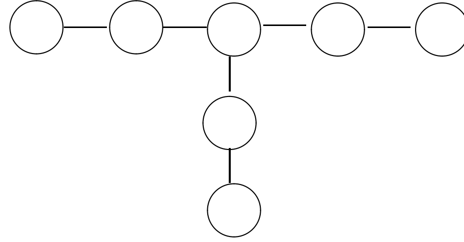

Figure 21: Graph with interval routing but where no linear interval routing exists.

6.3 Answers to Exercises

Partial Answer to **Exercise 6.23**.
Choose the size of the strip to be $k = \sqrt{d(G)}$. A *strip cover* is a collection of trees that span all the source nodes of a strip. In iteration i, first of all construct a "good" cover of strip i

Answer to **Exercise 6.26**.
Observe that, for any spanning tree T of G, $\mathit{Traffic}(T) = \sum_{u,v \in V} d_T(u,v)$ (Exercise 6.25). Let $SumDist(x) = \sum_{u \in V} d_G(u,x)$; clearly $\mathit{Traffic}(T\star) \geq \sum_{x \in V} SumDist(x)$. Let z be a *median* of G (i.e., a node for which $SumDist$ is minimized); then $SumDist(z) \leq \frac{1}{n} Traffic(T\star)$. Thus we have that $\mathit{Traffic}(PT(z)) = \sum_{u,v \in V} d_{PT(z)}(u,v) \leq \sum_{u,v \in V} (d_{PT(z)}(u,z) + d_{PT(z)}(z,v))$ $\leq (n-1) \sum_{u \in V} (d_{PT(z)}(u,z) + (n-1) \sum_{v \in V} (d_{PT(z)}(v,z) = 2(n-1) SumDist(z) \leq 2\,\mathit{Traffic}(T\star)$.

Answer to **Exercise 6.40**.
In the table of node x, the interval associated to *right* always starts with $x+1$ while the one associate to *left* always ends with $x-1$. Hence, for each interval, it is sufficient to store only the other end value.

Partial Answer to **Exercise 6.51**.
Consider the graph shown in Figure 21.

References

[1] Noga Alon, Richard M. Karp, David Peleg, and Douglas West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal of Computing*, 24:78–100, 1995.

[2] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[3] Baruch Awerbuch. Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization. *Networks*, 15:425–437, 1985.

[4] Baruch Awerbuch. Distributed shortest path algorithms. In *Proc. 21st Ann. ACM Symp. on Theory of Computing*, pages 490–500, 1989.

[5] Baruch Awerbuch, B. Berger, L. Cowen, and David Peleg. Near-linear cost sequential and distributed constructions of sparse neighborhood covers. In *Proceedings. 34th Annual Symposium on Foundations of Computer Science*, pages 638–647, New York, 3-5 Nov. 1993.

[6] Baruch Awerbuch and Robert G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, 33:315–322, 1987.

[7] E.M. Bakker, Jan van Leeuwen, and Richard Tan. Linear interval routing. *Algorithms Review*, 2(2):45–61, 1991.

[8] C.C. Chen. A distributed algorithm for shortest paths. *IEEE Transactions on Computers*, C-31:898–899, 1982.

[9] To-Yat Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, 9:504–512, 1983.

[10] T. Eilam, Shlomo Moran, and Shmuel Zaks. The complexity of the characterization of networks supporting shortest-path interval routing. In *4th International Colloquium on Structural Information and Communication Complexity*, pages 99–11, Ascona, 1997.

[11] Paola Flocchini, Linda Pagli, Tony Mesa, Giuseppe Prencipe, and Nicola Santoro. An efficient protocol for computing the optimal swap edges of a shortest path tree. In *??*, 2004.

[12] Greg N. Frederickson. A distributed shortest path algorithm for a planar network. *Information and Computation*, 86(2):140–159, June 1990.

[13] Greg N. Frederickson and Ravi Janardan. Designing networks with compact routing tables. *Algorithmica*, 3:171–190, June 1988.

[14] Eli Gafni and D.P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE Transactions on Communication*, Com-29:11–18, January 1981.

[15] Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1982.

[16] S. Haldar. An 'all pairs shortest paths' distributed algorithm using $2n^2$ messages. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, Utrecht, Netherlands, June 1993.

[17] Alon Itai and M. Rodeh. The multi-tree approach to reliability in distributed networks. *Information and Computation*, 79:43–59, 1988.

[18] H. Ito, K. Iwama, Y. Okabe, and T. Yoshihiro. Polynomial-time computable backup tables for shortest-path routing. In *Proc. of 10th Colloquium on Structural Information and Communication Complexity (SIROCCO 2003)*, pages 163–177, 2003.

[19] J. Misra K.M. Chandi. Distributed computations on graphs: shortest path algorithms. *Communications of ACM*, 25(11):833–837, November 1982.

[20] Jan van Leeuwen and Richard B. Tan. Interval routing. *The Computer Journal*, 30:298–307, 1987.

[21] H. Mohanty and G.P.Bhattacharjee. A distributed algorithm for edge-disjoint path problem. In *6th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 241 of *Lecture Notes in Computer Science*, pages 344–361, New Delhi, 1986. Springer.

[22] Lata Narayanan and Sunil Shende. Characterization of networks supporting shortest-path interval labelling schemes. In *3rd International Colloquium on Structural Information and Communication Complexity*, pages 73–87, 1996.

[23] Enrico Nardelli, Guido Proietti, and Peter Widmayer. Swapping a failing edge of a single source shortest paths tree is good and fast. *Algoritmica*, 35:56–74, 2003.

[24] J. Park, N. Tokura, T. Masuzawa, and K. Hagihara. An efficient distributed algorithm for constructing a breadth-first search tree. *Systems and Computers in Japan*, 20:15–30, 1989.

[25] Nicola Santoro and Ramez Khatib. Labeling and implicit routing in networks. *The Computer Journal*, 28:5–8, 1985.

[26] David W. Wall and Susanna Owicki. Construction of centered shortest-path trees in networks. *Networks*, 13(2):207–332, 1983.

[27] Y. Zhu and To-Yat Cheung. A new distributed breadth-first-search algorithm. *Information Processing Letters*, 25:329–333, 1987.