Minimum hop route maintenance in static and dynamic networks

A.A. Schoone

RUU-CS-90-18
April 1990

MINIMUM HOP ROUTE MAINTENANCE IN STATIC AND DYNAMIC NETWORKS*

Anneke A. Schoone

Department of Computer Science, University of Utrecht,
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract. We discuss distributed algorithms for computing minimum hop distances in a network from a general viewpoint and apply this to minimum hop routing. We show how different models of computation lead to different algorithms known from the literature. We then discuss the effect of various network assumptions (static, dynamic) upon minimum hop route maintenance. Using the Krogdahl-Knuth technique of system-wide invariants, we prove the following distributed algorithms correct: the minimum hop route determination algorithms of Gallager and Friedman for static networks and the route maintenance algorithm of Chu for dynamic networks.

1. Introduction. A basic problem that must be addressed in any design of a distributed network is the routing of messages. That is, if some node in the network decides it wants to send a message to some other node in the network or receives a message destined for some other node, a method is needed to enable this node to decide over which outgoing link it has to send this message. Algorithms for this problem are called *routing algorithms*. In the sequel we will only consider *distributed* routing algorithms which depend on the cooperative behavior of the local routing protocols of the nodes to guarantee effective message handling and delivery.

Desirable properties of routing algorithms are for example correctness, optimality, and robustness. *Correctness* seems easy to achieve in a static network, but the problem is far less trivial in case links and nodes are allowed to go down and come up like they tend to do in practice. *Optimality* is concerned with finding the "quickest" routes. Ideally, a route should be chosen for a message on which it will encounter the least delay but, as this depends on the amount of traffic on the way, this is hardly to foresee and hence is actually difficult to achieve as well. A frequent compromise is to minimize the number of *hops*, i.e., the number of links over which the message travels from sender to destination. We will restrict our study to *minimum hop routing*. *Robustness* is concerned with the ease with which the routing scheme is adapted in case of topological changes.

Our aim in this paper is twofold. First we present a systematic development of a number of distributed algorithms for minimum hop route determination and maintenance, including a re-appraisal of several existing methods for the static case and a detailed analysis

*This research was supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (Project ALCOM).

of some dynamic algorithms. Secondly, we present correctness proofs for these algorithms, which tends to be hard for distributed algorithms and indeed was not presented before in most cases that we consider. This applies in particular to the interesting distributed algorithm for minimum hop route maintenance due to Chu [C] (see also [Ta] and [Sch]), for which we will develop both a complete program skeleton and a correctness proof. In all cases we employ the Krogdahl-Knuth method of system-wide invariants.

The paper is organized as follows. In section 2 we state the assumptions about the networks that we consider and define the concept of a *program skeleton*. Different models of computation are then given. In the remainder we first concentrate on computing minimum hop distances in a static network (section 3). We begin by deriving general properties of distributed minimum hop algorithms, and then discuss different ways -related to different models of computation- for deciding whether an estimate of a minimum hop distance is correct. We conclude this section with the correctness proof of the minimum hop route determination algorithms of Gallager and Friedman [Fr] (section 3.2).

In section 4 we consider the problem of maintaining routes in dynamic networks, i.e., networks with links going down and coming up. A global acquaintance with the contents of sections 3.1 and 3.2.1 is assumed here. In section 4.1 we discuss some typical problems in adapting distributed algorithms for static networks to distributed algorithms for dynamic networks, both in general and specifically for minimum hop route maintenance. We present the well-known dynamic routing algorithm of Tajibnapis [Tj, La] for comparison, and in section 4.2 the dynamic routing algorithm of Chu [C]. The latter was presented as an improvement of the algorithm of Tajibnapis by Tanenbaum [Ta] and Schwartz [Sch] but, while Tajibnapis' algorithm has been proved correct by Lamport [La], it is not clear at all that the algorithm of Chu is correct. Moreover, the presentation of Chu's algorithm in the original report is very imprecise. In section 4.2 we give a complete specification of Chu's algorithm and a correctness proof.

2. Models of computation.

We are interested in the relation between a distributed algorithm and the *model of computation* that is used for its formulation. Generally speaking, a model of computation can be viewed as a set of assumptions and restrictions about the nature of the distributed computation in the network and the communication which takes place between the network nodes.

These assumptions are not meant as a criterion for matching specific networks. Rather, it is a way to focus attention on some aspects of distributed computing, while abstracting away from others. For example, an assumption that is made in this paper is that a link between two network nodes behaves like two FIFO queues of messages, one queue for each direction. This does not mean that we restrict ourselves to networks where this is indeed the case, but that we assume that this can be achieved by communication protocols present in the network, and that the question how to achieve it is not our concern now.

The restrictions about the distributed computation and the communication between the network nodes are restrictions in the order of events that are permitted in an actual execution. Such a permissible order of events is stated by means of a *program skeleton*, which can be refined to an algorithm. Thus a program skeleton stands for a class of algorithms, all of which can be obtained from that same skeleton by some refinement. The restriction in the order of events specified in a program skeleton can add extra structure to the computation at hand, i.e., the computation of minimum hop distances in this case, from which extra information can be derived that is exploited in actual algorithms. We investigate the interaction between the computation per se and the model of computation used. We define some program skeletons together with their assumptions in section 2.1. How these assumptions are modeled in the communication network is discussed in section 2.2.

2.1. Program skeletons. For an appraisal of the general features of distributed programs that compute minimum hop distances, we start by considering the essential building blocks and only use complete programs as illustrations. We do this by means of so-called *program skeletons*, which are generic descriptions for classes of algorithms all of which have some underlying structure in common. A program skeleton consists of a number of *operations*, each of which consists of a piece of program. Operations can be carried out any number of times, by any processor, and at any time. An operation is viewed as an atomic action, i.e., it is not interruptable. We do not specify anything about an assumed order in which the operations may take place, but an operation can contain a so-called *guard*: a boolean expression between braces { }. An operation may only be executed if its guard is true, otherwise nothing happens. For example, a process may only execute the code for receiving a message if there is indeed a message present to receive.

The most basic operations in a distributed program one can think of for a node i are: send a message to j (S_i), receive a message from j (R_i), and do an internal (local) computation (I_i), where j is any node connected to i by a link. Operations and variables are subscripted by the identity of the node that performs and maintains them, respectively. This yields the following program skeleton.

```
S_i :  begin send a message to some neighbor j end

R_i :  {a message has arrived from j }
       begin receive the message from j end

I_i :  begin compute end
```

We can use these atomic operations as *building blocks* for bigger atomic operations, thereby adding extra structure in the order of computation and/or the order of communication. This can be done in different ways, and yields different program skeletons. Some ways are more or less standard, and the resulting program skeletons together with the appropriate network assumptions will be referred to as *models of computation*. Examples are the message-driven model and the synchronous model of computation. Given a model of computation, an idea what information a message should contain, and a way to compute the wanted information from the received information, we have a general framework for an algorithm. For example, the message-driven model of computation with messages which contain the name of a destination node together with the estimated distance of the sender of the message to that destination node forms the basis of both the algorithms of Tajibnapis and Chu. (The basic computation is: take the minimum of the local estimate of the distance to the destination node and $1+$the distance value in the received message.)

As it is usually necessary to specify what the initial values of the node variables are, we will do so for the variables used directly before the code of the atomic operations. In the sequel we distinguish the following four program skeletons. We will give them for static networks now and discuss the extension to the case for dynamic networks in the next section.

2.1.1. Phasing. The idea of phasing is to divide all the work to be done over different phases, and to allow a node to begin working on another phase only if all the work of the current phase is completed. Thus phasing adds some structure to the order of events in a computation, which was totally arbitrary up till now. We add a variable for the current phase at each node i: $phase_i$, and a new atomic operation is added for the transition to the next phase: P_i. The most general program skeleton which makes use of phasing (P) is as follows.

```
Initially   $\forall i:\ phase_i = 0$.

$S_i^P$ :  begin send a message belonging to $phase_i$ to some neighbor $j$ end

$R_i^P$ :  {a message has arrived from $j$ }
       begin receive the message from $j$ and record it;  compute end

$P_i^P$ :  {all messages of phase = $phase_i$ are received from all neighbors }
       begin $phase_i := phase_i + 1$;  compute end
```

Note that the internal computation operation I_i is now divided over operation R_i^P where the computation pertaining to the received message is done, and the operation P_i^P, the internal computation which effectuates the phase transition. Comparison of the operations S_i^P and R_i^P reveals a problem introduced by phasing. While in operation S_i^P a message belonging to $phase_i$ is sent, nothing is mentioned about a phase number of a message in operation R_i^P. Ideally, the messages node i receives while $phase_i = p$, would be the messages that i's neighbors had sent while their phase number had the same value p. Now the problem is, what do we want node i to do when a message belonging to a different phase has arrived? There are several possibilities to deal with this problem.

First, it might be the case that the assumptions about communication on the links combined with the properties of a more specific program skeleton suffice to prove that the arrival of messages of the wrong phase cannot happen. Secondly, we could just refuse to receive the message if it belongs to a different phase, and add a guard to that effect to the operation R_i^P. We should take care not to introduce the possibility of deadlock then. Thirdly, the message could be received but ignored, thus equaling the effect of the loss of a message. Fourth, if the message belongs to a later phase, we could buffer it until the node reaches the right phase. Fifth, we could just let the node receive the message and do the appropriate computation. It will depend on the circumstances what choice we will make.

Another problem is the guard of operation P_i^P: all messages of phase = $phase_i$ are received. There must be some way for node i to decide this. One possibility, if the number of messages per phase is not fixed, is to include the number of messages to expect in a phase in the first message belonging to a phase, or to somehow mark the last message belonging to a phase as such.

2.1.2. Message-driven computation. The added structure in the order of computation in this case is that messages may only be sent upon receipt of another message, and usually there is some relation specified between the contents of the received message and that of the messages sent. We will use the term 'appropriate' for this as long as we do not want to specify the relation. Thus we do not have a separate, autonomous sending operation any more. But now we need a way to start the sending of messages. Hence we introduce an operation A_i^M (awaken) which can either be executed spontaneously (after initialization), or upon receipt of the first message. Thus we include a test whether node i is awake yet in operation R_i^M, and if not, we let i awaken first, i.e., execute operation A_i^M. A node is clearly supposed to awaken only once, although this is only necessary for the termination of program skeleton M and not for its partial correctness. The boolean $awake_i$ records whether i is awake or not.

```
Initially   $\forall i$: $awake_i = false$.

$A_i^M$ : {not $awake_i$}
      begin $awake_i := true$;
            forall neighbors $j$ do send the appropriate messages to $j$ od
      end

$R_i^M$ : {a message has arrived from $j$ }
      begin if not $awake_i$ then do $A_i^M$ fi; receive the message from $j$; compute;
            forall neighbors $k$ do send the appropriate messages to $k$ od
      end
```

Lemma 2.1. In a static network, all nodes eventually awaken iff in every connected component of the network there is at least one node that awakens spontaneously, and messages are not lost, garbled, duplicated, or delayed infinitely long.

Proof. Obvious from program skeleton M. ■

Thus we will assume in the sequel that there is at least one node in each connected component of the network that awakens spontaneously. In a dynamic network, some action comparable to awakening will be taken upon a change in link status (i.e., the going down or coming up of a link). This will be discussed in the next section. We need the other assumption, too, in this model, namely that messages do not get lost, garbled, duplicated, or delayed infinitely long. This is because the sending of a message is now restricted and a node does not have the possibility of sending a message repeatedly until it gets through. Hence we have the following assumptions in this model.

Assumption 2.1. In a static network, at least one node in every connected component awakens spontaneously.

Assumption 2.2. Messages do not get lost, garbled, duplicated, or delayed infinitely long.

This form of computation is called message-driven, i.e., something only happens if a message is received. The situation that no more messages are around is of special interest in this model, as nothing will happen any more when this situation arises. This situation is called *termination* (TERM). Unless we make further assumptions about the network such as: all messages only have a bounded delay, it is sometimes necessary to add a so-called termination detection algorithm to be able to detect this state (see e.g. [Tel]). Otherwise nodes might not be able to conclude whether their computed values really reflect minimum hop distances or not, as this can depend on whether all messages are received and processed.

2.1.3. Simulated synchronous computation. In this model which is frequently used to simulate synchronous programs on an asynchronous network, we combine the added structure of phasing and of the message-driven model of computation. Hence assumptions 2.1 and 2.2 apply here, too. Although we did not demand with phasing that the message received was of the same phase as the receiver (because it was not necessary), we will do so now as it has the advantage that a message can contain less information in this case. However, we then need a way to ensure that no messages belonging to the next phase are received if they happen to arrive early, as is possible in this model. Thus we add a new variable in each node i: rec_i, which is a boolean array which records for each neighbor j in $rec_i[j]$ whether all messages of the current phase from neighbor j have already been received or not. Hence it is necessary that the receiver of a message has some way to decide whether a message received is the last message of the current phase or not.

There are several ways to implement this, for example: counting the number of messages and sending the numbers, too, or combining all messages of one phase over one link into one big message. Again we are not interested in an actual implementation, as long as a node can decide the question. This leads to program skeleton *SS*.

```
Initially   $\forall i$: $awake_i = false$, $phase_i = 0$, and $\forall$ neighbors $j$: $rec_i[j] = false$.

$A_i^{SS}$ : {not $awake_i$}
    begin $awake_i$ := $true$;
        forall neighbors $j$ do send the appropriate messages of $phase_i$ to $j$ od
    end

$R_i^{SS}$ : {a message (of $phase_i$) has arrived from $j$ $\wedge$ not $rec_i[j]$ }
    begin if not $awake_i$ then do $A_i^{SS}$ fi;
        receive the message from $j$ and record it; compute;
        if this was the last message of $phase_i$ from $j$ then $rec_i[j]$ := $true$ fi;
        if $\forall$ neighbors $k$: $rec_i[k]$
        then $phase_i$ := $phase_i + 1$; compute;
            forall neighbors $k$
            do send the appropriate messages of $phase_i$ to $k$; $rec_i[k]$ := $false$ od
        fi
    end
```

Note that operation P_i^P is now included in operation R_i^{SS} to ensure that all operations except awakening are only done upon receipt of a message. A link can contain in one direction messages of two different phases. Assuming that messages have to be received in the order in which they arrive at a node, we now need the assumption that they arrive in the same order as they were sent. Otherwise the guard of operation R_i^{SS} could cause deadlock. In section 3.1.3 we will prove for refinements of this program skeleton, that the phrase 'of $phase_i$' is not necessary in the guard of operation R_i^{SS}.

Assumption 2.3. On any link, messages in any one direction are not reordered.

2.1.4. Synchronous computation.

In synchronous computation it is usually assumed that all nodes awaken simultaneously, and that messages are transmitted with a fixed delay. For our purposes however, it is enough to make the assumption that all nodes awaken spontaneously in addition to assumptions 2.2 and 2.3.

Assumption 2.4. All nodes awaken spontaneously.

In synchronous computation not only all messages of one phase are sent "at the same time", but all messages of one phase are also received "at the same time". It is usually not assumed that more messages over one link can be received at the same time, hence the information of all messages is assumed to be lumped together in one (bigger) message. Like in program skeleton SS, the operation P_i^P is included in the receive operation.

The variable rec_i is not used any more, as the registration of which messages are received in the program skeleton is replaced by a registration of which messages have arrived outside the program skeleton, and thus not specified here. If this program skeleton is used in a really synchronous environment, i.e., messages have a fixed delay, and there is a way to let all nodes start (awaken) simultaneously, then the guard of R_i^S becomes true automaticly each time the fixed delay has elapsed. The program skeleton (S) is as follows.

```
Initially   $\forall\, i$:  $awake_i = false$  and  $phase_i = 0$.

$A_i^S$ : {not $awake_i$}
      begin $awake_i := true$;
            forall neighbors $j$ do send the message belonging to $phase_i$ to $j$ od
      end

$R_i^S$ : { from every neighbor the message belonging to $phase_i$ has arrived }
      begin forall neighbors $j$ do receive the message from $j$ od;
            $phase_i := phase_i + 1$;  compute;
            forall neighbors $j$ do send the message belonging to $phase_i$ to $j$ od
      end
```

These program skeletons together with the network assumptions stated in section 2.2 form the models of computation. In sections 3 and 4 we will further refine these program skeletons. We first specify the actual computation of minimum hop distances and specify the contents of a message. We then derive properties of the classes of algorithms that can be described in this way. Finally, we refine these program skeletons further to arrive at concrete algorithms.

The method used for the correctness proofs of these refined program skeletons is that of assertional verification or system-wide invariants, first introduced by Krogdahl [Kr] and Knuth [Kn]. The idea is that if a relation (between process variables for example) holds initially, and is kept invariant by all possible operations, then it will hold always in the distributed system, whatever order of operations takes place in an actual execution of the distributed algorithm. This approach has a definite advantage over operational reasoning for correctness proofs because it is almost impossible not to overlook some odd coincidence of events that can arise in the execution of a distributed algorithm that might make the proof invalid, as it makes use of checking all possible executions.

The advantage of the use of system-wide invariants for program skeletons is the following. If we have a more elaborate program skeleton or an algorithm which can be viewed as a special instance of some program skeleton, then any system-wide invariant which holds for the program skeleton, will hold also for the more elaborate program skeleton or the algorithm. This is the case simply because the invariant was proven correct for any order of operations in the general case, and hence also for the special order of operations which will take place in the more elaborate program skeleton or the algorithm. For example, as the simulated synchronous program skeleton is a special case of the phasing program skeleton, invariants which hold in the latter hold in the former, too. The approach was used successfully in e.g. [Tel] and [Scho].

System-wide invariants are very well suited to prove the partial correctness (or safety) of a program skeleton, i.e., that all variables contain the correct values upon termination. In general they are less suited to prove total correctness (or liveness), i.e., that there is termination in finite time. That an execution cannot go on infinitely long is usually proven by means of some counting argument. Although the freedom of deadlock for some program skeleton can be stated in a system-wide invariant, this does not mean that the freedom of deadlock for this program skeleton automaticly carries over to the freedom of deadlock for a refinement of that program skeleton. As the order of operations in the latter might be more restricted because extra guards were introduced, freedom of deadlock for the refined program skeleton will correspond to a *different* system-wide invariant which will have to be proved separately.

2.2. The network. We assume we have a network consisting of nodes connected by undirected communication links. We do not assume that the network is connected. Nodes can send messages to their neighbors over links. Unless otherwise stated, we assume that nodes know (the identity of) their neighbors.

As discussed in the previous section, we assume that messages sent are not lost, garbled, duplicated, or delayed infinitely long. Thus a message sent always arrives in finite time at its destination. We also assume that messages on one link for one direction are not reordered: links have the FIFO property, i.e., a message sent first on a link, arrives first. We mean to say that the underlying communication protocol(s) ensure those properties on this level. Hence communication is modeled as follows. A link (i,j) is represented by two FIFO queues of messages $Q[i,j]$ and $Q[j,i]$: the messages from i to j and from j to i, respectively. We denote the fact that a message M is on its way from node i to node j over the link (i,j) as $<M> \in Q[i,j]$.

We discuss algorithms for both static and dynamic networks, in sections 3 and 4, respectively. In a static network, links are fixed and known to their neighbors. In a dynamic network, links can go down and come up. We use the following model for communication in dynamic networks. We assume that the assumptions stated above for links in a static network, also apply to links in a dynamic network, whether they are up or down. To model the loss of messages when a link is down, we do not change the assumptions, but add an extra atomic operation local to a link for going down which provides for the possible loss of messages. Also the send procedure provides for possible loss of messages when a link is down. As a counterpart for the operation for going down, we also add an operation for the coming up of a link.

If the link (i,j) is up, then sending a message from i to j means: appending a message to $Q[i,j]$, and receiving by i of a message from j means: deleting the first message i.e., the head of $Q[j,i]$. If the link is down, sending a message from i to j means: possibly appending the message to $Q[i,j]$. This corresponds on the one hand to the message getting lost and on the other hand to the situation that the message was still in i's output buffer when the link came up again. If the link is down and $Q[j,i]$ not empty, receiving a message just means getting the head of $Q[j,i]$, corresponding to getting the next message in i's input buffer. Thus we get the following *send* and *receive* procedures.

```
proc send <M> to j by i =  begin if linkstate(i,j) = up
                                then append <M> to Q[i,j]
                                else possibly append <M> to Q[i,j]
                                fi
                           end

proc receive <M> from j by i =  {<M> head of Q[j,i]}
                                begin delete <M> from Q[j,i] end
```

We will model nodes going down and coming up by the going down of all incident links of a node that are still up, and the coming up of a node by the coming up of possibly only some links incident to a node, respectively. It is of course necessary that nodes somehow become aware of the changed status (up or down, respectively) of an incident link. We model this by adding extra messages <*up*> and <*down*> which we call *control messages*. We assume that when a link changes status, control messages are added to the message queues corresponding to that link, and that the nodes incident to the link eventually become aware of the changed link status when they receive the control message. Thus the status of the link cannot be observed directly by the incident nodes.

When link (i,j) comes up, an <*up*> message is appended to both $Q[i,j]$ and $Q[j,i]$. When link (i,j) goes down, a <*down*> message is appended to both $Q[i,j]$ and $Q[j,i]$, and moreover, we allow that arbitrary noncontrol messages are deleted from $Q[i,j]$ and $Q[j,i]$, corresponding to the situation that some or all messages are lost when the link goes down. It is probably more realistic to only delete messages after the last <*up*> message, but it is not necessary to make this assumption. We do not allow that control messages are deleted.

Thus we get the following atomic operations U (coming up) and D (going down) on links.

```
U_ij :  {linkstate(i,j) = down}
        begin append <up> to Q[i,j];  append <up> to Q[j,i];
              linkstate(i,j) := up
        end
```

```
D_ij :  {linkstate(i, j) = up}
     begin delete all, some or no non-control messages from Q[i, j];
           delete all, some or no non-control messages from Q[j, i];
           append <down> to Q[i, j];  append <down> to Q[j, i];
           linkstate(i, j) := down
     end
```

Here *linkstate*(i,j) is a ghost variable which can have values *up* and *down*. It is not a variable which can be accessed by any node, but it is introduced to be able to formulate some invariants more precisely.

Summarizing, the assumptions for a dynamic network are:

Assumption 2.2'. Messages do not get lost otherwise than by a D-operation or a send procedure. Messages that are not lost do not get garbled, duplicated, or delayed infinitely long.

Assumption 2.5. If a link (i,j) changes status, the appropriate control messages are appended to $Q[i,j]$ and $Q[j,i]$.

Assumption 2.6. Initially, all links are down and the initial topology of the network is defined by the appropriate U_{ij} operations.

To the program skeletons from the previous section we now must add the new atomic operations RU and RD : receive an <*up*> message and receive a <*down*> message, respectively. Alternatively, we have to extend the code for the R operation with a test whether the message received is a control message and the appropriate action to take in case it is. If the computation on hand computes something which depends on the actual topology of the network, such as for example minimum hop distances, part of the computation might have become obsolete if there is a change in topology. There are two ways to deal with that: partially recompute with the problem of deciding what part, or totally restarting the computation with the problem of reinitializing. Thus in general, there will be several variables in the program skeleton that have to be changed in an RU or an RD operation.

In the phasing program skeleton of the previous section, at least the phasing number will have to be reset. In a message-driven computation we now must be aware that messages might have become lost because of link failures, contrary to assumption 2.2. Furthermore we need operations comparable to awakening which will take place upon changes in topology. These will be incorporated of course in the operations RU and RD mentioned above.

There is a more philosophical difference between a dynamic network and a static network, namely that in the latter case it is sufficient to do a computation once, while in a dynamic network the computation might have to be redone for every change in topology. This usually takes the form of a continuous computation which will only terminate when there are no more changes in topology.

3. Minimum hop distances in a static network. We begin by applying the most general program skeleton of section 2.1 to the computation of minimum hop distances. That is, we specify the contents of a message, we define variables to record computed values, and we specify the computation which has to be done. After investigating what we can derive about estimated distances which are obtained in this way, we show in section 3.1 how to arrive at minimum hop distances if we refine the program skeletons P, M, SS, and S defined in section 2.1 in this way. In section 3.2 we then discuss some algorithms due to Gallager and Friedman [Fr].

We assume that the contents of a message are the identity of some destination node together with the estimated distance between the sender of the message and the destination. So messages (denoted as $<x,l>$) consist of two fields: the destination is in the first field and the (estimated) distance of the sender of the message to the destination is in the second field. We assume each node maintains an array D with (estimated) distances to all nodes. If we want to use the minimum hop distance for routing, we also need to remember the neighbor which sent a node the estimated minimal distance. This will be maintained in *dsn* (for *downstream neighbor*- the reason for this terminology will become clear later). We then get the following program skeleton (B).

```
Initially   $\forall i$: $D_i[i] = 0$ and $\forall x$ with $x \neq i$: $D_i[x] = \infty$.

$S_i^B$ : begin send $<x, D_i[x]>$ to some neighbor $j$ end

$R_i^B$ : {a message $<x,l>$ has arrived from $j$}
      begin receive $<x,l>$ from $j$;
            if $l+1 < D_i[x]$ then $D_i[x] := l+1$; $dsn_i[x] := j$ fi
      end
```

Although it is not really necessary, we suppose for ease of notation that the set of all nodes in the (static) network is known a priori to every node.

It might seem that we have not specified much about any program that contains these atomic actions as building blocks. However, we can already derive some statements about the relation between the values in $D_i[x]$ and the real (minimum hop) distance between i and x, denoted by $d(i,x)$.

Lemma 3.1. For all x and i the following holds invariantly.

(1) $D_i[x]$ is not increasing,

(2) $<x,l> \in Q[j,i] \Rightarrow l \geq D_j[x]$,

(3) $D_i[x] \geq d(i,x)$.

Proof. (1), (2). Obvious from program skeleton B.
(3). This is initially true, as $d(i,i) = D_i[i] = 0$ and $d(i,x) \leq \infty$ for all x. If a message $<x,l>$ is sent by j, j sends its value $D_j[x]$ as l. Hence by statement (3) for j and x, $l \geq d(j,x)$ for messages sent by j. In case i receives a message $<x,l>$ from j and adjusts $D_i[x]$, we know by the triangle inequality that $d(i,x) \leq d(i,j) + d(j,x) = 1 + d(j,x)$ and thus $1 + d(j,x) \leq 1 + l = D_i[x]$. ■

Lemma 3.2. For all i and x with $x \neq i$

$D_i[x] = k < \infty \Rightarrow$ there is a path of $\leq k$ hops via $dsn_i[x]$ and $D_{dsn_i[x]}[x] \leq k-1$.

Proof. Consider operation R_i^B, and let the message received be $<x,l>$. Then $l \geq D_j[x]$ by lemma 3.1(2). If $l = \infty$, i does not change $D_i[x]$ as $1+\infty \nless D_i[x]$ always. Thus if i changes $D_i[x]$ and sets $dsn_i[x]$ to j, $l < \infty$. We have two cases.
Case 1: $l = 0$.

As $l \geq d(j,x)$ we know $d(j,x) = 0$ and hence $j = x$. Thus x is neighbor of i and there is a path of one hop to x. Then $D_i[x]$ is set to 1 and $dsn_i[x] = j$ while $D_j[x] \leq 1\text{-}1 = 0$.
Case 2: $l > 0$.
As $D_j[j] = 0$ initially and is never changed (lemma 3.1), $l > 0$ implies $x \neq j$. Hence we can use the induction hypothesis for j and conclude that there is a path of $\leq l$ hops to x from j. We have $l \geq D_j[x]$, $dsn_i[x]$ is set to j, and $D_i[x]$ to l+1 while there is a path of $\leq l+1$ hops via j to x from i. ■

Baran's *perfect learning algorithm* [Ba] is essentially identical to the above program skeleton. On messages sent for other reasons, a hop count of the distance traveled is piggybacked. This hop count is used by the receiver of the message to adjust its distance table in the manner described above. This distance table then is used in routing, without bothering whether the distances recorded are minimum distances.

3.1. Structuring the program skeleton. From the preceding section we conclude that any program which sends (estimates of) minimum hop distances to neighbors, and adjusts distances upon receipt of messages in this way, yields estimates which are upper bounds of the real minimum distances, providing the initial values were upper bounds. In general however, one would like to know when the upper bounds are equal to the minima. Clearly, one has to send "enough" values around. Thus the problem reduces to deciding for each node when it can stop sending values of distances because all nodes (including the node itself) have the correct values. The way to achieve this is to add extra structure to the program skeleton. Both with phasing and in the message-driven model we can decide when the distance tables are correct.

3.1.1. Phasing. The idea of phasing was explained in section 2.1.1. In order to distinguish messages of different phases, messages will now have a third field containing the current phase of the sender. All the work of one phase in this case constitutes of sending around those messages which contain a distance field of l = $phase_i$, and receiving all those messages from all neighbors, thus gathering all information about nodes which lie at a distance of $phase_i + 1$. We refine program skeleton P to program skeleton $P1$.

```
Initially   $\forall i$: $D_i[i] = 0$, $phase_i = 0$, and $\forall x$ with $x \neq i$: $D_i[x] = \infty$.

$S_i^{P1}$ : begin send $<x, D_i[x], phase_i>$ to some neighbor $j$ end

$R_i^{P1}$ : {a message $<x,l,p>$ has arrived from $j$}
        begin receive $<x,l,p>$ from $j$; record it as received for phase = $p$ ;
            if $l+1 < D_i[x]$ then $D_i[x] := l+1$; $dsn_i[x] := j$ fi
        end

$P_i^{P1}$ : {all messages of phase = $phase_i$ from all neighbors are received}
        begin $phase_i := phase_i + 1$ end
```

The guard of operation P_i^{P1} still is rather informally stated. It depends on the actual implementation how this statement should be formulated exactly. For example, the guard could be "from all neighbors j and about all destinations x a message of $phase_i$ is received".

We are not interested in how this could be implemented, we only assume that a node can somehow decide this question.

Note that in operation R_i^{P1} we did not test the phase number p of the received message against the phase number $phase_i$ of the node i. Whether a message is buffered until i has reached the corresponding phase or is processed directly, or even is thrown away, is immaterial to the correctness, as we will see. Thus we did not pose an extra restriction on R_i^{P1}.

As this program skeleton is also a refinement of program skeleton B, lemmas 3.1 and 3.2 still hold.

Lemma 3.3. For all i and x the following holds invariantly.
(1) $phase_i$ is not decreasing,
(2) $d(i,x)>phase_i \vee d(i,x)\geq D_i[x]$.

Proof. (1). Obvious from program skeleton $P1$.
(2). Initially $d(i,x)>phase_i = 0$ for $i \neq x$ and $d(i,i) = D_i[i] = 0$.
-Operation S_i^{P1} does not change any variables.
-Operation R_i^{P1} can only decrease $D_i[x]$, hence the statement remains valid.
-Consider operation P_i^{P1}. We only need to consider the case that $phase_i$ is increased to the value that happens to be $d(i,x)$. Thus assume $d(i,x) = k \geq 1$. Hence there is a path of k hops from i to x. Let j be the first node after i on this path. Thus $d(j,x) = k-1$. As the operation is enabled, i has received a message $<x,l,p>$ from j with $p = k-1$. Together with the induction hypothesis we have $d(j,x)>p$ or $d(j,x)\geq l$. As $d(j,x) = k-1$, we know $k-1\geq l$. In R_i^{P1} the result of receiving $<x,l,p>$ is that $D_i[x]\leq l+1$ whether or not $D_i[x]$ is adjusted. As $D_i[x]$ is not increasing, this is still the case. Hence $D_i[x]\leq l+1\leq k = d(i,x)$. ■

Now it is clear from operation R_i^{P1} that if two messages $<x,l,p>$ and $<x,l,p'>$ are received from j, that the second one has no effect whatsoever. Hence it is not necessary for j to send the second message, as long as i is able to conclude when it has received "all" messages of a phase. This could be implemented for example by letting j send the total number of messages of each phase to be expected, or sending all messages over one link inside one large message per phase. The proof of lemma 3.3 is easily adjusted for the set of atomic operations where this different definition of "all messages per phase" is used in the guard of operation P_i^{P1}. This will be discussed in more detail in section 3.1.3.

Theorem 3.4. For all i and x the following holds invariantly.
(1) $d(i,x)\leq phase_i \Rightarrow D_i[x] = d(i,x)$,
(2) $D_i[x]\leq phase_i \Rightarrow D_i[x] = d(i,x)$.

Proof. Follows directly from lemma 3.3 combined with lemma 3.1(2). ■

Thus we know on the one hand that of all nodes x lying at a distance less or equal the current phase number the correct distance is known to i, and on the other hand, that if the value that i has for the distance to x is less or equal the current phase number, then this value is correct.

Corollary 3.5. For all i the following holds invariantly.

$$phase_i > \max_x \{D_i[x] \mid D_i[x] < \infty\} \Rightarrow \forall x: D_i[x] = d(i,x).$$

Proof. With theorem 3.4 we have that for all x with $d(i,x) \le phase_i$, $D_i[x] > d(i,x)$. For those x with $d(i,x) = \infty$ we have with lemma 3.1 that $D_i[x] = d(i,x)$. Assume there is an x with $d(i,x) = k > phase_i$, and $d(i,x) < \infty$. Then there is a path of length k from i to x. Thus on this path, there must be a node y with $d(i,y) = phase_i$. With lemma 3.3 we know $d(i,y) = D_i[y] < phase_i$ which is in contradiction with the premise. Thus for all nodes x we have $D_i[x] = d(i,x)$. ■

Hence a node can decide when all its distance values are indeed correct. In an actual algorithm for computing minimum hop distances this can be used for a stop criterion in the algorithm. Note that the program skeleton as given here is too general for being able to prove that it terminates and does not deadlock or goes on generating messages forever.

3.1.2. Message-driven computation. In this case the added structure in the order of computation is that it is specified which messages are to be sent if a certain message is received. If the receipt of a message leads to an adjustment in the distance table, this "news" is sent to all other neighbors. We obtain the following program skeleton ($M1$).

```
Initially   $\forall i$: $awake_i = false$, $D_i[i] = 0$, and $\forall x$ with $x \ne i$: $D_i[x] = \infty$.

$A_i^{M1}$ :{not $awake_i$}
     begin $awake_i := true$; forall neighbors $j$ do send $<i,0>$ to $j$ od end

$R_i^{M1}$ :{a message $<x,l>$ has arrived from $j$}
     begin receive $<x,l>$ from $j$; if not $awake_i$ then do $A_i^{M1}$ fi;
          if $l+1 < D_i[x]$
          then $D_i[x] := l+1$; $dsn_i[x] := j$;
               forall neighbors $k$ with $k \ne j$ do send $<x,D_i[x]>$ to $k$ od
          fi
     end
```

As it is also true for this program skeleton that it is a refinement of skeleton B, lemmas 3.1 and 3.2 also hold now.

Lemma 3.6. For all links (i,j) and all nodes x the following holds invariantly.

$$awake_i \wedge D_i[x] < \infty \Rightarrow <x,D_i[x]> \in Q[i,j] \vee D_j[x] \le D_i[x]+1.$$

Proof. Initially the premise is false.
-Operation A_i^{M1}. When i awakens, only $D_i[i] < \infty$, but then $<i,0>$ is sent to j, hence $<i,0> \in Q[i,j]$.
-Operation R_i^{M1}. Only there is $D_i[x]$ changed. If it is, it is done upon receipt of some message $<x,l>$ from node k, where now $D_i[x] = l+1$. We distinguish two cases.
Case 1: $j = k$. With lemma 3.1 we have $l \ge D_k[x]$, thus $D_j[x] \le l \le D_i[x]+1$ holds.
Case 2: $j \ne k$. Then the message $<x,D_i[x]>$ is sent to j hence $<x,D_i[x]> \in Q[i,j]$ holds.
-Operation R_j^{M1} can falsify the statement $<x,D_i[x]> \in Q[i,j]$ by receiving that specific

message, but then the result is $D_j[x] \le D_i[x]+1$. If this last statement holds, it can not be invalidated by an R_j^{M1} operation because $D_j[x]$ is not increasing (lemma 3.1). ■

Lemma 3.7. For all nodes i and x the following holds invariantly.

$$\begin{aligned} & d(i,x) = D_i[x] \ \vee \\ & d(i,x) < D_i[x] \wedge \exists j \text{ with } d(i,j) = 1 \wedge d(i,x) = d(j,x)+1 \wedge \\ & \qquad (d(j,x) = D_j[x] \wedge (<x, D_j[x]> \in Q[j,i] \vee \textbf{not } awake_j \wedge x = j) \vee \\ & \qquad d(j,x) < D_j[x]). \end{aligned}$$

Proof. In case $d(i,x) < D_i[x]$ it follows that $i \ne x$ and $d(i,x) < \infty$. It is a property of minimum hop distances that i has such a neighbor j on a path to x. By lemma 3.1 we have $d(j,x) \le D_j[x]$. For the case of $d(j,x) = D_j[x]$ use lemma 3.6. ■

Lemma 3.8. The number of messages sent in program skeleton $M1$ is finite.

Proof. First note that in any message $<x,l>$ which is sent, $0 \le l < \infty$. Secondly, let N be the total number of nodes in the network. Then $l < N$ (use lemma 3.2). Moreover, if two messages $<x,l>$ and $<x',l'>$ are sent in the same direction over a link, then either $x \ne x'$ or $l \ne l'$. This gives the desired result. ■

Theorem 3.9. TERM $\Rightarrow$ $\forall i, x$: $D_i[x] = d(i,x)$.

Proof. Use lemmas 2.1, 3.6, and 3.7. ■

Corollary 3.10. Program skeleton $M1$ is (totally) correct.

Proof. As messages can always be received when they arrive, and delays are finite, this program skeleton terminates in finite time. ■

The only problem now left is that a node i cannot see from the values of its variables whether there is termination or not, and hence whether its distance table is correct. Unless we make further assumptions about the network such as: all messages have a bounded delay only, we need to add a so-called termination detection algorithm to be able to detect this state (see e.g. [Tel]).

3.1.3. Simulated synchronous computation.

In this model, the feature of phasing is incorporated in the message-driven model of computation, as was discussed in section 2.1.3. We obtain program skeleton $SS1$ if we refine program skeleton SS with the distance computation.

```
Initially ∀ i:   $awake_i = false$, $phase_i = 0$, $D_i[i] = 0$, and
                 ∀ $x$ with $x \ne i$: $D_i[x] = \infty$, ∀ neighbors $j$: $rec_i[j] = false$.

$A_i^{SS1}$ : {not $awake_i$}
    begin $awake_i := true$;
        forall neighbors $j$ do forall nodes $x$ do send $<x, D_i[x], phase_i>$ to $j$ od od
    end
```

```
$R_i^{SS1}$ : {a message $<x,l,p>$ has arrived from $j$ $\wedge$ not $rec_i[j]$}
    begin receive $<x,l,p>$ from $j$ ; if not $awake_i$ $\wedge$ $p = 0$ then do $A_i^{SS1}$ fi;
        if $awake_i \wedge p = phase_i$
        then if $l+1<D_i[x]$ then $D_i[x] := l+1$; $dsn_i[x] := j$ fi;
             if this was the last message from $j$ belonging to $phase_i$
             then $rec_i[j] := true$
             fi;
             if $\forall$ neighbors $k$: $rec_i[k]$
             then $phase_i := phase_i + 1$;
                  forall neighbors $k$
                  do   $rec_i[k] := false$;
                       forall nodes $x$ do send $<x,D_i[x],phase_i>$ to $k$ od
                  od; if not $\exists x$ with $D_i[x] = phase_i$ then $awake_i := false$ fi
        fi   fi
    end
```

In the above program skeleton a lot of information turns out to be redundant. Hence we proceed to prove relations between the variables involved in order to arrive at a simpler program skeleton whose correctness follows from the correctness of the skeleton above. Note that the guard of operation R_i^{SS1} does not contain the test $p = phase_i$, contrary to the situation in R_i^{SS}. The next theorem states that this test is not necessary.

Lemma 3.11. For all links (i,j), all nodes x, and all integers $k \geq 0$ the following holds invariantly.

(1) $<x',l',p'>$ behind $<x,l,p>$ in $Q[i,j] \Rightarrow p' \geq p$,

(2) $\exists <x,l,p> \in Q[i,j]$ with $p = k \Rightarrow phase_i \geq k \wedge$
$phase_j < k \vee phase_j = k \wedge rec_j[i] = false$,

(3) $(awake_i \wedge phase_i = k = 0 \vee 0<k \leq phase_i) \wedge$ not $\exists <x,l,p> \in Q[i,j]$ with $p = k \Rightarrow$
$phase_j > k \vee phase_j = k \wedge rec_j[i] = true \vee$ not $awake_j \wedge phase_j \geq 1$,

(4) (not $awake_i \wedge phase_i = k = 0 \vee k > phase_i) \Rightarrow$
not $\exists <x,l,p> \in Q[i,j]$ with $p = k \wedge$
$(phase_j < k \vee phase_j = k \wedge rec_j[i] = false)$.

Proof. (1). Follows from the FIFO property of the message queues and lemma 3.3.
We prove the remaining statements simultaneously. Initially the premise of (2) and (3) is false, and (4) holds for all k as all queues are empty. Consider the effects of the different operations upon the statements.

-Operation A_i^{SS1}. Then the premise of (2) holds for $k = 0$ and $phase_j = 0$ and $rec_j[i] = false$. The premise of (3) is *false* for $k = 0$. (4) continues to hold for $k \geq 1$.

-Operation R_i^{SS1} where $phase_i$ is increased. (2) continues to hold for $phase_i > k$ as before, and for $k = phase_i$ messages $<x,l,p>$ with $p = k$ are sent to j. As $phase_j \leq phase_i$ before R_i^{SS1}, we now have $phase_j < k$. (3) continues to hold for $phase_i > k$ as before, and the premise is *false* for $k = phase_i$. (4). The premise holds for less values of k.

-Operation A_j^{SS1}, spontaneously or on receipt of a message which was not the last one of this phase from i. Hence as a result, $phase_j = 0$ and $rec_j[i] = false$. If (2) held, it still holds. The premise of (3) is *false*. (4) continues to hold as before.

-Operation R_j^{SS1} on receipt of a message which was not the last one of this phase from i. Then all statements remain valid as before.

-Operation R_j^{SS1} on receipt of a message $<x,l,p>$ from i which was the last one of this phase. Then before $rec_j[i] = false$ (statement (2)). We have two cases.

Case 1: $phase_j$ is not increased. Then $rec_j[i] = true$ and **not** $\exists <x,l,p> \in Q[i,j]$ **with** $p = phase_j$ holds. An exception is the case where $awake_j$ was *false*, since $rec_j[i] = false$ then.

Case 2: $phase_j$ is increased. Then again $rec_j[i] = false$ and **not** $\exists <x,l,p> \in Q[i,j]$ **with** $p = phase_j$ holds. If (2) held before for values of k with $phase_j < k$, then it now holds for $phase_j \le k$ as $rec_j[i] = false$. (4) cannot have held before R_j^{SS1} for $k = phase_j$, hence (4) continues to hold now $phase_j$ is increased. If $awake_j$ is set to *false*, $phase_j$ must at least be 1 as $phase_j$ is increased first.

-Operation R_j^{SS1} on receipt of a message from another neighbor than i which results in an increase in $phase_j$. This can only happen if before R_j^{SS1} $rec_j[i] = true$ already, and hence **not** $\exists <x,l,p> \in Q[i,j]$ **with** $p = phase_j$ before R_j^{SS1}. Afterwards, $rec_j[i] = false$. The premise of (2) only could hold for $k > phase_j$, hence (2) holds afterwards. In (3), $phase_j = k$ and $rec_j[i] = true$ held before, and now $phase_j > k$ holds. In (4), $phase_j = k$ and $rec_j[i] = false$ did not hold, hence increasing $phase_j$ by 1 does not invalidate (4).

Hence these statements remain invariant under all possible operations. ■

Theorem 3.12. For all links (i,j) and all nodes x the following holds invariantly.
$<x,l,p>$ **head of** $Q[i,j] \Rightarrow$
$$p = phase_j \wedge rec_j[i] = false \vee p > phase_j \wedge (rec_j[i] = true \vee awake_j = false).$$

Proof. Follows directly from lemma 3.10 (1) and (4), lemma 3.10 (2) with $k = p$ and lemma 3.10 (3) with $k \le p - 1$. ■

Hence we can conclude that the receiver of a message $<x,l,p>$ has no need for the information what in the third field. If the receiver is *awake* and the guard is enabled, the message is of the current phase, if the receiver is not *awake*, it is clear from its own phase number (=0 or >0) whether it should awaken and start participating in the algorithm or that it has finished already. In the last case the (redundant) message can be thrown away. Thus we can omit the third field from the messages. However, there is more information that is redundant.

Lemma 3.13. For all links (i,j) and all nodes x the following holds invariantly.
(1) $D_i[x] < \infty \Rightarrow D_i[x] \le phase_i + 1$,
(2) $<x,l,p> \in Q[i,j] \Rightarrow l = \infty \vee l = d(i,x) \le p$,
(3) $<x,l,p> \in Q[i,j] \wedge l < p = phase_j \Rightarrow D_j[x] = d(j,x)$.

Proof. We prove the first two statements simultaneously. Initially both are true. If a node awakens, it sends messages of the form $<x,0,0>$ which agrees with the second statement. If in an operation R_i^{SS1} $D_i[x]$ is changed, it is done upon receipt of some message $<x,l,p>$ from say j. Then $l \ne \infty$, $p = phase_i$, and $l = d(j,x) \le phase_i$. As $D_i[x]$ is set to $l+1 = d(j,x)+1 \le phase_i + 1$. If in operation R_i^{SS1} messages are sent, then $phase_i$ was increased, too. As for those x with $D_i[x] < \infty$, $D_i[x] \le phase_i + 1$, we have that after the increase

$D_i[x] \le phase_i$ and thus $l \le p$ in the messages sent, or $l = \infty$. With theorem 3.4(2) $D_i[x] \le phase_i$ implies $l = d(i,x)$. Hence (1) and (2) remain invariant.
Statement (3) follows from the triangle inequality $d(j,x) \le d(i,x)+1$ together with (1) and theorem 3.4(1). ■

Thus the only messages $<x,l,p>$ upon the receipt of which an entry $D_i[x]$ is changed, are those with $l = p = phase_i$, and hence $D_i[x]$ is always set to $phase_i+1$. Necessarily $D_i[x]$ was ∞ beforehand. Thus the test whether to change $D_i[x]$ can be stated otherwise. Only the name of the node x in a message $<x,l,p>$ is information we need. Now it is possible to do two things: one is to just send messages which only contain the name of a node. The problem with this is that the receiver now has no way of knowing when all messages of one phase have been received, as the number of them is not fixed any more, and even might be zero. The second way is to send only one message which contains a set of node names. Thus we avoid the previous problem, introducing however messages which do not have a fixed length.

We now give the simplified program skeleton for the second way in skeleton $SS2$, for easy comparison to the synchronous program skeleton (see section 3.1.4). Now X denotes a (possibly empty) set of node names.

```
Initially ∀ $i$:    $awake_i = false$, $phase_i = 0$, $D_i[i] = 0$, and
                    ∀ $x$ with $x \ne i$: $D_i[x] = \infty$, ∀ neighbors $j$: $rec_i[j] = false$.

$A_i^{SS2}$ : {not $awake_i$}
    begin $awake_i := true$; forall neighbors $j$ do send <{$i$}> to $j$ od end

$R_i^{SS2}$ : {a message $<X>$ has arrived from $j$ $\wedge$ not $rec_i[j]$}
    begin receive $<X>$ from $j$; if not $awake_i$ $\wedge$ $phase_i = 0$ then do $A_i^{SS2}$ fi;
        if $awake_i$
        then $rec_i[j] := true$;
             forall $x \in X$
             do if $D_i[x] = \infty$ then $D_i[x] := phase_i+1$; $dsn_i[x] := j$ fi od;
             if ∀ neighbors $k$: $rec_i[k]$
             then $phase_i := phase_i+1$; $X := \{x \mid D_i[x] = phase_i\}$;
                  forall neighbors $k$ do $rec_i[k] := false$; send $<X>$ to $k$ od;
                  if $X = \emptyset$ then $awake_i := false$ fi
        fi   fi
    end
```

Even now there is some redundant information sent. This is exploited in the algorithms of Gallager and Friedman [Fr]. We refer the reader to section 3.2 for more details.

We now proceed with the issue of correctness of this program skeleton, as this does not follow directly from corollaries 3.5 and 3.10. Since we introduced extra guards it is not obvious that no deadlock can occur.

Lemma 3.14. For all links (i,j) the following holds invariantly.
$awake_i \wedge rec_i[j] = false \wedge Q[j,i] = \emptyset \quad \Rightarrow$
$phase_j < phase_i \wedge awake_j \vee phase_j = phase_i = 0 \wedge \text{not } awake_j \wedge Q[i,j] \ne \emptyset$.

Proof. Using lemma 3.11(3) with i and j interchanged and with $k = phase_i$, we get a contradiction. Hence we conclude that $phase_j < phase_i$ or **not** $awake_j \wedge phase_i = phase_j = 0$. In the case of $phase_j < phase_i$, let us assume that **not** $awake_j$ holds. It is clear from the program skeleton that this implies **not** $\exists x$ **with** $D_j[x] = phase_j$. Hence there is no node x with $d(j,x) = phase_j$ (theorem 3.4). On the other hand, $awake_i$ holds, so there is some node x with $D_i[x] = d(i,x) = phase_i > phase_j$. As $d(j,x) \geq d(i,x) - 1$ because i and j are neighbors, this leads to a contradiction and j must be *awake* still. Upon awakening, i sends a message to j, hence $Q[i,j] \neq \varnothing$. As long as j does not receive the message, $rec_j[i] = false$ and j does not awake. On the other hand, if j awakens spontaneously, $Q[j,i] = \varnothing$ is invalidated. Hence the statement holds. ■

Theorem 3.15. The program skeletons $SS1$ and $SS2$ are correct.

Proof. Lemma 3.14 implies that if at least one node in each connected component of the network awakens spontaneously, there is always some node in that connected component which can go on because its R operation is enabled. Thus no deadlock can occur. That the values in the distance tables are correct follows from corollary 3.5. As the number of messages sent is finite: at most the number of phases (bounded by the diameter of the connected component) times the number of nodes, termination is in finite time. ■

3.1.4. Synchronous computation. In synchronous computation not only all messages of one phase are sent "at the same time", but all messages of one phase are also received "at the same time". Refining program skeleton S from section 2.1.4 leads to program skeleton $S1$.

```
Initially ∀ $i$:    $awake_i = false,\ phase_i = 0,\ D_i[i] = 0$, and
                    ∀ $x$ with $x \neq i$:  $D_i[x] = \infty$.

$A_i^{S1}$ : as $A_i^{SS2}$.

$R_i^{S1}$ : {$awake_i$ and from all neighbors a message has arrived }
    begin forall neighbors $j$
          do   receive <$X$> from $j$ ;
               forall $x \in X$
               do if $D_i[x] = \infty$ then $D_i[x] := phase_i + 1$;  $dsn_i[x] := j$ fi od
          od; $phase_i := phase_i + 1$;  $X := \{x \mid D_i[x] = phase_i\}$;
          forall neighbors $j$ do send <$X$> to $j$ od;
          if $X = \varnothing$ then $awake_i := false$ fi
    end
```

Lemma 3.16. For all links (i,j) the following holds invariantly.

$$awake_i \wedge Q[j,i] = \varnothing \quad \Rightarrow$$
$$phase_j < phase_i \wedge awake_j \vee phase_j = phase_i = 0 \wedge \textbf{not } awake_j \wedge Q[i,j] \neq \varnothing.$$

Proof. Initially the relation is true.
-Operation A_i^{S1} sets $awake_i$ and $Q[i,j] \neq \varnothing$.
-Operation A_j^{S1} falsifies the premise.
-Operation R_i^{S1} is only enabled if $Q[j,i] \neq \varnothing$. If afterwards $Q[j,i] = \varnothing$, then it contained

exactly one message. With a reformulation of lemma 3.11 and theorem 3.12 for this program skeleton we know that before R_i^{S1}, $phase_i = phase_j$ held. As $phase_i$ is increased, the result is $phase_j < phase_i$. The argument that $awake_j$ holds in the first part of the conclusion is the same as in the proof of lemma 3.14.
-Operation R_j^{S1} falsifies the premise. ■

Theorem 3.17. Program skeleton $S1$ is correct.

Proof. Consider the differences with program skeleton $SS2$. In operation R_i^{S1} there is no possibility for awakening upon receipt of a message. This is not necessary because of assumption 2.4: all nodes awaken spontaneously. The partial correctness follows because this is a refinement of program skeleton $SS2$. Lemma 3.16 implies that there is always a node with a minimal phase number for which some operation is enabled: either awakening or receiving messages over all links. Hence deadlock cannot occur. As the total number of messages sent is finite, this algorithm terminates in finite time (assuming that all nodes awaken in finite time.)
■

Although program skeleton $S1$ is the straightforward refinement of program skeleton $SS2$, it should be noted that the problem we consider, namely computing minimum hop distances, is not wholly suited for a really synchronous computation. Ideally, one would not only want all nodes to start simultaneously, but also to finish simultaneously. However, it is inherent to this problem that some nodes have more work to do than others, possibly twice as much. For example, for nodes on a path, the ones at the ends of the path go on twice as long as the one(s) in the middle. Moreover, in the present formulation, there are (redundant) messages left in the queues after termination of the algorithm which is not elegant. It is no problem of course to add some code to receive and throw away the remaining messages.

3.2. Concrete algorithms.

Friedman [Fr] discussed two algorithms for finding the minimum hop distances in a static network, which can be viewed more or less as special cases of program skeleton $S1$ from the previous section. The first of these algorithms is attributed to Gallager.

3.2.1. The algorithm of Gallager.

Gallager noted that in program skeleton $S1$ for the computation of minimum hop distances, there is still redundant information sent in messages. Consider the case that node i hears about a node x first from its neighbor j. Thus a shortest path from i to x leads via j. However, in the next phase i sends the newly learned identity of x to j too, which clearly is redundant information for j.

Lemma 3.18. For all links (i,j) and all nodes x the following holds invariantly.

$$dsn_i[x] = j \Rightarrow D_j[x] = d(j,x).$$

Proof. Initially the premise is false. If $dsn_i[x]$ is set to some value j in operation R_i^{S1}, then x was included in a message from j. When j sent this message, $D_j[x] = d(j,x)$, and $D_j[x]$ is not changed any more. ■

Thus it is not necessary to include node x in the set X sent to j if $dsn_i[x] = j$. However, from the program skeleton it is not clear which neighbor will be chosen as downstream neighbor in case there are more possibilities, as the order of operation in "**forall** neighbors j **do**..." is not specified. For every possibility j, i.e., every neighbor j with minimal distance to x, it is not necessary to include x in X to j. Thus the array of single values dsn_i is changed to an array of sets of neighbors all of which lie at a minimal distance.

We now rewrite the program in these terms which results (almost) in the algorithm of Gallager as stated by Friedman [Fr].

```
Initially $\forall i$:   $awake_i = false$, $phase_i = 0$, $D_i[i] = 0$, $dsn_i[i] = \emptyset$, and
                 $\forall x$ with $x \neq i$:  $D_i[x] = \infty$ and $dsn_i[x] = \emptyset$.

$A_i^G$ : as $A_i^{S1}$.

$R_i^G$ : {$awake_i \wedge$ from all neighbors a message has arrived}
    begin forall neighbors $j$
        do   receive $<X>$ from $j$ ;
             forall $x \in X$
             do   if $D_i[x] \geq phase_i + 1$
                  then $D_i[x] := phase_i + 1$;  $dsn_i[x] := dsn_i[x] \cup \{j\}$
                  fi
        od   od; $phase_i := phase_i + 1$;
        forall neighbors $j$
        do let $X = \{x \mid D_i[x] = phase_i \wedge j \notin dsn_i[x]\}$;  send $<X>$ to $j$ od;
        if not $\exists x$ with $D_i[x] = phase_i$ then $awake_i := false$ fi
    end
```

Corollary 3.19. The algorithm of Gallager is correct.

Proof. Follows from lemma 3.18 and theorem 3.17. ■

Note that there is one difference with the algorithm of Gallager: where we have: "**forall** neighbors j **do** receive $<X>$ from j", Gallager has "receive transmissions from all *active* neighbors". However, we showed in lemma 3.14 that all neighbors are *awake* or active long enough and hence that this test is unnecessary.

As Friedman already noted (without any arguments however) this algorithm can easily be adapted for use in an asynchronous environment. It is now a straightforward exercise to incorporate the idea of the sets of downstream neighbors $dsn_i[x]$ in program skeleton $SS2$ which makes use of messages about sets of nodes.

3.2.2. The algorithm of Friedman. Friedman [Fr] observed that there is still some redundant information sent around in the previous algorithm, in the case that the network is not a bipartite graph. This means that the graph contains cycles of odd length. Consider two neighbors i and j on a cycle of odd length. This means there are nodes (at least one, say x) with the same distance to i and j: $d(i,x) = d(j,x)$. Thus in phase $d(i,x)$ i and j send the name of node x to each other, while this is no new information for them. Friedman tried to adapt the previous algorithm in such a way that this is avoided.

The way he did that, is by ensuring that information is not sent in in both directions over a link simultaneously. For each link, a HI and a LO end is defined, the number of phases is doubled, and nodes alternately send and receive from HI and LO ends. To be able to discuss the relation between the phases in the previous program skeleton and the phases in Friedman's algorithm, we will give the latter variables a new name: *Fphase*. In the program skeleton we will write in comment what would have happened to the variable *phase*, thus simplifying the formulation of invariants. Note that we cannot choose HI and LO ends such that nodes have only HI or only LO ends, because that would mean that the network is bipartite and the problem we want to avoid does not occur.

Hence we distinguish two sets hi_i and lo_i which contain those links for which i is HI and LO end, respectively. It is arbitrary how we choose HI and LO ends, as long as both ends of a link decide consistently. We assume node names are distinct and can be ordered in some way, (lexicographically for example), and we take as HI end the end with the highest node name.

Furthermore, we need some way to remember which node identities were received by LO ends while they were candidates to be sent in the next phase, as those are the identities we want to avoid to send twice. Note that they are not remembered in Gallager's algorithm, because they have a shorter path via another neighbor. We will maintain an array nnn_i (for next nearest neighbor) of sets, where neighbor $j \in nnn_i[x]$ if $d(i,x) = d(j,x)$. If we incorporate these ideas in Gallager's algorithm we get the following program skeleton. Lines differing from the algorithm as stated by Friedman himself are marked with an asterisk (*) at the beginning of the line.

The atomic operation R_i^G is now split into three different atomic operations: R_i^0, R_i^{odd}, and R_i^{even}, as the guards of these operations now differ. In R_i^0, messages must have arrived over all links to receive all of them and decide the HI and LO ends of the links. In R_i^{odd} the *Fphase* number is odd and messages from LO link ends must be awaited before they can be received. If the *Fphase* number is even, messages from HI link ends are awaited in R_i^{even}. In order to be able to state invariants about messages of a certain phase, we add in comment messages with a second field containing the current *Fphase* number.

```
Initially ∀ i:   $awake_i = false,\ Fphase_i = 0,\ D_i[i] = 0,\ dsn_i[i] = \varnothing,$
                 $hi_i = \varnothing,\ lo_i = \varnothing,$ and                    co $phase_i = 0$ co
                 $\forall x$ with $x \neq i$:  $D_i[x] = \infty,\ dsn_i[x] = \varnothing,$ and $nnn_i[x] = \varnothing.$

$A_i^F$ : as $A_i^G$                                                  co send $<\{i\},0>$ co

$R_i^0$ : $\{awake_i \wedge Fphase_i = 0 \wedge$ over all links a message has arrived$\}$
      begin forall links
            do    receive $<\{j\}>$ over the link; $D_i[j] := 1;\ dsn_i[j] := \{j\};$
                  if $j < i$ then $hi_i := hi_i \cup \{j\}$ else $lo_i := lo_i \cup \{j\}$ fi
            od; $Fphase_i := 1;$                                      co $phase_i := 1$ co
            forall $j \in hi_i$
*           do let $X = \{x \mid D_i[x] = \frac{1}{2}(Fphase_i+1) \wedge j \notin dsn_i[x]\};$ send $<X>$ to $j$
            od;                                           co send $<X, Fphase_i>$ to $j$ co
            if not $\exists x$ with $D_i[x] = \frac{1}{2}(Fphase_i+1)$ then $awake_i := false$ fi
      end
```

```
$R_i^{odd}$ : { $awake_i \wedge$ odd$(Fphase_i) \wedge$ from all $j \in lo_i$ a message has arrived }
    begin forall $j \in lo_i$
          do    receive $<X>$ from $j$ ;
                forall $x \in X$
                do    if $D_i[x] \geq \frac{1}{2}(Fphase_i + 3)$
                      then $D_i[x] := \frac{1}{2}(Fphase_i + 3)$; $dsn_i[x] := dsn_i[x] \cup \{j\}$
*                     elif $D_i[x] = \frac{1}{2}(Fphase_i + 1)$
*                     then $nnn_i[x] := nnn_i[x] \cup \{j\}$
                      fi
          od    od; $Fphase_i := Fphase_i + 1$;
          forall $j \in lo_i$
*         do    let $X = \{x \mid D_i[x] = \frac{1}{2}Fphase_i \wedge j \notin dsn_i[x] \wedge j \notin nnn_i[x]\}$;
                send $<X>$ to $j$                    co send $<X, Fphase_i>$ to $j$ co
          od
    end

$R_i^{even}$ : { $awake_i \wedge$ even$(Fphi) \wedge Fphase_i > 0 \wedge$ from all $j \in hi_i$ a message has arrived}
    begin forall $j \in hi_i$
          do    receive $<X>$ from $j$ ;
                forall $x \in X$
                do    if $D_i[x] \geq \frac{1}{2}Fphase_i + 1$
                      then $D_i[x] := \frac{1}{2}Fphase_i + 1$; $dsn_i[x] := dsn_i[x] \cup \{j\}$
                      fi
          od    od; $Fphase_i := Fphase_i + 1$;            co $phase_i := phase_i + 1$ co
          forall $j \in hi_i$
*         do let $X = \{x \mid D_i[x] = \frac{1}{2}(Fphase_i + 1) \wedge j \notin dsn_i[x]\}$; send $<X>$ to $j$
          od;                                       co send $<X, Fphase_i>$ to $j$ co
          if not $\exists x$ with $D_i[x] = \frac{1}{2}(Fphase_i + 1)$
          then forall $j \in lo_i$ do send $< \emptyset >$ to $j$ od;
                $Fphase_i := Fphase_i + 1$; $awake_i := false$
          fi
    end
```

Note that the algorithm of Friedman is not a special case of the synchronous or simulated synchronous program skeleton because in R_i^{odd} messages of $phase_i$ are received while other messages of $phase_i$ are sent afterwards in the same operation.

Lemma 3.20. For all links (i, j) and all nodes x with $x \neq i$ the following holds invariantly.

(1) $Fphase_i > 0 \Rightarrow hi_i \cup lo_i = \{\text{all links incident to } i\} \wedge hi_i \cap lo_i = \emptyset$,

(2) $Fphase_i > 0 \wedge Fphase_j > 0 \Rightarrow j \in lo_i \wedge i \in hi_j \vee j \in hi_i \wedge i \in lo_j$,

(3) $<X, f> \in Q[i, j] \Rightarrow f = 0 \vee \text{odd}(f) \wedge j \in hi_i \vee \text{even}(f) \wedge j \in lo_i$,

(4) $<X, f>$ **tail of** $Q[i, j] \Rightarrow Fphase_i = f \vee Fphase_i = f + 1$,

(5) $<X',f'>$ **behind** $<X,f>$ **in** $Q[i,j] \Rightarrow f = 0 \wedge f' = 1 \vee f' = f+2$,
(6) $<X,f>$ **head of** $Q[i,j] \Rightarrow f = Fphase_j \vee f = Fphase_j + 1$,
(7) $phase_i = \lceil \frac{1}{2} Fphase_i \rceil$,
(8) $<X,f> \in Q[i,j] \wedge x \in X \Rightarrow d(i,x) = D_i[x] = \lceil \frac{1}{2} f \rceil$,
(9) $j \in dsn_i[x] \vee j \in nnn_i[x] \Rightarrow D_j[x] = d(j,x)$,
(10) **not** $\exists <X,f> \in Q[i,j]$ **with** $f = k \wedge (k = 0 \vee \text{even}(k) \wedge j \in lo_i \vee \text{odd}(k) \wedge j \in hi_i)$
$\Rightarrow$ **not** $awake_i \wedge Fphase_i = 0 \vee Fphase_i < k \vee Fphase_j > k$.

Proof. (1), (2), and (3) follow directly from the program skeleton.
(4). Note that R_i^{odd} and R_i^{even} can only become enabled alternately. If $Fphase_i$ is increased to $f+2$ and hence to a value with the same parity as f, a new last message is sent so the relation holds for the new message.
(5). Use statements (3) and (4).
(6) follows from program skeleton F and statements (4) and (5).
(7) is clear from program skeleton F.
(8). Although this is not a refinement of program skeleton SS, we claim that it is a special case of program skeleton $P1$: Messages of $phase_i$ (i.e., about nodes x with $d(i,x) = phase_i$) are sent when $phase_i = \frac{1}{2} Fphase_i$ and when $phase_i = \frac{1}{2}(Fphase_i + 1)$. As $phase_i = \lceil \frac{1}{2} Fphase_i \rceil$, messages of $phase_i$ are indeed sent in phase $phase_i$. Furthermore we have to show that the guard of operation P_i^{P1} in program skeleton $P1$ is true when $phase_i$ is increased in this program skeleton. In operation R_i^{even} $phase_i$ is increased. With statements (2), (3), (4) and (6) we have that $<X,f>$ is received when $f = Fphase_i$. As the messages of $phase_i$ are those with $f = 2.phase_i$ and $f = 2.phase_i - 1$ they have all been received when all messages on the HI links are received in operation R_i^{even}. Thus we can use theorem 3.4.
(9). A node j is added to a set $dsn_i[x]$ or $nnn_i[x]$ only upon receipt of a message $<X,f>$ from j where $x \in X$. With (8) we have the desired result.
(10). Initially **not** $awake_i \wedge Fphase_i = 0$ holds. Operation A_i^F falsifies the premise for $k = 0$ and for the other values of k $Fphase_i < k$ now holds. In general, $Fphase_i < k$ is falsified together with the premise. If the premise is validated by the receipt of a message $<X,k>$ by j, then afterwards $Fphase_j > k$ holds, which cannot be invalidated any more. ■

Hence the second field in messages containing the *Fphase* number can indeed be deleted.

Lemma 3.21. For all links (i,j) the following holds invariantly.
(1) $awake_i \wedge Fphase_i = 0 \wedge Q[j,i] = \emptyset \Rightarrow$ **not** $awake_j \wedge Fphase_j = 0 \wedge Q[i,j] \neq \emptyset$,
(2) $awake_i \wedge \text{odd}(Fphase_i) \wedge j \in lo_i \wedge Q[j,i] = \emptyset \Rightarrow awake_j \wedge Fphase_j < Fphase_i$,
(3) $awake_i \wedge \text{even}(Fphase_i) \wedge j \in hi_i \wedge Q[j,i] = \emptyset \Rightarrow awake_j \wedge Fphase_j < Fphase_i$.

Proof. (1) follows from lemma 3.20(10).
Except for the statement $awake_j$ in the conclusion, (2) and (3) also follow from lemma 3.20(10). Let $\text{odd}(Fphase_i)$ hold. Then $awake_i$ implies that there is some node x with $d(i,x) = \frac{1}{2}(Fphase_i + 1)$ and hence also a (possibly different) node x with $d(j,x) = \frac{1}{2}(Fphase_i - 1)$. As $Fphase_j < Fphase_i$ we have $awake_j$. On the other hand, let

even($Fphase_i$) hold. Then $awake_i$ implies that there is some node x with $d(i,x) = \frac{1}{2}Fphase_i$ and hence also a (possibly different) node x with $d(j,x) = \frac{1}{2}(Fphase_i - 2)$. Hence $awake_j$ holds for $Fphase_j \le Fphase_i - 2$, and as becoming **not** *awake* is only done in even *phases*, $awake_j$ also holds for $Fphase_j = Fphase_i - 1$. ■

Theorem 3.22. The algorithm of Friedman is correct.

Proof. That the second field in messages $<X, f>$ can be deleted follows from lemma 3.20(6). The partial correctness follows from lemma 3.20. The freedom of deadlock follows from lemma 3.21. ■

Although this is the direct translation of the ideas of Friedman for his algorithm, it is by no means the same as the algorithm he stated for this purpose, which we cite now:

Step 0:	Node pairs exchange identities and choose HI and LO.
Step l:	All HI nodes broadcast to their corresponding LO neighbors,
l odd	as in Gallager's algorithm, new identities learned at **Step** l-1.
Step l:	All LO nodes broadcast to their corresponding HI neighbors
l even	new identities learned at **Step** l-2.

Termination is as in Gallager's algorithm.

It is immediately clear that this cannot be correct, as only new identities learned in even steps are sent through. Hence information received in LO link ends will never be sent through. Moreover, as the algorithm is stated now, information is still sent twice over a link, as trying out the algorithm on a cycle of length 3 shows.

4. Minimum hop distances in a dynamic network.

In a dynamic network, links can go down and come up, as can the nodes themselves. Hence the minimum hop distance between two nodes can change in time. We have specified what we exactly mean by "going down" and "coming up" in section 2.2. The precise formulation of the assumptions is important, as people tend to make slightly different assumptions and it does make a difference for the correctness proofs.

In the next section we discuss some different problems one encounters in algorithm design for dynamic networks. We then give the algorithm of Tajibnapis [Tj] for comparison and in section 4.2 we give a complete correctness proof of the algorithm of Chu [C].

4.1. Comparison with the static case.

As a dynamic network is inherently asynchronous according to our assumptions, we only have the message-driven model of computation, and the simulated synchronous model of computation, the latter being a message-driven model of computation which incorporates the idea of phasing. In both models, the partial correctness relies heavily on the facts that the $D_i[x]$ are decreasing and approximate $d(i,x)$ from above (lemma 3.1). A consequence of this is lemma 3.2, which states that a finite entry in $D_i[x]$ reflects the existence of a path from x to i. As the minimum hop distance of two nodes can increase if a link goes down the relation $D_i[x] \ge d(i,x)$ cannot be an invariant. In fact, in all algorithms that we have seen, it is the case that, as soon as a node receives information that this relation might not hold any more, it sets $D_i[x]$ to ∞.

There are basicly two extremes in adapting static algorithms for use in a dynamic network, both with their own advantages and disadvantages. One extreme is to process all new information as it comes in, discarding the old information, and sending it on immediately. Simple though this might seem, it leads to inconsistent information between network nodes, due to the asynchronicity of the network. In the case of minimum hop routing, it is easy to construct examples such that node i has node j as its downstream neighbor for routing towards x, while node j has node i as its downstream neighbor. This problem is referred to as "not loop free" in the literature. The algorithm of Tajibnapis is an example of this. The algorithm of Chu, called the predecessor algorithm by Schwartz [Sch], is an adaptation of this which avoids some loops (those of length two).

The other extreme is, to first discard all old information in the whole network, and reset all nodes to an initial state before restarting the static algorithm anew. Clearly this makes routing in the whole network impossible for some time, even in those parts of the network that are not affected. However, in combination with a minimum hop algorithm such as that of Gallager, routing is loop free. Of course, any algorithm that defers all routing until lemma 3.2 holds again, is loop free. Resetting all nodes before restarting could be done by a resynch algorithm such as Finn's [Fi], but usually deferring all routing until the whole network is reset and all routing tables refilled, is too high a price to pay. Therefore people have tried to economize on this aspect while retaining loop freedom.

One approach, suitable for algorithms which work independently for all destinations, is to restrict the resetting of the network and deferring the routing for the affected destinations only. The algorithm of Merlin and Segall [MeSe] makes use of this. Another approach is to economize on complete resynchronization and only partially resynchronize, i.e., we do not demand that before restarting the minimum hop algorithm all nodes are reset as with complete resynchronization, but only the the neighbors of the node restarting. This is done in the failsafe version of Gallager's algorithm by Toueg [To]. Due to the very special order in which routing tables are filled, it is known during the execution of the algorithm which information is new and which is old.

A basic problem with running different versions of the same algorithm in an asynchronous environment is keeping them apart. One solution which usually is not choosen, is to wait with restarting a new version of the algorithm until the previous one has terminated. A reset algorithm by Afek et al. [AAG] could be used to force termination of the minimum hop algorithm together with resetting the subnetwork where the minimum hop algorithm was still in progress. This reset algorithm was proven correct in [DrS]. The usual solution is to number the different versions of the algorithm in some way, see for example Finn [Fi]. Toueg [To], in his adaptation of Gallager's algorithm, uses Lamport's concept of logical clocks [La1]. Both numberings use numbers which are not bounded. Note that algorithms such as an extension of the Merlin-Segall algorithm [Se] which rely on Finn's [Fi] idea of sending (bounded) differences of version numbers are probably not correct. As Soloway and Humblett [SoHu] showed, this algorithm of Finn is not correct as it can generate an infinite number of restarts after all topological changes have ceased. Soloway and Humblett introduced however a new algorithm to be able to use bounded sequence numbers, based on Gallager's minimum hop algorithm. As there is an essential difference in the assumptions about the network, we do not yet know if this works in our model, too.

Finally, Jaffe and Moss [JaMo] presented an algorithm based upon both the algorithm of Merlin and Segall and the algorithm of Tajibnapis, that is still loop free. They realized that the problem of looping only occurs when distances increase. Thus they use the algorithm of Tajibnapis for distance decreases, and the algorithm of Merlin-Segall for increases. It was recently repaired and proved correct by van Haaften and van Leeuwen.

4.2. Concrete algorithms. In the sequel we give a general introduction to the Tajibnapis' algorithm, together with the program skeleton and the most important invariants for comparison with program skeleton $M1$ (section 3.1.2) and the algorithm of Chu. The latter is an extension of the former and is proven correct in sections 4.2.2 and 4.2.3.

4.2.1. The algorithm of Tajibnapis. In program skeleton $M1$, if a message $<x,l>$ is received by node i from node j, we can recompute $D_i[x]$ as the minimum of the old value of $D_i[x]$ and $l+1$. This is due to the fact that we know that the estimate l of $D_j[x]$ is decreasing, hence we can simply take the minimum. In the case that links can go down however, this l might be larger than the previous estimate which i received from j. Thus node i now cannot recompute the new minimum, unless it has stored the latest estimates from its other neighbors to use for the computation. Hence nodes keep track of which distance information was received from which neighbor.

The algorithm of Tajibnapis contains one other new feature, which requires another assumption. It is necessary that the total number of nodes in the network, or at least an upper bound of this number, is known beforehand.

Assumption 4.1. All nodes in the network know an upper bound of the total number of nodes in the network.

We will denote this number known by N. The reason that we need this number is that in the case that the network becomes disconnected, the distances between nodes become infinity. As the algorithm tends to increase the estimates of distances with one hop at a time, we need a way to "jump to the conclusion" that the distance is infinity to prevent that the program will never terminate. What is used here is the observation that if the total number of nodes is N, the largest possible finite distance between nodes is $N-1$. Thus the number N can be interpreted as ∞ in this context.

Apart from the atomic operations U_{ij} and D_{ij} from section 2.2, which reflect link (i,j) coming up or going down, respectively, we add as extra atomic actions RU_i^T: node i receives the message $<up>$ and RD_i^T: node i receives the message $<down>$. Nodes maintain a table *Dtab*, in which $Dtab_i[x,j]$ contains for every destination $x \neq i$ and every (current) neighbor j, the last distance information it received from j. Since the set of neighbors is not fixed any more, this set is maintained in $nbrs_i$. Program skeleton T is as follows.

```
Initially ∀ i:    $nbrs_i = \emptyset$, $D_i[i] = 0$,
                  ∀ j with $i \neq j$:  $linkstate(i,j) =$ down, $Q[i,j] = \emptyset$,
                                        $D_i[j] = N$, $dsn_i[j] =$ none,
                                        ∀ x with $x \neq i$:  $Dtab_i[x,j] = \infty$.
```

```
$RU_i^T$ :{an $<up>$ has arrived from $j$}
    begin receive $<up>$ from $j$ ;  add $j$ to $nbrs_i$ ;
        $Dtab_i[j,j] := 0$;  $D_i[j] := 1$;  $dsn_i[j] := j$ ;
        forall $x \in nbrs_i$ with $x \neq j$ do send $<j,1>$ to $x$ od;
        if $|nbrs_i| = 1$
        then forall $x$ with $x \neq i \wedge x \neq j$ do $D_i[x] := N$;  $dsn_i[x] := j$ od
        fi;
        forall $x$ with $x \neq i \wedge x \neq j$
        do $Dtab_i[x,j] := N$ ;  send $<x,D_i[x]>$ to $j$ od
    end

$RD_i^T$ :{a $<down>$ has arrived from $j$}
    begin receive $<down>$ from $j$ ;  delete $j$ from $nbrs_i$ ;
        forall $x$ with $x \neq i$
        do    $Dtab_i[x,j] := \infty$;
              if $nbrs_i \neq \varnothing \wedge dsn_i[x] = j$
              then $olddist := D_i[x]$;
                   choose $ndsn \in nbrs_i$ such that
                        $Dtab_i[x,ndsn] = \min_{a \in nbrs_i} Dtab_i[x,a]$;
                   $dsn_i[x] := ndsn$ ;  $D_i[x] := \min(N, 1+Dtab_i[x,ndsn])$;
                   if $olddist \neq D_i[x]$
                   then forall $a \in nbrs_i$ do send $<x,D_i[x]>$ to $a$ od
                   fi
              elif $nbrs_i = \varnothing$ then $D_i[x] := N$;  $dsn_i[x] :=$ none
              fi
        od
    end

$R_i^T$ :  {a message $<x,l>$ has arrived from $j$}
    begin receive $<x,l>$ from $j$ ;
        if $x \neq i \wedge j \in nbrs_i$
        then $Dtab_i[x,j] := l$ ;
             if $dsn_i[x] = j \vee l+1 < D_i[x]$
             then $olddist := D_i[x]$;
                  choose $ndsn \in nbrs_i$ such that
                       $Dtab_i[x,ndsn] = \min_{a \in nbrs_i} Dtab_i[x,a]$;
                  $dsn_i[x] := ndsn$ ;  $D_i[x] := \min(N, 1+Dtab_i[x,ndsn])$;
                  if $D_i[x] \neq olddist$
                  then forall $a \in nbrs_i$ do send $<x,D_i[x]>$ to $a$ od
        fi    fi    fi
    end
```

We used the symbol ∞ as well as the symbol N for infinite distances. This is to reflect the difference between "throw all information away", e.g. delete column j from the array $Dtab_i$, and "set the value to N", e.g. as initialization for an added column j of the array $Dtab_i$. However, we are not interested in an actual implementation. It also depends on the implementation whether all variables mentioned actually have to be maintained or can be deduced from other values of variables. For example, if obsolete columns of the array $Dtab_i$ are actually thrown away, $nbrs_i$ corresponds to the columns actually present in $Dtab_i$. On the other hand, if the entire column is set to the value N, $j \in nbrs_i$ if and only if $Dtab_i[j,j] = 0$.

Note that the operation A_i^{M1} "awaken" of section 3.1.2 is divided as it were over all links and incorporated in RU_i^T for one link. Part of the work done in RU_i^T corresponds to what would be done in operation R_i^T if the message received from j would be $<j,0>$. In fact operations RU_i^T and RD_i^T could be incorporated in R_i^T if $<up>$ is coded as $<j,0>$ and $<down>$ as $<j,\infty>$, and the extra code added for the extra work to be done in those special cases. We feel that this formulation would obscure the special status of these messages.

We will now state but not prove the basic invariants that lead to the partial correctness of this algorithm, for comparison with lemma 3.6 and the invariants of the algorithm of Chu.

Theorem 4.1. (Lamport [La]) For all i, j, and x with $i \neq j$, $x \neq i$, and $x \neq j$ the following holds invariantly.

(1) $linkstate(i,j) = \text{down} \Rightarrow$ $<down>$ **last control message in** $Q[j,i] \vee$
$Dtab_i[x,j] = \infty \wedge Dtab_i[j,j] = \infty$,

(2) $linkstate(i,j) = \text{up} \Rightarrow$
$<up>$ **last control message in** $Q[j,i] \vee$
$Dtab_i[j,j] = 0 \wedge (Dtab_i[x,j] = D_j[x] \vee$
$<x,D_j[x]>$ **after any control message in** $Q[j,i]$).

4.2.2. The algorithm of Chu. The problem with the previous algorithm is that if a node i receives information from its neighbor j that it is l hops away from x, i has no way to know whether this route goes through i itself. This leads to a slow propagation of distance updates in case a link has gone down. The algorithm due to Chu [C] maintains this extra information, thus maintaining a sink tree for every destination.

In the algorithm of Tajibnapis $dsn_i[x]$ is maintained, the downstream neighbor to which i should route messages for x. Now we say that j is upstream from i for destination x if i is downstream from j. If i's downstream link for x happens to go down, it is clear that we should not choose an upstream neighbor as i's new downstream neighbor, but some other neighbor. If there is no non-upstream neighbor, i sends a message $<x,N,1>$ upstream saying "help, my route to destination x is blocked" and waits until a route to x is found via another node.

Node i maintains its sink tree information in a table T_i, where $T_i[x,j] = d$ or u means: neighbor j is downstream or upstream for destination x, respectively, and $T_i[x,j] = n$ means: neighbor j is neither downstream nor upstream for destination x.

To the messages that are sent an extra field is added to convey this extra information: node i sends messages $<x,D_i[x],1>$ to j if $T_i[x,j] = d$ (j is downstream neighbor) and messages $<x,D_i[x],0>$ to j if $T_i[x,j] \neq d$. We also have to send messages in a situation where we did not do so in the previous algorithm, namely in the case that the minimum hop distance to a node x stays the same but the downstream neighbor is changed, we have to inform the old and new downstream neighbor of this change.

We now give the text of the algorithm of Chu (program skeleton C). Apart from notating it in our own way for comparison to the other program skeletons, we had to make some slight changes to be able to prove the algorithm correct. These are marked with an asterisk (*) at the beginning of the line. The assumptions concerning the model of communication are still the same, hence these operations are to be augmented with operations D_{ij} and U_{ij} for all i and $j \neq i$ from section 2.2.

```
Initially $\forall i$:   $nbrs_i = \emptyset$, $D_i[i] = 0$,
                         $\forall j$ with $i \neq j$:  $linkstate(i,j) =$ down, $Q[i,j] = \emptyset$, $D_i[j] = N$,
                                          $\forall x$ with $x \neq i$:  $Dtab_i[x,j] = \infty$, $T_i[x,j] = n$.

RD$_i^C$ : {a <$down$> has arrived from $j$}
    begin receive <$down$> from $j$ ;  delete $j$ from $nbrs_i$ ;
        forall $x$ with $x \neq i$
        do    $Dtab_i[x,j] := \infty$;
              if $nbrs_i \neq \emptyset \wedge T_i[x,j] = d$
              then $olddist := D_i[x]$;
                   if $\exists\, a \in nbrs_i$ with $T_i[x,a] \neq u$
                   then choose $ndsn \in nbrs_i$ such that $Dtab_i[x,ndsn] = \min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x,a]$;
                        $T_i[x,ndsn] := d$ ;  $D_i[x] := \min(N, 1 + Dtab_i[x,ndsn])$;
                   else choose $ndsn \in nbrs_i$ ;  $Dtab_i[x,ndsn] := N$ ;
                        $T_i[x,ndsn] := d$ ;  $D_i[x] := N$ ;
                        forall $a \in nbrs_i$ with $a \neq ndsn$
                        do $Dtab_i[x,a] := N$ ;  $T_i[x,a] := n$ od
                   fi;  send $<x, D_i[x], 1>$ to $ndsn$ ;
                   if $olddist \neq D_i[x]$
                   then forall $a \in nbrs_i$ with $a \neq ndsn$
                        do send $<x, D_i[x], 0>$ to $a$ od
                   fi
*             elif $nbrs_i = \emptyset$ then $D_i[x] := N$
              fi;  $T_i[x,j] := n$
        od
    end
```

```
$R_i^C$ :  {a message $<x, l, t>$ has arrived from $j$}
     begin receive $<x, l, t>$ from $j$ ;
*          if $x \neq i \wedge j \in nbrs_i$
           then $olddist := D_i[x]$;  let $odsn$ such that $T_i[x, odsn] = d$ ;
                if $t = 1$
                then $Dtab_i[x, j] := l$;  $T_i[x, j] := u$ ;
                     if $olddist < N \wedge odsn = j$
                     then if $\exists\ a \in nbrs_i$ with $T_i[x, a] \neq u$
                          then choose $ndsn \in nbrs_i$ such that $Dtab_i[x, ndsn] = \min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x, a]$;
                               $T_i[x, ndsn] := d$;  $D_i[x] := \min(N, 1 + Dtab_i[x, ndsn])$
                          elif $|nbrs_i| > 1$
                          then choose $ndsn \in nbrs_i$ with $ndsn \neq j$;  $D_i[x] := N$ ;
                               $Dtab_i[x, ndsn] := N$;  $T_i[x, ndsn] := d$ ;
                               forall $a \in nbrs_i$ with $a \neq ndsn$
                               do $Dtab_i[x, a] := N$;  $T_i[x, a] := n$ od
                          else $Dtab_i[x, j] := N$;  $T_i[x, j] := d$ ;
                               $D_i[x] := N$;  $ndsn := j$
                          fi;  send $<x, D_i[x], 1>$ to $ndsn$ ;
                          if $D_i[x] \neq olddist$
                          then forall $a \in nbrs_i$ with $a \neq ndsn$
                               do send $<x, D_i[x], 0>$ to $a$ od
                          elif $j \neq ndsn$ then send $<x, D_i[x], 0>$ to $j$
                          fi
*                    elif $odsn = j$ then $T_i[x, j] := d$;  $Dtab_i[x, j] := N$
                     fi
                else $Dtab_i[x, j] := l$;  $T_i[x, j] := n$ ;
                     choose $ndsn \in nbrs_i$ such that $Dtab_i[x, ndsn] = \min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x, a]$;
*                    if $Dtab_i[x, ndsn] = Dtab_i[x, odsn]$ then $ndsn := odsn$ fi;
                     $T_i[x, ndsn] := d$;  $D_i[x] := \min(N, 1 + Dtab_i[x, ndsn])$;
                     if $odsn \neq ndsn$
                     then $T_i[x, odsn] := n$;  send $<x, D_i[x], 1>$ to $ndsn$
                     fi;
                     if $D_i[x] \neq olddist$
                     then forall $a \in nbrs_i$ with $a \neq ndsn$ do send $<x, D_i[x], 0>$ to $a$ od
                     elif $odsn \neq ndsn$ then send $<x, D_i[x], 0>$ to $odsn$
           fi   fi   fi
      end
```

```
RU$_i^C$ : {an $<up>$ has arrived from $j$}
        begin receive $<up>$ from $j$ ; add $j$ to $nbrs_i$ ;
*               if $|nbrs_i| > 1$ then let $odsn$ such that $T_i[j,odsn] = d$ ; $T_i[j,odsn] := n$ fi;
                $T_i[j,j] := d$ ; $Dtab_i[j,j] := 0$; $D_i[j] := 1$;
                forall $x \in nbrs_i$ with $x \neq j$ do send $<j,1,0>$ to $x$ od;
                forall $x$ with $x \neq i \wedge x \neq j$
                do      $Dtab_i[x,j] := N$ ;
*                       if $|nbrs_i| = 1$
*                       then $T_i[x,j] := d$ ; $D_i[x] := N$ ; send $<x,D_i[x],1>$ to $j$
*                       else $T_i[x,j] := n$ ; send $<x,D_i[x],0>$ to $j$
                        fi
                od
        end
```

4.2.2.1. Partial correctness. For the partial correctness we begin with proving some technical lemmas.

Lemma 4.2. For all i and j with $i \neq j$ the following holds invariantly.

(1) $linkstate(i,j) =$ up $\Leftrightarrow j \in nbrs_i \vee <up>$ **last control message in** $Q[j,i]$,

(2) $linkstate(i,j) =$ down $\Leftrightarrow j \notin nbrs_i \vee <down>$ **last control message in** $Q[j,i]$.

Proof. Obvious from operations U_{ij}, D_{ij}, RU$_i^C$, and RD$_i^C$. ■

Lemma 4.3. For all i, j, and x with $j \neq i$ and $x \neq i$, the following holds invariantly.

(1) $T_i[x,j] = d \vee T_i[x,j] = u \Rightarrow j \in nbrs_i$,

(2) $\exists! j$ **with** $T_i[x,j] = u \Leftrightarrow nbrs_i \neq \emptyset$,

(3) $j \in nbrs_i \Leftrightarrow Dtab_i[j,j] = 0$,

(4) $T_i[x,j] = d \Rightarrow$

$$D_i[x] = \min(N, 1 + Dtab_i[x,j]) \wedge Dtab_i[x,j] = \min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x,a],$$

(5) $D_i[i] = 0 \wedge D_i[x] > 0$,

(6) $<x,l,t> \in Q[j,i] \Rightarrow l > 0 \wedge x \neq j$,

(7) $x \neq j \Rightarrow Dtab_i[x,j] > 0$.

Proof. (1) and (2) are obvious from operations RU$_i^C$, RD$_i^C$, and R$_i^C$.

(3). Use statement (6) with operation R$_i^C$.

(4). Obvious from operations RU$_i^C$, RD$_i^C$, and R$_i^C$.

(5). $D_i[i] = 0$ initially and is never changed anymore. For $nbrs_i = \emptyset$ we have $D_i[x] = N$, otherwise statement (4) can be used together with statement (7).

(6). As $<x,l,t> \in Q[j,i]$ is only validated when j sends a message to i and j only sends messages with $x \neq j$ and $l = D_j[x]$, we have $l > 0$ with statement (5).

(7). For $x \neq j$, $Dtab_i[x,j]$ entries are only set to ∞, N, or to some value l from a received message $<x,l,t>$. With statement (6) we have $l > 0$. ■

Lemma 4.4. For all i, j, and x with $i \neq j$ the following holds invariantly.
linkstate(i,j) = down $\vee$ (1)
$<up>$ **last control message in** $Q[i,j] \vee$ (2)
$<x,l,t> \in Q[j,i] \Rightarrow$ (3)
the last message $<a,b,c> \in Q[j,i]$ with $a = x$ has $b = D_j[x] \wedge$
$(c = 1 \Leftrightarrow T_j[x,i] = d) \wedge <a,b,c>$ **after any control message in** $Q[j,i]$. (4)

Proof. Initially (1) holds. If (1) holds, it can be invalidated by operation U_{ij}, but then (2) will hold. If (2) holds, it can be invalidated by D_{ij}, but then (1) will hold. (2) can also be invalidated by operation RU_j^C, but then for those x that (3) is validated, (4) is validated for the same message. If (4) holds and is invalidated by operation R_i^C, then (3) is also invalidated. If (4) holds and is invalidated because j changes $D_j[x]$ or $T_j[x,i]$, we have the following cases.
Case 1: $i \notin nbrs_j$. Then (1) or (2) holds (lemma 4.1).
Case 2: $i \in nbrs_j$. Then a message is sent such that (4) holds. If (4) is invalidated because a control message is placed in $Q[j,i]$, then (1) or (2) now holds. If neither (1), (2) nor (3) holds for some x and a message is sent such that (3) is validated, this message is such that (4) holds for this message. ■

Lemma 4.5. For all i, j, and x with $i \neq j$ and $i \neq x$ the following holds invariantly.
linkstate(i,j) = down $\Rightarrow$
$Dtab_i[x,j] = \infty \wedge$ **no control message in** $Q[j,i] \vee$
$<down>$ **last control message in** $Q[j,i]$.

Proof. -Operation D_{ij} validates the premiss and places a $<down>$ in $Q[j,i]$ as last control message.
-Operation U_{ij} invalidates the premiss.
-Operation RU_i^C can only occur if $<down> \in Q[j,i]$ as last control message, if the premiss is true.
-If RD_i^C receives the last $<down>$ left in $Q[j,i]$, there will be no control message left and $Dtab_i[x,j]$ is set to ∞.
Other operations do not influence the variables involved. ■

Lemma 4.6. For all i, j, and x with $i \neq j$, $i \neq x$ and $j \neq x$, the following holds invariantly.
linkstate(i,j) = up $\Rightarrow$
$<up>$ **last control message in** $Q[i,j] \vee$ (1)
$<x,D_j[x],t>$ **after any control message in** $Q[j,i]$ **with** $(t = 1 \Leftrightarrow T_j[x,i] = d) \vee$ (2)
$Dtab_i[x,j] = N \wedge T_j[x,i] = d \wedge$ **no control message in** $Q[j,i] \wedge$
$<x,N,t>$ **after any control message in** $Q[i,j] \vee$ (3)
$Dtab_i[x,j] = D_j[x] \wedge$ **no control message in** $Q[j,i] \wedge$
$(T_i[x,j] = u \Rightarrow T_j[x,i] = d)$. (4)

Proof. -Operation D_{ij} invalidates the premiss.
-Operation U_{ij} validates the premiss and (1).
-Operation RU_j^C can only be performed if (1) held. Afterwards (1) can still hold, or otherwise i is added to $nbrs_j$ and $<x,D_j[x],t>$ is sent to i. As $Q[j,i]$ is a queue, the message is placed after any control message. Hence (2) holds now.

-Operation RU_i^C. If (1) holds, it will still hold afterwards. If (2) holds, it will still hold afterwards, since the $<up>$ received occurred before the message under consideration. (3) nor (4) could hold beforehand.
-Operation RD_j^C. Either the premiss is not true or (1) continues to hold.
-Operation RD_i^C. (3) nor (4) could hold beforehand. (1) and (2) remain to hold.
-Operation R_i^C with a message from j. If (1) held, it still holds. If (2) held, it either still holds, or the message received is $<x, D_j[x], t>$ with $t = 1 \Leftrightarrow T_j[x,i] = d$. In the latter case we know that there can be no control message in $Q[j,i]$. R_i sets $Dtab_i[x,j] = D_j[x]$ and $T_i[x,j] = u \Leftrightarrow T_j[x,i] = d$, so (4) holds now, except in the very special case that $t = 1$ (hence $T_j[x,i] = d$), $olddist < N$, $odsn = j$, and $\forall a \in nbrs_i\ T_i[x,a] = u$, in which case $Dtab_i[x,j]$ is set to N, $T_i[x,j]$ to n or d, and $D_i[x]$ to N, and $<x, N, t>$ is sent to j. Hence (3) holds now. The other exception is the case that $D_j[x] < N$, $t = 1$, $olddist = N$, and $odsn = j$, where no messages are sent but $Dtab_i[x,j]$ is reset to N and $T_i[x,j]$ to d. (Otherwise there would be no downstream neighbor left for x.) For this case, we not only assume that lemma 4.5 held before operation R_i^C, but also that lemma 4.5 held with i and j interchanged. Let the statements (1) to (4) with i and j interchanged be statements (1') to (4'), respectively. We then know that (1') cannot hold, as $Q[j,i]$ contains no control messages. As $olddist = N$, $D_i[x] = N$, and $odsn = j$, we have $T_i[x,j] = d$. The message received was $<x, D_j[x], 1>$ with $D_j[x] < N$ and $T_j[x,i] = d$, hence $Dtab_j[x,i] < N$. Thus neither (3') nor (4') can hold, and (2') must hold. Thus $<x, N, t> \in Q[i,j]$. Since operation R_i^C cannot invalidate this, (3) holds now. If (3) or (4) held, we know with lemma 4.3 that (2) holds also.
-Operation R_i^C with a message from $k \neq j$. (1), (2), and (3) remain to hold. If (4) held, it either remains to hold or $Dtab_i[x,j]$ can be set to N and $T_i[x,j]$ to n or d in the case that $T_i[x,a] = u\ \forall a \in nbrs_i$, $olddist < N$ and $odsn = k$. Hence we can conclude $T_j[x,i] = d$ and $<x, N, t>$ is sent to j. Thus (3) holds now.
-Operation R_j^C with a message from i. (1) remains to hold. If (2) holds, and $D_j[x]$ and/or $T_j[x,i]$ change, a new message is sent which reflects these changes. Thus (2) now holds for this new message. If (3) holds, we have the following two cases.
Case 1: only $T_j[x,i]$ changes, then a message to reflect this change is sent to i and (2) holds now.
Case 2: the message $<x, N, t>$ is received. If (4) did not hold, we had $D_j[x] < N$ and $T_j[x,i] = d$. $Dtab_i[x,j]$ is set to N. Thus either $D_j[x]$ or $T_j[x,i]$ changes so a message is sent to i to this effect and (2) now holds.
If (4) holds, it continues to hold unless $D_j[x]$ or $T_j[x,i]$ is changed. In this case a message is sent to i so (2) holds.
-Operation R_j^C with a message from $k \neq i$. (1) remains to hold. (2) remains to hold unless $D_j[x]$ or $T_j[x,i]$ are changed, in which case (2) will hold for the new message sent. In case (3) holds, only $T_j[x,i]$ could be changed, but then a message will be sent such that (2) holds. If (4) holds, it will continue to hold unless (2) holds with changed $D_j[x]$ or $T_j[x,i]$. ■

Lemma 4.7. For all i and j with $i \neq j$ the following holds invariantly.
linkstate$(i,j) =$ up $\Rightarrow$ $<up>$ **last control message in** $Q[j,i] \vee$
$Dtab_i[j,j] = 0 \wedge T_i[j,j] = d \wedge D_i[j] = 1 \wedge$ **no control message in** $Q[j,i]$.

Proof. If the last $<up>$ is received in RU_i^C, the variables specified are set to the right values. With lemma 4.2(7) we have that $Dtab_i[j,k]$ for $k \neq j$ is > 0, hence $T_i[j,j]$ does not change. Node j does not send any message about destination j (lemma 4.2(6)), hence $Dtab_i[j,j]$ is not changed. As $T_i[j,j] \neq u$, $Dtab_i[j,j]$ cannot be set to N either. ■

Lemma 4.8. For all i, j, and x with $i \neq j$ and $i \neq x$ the following holds invariantly.

$$T_i[x,j] = u \wedge Q[i,j] = \varnothing \wedge Q[j,i] = \varnothing \Rightarrow Dtab_i[x,j] = \min(N, D_i[x]+1).$$

Proof. Use lemma 4.2(1), lemma 4.1, lemma 4.5 and lemma 4.2 (4). ■

Lemma 4.9.

$$\begin{array}{lll} \text{TERM} \Rightarrow \forall i,j \text{ with } i \neq j: & linkstate(i,j) = \text{up} \Leftrightarrow j \in nbrs_i \wedge \\ & j \in nbrs_i \Rightarrow \forall x \neq i \;\; Dtab_i[x,j] = D_j[x] \wedge \\ & j \notin nbrs_i \Rightarrow \forall x \neq i \;\; Dtab_i[x,j] = \infty \wedge \\ & \forall x \neq i \;\; D_i[x] = \min(N, 1+\min\limits_{\forall j \neq i} Dtab_i[x,j]). \end{array}$$

Proof. TERM is equivalent with all queues are empty. Use lemmas 4.1, 4.2(4), 4.4, 4.5, 4.6 and 4.7. ■

Theorem 4.10. TERM $\Rightarrow \forall i,x$: $D_i[x] = \min(N, d(i,x))$.

Proof. We prove this by first proving that $D_i[x] \leq d(i,x)$ and secondly proving that $D_i[x] \geq \min(N, d(i,x))$.

(1). Let $d(i,x) = \infty$. Then for all possible values of $D_i[x]$, we have $D_i[x] \leq d(i,x)$. Let $d(i,x) = k < \infty$. $k = 0$ implies $i = x$ and $D_i[x] = 0$. For $k > 0$ there is some path $x = x_0$, x_1, ..., $x_k = i$ from x to i of length k. Thus all links (x_j, x_{j-1}) for $1 \leq j \leq k$ are up and $x_{j-1} \in nbrs_{x_j}$. Thus $Dtab_{x_j}[x, x_{j-1}] = D_{x_{j-1}}[x]$ and $D_{x_j}[x] \leq \min(N, 1+D_{x_{j-1}}[x])$. Hence $D_i[x] = D_{x_k}[x] \leq D_{x_0}[x]+k = D_x[x]+k = k$. Note that this is not necessarily the path designated by the downstream neighbors.

(2). We use induction over k for the hypothesis $d(i,j) \geq k \Rightarrow D_i[j] \geq \min(N,k)$. For $k = 0$ we have $d(i,j) \geq 0$ and $D_i[j] \geq 0$. Assume $d(u,v) \geq k+1$. For all neighbors $a \in nbrs_u$ we have $d(u,a) = 1$ and thus $d(v,a) \geq k$ (triangle inequality). Hence $D_a[v] \geq \min(N,k)$. $d_u[v] = \min(N, 1+\min\limits_{a \in nbrs_u} Dtab_u[v,a]) = \min(N, 1+\min\limits_{a \in nbrs_u} D_a[v]) \geq \min(N, k+1)$. ■

Corollary 4.11. If we interpret N as ∞, then TERM implies $D_i[x] = d(i,x)$ for all i and x.

Proof. The longest finite distance between two nodes is at most $N-1$. Hence $d(i,x) \geq N$ implies $d(i,x) = \infty$. ■

This completes the proof of the partial correctness of the algorithm of Chu.

4.2.2.2. Total correctness. For the total correctness, we still have to prove that if there are no more topological changes, the algorithm indeed terminates in finite time.

Theorem 4.12. The algorithm cannot deadlock.

Proof. If there is a queue which contains a message, then there is always an operation which can receive that message: either RU_i^C, RD_i^C, or R_i^C for $Q[j,i]$, depending on the nature of the message. If all queues are empty, there is termination by definition. ■

Thus we have to show that the algorithm cannot go on generating messages forever, in the case that there are no more topological changes after some time. For this purpose we define a function F of the system state to the set W of N+1 tuples of nonnegative integers. We define the following total ordering $<_W$ on W :

$(a_0, a_1, \ldots, a_N) <_W (b_0, b_1, \ldots, b_N)$ if

$\exists i$ with $0 \le i \le N$: $(a_i < b_i \wedge \forall j$ with $0 \le j < i$: $a_j = b_j)$.

As the a_i and b_i are nonnegative integers, this order relation on W is well-founded (i.e., there is no infinite decreasing chain). Thus, in order to prove the total correctness, it is sufficient to find a function F from the system state to W which is decreased by every operation if there are no more topological changes. We define F as follows:

$F = (cm, m(1)+2d(1), \ldots, m(N-1)+2d(N-1), 2m(N)+d(N))$,

where

cm = the total number of control messages ($<up>$ or $<down>$) in all message queues,

$m(k) = \sum_x m_x(k)$,

$d(k) = \sum_x d_x(k)$,

$m_x(k)$= the total number of messages $<y,l,t>$ with $x = y$ and $l = k$ in all message queues,

$d_x(k) = \sum_i \left[1 + | \{ j \mid j \in nbrs_i \wedge D_i[x] = k \wedge Dtab_i[x,j] = Dtab_i[x,dsn] \} | \right]$,

dsn = downstream neighbor, i.e., $T_i[x,dsn] = d$.

Theorem 4.13. For all i, F is strictly decreased by operations RU_i^C, RD_i^C, and R_i^C.

Proof. RU_i^C decreases cm by one, as does RD_i^C.

Consider operation R_i^C. Let the received message be $<x,l,t>$. cm cannot be changed, nor $m_y(k)$ or $d_y(k)$ with $y \ne x$. Note that R_i^C cannot change the set $nbrs_i$. As it depends on the old and new values of $D_i[x]$ and the old and new downstream neighbor of i for x how F changes, we define for the moment *olddist* as the value of $D_i[x]$ before operation R_i^C, *newdist* as the value of $D_i[x]$ after operation R_i^C, and *odsn* and *ndsn* as the neighbor j of i with $T_i[x,j] = d$ before and after operation R_i^C, respectively. We distinguish the following cases.

Case 1: $olddist < newdist$. Then $d(olddist)$ decreases, and $d(newdist)$ and $m(newdist)$ increase. Hence F decreases.

Case 2: $olddist > newdist$. This is only possible if in the received message $<x,l,t>$ $t = 0$ and $l = newdist-1$. Thus $m(newdist-1)$ decreases, while $m(newdist)$ and $d(newdist)$ increase.

Case 3: $olddist = newdist$.

Case 3.1: $olddist = newdist < N$.

Case 3.1.1: $odsn = ndsn$. Then $m(newdist)$ is not increased because no new messages are sent. However, $d(newdist)$ could increase, if there is now one more neighbor with minimal distance to x. This can only happen if the received message had $l = newdist-1$. Hence $m(newdist-1)$ is decreased and F decreases. If $d(newdist)$ does not change, F decreases because $m(l)$ decreases.

Case 3.1.2: $odsn \ne ndsn$. Then $m(newdist)$ is increased by 2. If $d(newdist)$ increases, we have $l = newdist-1$ as above, and $m(newdist-1)$ decreases. If $d(newdist)$ does not change, it must be the case that $Dtab_i[x, ndsn] = Dtab_i[x, odsn]$, as R_i^C can only change one value of $Dtab_i$ (unless $newdist = N$) at the time, hence this case cannot occur. Thus $d(newdist)$ decreases by one. Hence $2d(newdist)+m(newdist)$ does not change, and as $m(l)$ decreases, so does F.

Case 3.2: $olddist = newdist = N$.

Case 3.2.1: $odsn = ndsn$. Thus no new messages are sent. However, $d(newdist)$ could change. If $t = 1$ in the received message, then $d(newdist)$ can only decrease. $d(newdist)$ can increase by one if the message received from j was $<x, N, 0>$ and $T_i[x, j] = u$ before R_i^C. Then $m(newdist)$ decreases by one and F is decreased because $2m_x(N)+d_x(N)$ is decreased.

Case 3.2.2: $odsn \neq ndsn$. If $olddist = N$ and $t = 1$, nothing happens in R_i^C, so $odsn = ndsn$. Hence $t = 0$. If $l < N$, then we would have $newdist < N$, thus $l = N$. So $d(newdist)$ cannot decrease, which implies $Dtab_i[x, odsn]$ remains N. Hence the downstream neighbor is not changed, and we conclude that this case cannot occur. ■

Corollary 4.14. If topological changes cease then the algorithm of Chu terminates in finite time.

5. References.

[AAG] Afek, Y., B. Awerbuch, and E. Gafni, *Applying static network protocols to dynamic networks,* FOCS 87.

[Ba] Baran, P., *On distributed communications networks,* IEEE Trans. Commun. Syst., CS-12 (1964), 1-9.

[C] Chu, K., *A distributed protocol for updating network topology,* Report RC7235, IBM T.J.Watson Research Center, Yorktown Heights, 1978.

[DrS] Drost, N.J., and A.A. Schoone, *Assertional verification of a reset algorithm,* Techn. Report RUU-CS-88-5, Dept. of Computer Science, University of Utrecht, Utrecht, 1988.

[Fi] Finn, S.G., *Resynch procedures and a failsafe network protocol,* IEEE Trans. on Comm., COM-27 (1979), 840-845.

[Fr] Friedman, D.U., *Communication complexity of distributed shortest path algorithms,* MIT thesis, 1978.

[JaMo] Jaffe, J.M., and F.H. Moss, *A responsive distributed routing algorithm for computer networks,* Report RC8479, IBM T.J.Watson Research Center, Yorktown Heights, 1980.

[Kn] Knuth, D.E., *Verification of link-level protocols,* BIT 21 (1981), 31-36.

[Kr] Krogdahl, S., *Verification of a class of link-level protocols,* BIT 18 (1978), 436-448.

[La] Lamport, L., *An assertional correctness proof of a distributed algorithm,* Science of Computer Programming 2 (1982), 175-206.

[La1] Lamport, L., *Time, clocks, and the ordering of events in a distributed system,* Comm. ACM 21 (1978), 558-565.

[MeSe] Merlin, P.M., and A. Segall, *A failsafe distributed routing protocol,* IEEE Trans. on Comm. COM-27 (1979), 1280-1287.

[Sch] Schwartz, M., *Routing and flow control in data networks,* Report RC8353, IBM

T.J.Watson Research Center, Yorktown Heights, 1980.

[Scho] Schoone, A.A., *Verification of Connection-Management Protocols*, Techn. Report RUU-CS-87-14, Dept. of Computer Science, University of Utrecht, Utrecht, 1987.

[Se] Segall, A., *Advances in verifiable fail-safe routing procedures*, IEEE Trans. on Comm. COM-29 (1981), 491-497.

[SoHu] Soloway, F.R., and P.A. Humblett, *On distributed network protocols for changing topologies*, Techn. Report LIDS-P-1564, MIT, Cambridge, Mass., 1986.

[Ta] Tanenbaum, A.S., *Computer Networks*, Prentice-hall, Englewood Cliffs, 1981, pp. 209-210.

[Tel] Tel, G., *The structure of distributed algorithms*, Ph.D. thesis, Dept. of Computer Science, University of Utrecht, Utrecht, 1989.

[Tj] Tajibnapis, W.D., *A correctness proof of a topology information maintenance protocol for a distributed computer network*, Comm. ACM 20 (1977), 477-485.

[To] Toueg, S., *A minimum hop path failsafe and loop-free distributed algorithm*, Report RC8530, IBM T.J.Watson Research Center, Yorktown Heights, 1980.

Minimum hop route maintenance in static and dynamic networks

A.A. Schoone

RUU-CS-90-18
April 1990

Minimum hop route maintenance in static and dynamic networks

A.A. Schoone

Technical Report RUU-CS-90-18
April 1990

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

MINIMUM HOP ROUTE MAINTENANCE IN STATIC AND DYNAMIC NETWORKS[*]

Anneke A. Schoone

Department of Computer Science, University of Utrecht,
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract. We discuss distributed algorithms for computing minimum hop distances in a network from a general viewpoint and apply this to minimum hop routing. We show how different models of computation lead to different algorithms known from the literature. We then discuss the effect of various network assumptions (static, dynamic) upon minimum hop route maintenance. Using the Krogdahl-Knuth technique of system-wide invariants, we prove the following distributed algorithms correct: the minimum hop route determination algorithms of Gallager and Friedman for static networks and the route maintenance algorithm of Chu for dynamic networks.

1. Introduction. A basic problem that must be addressed in any design of a distributed network is the routing of messages. That is, if some node in the network decides it wants to send a message to some other node in the network or receives a message destined for some other node, a method is needed to enable this node to decide over which outgoing link it has to send this message. Algorithms for this problem are called *routing algorithms*. In the sequel we will only consider *distributed* routing algorithms which depend on the cooperative behavior of the local routing protocols of the nodes to guarantee effective message handling and delivery.

Desirable properties of routing algorithms are for example correctness, optimality, and robustness. *Correctness* seems easy to achieve in a static network, but the problem is far less trivial in case links and nodes are allowed to go down and come up like they tend to do in practice. *Optimality* is concerned with finding the "quickest" routes. Ideally, a route should be chosen for a message on which it will encounter the least delay but, as this depends on the amount of traffic on the way, this is hardly to foresee and hence is actually difficult to achieve as well. A frequent compromise is to minimize the number of *hops*, i.e., the number of links over which the message travels from sender to destination. We will restrict our study to *minimum hop routing*. *Robustness* is concerned with the ease with which the routing scheme is adapted in case of topological changes.

Our aim in this paper is twofold. First we present a systematic development of a number of distributed algorithms for minimum hop route determination and maintenance, including a re-appraisal of several existing methods for the static case and a detailed analysis

[*]This research was supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (Project ALCOM).

of some dynamic algorithms. Secondly, we present correctness proofs for these algorithms, which tends to be hard for distributed algorithms and indeed was not presented before in most cases that we consider. This applies in particular to the interesting distributed algorithm for minimum hop route maintenance due to Chu [C] (see also [Ta] and [Sch]), for which we will develop both a complete program skeleton and a correctness proof. In all cases we employ the Krogdahl-Knuth method of system-wide invariants.

The paper is organized as follows. In section 2 we state the assumptions about the networks that we consider and define the concept of a *program skeleton*. Different models of computation are then given. In the remainder we first concentrate on computing minimum hop distances in a static network (section 3). We begin by deriving general properties of distributed minimum hop algorithms, and then discuss different ways -related to different models of computation- for deciding whether an estimate of a minimum hop distance is correct. We conclude this section with the correctness proof of the minimum hop route determination algorithms of Gallager and Friedman [Fr] (section 3.2).

In section 4 we consider the problem of maintaining routes in dynamic networks, i.e., networks with links going down and coming up. A global acquaintance with the contents of sections 3.1 and 3.2.1 is assumed here. In section 4.1 we discuss some typical problems in adapting distributed algorithms for static networks to distributed algorithms for dynamic networks, both in general and specifically for minimum hop route maintenance. We present the well-known dynamic routing algorithm of Tajibnapis [Tj, La] for comparison, and in section 4.2 the dynamic routing algorithm of Chu [C]. The latter was presented as an improvement of the algorithm of Tajibnapis by Tanenbaum [Ta] and Schwartz [Sch] but, while Tajibnapis' algorithm has been proved correct by Lamport [La], it is not clear at all that the algorithm of Chu is correct. Moreover, the presentation of Chu's algorithm in the original report is very imprecise. In section 4.2 we give a complete specification of Chu's algorithm and a correctness proof.

2. Models of computation.

We are interested in the relation between a distributed algorithm and the *model of computation* that is used for its formulation. Generally speaking, a model of computation can be viewed as a set of assumptions and restrictions about the nature of the distributed computation in the network and the communication which takes place between the network nodes.

These assumptions are not meant as a criterion for matching specific networks. Rather, it is a way to focus attention on some aspects of distributed computing, while abstracting away from others. For example, an assumption that is made in this paper is that a link between two network nodes behaves like two FIFO queues of messages, one queue for each direction. This does not mean that we restrict ourselves to networks where this is indeed the case, but that we assume that this can be achieved by communication protocols present in the network, and that the question how to achieve it is not our concern now.

The restrictions about the distributed computation and the communication between the network nodes are restrictions in the order of events that are permitted in an actual execution. Such a permissible order of events is stated by means of a *program skeleton*, which can be refined to an algorithm. Thus a program skeleton stands for a class of algorithms, all of which can be obtained from that same skeleton by some refinement. The restriction in the order of events specified in a program skeleton can add extra structure to the computation at hand, i.e., the computation of minimum hop distances in this case, from which extra information can be derived that is exploited in actual algorithms. We investigate the interaction between the computation per se and the model of computation used. We define some program skeletons together with their assumptions in section 2.1. How these assumptions are modeled in the communication network is discussed in section 2.2.

2.1. Program skeletons. For an appraisal of the general features of distributed programs that compute minimum hop distances, we start by considering the essential building blocks and only use complete programs as illustrations. We do this by means of so-called *program skeletons,* which are generic descriptions for classes of algorithms all of which have some underlying structure in common. A program skeleton consists of a number of *operations,* each of which consists of a piece of program. Operations can be carried out any number of times, by any processor, and at any time. An operation is viewed as an atomic action, i.e., it is not interruptable. We do not specify anything about an assumed order in which the operations may take place, but an operation can contain a so-called *guard*: a boolean expression between braces { }. An operation may only be executed if its guard is true, otherwise nothing happens. For example, a process may only execute the code for receiving a message if there is indeed a message present to receive.

The most basic operations in a distributed program one can think of for a node i are: send a message to j (S_i), receive a message from j (R_i), and do an internal (local) computation (I_i), where j is any node connected to i by a link. Operations and variables are subscripted by the identity of the node that performs and maintains them, respectively. This yields the following program skeleton.

```
S_i :  begin send a message to some neighbor j end

R_i :  {a message has arrived from j }
       begin receive the message from j end

I_i :  begin compute end
```

We can use these atomic operations as *building blocks* for bigger atomic operations, thereby adding extra structure in the order of computation and/or the order of communication. This can be done in different ways, and yields different program skeletons. Some ways are more or less standard, and the resulting program skeletons together with the appropriate network assumptions will be referred to as *models of computation.* Examples are the message-driven model and the synchronous model of computation. Given a model of computation, an idea what information a message should contain, and a way to compute the wanted information from the received information, we have a general framework for an algorithm. For example, the message-driven model of computation with messages which contain the name of a destination node together with the estimated distance of the sender of the message to that destination node forms the basis of both the algorithms of Tajibnapis and Chu. (The basic computation is: take the minimum of the local estimate of the distance to the destination node and 1 + the distance value in the received message.)

As it is usually necessary to specify what the initial values of the node variables are, we will do so for the variables used directly before the code of the atomic operations. In the sequel we distinguish the following four program skeletons. We will give them for static networks now and discuss the extension to the case for dynamic networks in the next section.

2.1.1. Phasing. The idea of phasing is to divide all the work to be done over different phases, and to allow a node to begin working on another phase only if all the work of the current phase is completed. Thus phasing adds some structure to the order of events in a computation, which was totally arbitrary up till now. We add a variable for the current phase at each node i: $phase_i$, and a new atomic operation is added for the transition to the next phase: P_i. The most general program skeleton which makes use of phasing (P) is as follows.

```
Initially    ∀ $i$: $phase_i = 0$.

$S_i^P$ : begin send a message belonging to $phase_i$ to some neighbor $j$ end

$R_i^P$ : {a message has arrived from $j$ }
       begin receive the message from $j$ and record it; compute end

$P_i^P$ : {all messages of phase = $phase_i$ are received from all neighbors }
       begin $phase_i := phase_i + 1$; compute end
```

Note that the internal computation operation I_i is now divided over operation R_i^P where the computation pertaining to the received message is done, and the operation P_i^P, the internal computation which effectuates the phase transition. Comparison of the operations S_i^P and R_i^P reveals a problem introduced by phasing. While in operation S_i^P a message belonging to $phase_i$ is sent, nothing is mentioned about a phase number of a message in operation R_i^P. Ideally, the messages node i receives while $phase_i = p$, would be the messages that i's neighbors had sent while their phase number had the same value p. Now the problem is, what do we want node i to do when a message belonging to a different phase has arrived? There are several possibilities to deal with this problem.

First, it might be the case that the assumptions about communication on the links combined with the properties of a more specific program skeleton suffice to prove that the arrival of messages of the wrong phase cannot happen. Secondly, we could just refuse to receive the message if it belongs to a different phase, and add a guard to that effect to the operation R_i^P. We should take care not to introduce the possibility of deadlock then. Thirdly, the message could be received but ignored, thus equaling the effect of the loss of a message. Fourth, if the message belongs to a later phase, we could buffer it until the node reaches the right phase. Fifth, we could just let the node receive the message and do the appropriate computation. It will depend on the circumstances what choice we will make.

Another problem is the guard of operation P_i^P: all messages of phase = $phase_i$ are received. There must be some way for node i to decide this. One possibility, if the number of messages per phase is not fixed, is to include the number of messages to expect in a phase in the first message belonging to a phase, or to somehow mark the last message belonging to a phase as such.

2.1.2. Message-driven computation. The added structure in the order of computation in this case is that messages may only be sent upon receipt of another message, and usually there is some relation specified between the contents of the received message and that of the messages sent. We will use the term 'appropriate' for this as long as we do not want to specify the relation. Thus we do not have a separate, autonomous sending operation any more. But now we need a way to start the sending of messages. Hence we introduce an operation A_i^M (awaken) which can either be executed spontaneously (after initialization), or upon receipt of the first message. Thus we include a test whether node i is awake yet in operation R_i^M, and if not, we let i awaken first, i.e., execute operation A_i^M. A node is clearly supposed to awaken only once, although this is only necessary for the termination of program skeleton M and not for its partial correctness. The boolean $awake_i$ records whether i is awake or not.

```
Initially   ∀ $i$: $awake_i = false$.

$A_i^M$ : {not $awake_i$}
    begin $awake_i$ := $true$;
        forall neighbors $j$ do send the appropriate messages to $j$ od
    end

$R_i^M$ : {a message has arrived from $j$ }
    begin if not $awake_i$ then do $A_i^M$ fi; receive the message from $j$; compute;
        forall neighbors $k$ do send the appropriate messages to $k$ od
    end
```

Lemma 2.1. In a static network, all nodes eventually awaken iff in every connected component of the network there is at least one node that awakens spontaneously, and messages are not lost, garbled, duplicated, or delayed infinitely long.

Proof. Obvious from program skeleton M. ■

Thus we will assume in the sequel that there is at least one node in each connected component of the network that awakens spontaneously. In a dynamic network, some action comparable to awakening will be taken upon a change in link status (i.e., the going down or coming up of a link). This will be discussed in the next section. We need the other assumption, too, in this model, namely that messages do not get lost, garbled, duplicated, or delayed infinitely long. This is because the sending of a message is now restricted and a node does not have the possibility of sending a message repeatedly until it gets through. Hence we have the following assumptions in this model.

Assumption 2.1. In a static network, at least one node in every connected component awakens spontaneously.

Assumption 2.2. Messages do not get lost, garbled, duplicated, or delayed infinitely long.

This form of computation is called message-driven, i.e., something only happens if a message is received. The situation that no more messages are around is of special interest in this model, as nothing will happen any more when this situation arises. This situation is called *termination* (TERM). Unless we make further assumptions about the network such as: all messages only have a bounded delay, it is sometimes necessary to add a so-called termination detection algorithm to be able to detect this state (see e.g. [Tel]). Otherwise nodes might not be able to conclude whether their computed values really reflect minimum hop distances or not, as this can depend on whether all messages are received and processed.

2.1.3. Simulated synchronous computation. In this model which is frequently used to simulate synchronous programs on an asynchronous network, we combine the added structure of phasing and of the message-driven model of computation. Hence assumptions 2.1 and 2.2 apply here, too. Although we did not demand with phasing that the message received was of the same phase as the receiver (because it was not necessary), we will do so now as it has the advantage that a message can contain less information in this case. However, we then need a way to ensure that no messages belonging to the next phase are received if they happen to arrive early, as is possible in this model. Thus we add a new variable in each node i: rec_i, which is a boolean array which records for each neighbor j in $rec_i[j]$ whether all messages of the current phase from neighbor j have already been received or not. Hence it is necessary that the receiver of a message has some way to decide whether a message received is the last message of the current phase or not.

There are several ways to implement this, for example: counting the number of messages and sending the numbers, too, or combining all messages of one phase over one link into one big message. Again we are not interested in an actual implementation, as long as a node can decide the question. This leads to program skeleton SS.

```
Initially   ∀ $i$: $awake_i = false$, $phase_i = 0$, and ∀ neighbors $j$: $rec_i[j] = false$.

$A_i^{SS}$ : {not $awake_i$}
    begin $awake_i := true$;
        forall neighbors $j$ do send the appropriate messages of $phase_i$ to $j$ od
    end

$R_i^{SS}$ : {a message (of $phase_i$) has arrived from $j$ $\wedge$ not $rec_i[j]$ }
    begin if not $awake_i$ then do $A_i^{SS}$ fi;
        receive the message from $j$ and record it; compute;
        if this was the last message of $phase_i$ from $j$ then $rec_i[j] := true$ fi;
        if ∀ neighbors $k$: $rec_i[k]$
        then $phase_i := phase_i + 1$; compute;
            forall neighbors $k$
            do send the appropriate messages of $phase_i$ to $k$; $rec_i[k] := false$ od
        fi
    end
```

Note that operation P_i^P is now included in operation R_i^{SS} to ensure that all operations except awakening are only done upon receipt of a message. A link can contain in one direction messages of two different phases. Assuming that messages have to be received in the order in which they arrive at a node, we now need the assumption that they arrive in the same order as they were sent. Otherwise the guard of operation R_i^{SS} could cause deadlock. In section 3.1.3 we will prove for refinements of this program skeleton, that the phrase 'of *phase*$_i$' is not necessary in the guard of operation R_i^{SS}.

Assumption 2.3. On any link, messages in any one direction are not reordered.

2.1.4. Synchronous computation.

In synchronous computation it is usually assumed that all nodes awaken simultaneously, and that messages are transmitted with a fixed delay. For our purposes however, it is enough to make the assumption that all nodes awaken spontaneously in addition to assumptions 2.2 and 2.3.

Assumption 2.4. All nodes awaken spontaneously.

In synchronous computation not only all messages of one phase are sent "at the same time", but all messages of one phase are also received "at the same time". It is usually not assumed that more messages over one link can be received at the same time, hence the information of all messages is assumed to be lumped together in one (bigger) message. Like in program skeleton *SS*, the operation P_i^P is included in the receive operation.

The variable rec_i is not used any more, as the registration of which messages are received in the program skeleton is replaced by a registration of which messages have arrived outside the program skeleton, and thus not specified here. If this program skeleton is used in a really synchronous environment, i.e., messages have a fixed delay, and there is a way to let all nodes start (awaken) simultaneously, then the guard of R_i^S becomes true automaticly each time the fixed delay has elapsed. The program skeleton (S) is as follows.

```
Initially    $\forall i$: $awake_i = false$ and $phase_i = 0$.

$A_i^S$ : {not $awake_i$}
      begin $awake_i := true$;
            forall neighbors $j$ do send the message belonging to $phase_i$ to $j$ od
      end

$R_i^S$ : { from every neighbor the message belonging to $phase_i$ has arrived }
      begin forall neighbors $j$ do receive the message from $j$ od;
            $phase_i := phase_i + 1$; compute;
            forall neighbors $j$ do send the message belonging to $phase_i$ to $j$ od
      end
```

These program skeletons together with the network assumptions stated in section 2.2 form the models of computation. In sections 3 and 4 we will further refine these program skeletons. We first specify the actual computation of minimum hop distances and specify the contents of a message. We then derive properties of the classes of algorithms that can be described in this way. Finally, we refine these program skeletons further to arrive at concrete algorithms.

The method used for the correctness proofs of these refined program skeletons is that of assertional verification or system-wide invariants, first introduced by Krogdahl [Kr] and Knuth [Kn]. The idea is that if a relation (between process variables for example) holds initially, and is kept invariant by all possible operations, then it will hold always in the distributed system, whatever order of operations takes place in an actual execution of the distributed algorithm. This approach has a definite advantage over operational reasoning for correctness proofs because it is almost impossible not to overlook some odd coincidence of events that can arise in the execution of a distributed algorithm that might make the proof invalid, as it makes use of checking all possible executions.

The advantage of the use of system-wide invariants for program skeletons is the following. If we have a more elaborate program skeleton or an algorithm which can be viewed as a special instance of some program skeleton, then any system-wide invariant which holds for the program skeleton, will hold also for the more elaborate program skeleton or the algorithm. This is the case simply because the invariant was proven correct for any order of operations in the general case, and hence also for the special order of operations which will take place in the more elaborate program skeleton or the algorithm. For example, as the simulated synchronous program skeleton is a special case of the phasing program skeleton, invariants which hold in the latter hold in the former, too. The approach was used successfully in e.g. [Tel] and [Scho].

System-wide invariants are very well suited to prove the partial correctness (or safety) of a program skeleton, i.e., that all variables contain the correct values upon termination. In general they are less suited to prove total correctness (or liveness), i.e., that there is termination in finite time. That an execution cannot go on infinitely long is usually proven by means of some counting argument. Although the freedom of deadlock for some program skeleton can be stated in a system-wide invariant, this does not mean that the freedom of deadlock for this program skeleton automaticly carries over to the freedom of deadlock for a refinement of that program skeleton. As the order of operations in the latter might be more restricted because extra guards were introduced, freedom of deadlock for the refined program skeleton will correspond to a *different* system-wide invariant which will have to be proved separately.

2.2. The network. We assume we have a network consisting of nodes connected by undirected communication links. We do not assume that the network is connected. Nodes can send messages to their neighbors over links. Unless otherwise stated, we assume that nodes know (the identity of) their neighbors.

As discussed in the previous section, we assume that messages sent are not lost, garbled, duplicated, or delayed infinitely long. Thus a message sent always arrives in finite time at its destination. We also assume that messages on one link for one direction are not reordered: links have the FIFO property, i.e., a message sent first on a link, arrives first. We mean to say that the underlying communication protocol(s) ensure those properties on this level. Hence communication is modeled as follows. A link (i,j) is represented by two FIFO queues of messages $Q[i,j]$ and $Q[j,i]$: the messages from i to j and from j to i, respectively. We denote the fact that a message M is on its way from node i to node j over the link (i,j) as $<M> \in Q[i,j]$.

We discuss algorithms for both static and dynamic networks, in sections 3 and 4, respectively. In a static network, links are fixed and known to their neighbors. In a dynamic network, links can go down and come up. We use the following model for communication in dynamic networks. We assume that the assumptions stated above for links in a static network, also apply to links in a dynamic network, whether they are up or down. To model the loss of messages when a link is down, we do not change the assumptions, but add an extra atomic operation local to a link for going down which provides for the possible loss of messages. Also the send procedure provides for possible loss of messages when a link is down. As a counterpart for the operation for going down, we also add an operation for the coming up of a link.

If the link (i,j) is up, then sending a message from i to j means: appending a message to $Q[i,j]$, and receiving by i of a message from j means: deleting the first message i.e., the head of $Q[j,i]$. If the link is down, sending a message from i to j means: possibly appending the message to $Q[i,j]$. This corresponds on the one hand to the message getting lost and on the other hand to the situation that the message was still in i's output buffer when the link came up again. If the link is down and $Q[j,i]$ not empty, receiving a message just means getting the head of $Q[j,i]$, corresponding to getting the next message in i's input buffer. Thus we get the following *send* and *receive* procedures.

```
proc send <M> to j by i =  begin if linkstate(i,j) = up
                                  then append <M> to Q[i,j]
                                  else possibly append <M> to Q[i,j]
                                  fi
                           end

proc receive <M> from j by i =  {<M> head of Q[j,i]}
                                begin delete <M> from Q[j,i] end
```

We will model nodes going down and coming up by the going down of all incident links of a node that are still up, and the coming up of a node by the coming up of possibly only some links incident to a node, respectively. It is of course necessary that nodes somehow become aware of the changed status (up or down, respectively) of an incident link. We model this by adding extra messages <*up*> and <*down*> which we call *control messages.* We assume that when a link changes status, control messages are added to the message queues corresponding to that link, and that the nodes incident to the link eventually become aware of the changed link status when they receive the control message. Thus the status of the link cannot be observed directly by the incident nodes.

When link (i,j) comes up, an <*up*> message is appended to both $Q[i,j]$ and $Q[j,i]$. When link (i,j) goes down, a <*down*> message is appended to both $Q[i,j]$ and $Q[j,i]$, and moreover, we allow that arbitrary noncontrol messages are deleted from $Q[i,j]$ and $Q[j,i]$, corresponding to the situation that some or all messages are lost when the link goes down. It is probably more realistic to only delete messages after the last <*up*> message, but it is not necessary to make this assumption. We do not allow that control messages are deleted.

Thus we get the following atomic operations U (coming up) and D (going down) on links.

```
U_ij :  {linkstate(i,j) = down}
        begin append <up> to Q[i,j];  append <up> to Q[j,i];
              linkstate(i,j) := up
        end
```

```
$D_{ij}$ :  {$linkstate(i,j)$ = up}
        begin delete all, some or no non-control messages from $Q[i,j]$;
              delete all, some or no non-control messages from $Q[j,i]$;
              append <down> to $Q[i,j]$; append <down> to $Q[j,i]$;
              $linkstate(i,j)$ := down
        end
```

Here *linkstate*(i,j) is a ghost variable which can have values *up* and *down*. It is not a variable which can be accessed by any node, but it is introduced to be able to formulate some invariants more precisely.

Summarizing, the assumptions for a dynamic network are:

Assumption 2.2'. Messages do not get lost otherwise than by a D-operation or a send procedure. Messages that are not lost do not get garbled, duplicated, or delayed infinitely long.

Assumption 2.5. If a link (i,j) changes status, the appropriate control messages are appended to $Q[i,j]$ and $Q[j,i]$.

Assumption 2.6. Initially, all links are down and the initial topology of the network is defined by the appropriate U_{ij} operations.

To the program skeletons from the previous section we now must add the new atomic operations RU and RD : receive an <*up*> message and receive a <*down*> message, respectively. Alternatively, we have to extend the code for the R operation with a test whether the message received is a control message and the appropriate action to take in case it is. If the computation on hand computes something which depends on the actual topology of the network, such as for example minimum hop distances, part of the computation might have become obsolete if there is a change in topology. There are two ways to deal with that: partially recompute with the problem of deciding what part, or totally restarting the computation with the problem of reinitializing. Thus in general, there will be several variables in the program skeleton that have to be changed in an RU or an RD operation.

In the phasing program skeleton of the previous section, at least the phasing number will have to be reset. In a message-driven computation we now must be aware that messages might have become lost because of link failures, contrary to assumption 2.2. Furthermore we need operations comparable to awakening which will take place upon changes in topology. These will be incorporated of course in the operations RU and RD mentioned above.

There is a more philosophical difference between a dynamic network and a static network, namely that in the latter case it is sufficient to do a computation once, while in a dynamic network the computation might have to be redone for every change in topology. This usually takes the form of a continuous computation which will only terminate when there are no more changes in topology.

3. Minimum hop distances in a static network.

We begin by applying the most general program skeleton of section 2.1 to the computation of minimum hop distances. That is, we specify the contents of a message, we define variables to record computed values, and we specify the computation which has to be done. After investigating what we can derive about estimated distances which are obtained in this way, we show in section 3.1 how to arrive at minimum hop distances if we refine the program skeletons P, M, SS, and S defined in section 2.1 in this way. In section 3.2 we then discuss some algorithms due to Gallager and Friedman [Fr].

We assume that the contents of a message are the identity of some destination node together with the estimated distance between the sender of the message and the destination. So messages (denoted as $<x,l>$) consist of two fields: the destination is in the first field and the (estimated) distance of the sender of the message to the destination is in the second field. We assume each node maintains an array D with (estimated) distances to all nodes. If we want to use the minimum hop distance for routing, we also need to remember the neighbor which sent a node the estimated minimal distance. This will be maintained in *dsn* (for *downstream neighbor*- the reason for this terminology will become clear later). We then get the following program skeleton (B).

```
Initially    ∀$i$: $D_i[i] = 0$ and ∀$x$ with $x \neq i$: $D_i[x] = \infty$.

$S_i^B$ : begin send $<x, D_i[x]>$ to some neighbor $j$ end

$R_i^B$ : {a message $<x,l>$ has arrived from $j$}
        begin receive $<x,l>$ from $j$;
            if $l+1 < D_i[x]$ then $D_i[x] := l+1$; $dsn_i[x] := j$ fi
        end
```

Although it is not really necessary, we suppose for ease of notation that the set of all nodes in the (static) network is known a priori to every node.

It might seem that we have not specified much about any program that contains these atomic actions as building blocks. However, we can already derive some statements about the relation between the values in $D_i[x]$ and the real (minimum hop) distance between i and x, denoted by $d(i,x)$.

Lemma 3.1. For all x and i the following holds invariantly.
(1) $D_i[x]$ is not increasing,
(2) $<x,l> \in Q[j,i] \Rightarrow l \geq D_j[x]$,
(3) $D_i[x] \geq d(i,x)$.

Proof. (1), (2). Obvious from program skeleton B.
(3). This is initially true, as $d(i,i) = D_i[i] = 0$ and $d(i,x) \leq \infty$ for all x. If a message $<x,l>$ is sent by j, j sends its value $D_j[x]$ as l. Hence by statement (3) for j and x, $l \geq d(j,x)$ for messages sent by j. In case i receives a message $<x,l>$ from j and adjusts $D_i[x]$, we know by the triangle inequality that $d(i,x) \leq d(i,j) + d(j,x) = 1 + d(j,x)$ and thus $1 + d(j,x) \leq 1 + l = D_i[x]$. ■

Lemma 3.2. For all i and x with $x \neq i$

$D_i[x] = k < \infty \Rightarrow$ there is a path of $\leq k$ hops via $dsn_i[x]$ and $D_{dsn_i[x]}[x] \leq k-1$.

Proof. Consider operation R_i^B, and let the message received be $<x,l>$. Then $l \geq D_j[x]$ by lemma 3.1(2). If $l = \infty$, i does not change $D_i[x]$ as $1+\infty \not< D_i[x]$ always. Thus if i changes $D_i[x]$ and sets $dsn_i[x]$ to j, $l < \infty$. We have two cases.
Case 1: $l = 0$.

As $l \geq d(j,x)$ we know $d(j,x) = 0$ and hence $j = x$. Thus x is neighbor of i and there is a path of one hop to x. Then $D_i[x]$ is set to 1 and $dsn_i[x] = j$ while $D_j[x] \leq 1\text{-}1 = 0$.
Case 2: $l > 0$.
As $D_j[j] = 0$ initially and is never changed (lemma 3.1), $l > 0$ implies $x \neq j$. Hence we can use the induction hypothesis for j and conclude that there is a path of $\leq l$ hops to x from j. We have $l \geq D_j[x]$, $dsn_i[x]$ is set to j, and $D_i[x]$ to l+1 while there is a path of $\leq l+1$ hops via j to x from i. ■

Baran's *perfect learning algorithm* [Ba] is essentially identical to the above program skeleton. On messages sent for other reasons, a hop count of the distance traveled is piggybacked. This hop count is used by the receiver of the message to adjust its distance table in the manner described above. This distance table then is used in routing, without bothering whether the distances recorded are minimum distances.

3.1. Structuring the program skeleton. From the preceding section we conclude that any program which sends (estimates of) minimum hop distances to neighbors, and adjusts distances upon receipt of messages in this way, yields estimates which are upper bounds of the real minimum distances, providing the initial values were upper bounds. In general however, one would like to know when the upper bounds are equal to the minima. Clearly, one has to send "enough" values around. Thus the problem reduces to deciding for each node when it can stop sending values of distances because all nodes (including the node itself) have the correct values. The way to achieve this is to add extra structure to the program skeleton. Both with phasing and in the message-driven model we can decide when the distance tables are correct.

3.1.1. Phasing. The idea of phasing was explained in section 2.1.1. In order to distinguish messages of different phases, messages will now have a third field containing the current phase of the sender. All the work of one phase in this case constitutes of sending around those messages which contain a distance field of $l = phase_i$, and receiving all those messages from all neighbors, thus gathering all information about nodes which lie at a distance of $phase_i + 1$. We refine program skeleton P to program skeleton $P1$.

```
Initially   $\forall i$: $D_i[i] = 0$, $phase_i = 0$, and $\forall x$ with $x \neq i$: $D_i[x] = \infty$.

$S_i^{P1}$ : begin send $<x, D_i[x], phase_i>$ to some neighbor $j$ end

$R_i^{P1}$ : {a message $<x,l,p>$ has arrived from $j$}
    begin receive $<x,l,p>$ from $j$; record it as received for phase = $p$;
        if $l$+1 $< D_i[x]$ then $D_i[x] := l$+1; $dsn_i[x] := j$ fi
    end

$P_i^{P1}$ : {all messages of phase = $phase_i$ from all neighbors are received}
    begin $phase_i := phase_i + 1$ end
```

The guard of operation P_i^{P1} still is rather informally stated. It depends on the actual implementation how this statement should be formulated exactly. For example, the guard could be "from all neighbors j and about all destinations x a message of $phase_i$ is received".

We are not interested in how this could be implemented, we only assume that a node can somehow decide this question.

Note that in operation R_i^{P1} we did not test the phase number p of the received message against the phase number $phase_i$ of the node i. Whether a message is buffered until i has reached the corresponding phase or is processed directly, or even is thrown away, is immaterial to the correctness, as we will see. Thus we did not pose an extra restriction on R_i^{P1}.

As this program skeleton is also a refinement of program skeleton B, lemmas 3.1 and 3.2 still hold.

Lemma 3.3. For all i and x the following holds invariantly.

(1) $phase_i$ is not decreasing,

(2) $d(i,x) > phase_i \vee d(i,x) \geq D_i[x]$.

Proof. (1). Obvious from program skeleton $P1$.

(2). Initially $d(i,x) > phase_i = 0$ for $i \neq x$ and $d(i,i) = D_i[i] = 0$.

-Operation S_i^{P1} does not change any variables.

-Operation R_i^{P1} can only decrease $D_i[x]$, hence the statement remains valid.

-Consider operation P_i^{P1}. We only need to consider the case that $phase_i$ is increased to the value that happens to be $d(i,x)$. Thus assume $d(i,x) = k \geq 1$. Hence there is a path of k hops from i to x. Let j be the first node after i on this path. Thus $d(j,x) = k-1$. As the operation is enabled, i has received a message $<x,l,p>$ from j with $p = k-1$. Together with the induction hypothesis we have $d(j,x) > p$ or $d(j,x) \geq l$. As $d(j,x) = k-1$, we know $k-1 \geq l$. In R_i^{P1} the result of receiving $<x,l,p>$ is that $D_i[x] \leq l+1$ whether or not $D_i[x]$ is adjusted. As $D_i[x]$ is not increasing, this is still the case. Hence $D_i[x] \leq l+1 \leq k = d(i,x)$. ■

Now it is clear from operation R_i^{P1} that if two messages $<x,l,p>$ and $<x,l,p'>$ are received from j, that the second one has no effect whatsoever. Hence it is not necessary for j to send the second message, as long as i is able to conclude when it has received "all" messages of a phase. This could be implemented for example by letting j send the total number of messages of each phase to be expected, or sending all messages over one link inside one large message per phase. The proof of lemma 3.3 is easily adjusted for the set of atomic operations where this different definition of "all messages per phase" is used in the guard of operation P_i^{P1}. This will be discussed in more detail in section 3.1.3.

Theorem 3.4. For all i and x the following holds invariantly.

(1) $d(i,x) \leq phase_i \Rightarrow D_i[x] = d(i,x)$,

(2) $D_i[x] \leq phase_i \Rightarrow D_i[x] = d(i,x)$.

Proof. Follows directly from lemma 3.3 combined with lemma 3.1(2). ■

Thus we know on the one hand that of all nodes x lying at a distance less or equal the current phase number the correct distance is known to i, and on the other hand, that if the value that i has for the distance to x is less or equal the current phase number, then this value is correct.

Corollary 3.5. For all i the following holds invariantly.

$$phase_i > \max_x \{D_i[x] \mid D_i[x] < \infty\} \Rightarrow \forall x: D_i[x] = d(i,x).$$

Proof. With theorem 3.4 we have that for all x with $d(i,x) \leq phase_i$, $D_i[x] > d(i,x)$. For those x with $d(i,x) = \infty$ we have with lemma 3.1 that $D_i[x] = d(i,x)$. Assume there is an x with $d(i,x) = k > phase_i$, and $d(i,x) < \infty$. Then there is a path of length k from i to x. Thus on this path, there must be a node y with $d(i,y) = phase_i$. With lemma 3.3 we know $d(i,y) = D_i[y] < phase_i$ which is in contradiction with the premise. Thus for all nodes x we have $D_i[x] = d(i,x)$. ■

Hence a node can decide when all its distance values are indeed correct. In an actual algorithm for computing minimum hop distances this can be used for a stop criterion in the algorithm. Note that the program skeleton as given here is too general for being able to prove that it terminates and does not deadlock or goes on generating messages forever.

3.1.2. Message-driven computation. In this case the added structure in the order of computation is that it is specified which messages are to be sent if a certain message is received. If the receipt of a message leads to an adjustment in the distance table, this "news" is sent to all other neighbors. We obtain the following program skeleton ($M1$).

```
Initially   ∀ $i$: $awake_i = false$, $D_i[i] = 0$, and ∀ $x$ with $x \neq i$: $D_i[x] = \infty$.

$A_i^{M1}$ :{not $awake_i$}
    begin $awake_i := true$; forall neighbors $j$ do send $<i,0>$ to $j$ od end

$R_i^{M1}$ :{a message $<x,l>$ has arrived from $j$}
    begin receive $<x,l>$ from $j$; if not $awake_i$ then do $A_i^{M1}$ fi;
        if $l+1 < D_i[x]$
        then $D_i[x] := l+1$; $dsn_i[x] := j$;
            forall neighbors $k$ with $k \neq j$ do send $<x,D_i[x]>$ to $k$ od
        fi
    end
```

As it is also true for this program skeleton that it is a refinement of skeleton B, lemmas 3.1 and 3.2 also hold now.

Lemma 3.6. For all links (i,j) and all nodes x the following holds invariantly.

$$awake_i \wedge D_i[x] < \infty \Rightarrow <x,D_i[x]> \in Q[i,j] \vee D_j[x] \leq D_i[x]+1.$$

Proof. Initially the premise is false.
-Operation A_i^{M1}. When i awakens, only $D_i[i] < \infty$, but then $<i,0>$ is sent to j, hence $<i,0> \in Q[i,j]$.
-Operation R_i^{M1}. Only there is $D_i[x]$ changed. If it is, it is done upon receipt of some message $<x,l>$ from node k, where now $D_i[x] = l+1$. We distinguish two cases.
Case 1: $j = k$. With lemma 3.1 we have $l \geq D_k[x]$, thus $D_j[x] \leq l \leq D_i[x]+1$ holds.
Case 2: $j \neq k$. Then the message $<x,D_i[x]>$ is sent to j hence $<x,D_i[x]> \in Q[i,j]$ holds.
-Operation R_j^{M1} can falsify the statement $<x,D_i[x]> \in Q[i,j]$ by receiving that specific

message, but then the result is $D_j[x] \le D_i[x]+1$. If this last statement holds, it can not be invalidated by an R_j^{M1} operation because $D_j[x]$ is not increasing (lemma 3.1). ■

Lemma 3.7. For all nodes i and x the following holds invariantly.
$d(i,x) = D_i[x] \ \vee$
$d(i,x) < D_i[x] \wedge \exists j$ **with** $d(i,j) = 1 \wedge d(i,x) = d(j,x)+1 \wedge$
$(d(j,x) = D_j[x] \wedge (<x,D_j[x]> \in Q[j,i] \vee$ **not** $awake_j \wedge x = j) \vee$
$d(j,x) < D_j[x])$.

Proof. In case $d(i,x) < D_i[x]$ it follows that $i \ne x$ and $d(i,x) < \infty$. It is a property of minimum hop distances that i has such a neighbor j on a path to x. By lemma 3.1 we have $d(j,x) \le D_j[x]$. For the case of $d(j,x) = D_j[x]$ use lemma 3.6. ■

Lemma 3.8. The number of messages sent in program skeleton $M1$ is finite.

Proof. First note that in any message $<x,l>$ which is sent, $0 \le l < \infty$. Secondly, let N be the total number of nodes in the network. Then $l < N$ (use lemma 3.2). Moreover, if two messages $<x,l>$ and $<x',l'>$ are sent in the same direction over a link, then either $x \ne x'$ or $l \ne l'$. This gives the desired result. ■

Theorem 3.9. TERM $\Rightarrow \forall i,x\colon D_i[x] = d(i,x)$.

Proof. Use lemmas 2.1, 3.6, and 3.7. ■

Corollary 3.10. Program skeleton $M1$ is (totally) correct.

Proof. As messages can always be received when they arrive, and delays are finite, this program skeleton terminates in finite time. ■

The only problem now left is that a node i cannot see from the values of its variables whether there is termination or not, and hence whether its distance table is correct. Unless we make further assumptions about the network such as: all messages have a bounded delay only, we need to add a so-called termination detection algorithm to be able to detect this state (see e.g. [Tel]).

3.1.3. Simulated synchronous computation. In this model, the feature of phasing is incorporated in the message-driven model of computation, as was discussed in section 2.1.3. We obtain program skeleton $SS1$ if we refine program skeleton SS with the distance computation.

```
Initially ∀ $i$:    $awake_i = false$, $phase_i = 0$, $D_i[i] = 0$, and
                    ∀ $x$ with $x \ne i$: $D_i[x] = \infty$, ∀ neighbors $j$: $rec_i[j] = false$.

$A_i^{SS1}$ : {not $awake_i$}
    begin $awake_i := true$;
        forall neighbors $j$ do forall nodes $x$ do send $<x,D_i[x],phase_i>$ to $j$ od od
    end
```

```
$R_i^{SS1}$ : {a message $<x,l,p>$ has arrived from $j$ $\wedge$ not $rec_i[j]$}
    begin receive $<x,l,p>$ from $j$ ; if not $awake_i$ $\wedge$ $p = 0$ then do $A_i^{SS1}$ fi ;
        if $awake_i \wedge p = phase_i$
        then if $l+1<D_i[x]$ then $D_i[x] := l+1$; $dsn_i[x] := j$ fi ;
             if this was the last message from $j$ belonging to $phase_i$
             then $rec_i[j] := true$
             fi ;
             if $\forall$ neighbors $k$: $rec_i[k]$
             then $phase_i := phase_i + 1$ ;
                  forall neighbors $k$
                  do   $rec_i[k] := false$;
                       forall nodes $x$ do send $<x,D_i[x],phase_i>$ to $k$ od
                  od ; if not $\exists x$ with $D_i[x] = phase_i$ then $awake_i := false$ fi
        fi   fi
    end
```

In the above program skeleton a lot of information turns out to be redundant. Hence we proceed to prove relations between the variables involved in order to arrive at a simpler program skeleton whose correctness follows from the correctness of the skeleton above. Note that the guard of operation R_i^{SS1} does not contain the test $p = phase_i$, contrary to the situation in R_i^{SS}. The next theorem states that this test is not necessary.

Lemma 3.11. For all links (i,j), all nodes x, and all integers $k \geq 0$ the following holds invariantly.

(1) $<x',l',p'>$ **behind** $<x,l,p>$ **in** $Q[i,j] \Rightarrow p' \geq p$,

(2) $\exists <x,l,p> \in Q[i,j]$ **with** $p = k \Rightarrow phase_i \geq k \wedge$

$\qquad\qquad phase_j < k \vee phase_j = k \wedge rec_j[i] = false$,

(3) $(awake_i \wedge phase_i = k = 0 \vee 0<k \leq phase_i) \wedge$ **not** $\exists <x,l,p> \in Q[i,j]$ **with** $p = k \Rightarrow$

$\qquad phase_j > k \vee phase_j = k \wedge rec_j[i] = true \vee$ **not** $awake_j \wedge phase_j \geq 1$,

(4) (**not** $awake_i \wedge phase_i = k = 0 \vee k > phase_i) \Rightarrow$

$\qquad$ **not** $\exists <x,l,p> \in Q[i,j]$ **with** $p = k \wedge$

$\qquad (phase_j < k \vee phase_j = k \wedge rec_j[i] = false)$.

Proof. (1). Follows from the FIFO property of the message queues and lemma 3.3.
We prove the remaining statements simultaneously. Initially the premise of (2) and (3) is false, and (4) holds for all k as all queues are empty. Consider the effects of the different operations upon the statements.

-Operation A_i^{SS1}. Then the premise of (2) holds for $k = 0$ and $phase_j = 0$ and $rec_j[i] = false$. The premise of (3) is *false* for $k = 0$. (4) continues to hold for $k \geq 1$.

-Operation R_i^{SS1} where $phase_i$ is increased. (2) continues to hold for $phase_i > k$ as before, and for $k = phase_i$ messages $<x,l,p>$ with $p = k$ are sent to j. As $phase_j \leq phase_i$ before R_i^{SS1}, we now have $phase_j < k$. (3) continues to hold for $phase_i > k$ as before, and the premise is *false* for $k = phase_i$. (4). The premise holds for less values of k.

-Operation A_j^{SS1}, spontaneously or on receipt of a message which was not the last one of this phase from i. Hence as a result, $phase_j = 0$ and $rec_j[i] = false$. If (2) held, it still holds. The premise of (3) is *false*. (4) continues to hold as before.

-Operation R_j^{SS1} on receipt of a message which was not the last one of this phase from i. Then all statements remain valid as before.

-Operation R_j^{SS1} on receipt of a message $<x,l,p>$ from i which was the last one of this phase. Then before $rec_j[i] = false$ (statement (2)). We have two cases.

Case 1: $phase_j$ is not increased. Then $rec_j[i] = true$ and **not** $\exists <x,l,p> \in Q[i,j]$ with $p = phase_j$ holds. An exception is the case where $awake_j$ was *false*, since $rec_j[i] = false$ then.

Case 2: $phase_j$ is increased. Then again $rec_j[i] = false$ and **not** $\exists <x,l,p> \in Q[i,j]$ with $p = phase_j$ holds. If (2) held before for values of k with $phase_j < k$, then it now holds for $phase_j \le k$ as $rec_j[i] = false$. (4) cannot have held before R_j^{SS1} for $k = phase_j$, hence (4) continues to hold now $phase_j$ is increased. If $awake_j$ is set to *false*, $phase_j$ must at least be 1 as $phase_j$ is increased first.

-Operation R_j^{SS1} on receipt of a message from another neighbor than i which results in an increase in $phase_j$. This can only happen if before R_j^{SS1} $rec_j[i] = true$ already, and hence **not** $\exists <x,l,p> \in Q[i,j]$ with $p = phase_j$ before R_j^{SS1}. Afterwards, $rec_j[i] = false$. The premise of (2) only could hold for $k > phase_j$, hence (2) holds afterwards. In (3), $phase_j = k$ and $rec_j[i] = true$ held before, and now $phase_j > k$ holds. In (4), $phase_j = k$ and $rec_j[i] = false$ did not hold, hence increasing $phase_j$ by 1 does not invalidate (4).

Hence these statements remain invariant under all possible operations. ■

Theorem 3.12. For all links (i,j) and all nodes x the following holds invariantly.
$<x,l,p>$ **head of** $Q[i,j] \Rightarrow$
$$p = phase_j \wedge rec_j[i] = false \vee p > phase_j \wedge (rec_j[i] = true \vee awake_j = false).$$

Proof. Follows directly from lemma 3.10 (1) and (4), lemma 3.10 (2) with $k = p$ and lemma 3.10 (3) with $k \le p - 1$. ■

Hence we can conclude that the receiver of a message $<x,l,p>$ has no need for the information what in the third field. If the receiver is *awake* and the guard is enabled, the message is of the current phase, if the receiver is not *awake*, it is clear from its own phase number (=0 or >0) whether it should awaken and start participating in the algorithm or that it has finished already. In the last case the (redundant) message can be thrown away. Thus we can omit the third field from the messages. However, there is more information that is redundant.

Lemma 3.13. For all links (i,j) and all nodes x the following holds invariantly.
(1) $D_i[x] < \infty \Rightarrow D_i[x] \le phase_i + 1$,
(2) $<x,l,p> \in Q[i,j] \Rightarrow l = \infty \vee l = d(i,x) \le p$,
(3) $<x,l,p> \in Q[i,j] \wedge l < p = phase_j \Rightarrow D_j[x] = d(j,x)$.

Proof. We prove the first two statements simultaneously. Initially both are true. If a node awakens, it sends messages of the form $<x,0,0>$ which agrees with the second statement. If in an operation R_i^{SS1} $D_i[x]$ is changed, it is done upon receipt of some message $<x,l,p>$ from say j. Then $l \ne \infty$, $p = phase_i$, and $l = d(j,x) \le phase_i$. As $D_i[x]$ is set to $l+1 = d(j,x)+1 \le phase_i + 1$. If in operation R_i^{SS1} messages are sent, then $phase_i$ was increased, too. As for those x with $D_i[x] < \infty$, $D_i[x] \le phase_i + 1$, we have that after the increase

$D_i[x] \leq phase_i$ and thus $l \leq p$ in the messages sent, or $l = \infty$. With theorem 3.4(2) $D_i[x] \leq phase_i$ implies $l = d(i,x)$. Hence (1) and (2) remain invariant.
Statement (3) follows from the triangle inequality $d(j,x) \leq d(i,x)+1$ together with (1) and theorem 3.4(1). ■

Thus the only messages $<x,l,p>$ upon the receipt of which an entry $D_i[x]$ is changed, are those with $l = p = phase_i$, and hence $D_i[x]$ is always set to $phase_i+1$. Necessarily $D_i[x]$ was ∞ beforehand. Thus the test whether to change $D_i[x]$ can be stated otherwise. Only the name of the node x in a message $<x,l,p>$ is information we need. Now it is possible to do two things: one is to just send messages which only contain the name of a node. The problem with this is that the receiver now has no way of knowing when all messages of one phase have been received, as the number of them is not fixed any more, and even might be zero. The second way is to send only one message which contains a set of node names. Thus we avoid the previous problem, introducing however messages which do not have a fixed length.

We now give the simplified program skeleton for the second way in skeleton $SS2$, for easy comparison to the synchronous program skeleton (see section 3.1.4). Now X denotes a (possibly empty) set of node names.

```
Initially $\forall\, i$:    $awake_i = false,\ phase_i = 0,\ D_i[i] = 0$, and
                $\forall\, x$ with $x \neq i$: $D_i[x] = \infty$, $\forall$ neighbors $j$: $rec_i[j] = false$.

  $A_i^{SS2}$ : {not $awake_i$}
      begin $awake_i := true$; forall neighbors $j$ do send $<\{i\}>$ to $j$ od end

  $R_i^{SS2}$ : {a message $<X>$ has arrived from $j$ $\wedge$ not $rec_i[j]$}
      begin receive $<X>$ from $j$; if not $awake_i$ $\wedge$ $phase_i = 0$ then do $A_i^{SS2}$ fi;
          if $awake_i$
          then $rec_i[j] := true$;
               forall $x \in X$
               do if $D_i[x] = \infty$ then $D_i[x] := phase_i+1$; $dsn_i[x] := j$ fi od;
               if $\forall$ neighbors $k$: $rec_i[k]$
               then $phase_i := phase_i+1$; $X := \{x \mid D_i[x] = phase_i\}$;
                    forall neighbors $k$ do $rec_i[k] := false$; send $<X>$ to $k$ od;
                    if $X = \emptyset$ then $awake_i := false$ fi
          fi   fi
      end
```

Even now there is some redundant information sent. This is exploited in the algorithms of Gallager and Friedman [Fr]. We refer the reader to section 3.2 for more details.

We now proceed with the issue of correctness of this program skeleton, as this does not follow directly from corollaries 3.5 and 3.10. Since we introduced extra guards it is not obvious that no deadlock can occur.

Lemma 3.14. For all links (i,j) the following holds invariantly.
$awake_i \wedge rec_i[j] = false \wedge Q[j,i] = \emptyset \quad \Rightarrow$
$\quad phase_j < phase_i \wedge awake_j \vee phase_j = phase_i = 0 \wedge \text{not } awake_j \wedge Q[i,j] \neq \emptyset$.

Proof. Using lemma 3.11(3) with i and j interchanged and with $k = phase_i$, we get a contradiction. Hence we conclude that $phase_j < phase_i$ or **not** $awake_j \wedge phase_i = phase_j = 0$. In the case of $phase_j < phase_i$, let us assume that **not** $awake_j$ holds. It is clear from the program skeleton that this implies **not** $\exists x$ **with** $D_j[x] = phase_j$. Hence there is no node x with $d(j,x) = phase_j$ (theorem 3.4). On the other hand, $awake_i$ holds, so there is some node x with $D_i[x] = d(i,x) = phase_i > phase_j$. As $d(j,x) \geq d(i,x) - 1$ because i and j are neighbors, this leads to a contradiction and j must be *awake* still. Upon awakening, i sends a message to j, hence $Q[i,j] \neq \varnothing$. As long as j does not receive the message, $rec_j[i] = false$ and j does not awake. On the other hand, if j awakens spontaneously, $Q[j,i] = \varnothing$ is invalidated. Hence the statement holds. ■

Theorem 3.15. The program skeletons $SS1$ and $SS2$ are correct.

Proof. Lemma 3.14 implies that if at least one node in each connected component of the network awakens spontaneously, there is always some node in that connected component which can go on because its R operation is enabled. Thus no deadlock can occur. That the values in the distance tables are correct follows from corollary 3.5. As the number of messages sent is finite: at most the number of phases (bounded by the diameter of the connected component) times the number of nodes, termination is in finite time. ■

3.1.4. Synchronous computation. In synchronous computation not only all messages of one phase are sent "at the same time", but all messages of one phase are also received "at the same time". Refining program skeleton S from section 2.1.4 leads to program skeleton $S1$.

```
Initially ∀ $i$:    $awake_i = false$, $phase_i = 0$, $D_i[i] = 0$, and
                   ∀ $x$ with $x \neq i$: $D_i[x] = \infty$.

$A_i^{S1}$ : as $A_i^{SS2}$.

$R_i^{S1}$ : {$awake_i$ and from all neighbors a message has arrived }
    begin forall neighbors $j$
          do   receive <$X$> from $j$ ;
               forall $x \in X$
               do if $D_i[x] = \infty$ then $D_i[x] := phase_i + 1$; $dsn_i[x] := j$ fi od
          od; $phase_i := phase_i + 1$; $X := \{x \mid D_i[x] = phase_i\}$;
          forall neighbors $j$ do send <$X$> to $j$ od;
          if $X = \varnothing$ then $awake_i := false$ fi
    end
```

Lemma 3.16. For all links (i,j) the following holds invariantly.
$awake_i \wedge Q[j,i] = \varnothing \quad \Rightarrow$
$$phase_j < phase_i \wedge awake_j \vee phase_j = phase_i = 0 \wedge \textbf{not } awake_j \wedge Q[i,j] \neq \varnothing .$$

Proof. Initially the relation is true.
-Operation A_i^{S1} sets $awake_i$ and $Q[i,j] \neq \varnothing$.
-Operation A_j^{S1} falsifies the premise.
-Operation R_i^{S1} is only enabled if $Q[j,i] \neq \varnothing$. If afterwards $Q[j,i] = \varnothing$, then it contained

exactly one message. With a reformulation of lemma 3.11 and theorem 3.12 for this program skeleton we know that before R_i^{S1}, $phase_i = phase_j$ held. As $phase_i$ is increased, the result is $phase_j < phase_i$. The argument that $awake_j$ holds in the first part of the conclusion is the same as in the proof of lemma 3.14.
-Operation R_j^{S1} falsifies the premise. ■

Theorem 3.17. Program skeleton $S1$ is correct.

Proof. Consider the differences with program skeleton $SS2$. In operation R_i^{S1} there is no possibility for awakening upon receipt of a message. This is not necessary because of assumption 2.4: all nodes awaken spontaneously. The partial correctness follows because this is a refinement of program skeleton $SS2$. Lemma 3.16 implies that there is always a node with a minimal phase number for which some operation is enabled: either awakening or receiving messages over all links. Hence deadlock cannot occur. As the total number of messages sent is finite, this algorithm terminates in finite time (assuming that all nodes awaken in finite time.) ■

Although program skeleton $S1$ is the straightforward refinement of program skeleton $SS2$, it should be noted that the problem we consider, namely computing minimum hop distances, is not wholly suited for a really synchronous computation. Ideally, one would not only want all nodes to start simultaneously, but also to finish simultaneously. However, it is inherent to this problem that some nodes have more work to do than others, possibly twice as much. For example, for nodes on a path, the ones at the ends of the path go on twice as long as the one(s) in the middle. Moreover, in the present formulation, there are (redundant) messages left in the queues after termination of the algorithm which is not elegant. It is no problem of course to add some code to receive and throw away the remaining messages.

3.2. Concrete algorithms.

Friedman [Fr] discussed two algorithms for finding the minimum hop distances in a static network, which can be viewed more or less as special cases of program skeleton $S1$ from the previous section. The first of these algorithms is attributed to Gallager.

3.2.1. The algorithm of Gallager.

Gallager noted that in program skeleton $S1$ for the computation of minimum hop distances, there is still redundant information sent in messages. Consider the case that node i hears about a node x first from its neighbor j. Thus a shortest path from i to x leads via j. However, in the next phase i sends the newly learned identity of x to j too, which clearly is redundant information for j.

Lemma 3.18. For all links (i,j) and all nodes x the following holds invariantly.

$$dsn_i[x] = j \Rightarrow D_j[x] = d(j,x).$$

Proof. Initially the premise is false. If $dsn_i[x]$ is set to some value j in operation R_i^{S1}, then x was included in a message from j. When j sent this message, $D_j[x] = d(j,x)$, and $D_j[x]$ is not changed any more. ■

Thus it is not necessary to include node x in the set X sent to j if $dsn_i[x] = j$. However, from the program skeleton it is not clear which neighbor will be chosen as downstream neighbor in case there are more possibilities, as the order of operation in "**forall** neighbors j **do**..." is not specified. For every possibility j, i.e., every neighbor j with minimal distance to x, it is not necessary to include x in X to j. Thus the array of single values dsn_i is changed to an array of sets of neighbors all of which lie at a minimal distance.

We now rewrite the program in these terms which results (almost) in the algorithm of Gallager as stated by Friedman [Fr].

```
Initially ∀ i:  $awake_i = false$, $phase_i = 0$, $D_i[i] = 0$, $dsn_i[i] = \emptyset$, and
                $\forall x$ with $x \neq i$: $D_i[x] = \infty$ and $dsn_i[x] = \emptyset$.

$A_i^G$ : as $A_i^{S1}$.

$R_i^G$ : $\{awake_i \wedge$ from all neighbors a message has arrived$\}$
    begin forall neighbors $j$
        do  receive $<X>$ from $j$ ;
            forall $x \in X$
            do  if $D_i[x] \geq phase_i + 1$
                then $D_i[x] := phase_i + 1$; $dsn_i[x] := dsn_i[x] \cup \{j\}$
                fi
        od  od; $phase_i := phase_i + 1$;
        forall neighbors $j$
        do let $X = \{x \mid D_i[x] = phase_i \wedge j \notin dsn_i[x]\}$; send $<X>$ to $j$ od;
        if not $\exists x$ with $D_i[x] = phase_i$ then $awake_i := false$ fi
    end
```

Corollary 3.19. The algorithm of Gallager is correct.

Proof. Follows from lemma 3.18 and theorem 3.17. ■

Note that there is one difference with the algorithm of Gallager: where we have: "**forall** neighbors j **do** receive $<X>$ from j", Gallager has "receive transmissions from all *active* neighbors". However, we showed in lemma 3.14 that all neighbors are *awake* or active long enough and hence that this test is unnecessary.

As Friedman already noted (without any arguments however) this algorithm can easily be adapted for use in an asynchronous environment. It is now a straightforward exercise to incorporate the idea of the sets of downstream neighbors $dsn_i[x]$ in program skeleton $SS2$ which makes use of messages about sets of nodes.

3.2.2. The algorithm of Friedman.

Friedman [Fr] observed that there is still some redundant information sent around in the previous algorithm, in the case that the network is not a bipartite graph. This means that the graph contains cycles of odd length. Consider two neighbors i and j on a cycle of odd length. This means there are nodes (at least one, say x) with the same distance to i and j: $d(i,x) = d(j,x)$. Thus in phase $d(i,x)$ i and j send the name of node x to each other, while this is no new information for them. Friedman tried to adapt the previous algorithm in such a way that this is avoided.

The way he did that, is by ensuring that information is not sent in in both directions over a link simultaneously. For each link, a HI and a LO end is defined, the number of phases is doubled, and nodes alternately send and receive from HI and LO ends. To be able to discuss the relation between the phases in the previous program skeleton and the phases in Friedman's algorithm, we will give the latter variables a new name: *Fphase*. In the program skeleton we will write in comment what would have happened to the variable *phase*, thus simplifying the formulation of invariants. Note that we cannot choose HI and LO ends such that nodes have only HI or only LO ends, because that would mean that the network is bipartite and the problem we want to avoid does not occur.

Hence we distinguish two sets hi_i and lo_i which contain those links for which i is HI and LO end, respectively. It is arbitrary how we choose HI and LO ends, as long as both ends of a link decide consistently. We assume node names are distinct and can be ordered in some way, (lexicographically for example), and we take as HI end the end with the highest node name.

Furthermore, we need some way to remember which node identities were received by LO ends while they were candidates to be sent in the next phase, as those are the identities we want to avoid to send twice. Note that they are not remembered in Gallager's algorithm, because they have a shorter path via another neighbor. We will maintain an array nnn_i (for next nearest neighbor) of sets, where neighbor $j \in nnn_i[x]$ if $d(i,x) = d(j,x)$. If we incorporate these ideas in Gallager's algorithm we get the following program skeleton. Lines differing from the algorithm as stated by Friedman himself are marked with an asterisk (*) at the beginning of the line.

The atomic operation R_i^G is now split into three different atomic operations: R_i^0, R_i^{odd}, and R_i^{even}, as the guards of these operations now differ. In R_i^0, messages must have arrived over all links to receive all of them and decide the HI and LO ends of the links. In R_i^{odd} the *Fphase* number is odd and messages from LO link ends must be awaited before they can be received. If the *Fphase* number is even, messages from HI link ends are awaited in R_i^{even}. In order to be able to state invariants about messages of a certain phase, we add in comment messages with a second field containing the current *Fphase* number.

```
Initially ∀ i:   $awake_i = false,\ Fphase_i = 0,\ D_i[i] = 0,\ dsn_i[i] = \varnothing,$
                 $hi_i = \varnothing,\ lo_i = \varnothing,$ and                co $phase_i = 0$ co
                 $\forall x$ with $x \neq i$:  $D_i[x] = \infty,\ dsn_i[x] = \varnothing,$ and $nnn_i[x] = \varnothing.$

$A_i^F$ : as $A_i^G$                                          co send <{ $i$ },0> co

$R_i^0$ : {$awake_i \wedge Fphase_i = 0 \wedge$ over all links a message has arrived}
      begin forall links
            do    receive < { $j$ } > over the link; $D_i[j] := 1$; $dsn_i[j] := \{j\}$;
                  if $j < i$ then $hi_i := hi_i \cup \{j\}$ else $lo_i := lo_i \cup \{j\}$ fi
            od; $Fphase_i := 1$;                         co $phase_i := 1$ co
            forall $j \in hi_i$
*           do let $X = \{x \mid D_i[x] = \frac{1}{2}(Fphase_i+1) \wedge j \notin dsn_i[x]\}$; send <$X$> to $j$
            od;                                 co send <$X, Fphase_i$> to $j$ co
            if not $\exists x$ with $D_i[x] = \frac{1}{2}(Fphase_i+1)$ then $awake_i := false$ fi
      end
```

```
$R_i^{odd}$ : { $awake_i \wedge$ odd$(Fphase_i) \wedge$ from all $j \in lo_i$ a message has arrived }
      begin forall $j \in lo_i$
            do    receive $<X>$ from $j$ ;
                  forall $x \in X$
                  do    if $D_i[x] \geq \frac{1}{2}(Fphase_i + 3)$
                        then $D_i[x] := \frac{1}{2}(Fphase_i + 3)$; $dsn_i[x] := dsn_i[x] \cup \{j\}$
*                       elif $D_i[x] = \frac{1}{2}(Fphase_i + 1)$
*                       then $nnn_i[x] := nnn_i[x] \cup \{j\}$
                        fi
            od    od; $Fphase_i := Fphase_i + 1$;
            forall $j \in lo_i$
*           do    let $X = \{x \mid D_i[x] = \frac{1}{2}Fphase_i \wedge j \notin dsn_i[x] \wedge j \notin nnn_i[x]\}$;
                  send $<X>$ to $j$                          co send $<X, Fphase_i>$ to $j$ co
            od
      end

$R_i^{even}$ : { $awake_i \wedge$ even$(Fphi) \wedge Fphase_i > 0 \wedge$ from all $j \in hi_i$ a message has arrived}
      begin forall $j \in hi_i$
            do    receive $<X>$ from $j$ ;
                  forall $x \in X$
                  do    if $D_i[x] \geq \frac{1}{2}Fphase_i + 1$
                        then $D_i[x] := \frac{1}{2}Fphase_i + 1$; $dsn_i[x] := dsn_i[x] \cup \{j\}$
                        fi
            od    od; $Fphase_i := Fphase_i + 1$;                co $phase_i := phase_i + 1$ co
            forall $j \in hi_i$
*           do let $X = \{x \mid D_i[x] = \frac{1}{2}(Fphase_i + 1) \wedge j \notin dsn_i[x]\}$; send $<X>$ to $j$
            od;                                              co send $<X, Fphase_i>$ to $j$ co
            if not $\exists x$ with $D_i[x] = \frac{1}{2}(Fphase_i + 1)$
            then  forall $j \in lo_i$ do send $<\emptyset>$ to $j$ od;
                  $Fphase_i := Fphase_i + 1$; $awake_i := false$
            fi
      end
```

Note that the algorithm of Friedman is not a special case of the synchronous or simulated synchronous program skeleton because in R_i^{odd} messages of $phase_i$ are received while other messages of $phase_i$ are sent afterwards in the same operation.

Lemma 3.20. For all links (i, j) and all nodes x with $x \neq i$ the following holds invariantly.

(1) $Fphase_i > 0 \Rightarrow hi_i \cup lo_i = \{\text{all links incident to } i\} \wedge hi_i \cap lo_i = \emptyset$,

(2) $Fphase_i > 0 \wedge Fphase_j > 0 \Rightarrow j \in lo_i \wedge i \in hi_j \vee j \in hi_i \wedge i \in lo_j$,

(3) $<X, f> \in Q[i, j] \Rightarrow f = 0 \vee \text{odd}(f) \wedge j \in hi_i \vee \text{even}(f) \wedge j \in lo_i$,

(4) $<X, f>$ tail of $Q[i, j] \Rightarrow Fphase_i = f \vee Fphase_i = f + 1$,

(5) $<X', f'>$ **behind** $<X,f>$ **in** $Q[i,j] \;\Rightarrow\; f = 0 \wedge f' = 1 \vee f' = f+2$,
(6) $<X,f>$ **head of** $Q[i,j] \;\Rightarrow\; f = Fphase_j \vee f = Fphase_j + 1$,
(7) $phase_i = \lceil \frac{1}{2} Fphase_i \rceil$,
(8) $<X,f> \in Q[i,j] \wedge x \in X \;\Rightarrow\; d(i,x) = D_i[x] = \lceil \frac{1}{2} f \rceil$,
(9) $j \in dsn_i[x] \vee j \in nnn_i[x] \;\Rightarrow\; D_j[x] = d(j,x)$,
(10) **not** $\exists <X,f> \in Q[i,j]$ **with** $f = k \wedge (k = 0 \vee \text{even}(k) \wedge j \in lo_i \vee \text{odd}(k) \wedge j \in hi_i)$
$\Rightarrow$ **not** $awake_i \wedge Fphase_i = 0 \vee Fphase_i < k \vee Fphase_j > k$.

Proof. (1), (2), and (3) follow directly from the program skeleton.
(4). Note that R_i^{odd} and R_i^{even} can only become enabled alternately. If $Fphase_i$ is increased to $f+2$ and hence to a value with the same parity as f, a new last message is sent so the relation holds for the new message.
(5). Use statements (3) and (4).
(6) follows from program skeleton F and statements (4) and (5).
(7) is clear from program skeleton F.
(8). Although this is not a refinement of program skeleton SS, we claim that it is a special case of program skeleton $P1$: Messages of $phase_i$ (i.e., about nodes x with $d(i,x) = phase_i$) are sent when $phase_i = \frac{1}{2} Fphase_i$ and when $phase_i = \frac{1}{2}(Fphase_i + 1)$. As $phase_i = \lceil \frac{1}{2} Fphase_i \rceil$, messages of $phase_i$ are indeed sent in phase $phase_i$. Furthermore we have to show that the guard of operation P_i^{P1} in program skeleton $P1$ is true when $phase_i$ is increased in this program skeleton. In operation R_i^{even} $phase_i$ is increased. With statements (2), (3), (4) and (6) we have that $<X,f>$ is received when $f = Fphase_i$. As the messages of $phase_i$ are those with $f = 2.phase_i$ and $f = 2.phase_i - 1$ they have all been received when all messages on the HI links are received in operation R_i^{even}. Thus we can use theorem 3.4.
(9). A node j is added to a set $dsn_i[x]$ or $nnn_i[x]$ only upon receipt of a message $<X,f>$ from j where $x \in X$. With (8) we have the desired result.
(10). Initially **not** $awake_i \wedge Fphase_i = 0$ holds. Operation A_i^F falsifies the premise for $k = 0$ and for the other values of k $Fphase_i < k$ now holds. In general, $Fphase_i < k$ is falsified together with the premise. If the premise is validated by the receipt of a message $<X,k>$ by j, then afterwards $Fphase_j > k$ holds, which cannot be invalidated any more. ■

Hence the second field in messages containing the *Fphase* number can indeed be deleted.

Lemma 3.21. For all links (i,j) the following holds invariantly.
(1) $awake_i \wedge Fphase_i = 0 \wedge Q[j,i] = \emptyset \;\Rightarrow$ **not** $awake_j \wedge Fphase_j = 0 \wedge Q[i,j] \neq \emptyset$,
(2) $awake_i \wedge \text{odd}(Fphase_i) \wedge j \in lo_i \wedge Q[j,i] = \emptyset \;\Rightarrow\; awake_j \wedge Fphase_j < Fphase_i$,
(3) $awake_i \wedge \text{even}(Fphase_i) \wedge j \in hi_i \wedge Q[j,i] = \emptyset \;\Rightarrow\; awake_j \wedge Fphase_j < Fphase_i$.

Proof. (1) follows from lemma 3.20(10).
Except for the statement $awake_j$ in the conclusion, (2) and (3) also follow from lemma 3.20(10). Let $\text{odd}(Fphase_i)$ hold. Then $awake_i$ implies that there is some node x with $d(i,x) = \frac{1}{2}(Fphase_i + 1)$ and hence also a (possibly different) node x with $d(j,x) = \frac{1}{2}(Fphase_i - 1)$. As $Fphase_j < Fphase_i$ we have $awake_j$. On the other hand, let

even$(Fphase_i)$ hold. Then $awake_i$ implies that there is some node x with $d(i,x) = \frac{1}{2}Fphase_i$ and hence also a (possibly different) node x with $d(j,x) = \frac{1}{2}(Fphase_i - 2)$. Hence $awake_j$ holds for $Fphase_j \leq Fphase_i - 2$, and as becoming **not** *awake* is only done in even *phases*, $awake_j$ also holds for $Fphase_j = Fphase_i - 1$. ■

Theorem 3.22. The algorithm of Friedman is correct.

Proof. That the second field in messages $<X,f>$ can be deleted follows from lemma 3.20(6). The partial correctness follows from lemma 3.20. The freedom of deadlock follows from lemma 3.21. ■

Although this is the direct translation of the ideas of Friedman for his algorithm, it is by no means the same as the algorithm he stated for this purpose, which we cite now:

Step 0:	Node pairs exchange identities and choose HI and LO.
Step l:	All HI nodes broadcast to their corresponding LO neighbors,
l odd	as in Gallager's algorithm, new identities learned at **Step** l-1.
Step l:	All LO nodes broadcast to their corresponding HI neighbors
l even	new identities learned at **Step** l-2.

Termination is as in Gallager's algorithm.

It is immediately clear that this cannot be correct, as only new identities learned in even steps are sent through. Hence information received in LO link ends will never be sent through. Moreover, as the algorithm is stated now, information is still sent twice over a link, as trying out the algorithm on a cycle of length 3 shows.

4. Minimum hop distances in a dynamic network.

In a dynamic network, links can go down and come up, as can the nodes themselves. Hence the minimum hop distance between two nodes can change in time. We have specified what we exactly mean by "going down" and "coming up" in section 2.2. The precise formulation of the assumptions is important, as people tend to make slightly different assumptions and it does make a difference for the correctness proofs.

In the next section we discuss some different problems one encounters in algorithm design for dynamic networks. We then give the algorithm of Tajibnapis [Tj] for comparison and in section 4.2 we give a complete correctness proof of the algorithm of Chu [C].

4.1. Comparison with the static case.

As a dynamic network is inherently asynchronous according to our assumptions, we only have the message-driven model of computation, and the simulated synchronous model of computation, the latter being a message-driven model of computation which incorporates the idea of phasing. In both models, the partial correctness relies heavily on the facts that the $D_i[x]$ are decreasing and approximate $d(i,x)$ from above (lemma 3.1). A consequence of this is lemma 3.2, which states that a finite entry in $D_i[x]$ reflects the existence of a path from x to i. As the minimum hop distance of two nodes can increase if a link goes down the relation $D_i[x] \geq d(i,x)$ cannot be an invariant. In fact, in all algorithms that we have seen, it is the case that, as soon as a node receives information that this relation might not hold any more, it sets $D_i[x]$ to ∞.

There are basicly two extremes in adapting static algorithms for use in a dynamic network, both with their own advantages and disadvantages. One extreme is to process all new information as it comes in, discarding the old information, and sending it on immediately. Simple though this might seem, it leads to inconsistent information between network nodes, due to the asynchronicity of the network. In the case of minimum hop routing, it is easy to construct examples such that node i has node j as its downstream neighbor for routing towards x, while node j has node i as its downstream neighbor. This problem is referred to as "not loop free" in the literature. The algorithm of Tajibnapis is an example of this. The algorithm of Chu, called the predecessor algorithm by Schwartz [Sch], is an adaptation of this which avoids some loops (those of length two).

The other extreme is, to first discard all old information in the whole network, and reset all nodes to an initial state before restarting the static algorithm anew. Clearly this makes routing in the whole network impossible for some time, even in those parts of the network that are not affected. However, in combination with a minimum hop algorithm such as that of Gallager, routing is loop free. Of course, any algorithm that defers all routing until lemma 3.2 holds again, is loop free. Resetting all nodes before restarting could be done by a resynch algorithm such as Finn's [Fi], but usually deferring all routing until the whole network is reset and all routing tables refilled, is too high a price to pay. Therefore people have tried to economize on this aspect while retaining loop freedom.

One approach, suitable for algorithms which work independently for all destinations, is to restrict the resetting of the network and deferring the routing for the affected destinations only. The algorithm of Merlin and Segall [MeSe] makes use of this. Another approach is to economize on complete resynchronization and only partially resynchronize, i.e., we do not demand that before restarting the minimum hop algorithm all nodes are reset as with complete resynchronization, but only the the neighbors of the node restarting. This is done in the fail-safe version of Gallager's algorithm by Toueg [To]. Due to the very special order in which routing tables are filled, it is known during the execution of the algorithm which information is new and which is old.

A basic problem with running different versions of the same algorithm in an asynchronous environment is keeping them apart. One solution which usually is not choosen, is to wait with restarting a new version of the algorithm until the previous one has terminated. A reset algorithm by Afek et al. [AAG] could be used to force termination of the minimum hop algorithm together with resetting the subnetwork where the minimum hop algorithm was still in progress. This reset algorithm was proven correct in [DrS]. The usual solution is to number the different versions of the algorithm in some way, see for example Finn [Fi]. Toueg [To], in his adaptation of Gallager's algorithm, uses Lamport's concept of logical clocks [La1]. Both numberings use numbers which are not bounded. Note that algorithms such as an extension of the Merlin-Segall algorithm [Se] which rely on Finn's [Fi] idea of sending (bounded) differences of version numbers are probably not correct. As Soloway and Humblett [SoHu] showed, this algorithm of Finn is not correct as it can generate an infinite number of restarts after all topological changes have ceased. Soloway and Humblett introduced however a new algorithm to be able to use bounded sequence numbers, based on Gallager's minimum hop algorithm. As there is an essential difference in the assumptions about the network, we do not yet know if this works in our model, too.

Finally, Jaffe and Moss [JaMo] presented an algorithm based upon both the algorithm of Merlin and Segall and the algorithm of Tajibnapis, that is still loop free. They realized that the problem of looping only occurs when distances increase. Thus they use the algorithm of Tajibnapis for distance decreases, and the algorithm of Merlin-Segall for increases. It was recently repaired and proved correct by van Haaften and van Leeuwen.

4.2. Concrete algorithms. In the sequel we give a general introduction to the Tajibnapis' algorithm, together with the program skeleton and the most important invariants for comparison with program skeleton $M1$ (section 3.1.2) and the algorithm of Chu. The latter is an extension of the former and is proven correct in sections 4.2.2 and 4.2.3.

4.2.1. The algorithm of Tajibnapis. In program skeleton $M1$, if a message $<x,l>$ is received by node i from node j, we can recompute $D_i[x]$ as the minimum of the old value of $D_i[x]$ and $l+1$. This is due to the fact that we know that the estimate l of $D_j[x]$ is decreasing, hence we can simply take the minimum. In the case that links can go down however, this l might be larger than the previous estimate which i received from j. Thus node i now cannot recompute the new minimum, unless it has stored the latest estimates from its other neighbors to use for the computation. Hence nodes keep track of which distance information was received from which neighbor.

The algorithm of Tajibnapis contains one other new feature, which requires another assumption. It is necessary that the total number of nodes in the network, or at least an upper bound of this number, is known beforehand.

Assumption 4.1. All nodes in the network know an upper bound of the total number of nodes in the network.

We will denote this number known by N. The reason that we need this number is that in the case that the network becomes disconnected, the distances between nodes become infinity. As the algorithm tends to increase the estimates of distances with one hop at a time, we need a way to "jump to the conclusion" that the distance is infinity to prevent that the program will never terminate. What is used here is the observation that if the total number of nodes is N, the largest possible finite distance between nodes is $N-1$. Thus the number N can be interpreted as ∞ in this context.

Apart from the atomic operations U_{ij} and D_{ij} from section 2.2, which reflect link (i,j) coming up or going down, respectively, we add as extra atomic actions RU_i^T: node i receives the message $<up>$ and RD_i^T: node i receives the message $<down>$. Nodes maintain a table *Dtab*, in which $Dtab_i[x,j]$ contains for every destination $x \neq i$ and every (current) neighbor j, the last distance information it received from j. Since the set of neighbors is not fixed any more, this set is maintained in $nbrs_i$. Program skeleton T is as follows.

```
Initially ∀ i:   $nbrs_i = \varnothing,\ D_i[i] = 0,$
                 ∀ j with $i \neq j$:  $linkstate(i,j) = \text{down},\ Q[i,j] = \varnothing,$
                                $D_i[j] = N,\ dsn_i[j] = \text{none},$
                                ∀ x with $x \neq i$:  $Dtab_i[x,j] = \infty.$
```

```
RU$_i^T$ :{an <$up$> has arrived from $j$}
    begin receive <$up$> from $j$ ; add $j$ to $nbrs_i$ ;
        $Dtab_i[j,j] := 0$; $D_i[j] := 1$; $dsn_i[j] := j$ ;
        forall $x \in nbrs_i$ with $x \neq j$ do send <$j$, 1> to $x$ od;
        if $| nbrs_i | = 1$
        then forall $x$ with $x \neq i \wedge x \neq j$ do $D_i[x] := N$ ; $dsn_i[x] := j$ od
        fi;
        forall $x$ with $x \neq i \wedge x \neq j$
        do $Dtab_i[x,j] := N$ ; send <$x$,$D_i[x]$> to $j$ od
    end

RD$_i^T$ :{a <$down$> has arrived from $j$}
    begin receive <$down$> from $j$ ; delete $j$ from $nbrs_i$ ;
        forall $x$ with $x \neq i$
        do    $Dtab_i[x,j] := \infty$;
              if $nbrs_i \neq \emptyset \wedge dsn_i[x] = j$
              then $olddist := D_i[x]$;
                   choose $ndsn \in nbrs_i$ such that
                         $Dtab_i[x,ndsn] = \min_{a \in nbrs_i} Dtab_i[x,a]$;
                   $dsn_i[x] := ndsn$ ; $D_i[x] := \min(N, 1+Dtab_i[x,ndsn])$;
                   if $olddist \neq D_i[x]$
                   then forall $a \in nbrs_i$ do send <$x$,$D_i[x]$> to $a$ od
                   fi
              elif $nbrs_i = \emptyset$ then $D_i[x] := N$ ; $dsn_i[x] :=$ none
              fi
        od
    end

R$_i^T$ : {a message <$x$,$l$> has arrived from $j$}
    begin receive <$x$,$l$> from $j$ ;
        if $x \neq i \wedge j \in nbrs_i$
        then $Dtab_i[x,j] := l$ ;
             if $dsn_i[x] = j \vee l+1 < D_i[x]$
             then $olddist := D_i[x]$;
                  choose $ndsn \in nbrs_i$ such that
                        $Dtab_i[x,ndsn] = \min_{a \in nbrs_i} Dtab_i[x,a]$;
                  $dsn_i[x] := ndsn$ ; $D_i[x] := \min(N, 1+Dtab_i[x,ndsn])$;
                  if $D_i[x] \neq olddist$
                  then forall $a \in nbrs_i$ do send <$x$,$D_i[x]$> to $a$ od
        fi    fi    fi
    end
```

We used the symbol ∞ as well as the symbol N for infinite distances. This is to reflect the difference between "throw all information away", e.g. delete column j from the array $Dtab_i$, and "set the value to N", e.g. as initialization for an added column j of the array $Dtab_i$. However, we are not interested in an actual implementation. It also depends on the implementation whether all variables mentioned actually have to be maintained or can be deduced from other values of variables. For example, if obsolete columns of the array $Dtab_i$ are actually thrown away, $nbrs_i$ corresponds to the columns actually present in $Dtab_i$. On the other hand, if the entire column is set to the value N, $j \in nbrs_i$ if and only if $Dtab_i[j,j] = 0$.

Note that the operation A_i^{M1} "awaken" of section 3.1.2 is divided as it were over all links and incorporated in RU_i^T for one link. Part of the work done in RU_i^T corresponds to what would be done in operation R_i^T if the message received from j would be $<j,0>$. In fact operations RU_i^T and RD_i^T could be incorporated in R_i^T if $<up>$ is coded as $<j,0>$ and $<down>$ as $<j,\infty>$, and the extra code added for the extra work to be done in those special cases. We feel that this formulation would obscure the special status of these messages.

We will now state but not prove the basic invariants that lead to the partial correctness of this algorithm, for comparison with lemma 3.6 and the invariants of the algorithm of Chu.

Theorem 4.1. (Lamport [La]) For all i, j, and x with $i \neq j$, $x \neq i$, and $x \neq j$ the following holds invariantly.

(1) $linkstate(i,j) = \text{down} \Rightarrow <down>$ **last control message in** $Q[j,i] \vee$
$Dtab_i[x,j] = \infty \wedge Dtab_i[j,j] = \infty$,

(2) $linkstate(i,j) = \text{up} \Rightarrow$
$<up>$ **last control message in** $Q[j,i] \vee$
$Dtab_i[j,j] = 0 \wedge (Dtab_i[x,j] = D_j[x] \vee$
$<x,D_j[x]>$ **after any control message in** $Q[j,i])$.

4.2.2. The algorithm of Chu.

The problem with the previous algorithm is that if a node i receives information from its neighbor j that it is l hops away from x, i has no way to know whether this route goes through i itself. This leads to a slow propagation of distance updates in case a link has gone down. The algorithm due to Chu [C] maintains this extra information, thus maintaining a sink tree for every destination.

In the algorithm of Tajibnapis $dsn_i[x]$ is maintained, the downstream neighbor to which i should route messages for x. Now we say that j is upstream from i for destination x if i is downstream from j. If i's downstream link for x happens to go down, it is clear that we should not choose an upstream neighbor as i's new downstream neighbor, but some other neighbor. If there is no non-upstream neighbor, i sends a message $<x,N,1>$ upstream saying "help, my route to destination x is blocked" and waits until a route to x is found via another node.

Node i maintains its sink tree information in a table T_i, where $T_i[x,j] = d$ or u means: neighbor j is downstream or upstream for destination x, respectively, and $T_i[x,j] = n$ means: neighbor j is neither downstream nor upstream for destination x.

To the messages that are sent an extra field is added to convey this extra information: node i sends messages $<x,D_i[x],1>$ to j if $T_i[x,j] = d$ (j is downstream neighbor) and messages $<x,D_i[x],0>$ to j if $T_i[x,j] \neq d$. We also have to send messages in a situation where we did not do so in the previous algorithm, namely in the case that the minimum hop distance to a node x stays the same but the downstream neighbor is changed, we have to inform the old and new downstream neighbor of this change.

We now give the text of the algorithm of Chu (program skeleton C). Apart from notating it in our own way for comparison to the other program skeletons, we had to make some slight changes to be able to prove the algorithm correct. These are marked with an asterisk (*) at the beginning of the line. The assumptions concerning the model of communication are still the same, hence these operations are to be augmented with operations D_{ij} and U_{ij} for all i and $j \neq i$ from section 2.2.

```
    Initially ∀ i:   $nbrs_i = \emptyset,\ D_i[i] = 0,$
                     ∀ j with $i \neq j$:  $linkstate(i,j) =$ down, $Q[i,j] = \emptyset,\ D_i[j] = N,$
                                  ∀ x with $x \neq i$:  $Dtab_i[x,j] = \infty,\ T_i[x,j] = n.$

    $RD_i^C$ : {a <down> has arrived from $j$}
        begin receive <down> from $j$ ; delete $j$ from $nbrs_i$ ;
            forall $x$ with $x \neq i$
            do   $Dtab_i[x,j] := \infty$;
                 if $nbrs_i \neq \emptyset \wedge T_i[x,j] = d$
                 then $olddist := D_i[x]$;
                      if $\exists\, a \in nbrs_i$ with $T_i[x,a] \neq u$
                      then choose $ndsn \in nbrs_i$ such that $Dtab_i[x,ndsn] = \min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x,a]$;
                           $T_i[x,ndsn] := d$ ; $D_i[x] := \min(N, 1+Dtab_i[x,ndsn])$;
                      else choose $ndsn \in nbrs_i$ ; $Dtab_i[x,ndsn] := N$ ;
                           $T_i[x,ndsn] := d$ ; $D_i[x] := N$ ;
                           forall $a \in nbrs_i$ with $a \neq ndsn$
                           do $Dtab_i[x,a] := N$ ; $T_i[x,a] := n$ od
                      fi; send $<x, D_i[x], 1>$ to $ndsn$ ;
                      if $olddist \neq D_i[x]$
                      then forall $a \in nbrs_i$ with $a \neq ndsn$
                           do send $<x, D_i[x], 0>$ to $a$ od
                      fi
*                elif $nbrs_i = \emptyset$ then $D_i[x] := N$
                 fi; $T_i[x,j] := n$
            od
        end
```

```
R$_i^C$ :  {a message <$x,l,t$> has arrived from $j$}
     begin receive <$x,l,t$> from $j$ ;
*          if $x \neq i \wedge j \in nbrs_i$
           then  $olddist$ := $D_i[x]$;  let $odsn$ such that $T_i[x,odsn] = d$ ;
                 if $t = 1$
                 then  $Dtab_i[x,j] := l$;  $T_i[x,j] := u$ ;
                       if $olddist < N \wedge odsn = j$
                       then  if $\exists\ a \in nbrs_i$ with $T_i[x,a] \neq u$
                             then choose $ndsn \in nbrs_i$ such that $Dtab_i[x,ndsn] =$
                                   $\min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x,a]$;
                                   $T_i[x,ndsn] := d$;  $D_i[x] := \min(N, 1+Dtab_i[x,ndsn])$
                             elif $|nbrs_i| > 1$
                             then  choose $ndsn \in nbrs_i$ with $ndsn \neq j$;  $D_i[x] := N$ ;
                                   $Dtab_i[x,ndsn] := N$;  $T_i[x,ndsn] := d$ ;
                                   forall $a \in nbrs_i$ with $a \neq ndsn$
                                   do $Dtab_i[x,a] := N$;  $T_i[x,a] := n$ od
                             else  $Dtab_i[x,j] := N$;  $T_i[x,j] := d$ ;
                                   $D_i[x] := N$;  $ndsn := j$
                             fi;  send <$x,D_i[x]$,1> to $ndsn$ ;
                             if $D_i[x] \neq olddist$
                             then  forall $a \in nbrs_i$ with $a \neq ndsn$
                                   do send <$x,D_i[x]$,0> to $a$ od
                             elif $j \neq ndsn$ then send <$x,D_i[x]$,0> to $j$
                             fi
*                      elif $odsn = j$ then $T_i[x,j] := d$;  $Dtab_i[x,j] := N$
                       fi
                 else  $Dtab_i[x,j] := l$;  $T_i[x,j] := n$ ;
                       choose $ndsn \in nbrs_i$ such that $Dtab_i[x,ndsn] =$
                             $\min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x,a]$;
*                      if $Dtab_i[x,ndsn] = Dtab_i[x,odsn]$ then $ndsn := odsn$ fi;
                       $T_i[x,ndsn] := d$;  $D_i[x] := \min(N, 1+Dtab_i[x,ndsn])$;
                       if $odsn \neq ndsn$
                       then $T_i[x,odsn] := n$;  send <$x,D_i[x]$,1> to $ndsn$
                       fi;
                       if $D_i[x] \neq olddist$
                       then  forall $a \in nbrs_i$ with $a \neq ndsn$ do send <$x,D_i[x]$,0> to $a$ od
                       elif $odsn \neq ndsn$ then send <$x,D_i[x]$,0> to $odsn$
           fi    fi    fi
     end
```

```
$\mathrm{RU}_i^C$ : {an <up> has arrived from $j$}
    begin receive <up> from $j$; add $j$ to $nbrs_i$;
*       if $|nbrs_i| > 1$ then let $odsn$ such that $T_i[j,odsn] = d$; $T_i[j,odsn] := n$ fi;
        $T_i[j,j] := d$; $Dtab_i[j,j] := 0$; $D_i[j] := 1$;
        forall $x \in nbrs_i$ with $x \neq j$ do send <$j$,1,0> to $x$ od;
        forall $x$ with $x \neq i \wedge x \neq j$
        do  $Dtab_i[x,j] := N$;
*           if $|nbrs_i| = 1$
*           then $T_i[x,j] := d$; $D_i[x] := N$; send <$x$,$D_i[x]$,1> to $j$
*           else $T_i[x,j] := n$; send <$x$,$D_i[x]$,0> to $j$
            fi
        od
    end
```

4.2.2.1. Partial correctness. For the partial correctness we begin with proving some technical lemmas.

Lemma 4.2. For all i and j with $i \neq j$ the following holds invariantly.

(1) $linkstate(i,j) = \text{up} \Leftrightarrow j \in nbrs_i \vee$ <up> **last control message in** $Q[j,i]$,

(2) $linkstate(i,j) = \text{down} \Leftrightarrow j \notin nbrs_i \vee$ <$down$> **last control message in** $Q[j,i]$.

Proof. Obvious from operations U_{ij}, D_{ij}, RU_i^C, and RD_i^C. ■

Lemma 4.3. For all i, j, and x with $j \neq i$ and $x \neq i$, the following holds invariantly.

(1) $T_i[x,j] = d \vee T_i[x,j] = u \Rightarrow j \in nbrs_i$,

(2) $\exists! j$ **with** $T_i[x,j] = u \Leftrightarrow nbrs_i \neq \varnothing$,

(3) $j \in nbrs_i \Leftrightarrow Dtab_i[j,j] = 0$,

(4) $T_i[x,j] = d \Rightarrow$
$$D_i[x] = \min(N, 1 + Dtab_i[x,j]) \wedge Dtab_i[x,j] = \min_{a \in nbrs_i \text{ with } T_i[x,a] \neq u} Dtab_i[x,a],$$

(5) $D_i[i] = 0 \wedge D_i[x] > 0$,

(6) $<x,l,t> \in Q[j,i] \Rightarrow l > 0 \wedge x \neq j$,

(7) $x \neq j \Rightarrow Dtab_i[x,j] > 0$.

Proof. (1) and (2) are obvious from operations RU_i^C, RD_i^C, and R_i^C.

(3). Use statement (6) with operation R_i^C.

(4). Obvious from operations RU_i^C, RD_i^C, and R_i^C.

(5). $D_i[i] = 0$ initially and is never changed anymore. For $nbrs_i = \varnothing$ we have $D_i[x] = N$, otherwise statement (4) can be used together with statement (7).

(6). As $<x,l,t> \in Q[j,i]$ is only validated when j sends a message to i and j only sends messages with $x \neq j$ and $l = D_j[x]$, we have $l > 0$ with statement (5).

(7). For $x \neq j$, $Dtab_i[x,j]$ entries are only set to ∞, N, or to some value l from a received message $<x,l,t>$. With statement (6) we have $l > 0$. ■

Lemma 4.4. For all i, j, and x with $i \neq j$ the following holds invariantly.
linkstate(i,j) = down $\vee$ (1)
$<$*up*$>$ **last control message in** $Q[i,j] \vee$ (2)
$<x,l,t> \in Q[j,i] \Rightarrow$ (3)
the last message $<a,b,c> \in Q[j,i]$ with $a = x$ has $b = D_j[x] \wedge$
$(c = 1 \Leftrightarrow T_j[x,i] = d) \wedge <a,b,c>$ **after any control message in** $Q[j,i]$. (4)

Proof. Initially (1) holds. If (1) holds, it can be invalidated by operation U_{ij}, but then (2) will hold. If (2) holds, it can be invalidated by D_{ij}, but then (1) will hold. (2) can also be invalidated by operation RU_j^C, but then for those x that (3) is validated, (4) is validated for the same message. If (4) holds and is invalidated by operation R_i^C, then (3) is also invalidated. If (4) holds and is invalidated because j changes $D_j[x]$ or $T_j[x,i]$, we have the following cases.
Case 1: $i \notin nbrs_j$. Then (1) or (2) holds (lemma 4.1).
Case 2: $i \in nbrs_j$. Then a message is sent such that (4) holds. If (4) is invalidated because a control message is placed in $Q[j,i]$, then (1) or (2) now holds. If neither (1), (2) nor (3) holds for some x and a message is sent such that (3) is validated, this message is such that (4) holds for this message. ■

Lemma 4.5. For all i, j, and x with $i \neq j$ and $i \neq x$ the following holds invariantly.
linkstate(i,j) = down $\Rightarrow$
$Dtab_i[x,j] = \infty \wedge$ **no control message in** $Q[j,i] \vee$
$<$*down*$>$ **last control message in** $Q[j,i]$.

Proof. -Operation D_{ij} validates the premiss and places a $<$*down*$>$ in $Q[j,i]$ as last control message.
-Operation U_{ij} invalidates the premiss.
-Operation RU_i^C can only occur if $<$*down*$> \in Q[j,i]$ as last control message, if the premiss is true.
-If RD_i^C receives the last $<$*down*$>$ left in $Q[j,i]$, there will be no control message left and $Dtab_i[x,j]$ is set to ∞.
Other operations do not influence the variables involved. ■

Lemma 4.6. For all i, j, and x with $i \neq j$, $i \neq x$ and $j \neq x$, the following holds invariantly.
linkstate(i,j) = up $\Rightarrow$
$<$*up*$>$ **last control message in** $Q[i,j] \vee$ (1)
$<x,D_j[x],t>$ **after any control message in** $Q[j,i]$ **with** $(t = 1 \Leftrightarrow T_j[x,i] = d) \vee$(2)
$Dtab_i[x,j] = N \wedge T_j[x,i] = d \wedge$ **no control message in** $Q[j,i] \wedge$
$<x,N,t>$ **after any control message in** $Q[i,j] \vee$ (3)
$Dtab_i[x,j] = D_j[x] \wedge$ **no control message in** $Q[j,i] \wedge$
$(T_i[x,j] = u \Rightarrow T_j[x,i] = d)$. (4)

Proof. -Operation D_{ij} invalidates the premiss.
-Operation U_{ij} validates the premiss and (1).
-Operation RU_j^C can only be performed if (1) held. Afterwards (1) can still hold, or otherwise i is added to $nbrs_j$ and $<x,D_j[x],t>$ is sent to i. As $Q[j,i]$ is a queue, the message is placed after any control message. Hence (2) holds now.

-Operation RU_i^C. If (1) holds, it will still hold afterwards. If (2) holds, it will still hold afterwards, since the $<up>$ received occurred before the message under consideration. (3) nor (4) could hold beforehand.
-Operation RD_j^C. Either the premiss is not true or (1) continues to hold.
-Operation RD_i^C. (3) nor (4) could hold beforehand. (1) and (2) remain to hold.
-Operation R_i^C with a message from j. If (1) held, it still holds. If (2) held, it either still holds, or the message received is $<x, D_j[x], t>$ with $t = 1 \Leftrightarrow T_j[x,i] = d$. In the latter case we know that there can be no control message in $Q[j,i]$. R_i sets $Dtab_i[x,j] = D_j[x]$ and $T_i[x,j] = u \Leftrightarrow T_j[x,i] = d$, so (4) holds now, except in the very special case that $t = 1$ (hence $T_j[x,i] = d$), $olddist < N$, $odsn = j$, and $\forall a \in nbrs_i\ T_i[x,a] = u$, in which case $Dtab_i[x,j]$ is set to N, $T_i[x,j]$ to n or d, and $D_i[x]$ to N, and $<x,N,t>$ is sent to j. Hence (3) holds now. The other exception is the case that $D_j[x] < N$, $t = 1$, $olddist = N$, and $odsn = j$, where no messages are sent but $Dtab_i[x,j]$ is reset to N and $T_i[x,j]$ to d. (Otherwise there would be no downstream neighbor left for x.) For this case, we not only assume that lemma 4.5 held before operation R_i^C, but also that lemma 4.5 held with i and j interchanged. Let the statements (1) to (4) with i and j interchanged be statements (1') to (4'), respectively. We then know that (1') cannot hold, as $Q[j,i]$ contains no control messages. As $olddist = N$, $D_i[x] = N$, and $odsn = j$, we have $T_i[x,j] = d$. The message received was $<x, D_j[x], 1>$ with $D_j[x] < N$ and $T_j[x,i] = d$, hence $Dtab_j[x,i] < N$. Thus neither (3') nor (4') can hold, and (2') must hold. Thus $<x,N,t> \in Q[i,j]$. Since operation R_i^C cannot invalidate this, (3) holds now. If (3) or (4) held, we know with lemma 4.3 that (2) holds also.
-Operation R_i^C with a message from $k \neq j$. (1), (2), and (3) remain to hold. If (4) held, it either remains to hold or $Dtab_i[x,j]$ can be set to N and $T_i[x,j]$ to n or d in the case that $T_i[x,a] = u\ \forall a \in nbrs_i$, $olddist < N$ and $odsn = k$. Hence we can conclude $T_j[x,i] = d$ and $<x,N,t>$ is sent to j. Thus (3) holds now.
-Operation R_j^C with a message from i. (1) remains to hold. If (2) holds, and $D_j[x]$ and/or $T_j[x,i]$ change, a new message is sent which reflects these changes. Thus (2) now holds for this new message. If (3) holds, we have the following two cases.
Case 1: only $T_j[x,i]$ changes, then a message to reflect this change is sent to i and (2) holds now.
Case 2: the message $<x,N,t>$ is received. If (4) did not hold, we had $D_j[x] < N$ and $T_j[x,i] = d$. $Dtab_i[x,j]$ is set to N. Thus either $D_j[x]$ or $T_j[x,i]$ changes so a message is sent to i to this effect and (2) now holds.
If (4) holds, it continues to hold unless $D_j[x]$ or $T_j[x,i]$ is changed. In this case a message is sent to i so (2) holds.
-Operation R_j^C with a message from $k \neq i$. (1) remains to hold. (2) remains to hold unless $D_j[x]$ or $T_j[x,i]$ are changed, in which case (2) will hold for the new message sent. In case (3) holds, only $T_j[x,i]$ could be changed, but then a message will be sent such that (2) holds. If (4) holds, it will continue to hold unless (2) holds with changed $D_j[x]$ or $T_j[x,i]$. ■

Lemma 4.7. For all i and j with $i \neq j$ the following holds invariantly.
$linkstate(i,j) = \text{up} \Rightarrow$ $<up>$ **last control message in** $Q[j,i] \vee$
$Dtab_i[j,j] = 0 \wedge T_i[j,j] = d \wedge D_i[j] = 1 \wedge$ **no control message in** $Q[j,i]$.

Proof. If the last $<up>$ is received in RU_i^C, the variables specified are set to the right values. With lemma 4.2(7) we have that $Dtab_i[j,k]$ for $k \neq j$ is > 0, hence $T_i[j,j]$ does not change. Node j does not send any message about destination j (lemma 4.2(6)), hence $Dtab_i[j,j]$ is not changed. As $T_i[j,j] \neq u$, $Dtab_i[j,j]$ cannot be set to N either. ■

Lemma 4.8. For all i, j, and x with $i \neq j$ and $i \neq x$ the following holds invariantly.

$$T_i[x,j] = u \wedge Q[i,j] = \emptyset \wedge Q[j,i] = \emptyset \Rightarrow Dtab_i[x,j] = \min(N, D_i[x]+1).$$

Proof. Use lemma 4.2(1), lemma 4.1, lemma 4.5 and lemma 4.2 (4). ■

Lemma 4.9.

$$\begin{aligned} \text{TERM} \Rightarrow \forall i,j \text{ with } i \neq j:\ & linkstate(i,j) = \text{up} \Leftrightarrow j \in nbrs_i \wedge \\ & j \in nbrs_i \Rightarrow \forall x \neq i\ Dtab_i[x,j] = D_j[x] \wedge \\ & j \notin nbrs_i \Rightarrow \forall x \neq i\ Dtab_i[x,j] = \infty \wedge \\ & \forall x \neq i\ \ D_i[x] = \min(N, 1 + \min_{\forall j \neq i} Dtab_i[x,j]). \end{aligned}$$

Proof. TERM is equivalent with all queues are empty. Use lemmas 4.1, 4.2(4), 4.4, 4.5, 4.6 and 4.7. ■

Theorem 4.10. TERM $\Rightarrow \forall i,x$: $D_i[x] = \min(N, d(i,x))$.

Proof. We prove this by first proving that $D_i[x] \leq d(i,x)$ and secondly proving that $D_i[x] \geq \min(N, d(i,x))$.
(1). Let $d(i,x) = \infty$. Then for all possible values of $D_i[x]$, we have $D_i[x] \leq d(i,x)$. Let $d(i,x) = k < \infty$. $k = 0$ implies $i = x$ and $D_i[x] = 0$. For $k > 0$ there is some path $x = x_0$, x_1, ..., $x_k = i$ from x to i of length k. Thus all links (x_j, x_{j-1}) for $1 \leq j \leq k$ are up and $x_{j-1} \in nbrs_{x_j}$. Thus $Dtab_{x_j}[x, x_{j-1}] = D_{x_{j-1}}[x]$ and $D_{x_j}[x] \leq \min(N, 1 + D_{x_{j-1}}[x])$. Hence $D_i[x] = D_{x_k}[x] \leq D_{x_0}[x] + k = D_x[x] + k = k$. Note that this is not necessarily the path designated by the downstream neighbors.
(2). We use induction over k for the hypothesis $d(i,j) \geq k \Rightarrow D_i[j] \geq \min(N,k)$. For $k = 0$ we have $d(i,j) \geq 0$ and $D_i[j] \geq 0$. Assume $d(u,v) \geq k+1$. For all neighbors $a \in nbrs_u$ we have $d(u,a) = 1$ and thus $d(v,a) \geq k$ (triangle inequality). Hence $D_a[v] \geq \min(N,k)$. $d_u[v] = \min(N, 1 + \min_{a \in nbrs_u} Dtab_u[v,a]) = \min(N, 1 + \min_{a \in nbrs_u} D_a[v]) \geq \min(N, k+1)$. ■

Corollary 4.11. If we interpret N as ∞, then TERM implies $D_i[x] = d(i,x)$ for all i and x.

Proof. The longest finite distance between two nodes is at most $N-1$. Hence $d(i,x) \geq N$ implies $d(i,x) = \infty$. ■

This completes the proof of the partial correctness of the algorithm of Chu.

4.2.2.2. Total correctness.

For the total correctness, we still have to prove that if there are no more topological changes, the algorithm indeed terminates in finite time.

Theorem 4.12. The algorithm cannot deadlock.

Proof. If there is a queue which contains a message, then there is always an operation which can receive that message: either RU_i^C, RD_i^C, or R_i^C for $Q[j,i]$, depending on the nature of the message. If all queues are empty, there is termination by definition. ■

Thus we have to show that the algorithm cannot go on generating messages forever, in the case that there are no more topological changes after some time. For this purpose we define a function F of the system state to the set W of $N+1$ tuples of nonnegative integers. We define the following total ordering $<_W$ on W :

$(a_0, a_1, ..., a_N) <_W (b_0, b_1, ..., b_N)$ if

$\exists i$ **with** $0 \le i \le N$: $(a_i < b_i \wedge \forall j$ **with** $0 \le j < i$: $a_j = b_j)$.

As the a_i and b_i are nonnegative integers, this order relation on W is well-founded (i.e., there is no infinite decreasing chain). Thus, in order to prove the total correctness, it is sufficient to find a function F from the system state to W which is decreased by every operation if there are no more topological changes. We define F as follows:

$$F = (cm, m(1)+2d(1), ..., m(N-1)+2d(N-1), 2m(N)+d(N)),$$

where

cm = the total number of control messages ($<up>$ or $<down>$) in all message queues,

$m(k) = \sum_x m_x(k)$,

$d(k) = \sum_x d_x(k)$,

$m_x(k)$= the total number of messages $<y, l, t>$ with $x = y$ and $l = k$ in all message queues,

$d_x(k) = \sum_i \left[1+ | \{j \mid j \in nbrs_i \wedge D_i[x] = k \wedge Dtab_i[x,j] = Dtab_i[x,dsn]\} | \right]$,

dsn = downstream neighbor, i.e., $T_i[x,dsn] = d$.

Theorem 4.13. For all i, F is strictly decreased by operations RU_i^C, RD_i^C, and R_i^C.

Proof. RU_i^C decreases cm by one, as does RD_i^C.

Consider operation R_i^C. Let the received message be $<x, l, t>$. cm cannot be changed, nor $m_y(k)$ or $d_y(k)$ with $y \ne x$. Note that R_i^C cannot change the set $nbrs_i$. As it depends on the old and new values of $D_i[x]$ and the old and new downstream neighbor of i for x how F changes, we define for the moment *olddist* as the value of $D_i[x]$ before operation R_i^C, *newdist* as the value of $D_i[x]$ after operation R_i^C, and *odsn* and *ndsn* as the neighbor j of i with $T_i[x,j] = d$ before and after operation R_i^C, respectively. We distinguish the following cases.

Case 1: $olddist < newdist$. Then $d(olddist)$ decreases, and $d(newdist)$ and $m(newdist)$ increase. Hence F decreases.

Case 2: $olddist > newdist$. This is only possible if in the received message $<x, l, t>$ $t = 0$ and $l = newdist - 1$. Thus $m(newdist-1)$ decreases, while $m(newdist)$ and $d(newdist)$ increase.

Case 3: $olddist = newdist$.

Case 3.1: $olddist = newdist < N$.

Case 3.1.1: $odsn = ndsn$. Then $m(newdist)$ is not increased because no new messages are sent. However, $d(newdist)$ could increase, if there is now one more neighbor with minimal distance to x. This can only happen if the received message had $l = newdist - 1$. Hence $m(newdist-1)$ is decreased and F decreases. If $d(newdist)$ does not change, F decreases because $m(l)$ decreases.

Case 3.1.2: $odsn \ne ndsn$. Then $m(newdist)$ is increased by 2. If $d(newdist)$ increases, we have $l = newdist - 1$ as above, and $m(newdist-1)$ decreases. If $d(newdist)$ does not change,

it must be the case that $Dtab_i[x, ndsn] = Dtab_i[x, odsn]$, as R_i^C can only change one value of $Dtab_i$ (unless $newdist = N$) at the time, hence this case cannot occur. Thus $d(newdist)$ decreases by one. Hence $2d(newdist)+m(newdist)$ does not change, and as $m(l)$ decreases, so does F.
Case 3.2: $olddist = newdist = N$.
Case 3.2.1: $odsn = ndsn$. Thus no new messages are sent. However, $d(newdist)$ could change. If $t = 1$ in the received message, then $d(newdist)$ can only decrease. $d(newdist)$ can increase by one if the message received from j was $<x, N, 0>$ and $T_i[x, j] = u$ before R_i^C. Then $m(newdist)$ decreases by one and F is decreased because $2m_x(N)+d_x(N)$ is decreased.
Case 3.2.2: $odsn \neq ndsn$. If $olddist = N$ and $t = 1$, nothing happens in R_i^C, so $odsn = ndsn$. Hence $t = 0$. If $l < N$, then we would have $newdist < N$, thus $l = N$. So $d(newdist)$ cannot decrease, which implies $Dtab_i[x, odsn]$ remains N. Hence the downstream neighbor is not changed, and we conclude that this case cannot occur. ■

Corollary 4.14. If topological changes cease then the algorithm of Chu terminates in finite time.

5. References.

[AAG] Afek, Y., B. Awerbuch, and E. Gafni, *Applying static network protocols to dynamic networks*, FOCS 87.

[Ba] Baran, P., *On distributed communications networks*, IEEE Trans. Commun. Syst., CS-12 (1964), 1-9.

[C] Chu, K., *A distributed protocol for updating network topology*, Report RC7235, IBM T.J.Watson Research Center, Yorktown Heights, 1978.

[DrS] Drost, N.J., and A.A. Schoone, *Assertional verification of a reset algorithm*, Techn. Report RUU-CS-88-5, Dept. of Computer Science, University of Utrecht, Utrecht, 1988.

[Fi] Finn, S.G., *Resynch procedures and a failsafe network protocol*, IEEE Trans. on Comm., COM-27 (1979), 840-845.

[Fr] Friedman, D.U., *Communication complexity of distributed shortest path algorithms*, MIT thesis, 1978.

[JaMo] Jaffe, J.M., and F.H. Moss, *A responsive distributed routing algorithm for computer networks*, Report RC8479, IBM T.J.Watson Research Center, Yorktown Heights, 1980.

[Kn] Knuth, D.E., *Verification of link-level protocols*, BIT 21 (1981), 31-36.

[Kr] Krogdahl, S., *Verification of a class of link-level protocols*, BIT 18 (1978), 436-448.

[La] Lamport, L., *An assertional correctness proof of a distributed algorithm*, Science of Computer Programming 2 (1982), 175-206.

[La1] Lamport, L., *Time, clocks, and the ordering of events in a distributed system*, Comm. ACM 21 (1978), 558-565.

[MeSe] Merlin, P.M., and A. Segall, *A failsafe distributed routing protocol*, IEEE Trans. on Comm. COM-27 (1979), 1280-1287.

[Sch] Schwartz, M., *Routing and flow control in data networks*, Report RC8353, IBM

T.J.Watson Research Center, Yorktown Heights, 1980.

[Scho] Schoone, A.A., *Verification of Connection-Management Protocols*, Techn. Report RUU-CS-87-14, Dept. of Computer Science, University of Utrecht, Utrecht, 1987.

[Se] Segall, A., *Advances in verifiable fail-safe routing procedures*, IEEE Trans. on Comm. COM-29 (1981), 491-497.

[SoHu] Soloway, F,R., and P.A. Humblett, *On distributed network protocols for changing topologies*, Techn. Report LIDS-P-1564, MIT, Cambridge, Mass., 1986.

[Ta] Tanenbaum, A.S., *Computer Networks*, Prentice-hall, Englewood Cliffs, 1981, pp. 209-210.

[Tel] Tel, G., *The structure of distributed algorithms*, Ph.D. thesis, Dept. of Computer Science, University of Utrecht, Utrecht, 1989.

[Tj] Tajibnapis, W.D., *A correctness proof of a topology information maintenance protocol for a distributed computer network*, Comm. ACM 20 (1977), 477-485.

[To] Toueg, S., *A minimum hop path failsafe and loop-free distributed algorithm*, Report RC8530, IBM T.J.Watson Research Center, Yorktown Heights, 1980.