13 Distributed Control Algorithms for AI

Gerard Tel

13.1 Introduction

Centralized intelligence currently makes place for networked, or distributed intelligence. The `Webster` program on my computer illustrates it all: it has no built-in dictionary, but responds my queries by Internet access to an American server, yet outperforms any lookup in a paper version. Collecting resources in any computer is uneconomical, specialized resource servers are easier to maintain, and access cost is low due to cheap communication technology.

Network Computations. Computations in networks of processing nodes, each holding a part of the inputs and/or resources initially, can roughly be classified into *centralized*, *duplicated*, or *distributed* computations. A centralized solution relies on one node being designated as the computer node and possessing the resources to process the entire application locally. All input data and relevant resources are sent to this node, and after local processing the computer sends the relevant output data to each of the other nodes. A duplicated solution sends all input data to each node, after which each node processes the entire application and throws away all output data except those it needs itself. The flagrant waste of computing resources can be economically justified only if the output data (which is not transported here) far exceeds the input data in size. Duplicated computation is used to compute routing tables in the Internet [24, Sec. 5.5].

This chapter concerns distributed solutions, where the processing steps of the application are divided among the participating nodes. Even when not explicitly based on a sequential algorithm, each distributed solution can be seen as containing a sequential one consisting of the combined computation steps of the participants. In addition the distributed solution contains communication actions for the exchange of intermediate results and coordination; our goal is to minimise communication and computation cost.

Afek and Ricklin [1] observe cost benefits of an intermediate strategy where computation is concentrated in several computing centres. Awerbuch and Peleg [4] reach similar conclusions, but a discussion of such solutions, though we would still consider them as distributed, is not possible in this chapter.

13.1.1 Model of Computation

The distributed model is characterised by a collection of autonomous processing elements, called *nodes*. In addition to some computing and storage resources, each node has the possibility to exchange information with some of the other nodes; these are referred to as its *neighbors* and the communication takes place through a *link* (also called *edge*).

We denote by n the number of nodes (or *size*) of the network and by m the number of links and thus the network can be represented as an undirected graph on n vertices and with m edges. We use D for the *diameter* of this graph. We assume the graph to be connected, which implies $m \geq n - 1$. It is not assumed that the nodes know this graph; representing it in every node would be costly, and contradicts the aim of processing each bit of input where it belongs. Storing some derived topological information, such as n, m, or the diameter of the graph, would be feasible, but it is usually superfluous.

The neighbour relation can be defined by the hardware, for example in processor networks where the neighbors are those processors to which the node is physically connected. Alternatively, the application can define this relation, for example, in Belief Networks [31], where each node stores information about a stochastic variable and communicates with nodes storing some related variables.

Symmetry. In this chapter it is not necessary to make a distinction between nodes on the basis of their resources (computing or storage nodes) or role (such as clients or servers), but we do assume two distinctions. First, nodes are identified by unique, uninterpreted tags (names) and initially each node knows its own tag and those of its neighbors (*neighbourhood knowledge* is assumed). Second, a single node is distinguished to act as an initiator of computations; this only means that this node executes a special program (usually just an additional start procedure), not that is has extra capabilities or resources.

These assumptions are natural because they can be met at low cost when implementing a distributed system; distributed algorithms research has investigated their influence on the power of the model.

In terms of network computing power, one of these assumptions suffices and they are equivalent [28]. If only identities are given, we may use an *election* program to choose one node as an initiator; such a program should of course not rely on the existence of an initiator, and would output, for example, the largest identity [13, 29]. If no initial identifiers are known while an initiator is distinguished, it may start a network traversal to assign unique names. A different situation arises in *anonymous* networks, where neither identities nor initiator are given; Rosenstiehl *et al.* [21] established 25 years ago that these networks can compute fewer functions. No function that requires to break symmetry can be computed deterministically; with randomised algorithms, naming and election can be performed, but only if the nodes initially know the size of the network [27, Chap. 9].

Communication. In this chapter, the communication between nodes is by *message passing* and has two operations, *send* and *receive*. Parameters for the send are: the *recipient*, which is a neighbors of the calling node, and the *message*, which is some piece of information; it will be transported to the mailbox of the recipient. The receive operation removes a message from the node's mailbox; it can complete only if a message is available, and returns the message and its sender. If there are no messages the operation is suspended; if there is more than one message, either of them can be returned.

We further assume that communication is *asynchronous*, which means that the completion of a send operation does not imply that the message has been received, or even, that it was delivered in the recipient's mailbox. All we assume is that it will eventually be available for reception because we assume *reliable* communication. The time between sending a message and its delivery at the receiver is unpredictable and may vary between channels and even between messages.

Only one temporal relation can be derived, namely, that the message is sent *before* it is received; we also assume that each message is received only once. This distinguishes message passing from communication by writing and reading a shared variable; it is cumbersome to ensure that each item written to such a variable is read (and processed) by the reading node exactly once.

13.1.2 Complexity Measures

The asynchrony in the communication causes the execution model to be non-deterministic; indeed, a distributed program may allow different executions on the same data depending on the scheduling of the events. When discussing complexity we shall always consider the *worst case* over all possible schedules.

Communication, Time, and Storage. The first goal in analysis of distributed algorithms is to compute the amount of communication by the algorithm; usually, as the number of messages exchanged in a computation. Only if messages in some solution are exceptionally large (contain more than, say, a few data items), we must be more precise and count the bits in each message, for example, in the "linear" depth-first search algorithm of Section 13.2.1.

The time complexity represents the duration of the computation, and is expressed in terms of the *slowest* message in the computation. This compares with the classical sequential time complexity, which does not really measure time but instead counts consecutive operations. The parallelism in the distributed model complicates matters slightly. Consider node p sending messages m_1 and m_2 to neighbors q and r, respectively; after receiving m_1, node q sends message m_3 to r and this message arrives at r before message m_2. This small example, illustrated by the *space-time diagram* of Figure 13.1, contains a *message chain* of length 2, namely message m_1 followed by m_3 (sent after receipt of the former). However, the entire chain is formed during the transmission of the single message m_2, and hence we say the time complexity of the example is 1.

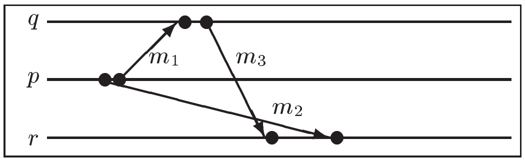

Figure 13.1 Two messages in one time unit.

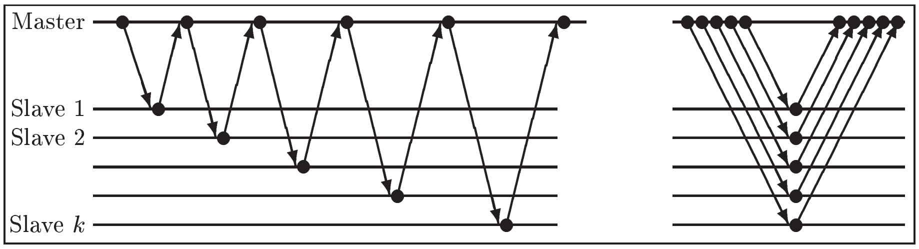

Figure 13.2 The local processing controversy.

We ignore local processing when computing time complexities; the time involved in processing is considered "small compared to the transmission delays." Two small examples may illustrate the controversy; see Figure 13.2. First, a server p polls k neighbors *one by one* by sending a message and receiving a reply; each request being sent upon receipt of the previous reply. The time complexity is $2k$ because polling each neighbors costs 2 time units, one for the request and one for the reply, and this repeats k times. Ignoring the processing time between receiving a message and sending the next one appears acceptable, because no waiting is involved, and it does not change the asymptotical complexity.

Alternatively the server may poll its neighbors in parallel by sending a request to each and then collect the answers; *now the time complexity is 2* because all requests are sent without waiting, and the last reply must arrive at the server after at most 2 time units. The definition of time complexity assumes that the time for sending the k requests, and the time for processing the k replies, is negligible.

The *storage complexity* expresses the amount of memory used by the algorithm; sometimes computed in bits, but it is usually convenient to assume larger units, "words," each capable of storing an identifier or integer.

Discussion. We might be tempted to compare the complexity of a distributed algorithm to the complexity of sequential algorithms for the same problem. When comparing, sequential *time* should not be compared to distributed *time*, but to message complexity. Indeed, in the sequential model, "time" actually measures the total amount of "work" because time actually counts the instructions executed sequentially. In the distributed model, "work" corresponds to messages, because the work performed by nodes can usually be charged to sending and receiving messages; see Algorithm 13.15–13.18 for an example. Distributed time accounts for the speedup that is achieved by the parallelism inherent in the model, but also penalizes for nodes that must wait for data before continuing their execution.

It is usually observed that the communication complexity for processing the network topology is at least linear in m, which is the input size for topological problems. For graph exploration, for example, this can be shown formally [27, Chap. 6] because each link must carry at least one message. Task that require processing a constant amount of information for each node (such as a sum of distributed inputs, see Alg. 13.11) can be performed with $O(n)$ messages using spanning trees or cycles. A message complexity below $O(n)$ is not possible for tasks that require cooperation of each node.

The worst case w.r.t. time complexity occurs when all messages are exchanged one after the other, and the time complexity then equals the message complexity. A straight-forward distribution of a sequential algorithm (see Section 13.2.1) often results in both message and time complexity being equal to the sequential time complexity (i.e., $\Theta(m)$). If all processing for each node can be performed in constant time, the overall time complexity becomes *linear* in the size of the network (n) and this is often possible, as we will see.

A *fast* algorithm is one that uses sub-linear (i.e., $o(n)$) time. Fast computing is not easy; Garay *et al.* [11] present an algorithm for Minimal Spanning Tree that runs in $O(D + n^{0.614})$ time, but the solution appears a bit artificial. The message complexity of the fast solution is large (unknown though), while message optimal solutions (exchanging $O(n \log n + m)$ messages) exist with linear time complexity [3, 10]. We shall discuss a distributed depth-first search algorithm whose time is proportional to the depth of the DFS tree, while again message complexity rises sky-high.

The network diameter serves as a time lower bound for all tasks that require coordination between all nodes (including every task that requires consensus in the output), because no information can be communicated across the network in $o(D)$ time. Linial [15] gives examples of tasks (Maximal Independent Set, Colouring) that can be solved by *local computations*, i.e., in sub-diameter time, and Litovsky *et al.* [16] have further investigated the power of local computations.

13.1.3 Examples of Distributed Architectures in AI

Distribution may be driven by several factors, such as the wish to speed up computations by using more hardware, or the availability of resources.

Multiprocessor computers. Sometimes an application can be processed by a single computer (in a sequential model) but this is just too slow. The solution is a *multiprocessor computer*, such as an array of 16, 128, or more processors connected by a high-speed communication network. The steps of the computation must be allocated over the available machines, but as the architecture does not match the application, this is usually a difficult task. The ideal allocation shares the computation load as good as possible, while achieving a low communication overhead (due to the exchange of intermediate results). See Section 13.4.5.

Resource distribution. In some situations distribution is not a choice, but is enforced by the availability of necessary resources at different locations. Consider, for example, the problem of planning several university committee meetings. Committees that share a professor may not overlap their meetings, but to decide if some date is available, the member's agenda must be consulted. Finding out if the members can go from one meeting to the other in time requires inspection of bus and train time tables. To see if rooms are available, the cooperation of the room reservation systems at the various universities is necessary.

Each of the mentioned resources runs at a fixed location, so the planning application must include distributed problem solving; we shall consider distributed constraint satisfaction in Section 13.4.

Belief Networks. A Belief Network models hypotheses and statistical dependencies between them in a graph. The computations to update the information in this network are naturally distributed over the nodes, where each node may need information from its neighbors to do the update. Each node in the graph can be described as a process, communicating with its neighbors processes. The physical location of the processes then becomes irrelevant: for the application it does not matter if all nodes are on the same machine, or distributed over various machines. In Section 13.5 we show that processing the network structure (computation of a loop cutset) can be described in the same model.

13.2 Graph Exploration

This section describes algorithms to compute spanning trees in an undirected network, that is, partition the set of edges into *tree* edges (these will be directed from parent to child) and *non-tree* edges. At the end no node will see the entire tree, but only the status of its own links (tree or non-tree).

The problem of computing weight-minimal trees has received attention in the literature [10], but where unit-cost links are assumed all trees are weight-minimal and we shall not address this problem. We illustrate the algorithms by giving pseudocode with Pascal-like (mostly self-explanatory) syntax set in typewriter font. As a convention we shall use a subscript u when reasoning about a variable of node u (as in la_u), but because in a distributed algorithm a node can access only its own variables, the subscript is omitted from the pseudocode.

13.2.1 Depth-First Search

In the sequential setting, depth-first search has been in wide use since the late 1950's, especially in Artificial Intelligence, as a technique for exploring solution

```
var visited[u]: bool init false ;

procedure dfs(u):
      if not visited[u]
      then  begin visited[u] := true ;
                  forall v in Neigh[u] do dfs(v)
            end

Start the algorithm: dfs(u0)
```

Algorithm 13.1 Sequential depth-first search.

spaces for problems. Its importance for graph processing was recognised by Hopcroft and Tarjan and results from the simplicity of the algorithm ($O(m)$ sequential time) combined with an attractive structural property of the constructed tree, namely, that the two endpoints of any non-tree edge are connected by a *directed* path in the tree.

Sequential depth-first search is implemented by a short recursive procedure (Alg. 13.1); the first call of $\mathit{dfs}(u)$ recurses on all neighbors, while subsequent calls return immediately. Calls to node u may be nested, i.e., a second call to u may occur while the first one is still active, but in this case the second call returns immediately because $\mathit{visited}[u]$ is set when entering the procedure. The start node u_0 is the root of the constructed tree, and each other node becomes a child of the neighbor from which the first call of $\mathit{dfs}(u)$ was made (this is not shown in Alg. 13.1). Alg. 13.1 makes two recursive calls through each edge.

First distributed solution. In the distributed model, control is passed from one node to the other by exchange of a message, so each recursive call uses two messages: one for the call and one for the return. Some saving can be achieved; node u will *not* place a call to its father, and node u will *not* call the procedure on neighbour v if a call was earlier received *from* v and returned. The reason is in both cases that the neighbour has already been visited and would return the call immediately.

To describe the operation of node u in more detail, consider the receipt of a message from neighbour v. If u has sent a message to v earlier, the received message is a return message and u selects a next neighbour for placing a call; when the neighbors are exhausted, u sends a return message to its father or, if u is the initiator, terminates. Otherwise, the message is a call from v; if this is the first call for u, designate v as the *father* and send a message to another neighbour. If a call was received before, a return message is sent to v immediately.

Alg. 13.2 uses variable $\mathit{status}_u[v]$ to indicate if the link from u to v is *unused*, *father*, or *cal* or *ret* if a call or return was sent through the link. It is not necessary to use different messages for a call and a return because the nature of the message can be derived from the context as argued above. Consequently, the algorithm uses only a single type of message, denoted `[dfs]`.

```
var   visited      bool   init false ;
      status[v]           init unused          (* for each neighbor *)

Start the algorithm (initiator only!):
      visited := true ;
      for some w in Neigh do
          begin send [dfs] to w ; status[w] := cal end

Upon receipt of [dfs] from v:
      if not visited then
          begin visited := true ; status[v] := father end ;
      if status[v] = unused then
          begin send [dfs] to v ; status[v] := ret end
      else if there is a w with status[w] = unused then
          begin send [dfs] to w ; status[w] := cal end
      else if there is a w with status[w] = father then
          begin send [dfs] to w end
      else (* initiator *) stop
```

Algorithm 13.2 Distributed depth-first search (for u).

At the end, the link status is interpreted as follows. Each non-initiator has one *father* link, leading to its father in the constructed dfs tree. A *ret* link was used for returning a second or later call and hence indicates a non-tree link leading to a descendant in the dfs tree. A *cal* link was used for placing a call, and this link is either a tree link (if the call was the first one made on the neighbour) or a non-tree link leading to an ascendant. If nodes must be able to distinguish between downward tree links and upward non-tree links, this can be done by using two different return messages for returning the first and the subsequent calls.

Regarding the complexity of the algorithm, we observe that two messages are exchanged through each link, hence the communication complexity is $2.m$ messages. As they are exchanged one after the other, the time complexity is also $2.m$. The algorithm uses in each node a number of bits proportional to its degree.

Awerbuch's linear-time solution. Exactly $n-1$ of the edges become tree edges, so in the case that m significantly exceeds n, the time complexity of the algorithm is dominated by the calls and returns through *non-tree* edges. These calls do not construct edges of the dfs tree; so *if* node u could be aware of its neighbour v being visited already, the call to v could be skipped without affecting the outcome, and the time complexity would be reduced significantly.

This is exploited in Awerbuch's algorithm [2]; each node informs its neighbors when it is visited for the first time, before forwarding any recursive calls. Of course we still communicate through each edge, but informing the neighbors can be parallelized and we save on time. When forwarding calls, the node now skips neighbors that are known to be visited already (status *done*); see Alg. 13.3.

```
Start the algorithm (initiator only!):
    visited := true ;
    forall x in Neigh do send [visit] to x ;
    forall x in Neigh do receive [ack] from x ;
    for some w in Neigh do
        begin send [dfs] to w ; status[w] := cal end

Upon receipt of [visit] from v :
    status[v] := done ; send [ack] to v

Upon receipt of dfs from v:
    if not visited then (* first dfs is first call *)
        begin visited := true ; status[v] := father ;
            forall x in Neigh – {v} do send [visit] to x ;
            forall x in Neigh – {v} do receive [ack] from x
        end ;
    if there is a w in Neigh with status[w] = unused then
        begin send [dfs] to w ; status[w] := cal end
    else if there is a w in Neigh with status[w] = father then
        begin send [dfs] to w end
    else (* initiator *) stop
```

Algorithm 13.3 Awerbuch's distributed depth-first search.

It uses three types of messages, namely `[dfs]` as before for the calls and returns, `[visit]` messages to indicate that the sender was visited, and `[ack]` messages to acknowledge these. The status of a link can be *unused*, *father*, *cal*, or *done* to indicate that no `[dfs]` message was exchanged, but the neighbour has been visited. The *ret* status is not used because no node ever receives a second call message; the corresponding clause of Alg. 13.2 is removed from the response to a `[dfs]` message.

The algorithm still communicates only a constant number of bits per edge, but the message complexity is now $4.m$, which is seen as follows. On a tree edge uv, u informs v about being visited at the expense of two messages, and the call on v by u costs two messages; no `[visit]` message is sent by v to its father. On a non-tree edge uv the nodes u and v mutually inform each other of being visited, both at the cost of two messages.

The algorithm exchanges 2 `[dfs]` messages through $n-1$ links, to a total of $2n-2$ messages and these are exchanged in a chain. Each time a node is visited for the first time the flow of `[dfs]` messages is interrupted for exchanging `[visit]` and `[ack]` messages, which takes two time units. Hence the time complexity is bounded by $4n-2$. A slightly better result was obtained by Cidon [6].

Linear-message solution. Calls and returns through non-tree edges can be avoided without sending additional messages; see Hélary *et al.* [12]. In these solutions a node is not informed about a neighbour being visited by receiving from that neighbour, but instead the call and return messages include a complete list

```
Start the algorithm (initiator only!):
    S := { u } ;
    for some w in Neigh do
        begin send [dfs,S] to w ; status[w] := cal end

Upon receipt of [dfs,S] from v :
    if not (u in S) then (* first message is first call *)
        begin S := S + {u} ; status[v] := father end ;
    if (exists w in Neigh with w notin S) then
        begin send [dfs,S] to w ; status[w] := cal end
    else if (exists w in Neigh with status[w] = father) then
        begin send [dfs,S] to w end
    else (* initiator *) stop
```

Algorithm 13.4 Linear-message depth-first search (node u).

of nodes already visited. Indeed, placing a call on any neighbour is avoided if that neighbour occurs in the list; see Algorithm 13.4, where we eliminated the *visited* variable because a node can inspect the message to find out if it was visited before.

The algorithm illustrates various observations regarding communication complexity and its relation to "amount of work." The total number of messages is reduced to $2(n-1)$, but at the expense of having very long messages; indeed the very last message received by the initiator contains the full list of all nodes. The total *length* of all transmitted lists is at least $n^2 - 1$ and at most $\frac{3}{2}n(n-1)$ node names; we observe a significant difference between *counting* messages (message complexity) and *weighing* them (bit complexity).

It is not reasonable to assume that Algorithm 13.4 requires only a constant amount of local processing per sent or received message, because the search for an unvisited neighbour requires to compare the received list of node names to the set of neighbors. Finally, the algorithm requires a lot of local storage to represent the S set. Concluding, the high bit complexity and the considerable local processing and storage, make the algorithm unpractical in most realistic situations.

Fast solution. The fastest algorithm for distributively computing a depth-first search tree is not obtained by simulating the sequential dfs algorithm, but by exploiting a characterisation of the resulting type of tree.

Definition 13.1
A rooted spanning tree of a graph satisfies the *dfs property* if for each edge uv, either u is an ancestor of v or v is an ancestor of u.

(The usual definition of dfs trees is based on the construction procedure, from which this property can be derived.)

Now assume an ordering on node names is available, and represent a path from the initiator to a node as a string enumerating the nodes in the path.

```
var la : string   init infty ;

For the initiator only:
    la := u ;
    forall x in Neigh do send [path,la] to x

Upon arrival of a [path,rho] message from v:
    receive [path,rho] from v ;
    if rho.u < la then
      begin la := rho.u ;
            forall x in Neigh s.t. x not in la
                  do send [path,la] to x
      end
```

Algorithm 13.5 The Relaxation algorithm.

Lemma 13.1
The set of edges formed by combining for all nodes u the lexically minimal simple path (lmsp) from the initiator to u is a dfs tree.

Proof (Sketch!) The selected edges connect the graph because for each node at least one path from the initiator is included. It is a tree because any prefix of the lmsp to u, say ending in vertex v, is the lmsp for v; this also implies that the tree path from the initiator to u is the lmsp to u.

To show that the dfs property is satisfied, consider neighbors u and v and let their lmsp's be $lmsp(u)$ and $lmsp(v)$, respectively; assume without loss of generality that $lmsp(u) < lmsp(v)$. If node v is in $lmsp(u)$, the prefix of $lmsp(u)$ up to v is a path to v that is lexically smaller than $lmsp(u)$, so assuming $lmsp(u) < lmsp(v)$, v is not contained in $lmsp(u)$.

But then $lmsp(u)$ concatenated with v, denoted $lmsp(u) \cdot v$, is a simple path to v and this implies $lmsp(v) \leq lmsp(u) \cdot v$. Consequently, $lmsp(u) < lmsp(v) \leq lmsp(u) \cdot v$, which implies that $lmsp(u)$ is a prefix of $lmsp(v)$, and u is an ancestor of v. ■

As a consequence, a dfs tree can be constructed with a variation of Chandy and Misra's algorithm [27, p. 120] for shortest path computation; see Alg. 13.5. Variable la_u is node u's *approximation* of its lmsp; the approximations are initialized to ∞, a string exceeding all other strings, and remain conservative in the sense that $la_u \geq lmsp(u)$.

The approximation is decreased when node u obtains information about a simple path to u that is lexically smaller than la_u; that is, upon receipt of a `[path,` ρ`]` message such that $\rho \cdot u < la_u$. The updated la_u is propagated to the neighbors because the smaller path to u may result in a smaller path to the neighbour also. This propagation and the subsequent processing of the message is called a *relaxation* over the edge to the neighbour.

Only finitely many messages are exchanged by the algorithm, because the messages sent by any node correspond to smaller and smaller paths, all of bounded length ($n-1$ hops) because they are simple. It is not particularly hard to construct an example where exponentially many messages are exchanged.

We call node u *ready* if $la_u = lmsp(u)$; no changes in la_u occur after u becomes ready, because no path smaller than $lmsp(u)$ is ever proposed to u. It can be shown that, for v the second last node in $lmsp(u)$, if edge vu is relaxed after v becomes ready, then u is ready also.

Lemma 13.2
Within t time units after the initialization by u_0, every node with an *lmsp* of length t or smaller is ready.

Proof This is done by induction on t; indeed, because no string starting with u_0 is ever lexically smaller than the string u_0, the initiator is ready immediately at the initialization.

Assume u has an lmsp of $t+1$ hops, with second last node v. Node v has an lmsp of t hops, hence at some point, within t time units after initialization, there is a relaxation that makes v ready. At this moment, v sends its new estimate la_v, now $lmsp(v)$, to u and this message is received within a time unit. After this relaxation, that is, within $t+1$ time units from initialization, u is ready. ■

We conclude that the algorithm constructs a dfs tree exchanging a large, but finite, amount of messages in time proportional to the depth of the tree. The algorithm can be fast in some cases, but other graphs have a dfs tree of linear depth.

The relaxation algorithm introduces another problem in distributed computing, namely that no node can directly observe the termination of the construction. Indeed, all nodes will end in the receiving state, where their approximations equal the actual minimal paths, but the nodes are never sure that no smaller paths will ever be proposed. We study the *termination detection* problem in Section 13.3.

The algorithm can also be used without prior definition of an initiator; if all nodes execute the initiating code, the network will converge towards a spanning tree with the smallest node as the root. Indeed, the paths starting in this node are *all* lexically smaller than the paths starting in any other node, so every node eventually accepts a path from the smallest node as the lexically minimal one.

Breadth-first search. A spanning tree has the *breadth-first search* property if the tree path from the root to each node is a shortest path in the network. Sequential computation of such a tree is very efficient ($O(m)$ work) but makes use of a data structure, a queue, to temporarily store nodes that have been discovered, but were not yet visited. The data structure plays an important role to ensure that the nodes are visited in the correct order and the use of this queue makes breadth-first search surprisingly difficult to distribute.

The simplest algorithms explore the network by sending an explore message through each edge ($2m$ messages). To synchronise the exploration, coordination

```
var rec    : integer ;
    father : neighbour ;

Algorithm for the initiator:
    rec := 0 ;
    forall v in Neigh do send [echo] to v ;
    while rec < |Neigh| do
          begin receive [echo] ; rec := rec + 1 end

Algorithm for other nodes:
    receive [echo] from w ; father := w ; rec := 1 ;
    forall v in Neigh – {w} do send [echo] to v ;
    while rec < |Neigh| do
          begin receive [echo] ; rec := rec + 1 end ;
    send [echo] to father
```

Algorithm 13.6 The Echo algorithm (for node u).

from the root takes place after each level (of which there can be D) at the expense of a linear number of messages. Consequently, the communication for the coordination is of order $D.n$. In the worst case, D is linear in n, so the overhead is quadratic and dominates the message complexity.

By exploring l levels between successive synchronisation rounds the number of coordination messages is reduced to $D.n/l$ but l exploration messages may be sent through each edge. The resulting $D.n/l + ml$ message complexity is minimised to $\sqrt{D.n.m}$ with $l = \sqrt{D.n/m}$; choosing the best l requires a priori knowledge of D and m. Even more sophisticated algorithms are known, but their complexity still exceeds the complexity of the sequential algorithm significantly.

The bottom line is that in the design of distributed algorithms, breadth-first search should be avoided if possible; fortunately, there are alternatives.

13.2.2 Pseudo-Fast Exploration: the Echo Algorithm

In practice, a very fast exploration and spanning tree construction algorithm is obtained if each node forwards exploration messages to all its neighbors in parallel. The algorithm (Alg. 13.6) floods `[echo]` messages to all nodes, exchanges them over non-tree edges, and "echoes" them back through tree edges.

In more detail, the Echo algorithm (Alg. 13.6) operates as follows. The initiator start the *exploration phase* of the algorithm by sending messages to its neighbors. Upon receipt of the first message, a non-initiator forwards messages to all neighbors except the sender of that first message, thus messages are flooded to all nodes in the network. Each node stores the neighbour from which the first message was received, and the corresponding links form a spanning tree in the network.

The *echo phase* of the algorithm consists of the replies sent by each non-initiator to its father; a node replies to its father after receiving one message from each neighbour (condition $rec_u = |Neigh_u|$). It must be shown that node u eventually receives a messages from each neighbour; for u's father this is obvious (it is u's first message) and for the non-tree links it is easy to see. Indeed, if uv is a non-tree link, then v sent a message to u upon its first receipt, hence u eventually receives this message.

We can now show that the echo phase starts from the leaves of the tree and propagates upwards to the initiator. Indeed, the leaves have no children and hence will send to their father by the argument in the previous paragraph. Then the nodes whose children are leaves can send to their fathers, and so on. This reasoning shows not only that *all* nodes will eventually receive from each neighbour, but also that the order in which this happens at the various nodes is determined by the tree shape, and the initiator is the last node to terminate.

The echo algorithm constructs an *arbitrary* spanning tree (it can be shown that *every* spanning tree of the network can be the result of the non-deterministic exploration), which limits its applicability. On the other hand, the algorithm is fast *in practice*; its time complexity in our model has frequently been misunderstood. Because the exploration phase forwards `[echo]` messages immediately, all nodes are reached by the exploration within D time units after initialization. The echo phase returns messages over the same paths, and it is easy to be mislead in thinking that this phase will also terminate in $O(D)$ time. It is easy to show that the time consumption is $O(D)$ under very weak additional assumptions about the timing of messages, and this explains why the algorithm is empirically fast; see the exercises 4 and 5.

Unfortunately, our theoretical model allows for worse executions [27, p. 217]. The $O(D)$ construction time of the tree does not imply that its depth is $O(D)$ because exploration messages over a long path may bypass messages over shorter paths. That the echo phase sends messages over the same path does not imply that they take the same time, because our model does not induce relations between various transmission delays over the same link. Exploiting the first observation yields an execution where a tree of depth $\Theta(n)$ is constructed in $O(D)$ time, after which the echo phase takes linear time.

13.2.3 Searching for Connectivity Certificates

We have seen that the construction of a spanning tree requires at least $\Omega(m)$ communication; it was recently discovered that the same amount of communication can result in a much richer structure. This subsection defines (edge) *connectivity certificates* of networks, and we shall show how to construct certificates sequentially and distributively. We also give applications of certificates.

Connectivity and Connectivity Certificates. The *local connectivity* of nodes u and v in G, denoted $\lambda_G(u, v)$, is defined as the maximal number of edge disjoint paths connecting u and v. (This function is related to transport capacity and reliability as explained at the end of this subsection.) A connectivity certificate is a subset of the edges preserving connectivity to a certain degree.

Definition 13.2
A subset $E' \subseteq E$ is a *k-connectivity certificate* if, with $G' = (V, E')$, for all nodes $u, v \in V$ $\lambda_{G'}(u, v) \geq \min(k, \lambda_G(u, v))$.

For example, a maximal forest preserves 1-connectivity, because nodes that are connected (through a path) in G are also connected in a maximal forest; nodes in different components of G remain unconnected in the forest, of course. Nodes joined by multiple paths in G are joined by only a single path in the forest, so higher connectivities are not certified by the forest. Now extend the forest to a set E' of edges such that every edge contained in a cycle in G is also contained in a cycle in E'. Then, if u and v are joined by *two* paths in G, the set E' also contains two such paths, and hence E' is a 2-certificate.

It is most attractive to have small size certificates, but the computation of minimal certificates is NP-Complete; a k-certificate is *sparse* if its size is $O(k.n)$.

Computation of Sparse Certificates. Nagamochi and Ibaraki have shown that sparse k-certificates can be computed efficiently, namely by computing and removing a maximal forest k times.

Theorem 13.1 [19]
Let E_i be any maximal forest in $(V, E \setminus \cup_{j<i} E_j)$; then $\cup_{j\leq k} E_j$ is a sparse k-connectivity certificate.

Computing k maximal forests can easily be done in $O(k.(n+m))$ time but Nagamochi and Ibaraki achieved an $O(n+m)$ algorithm by cleverly combining the construction of the various forests.

Computing a maximal forest. A maximal forest is obtained by starting with no edges ($E' = \emptyset$) and applying *test*(e) to every edge e (in arbitrary order), where *test*(e) means:

```
if e introduces no cycle in E'
then E' := E' + {e} else reject e
```

Regardless of the test order the obtained structure is a maximal forest, but different test orders may yield different forests.

In general it could require some effort to see if e introduces a cycle, but this effort is eliminated *by suitable test order strategy.* Call a node *active* if it has untested edges, and call a non-trivial tree of the forest active if it contains active nodes; the test strategy guarantees *at most one active tree* at any moment. The strategy is: if there is an active tree T, then select an active node u from it and test *all its untested edges*; testing all untested edges of u is called a *visit* to u. Then, if there is an active tree, adding some edges of u to E' only extends T but does not introduce an extra active tree, and only if there is no active tree, adding an edge may introduce one.

The uniqueness of the active tree implies that in a visit to u, edge uv introduces a cycle *if and only if* v is adjacent to an edge in E'. Indeed, v is adjacent to the untested edge uv, hence active, and so if it has an E' edge it is in an active tree;

because there is only one active tree, v is already connected to u through T. The construction of the forest is: repeatedly select and visit an unvisited node, if possible one that already has an adjacent E' edge. The visit to u is: consider its untested edges uv and include them if and only if v has no E' edges yet.

Computing all forests simultaneously. We start the construction with all forests empty ($E_i = \emptyset$) and apply a basic ranking step $rank(e)$ to every edge, where $rank(e)$ adds e to the first forest where e does not introduce a cycle.

```
i := smallest value s.t. e does not create a cycle in Ei ;
Ei := Ei + {e}
```

The ranking order will imply, as above, that every forest has at most one active tree, hence edge uv creates a cycle in E_j if and only if v already has an edge ranked j. Thus, when edge uv is ranked during a visit to u, its rank is the smallest rank at which v has no adjacent edges yet. A crucial property follows: if any node has an edge ranked i, it also has edges ranked j for all $j < i$; the highest rank of an edge of a node will be called the *level* of that node.

Each forest will have at most one active tree, and the mentioned property implies that a node active in forest i is also active in forest j for all $j < i$. Hence it suffices to select a node of maximal level and rank all its adjacent edges in order to construct the required sequence of maximal forests.

Nagamochi and Ibaraki have shown that the entire ranking can be completed in $O(n + m)$ time in the sequential model. Their solution uses a centralized data structure to store all unvisited nodes according to their level; the next visited node is selected from it in $O(1)$ time. Ranking an edge requires the data structure to be updated, because an unvisited node is increased in level; moving the node from one list to the list of next level is also done in $O(1)$ time.

Distributed Certificate Algorithm. At first sight the centralized data structure frustrates a distributed implementation, just as it is the case for breadth-first search. However, Evens *et al.* [9] showed that the central data structure can be replaced by a recursive search for unvisited nodes through the branches of the tree of the highest active level. To this end, if node u receives a search message through an edge of rank i, it forwards the message through all unsearched edges of rank i and higher, highest ranks first.

In Algorithm 13.7, node u stores the rank of its adjacent edge uv in $rank_u[v]$ (0 if the edge is unranked), and the flag $search_u[v]$ indicates if the search must still be forwarded to v. Verweij [32] shows that this search procedure indeed visits at each time the unvisited node of highest label. We summarise the properties of the algorithm.

Theorem 13.2
Algorithm 13.7 exchanges $4m$ messages in $4m$ time and assigns a rank to each edge in such a way that for each k, the edges with ranks 1 to k form a sparse k-connectivity certificate.

```
var rank[v]    : int     init 0 ;
    visited    : bool    init false ;
    search[v]  : bool    init true ;

procedure Visit:
    begin visited := true ;
          forall v s.t. rank[v] = 0
             do begin send [rnk] to v ;
                      receive [ranked,i] from v ;
                      rank[v] := i
                end
    end

procedure Search(v):
    begin if not visited then Visit ;
          forall w s.t. search[w] and rank[w] >= rank[v],
                      in decreasing order of rank[w]
             do begin search[w] := false ; send [srch] to w ;
                      receive [return] from w
                end ;
          if  v = u (* Initiator! *)
             then construction is terminated
             else send [return] to v
    end

Upon receipt of [rnk] from v:
    rank[v] := smallest i>0 s.t. u has no edge ranked i ;
    send [ranked,rank[v]] to v

Upon receipt of [srch] from v:
    Search(v)

To initiate the search (Only the initiator!):
    Search(u0)
```

Algorithm 13.7 The distributed certificate algorithm (for node u).

Ranking the unvisited edges of u (in procedure *Visit*) can be done in parallel to reduce the time complexity to $2m + 2n$. If only a certificate for one given value of k is required, the edges ranked higher than k need not be searched and the algorithm uses $2k.n + 2n$ time.

Applications. In communication networks, the local connectivity of u and v has two important operational meanings, related to *capacity* and to *reliability*. First, if each edge has a given data rate ρ, the existence of k disjoint paths between u and v implies that an overall data flow of $k.\rho$ can be transported from u to v. Second, the edge disjointness of the paths implies that, as long as $k - 1$ or fewer links fail, at least one path between u and v remains unaffected. Consequently, $\lambda_G(u, v)$ equals both the maximal data flow between u and v, and the number of link failures that can partition u from v.

Determining local connectivity. The local connectivity of u and v can be computed by repeatedly searching for an *augmenting path* in the graph, until no more paths are found. As this search may cost $O(m)$ messages, this way of computing the connectivity costs about $O(\lambda.m)$ messages ($\lambda = \lambda_G(u, v)$).

A better complexity is obtained with certificates; after ranking all the edges, the first uv path is searched in the edges of rank 1. The second augmenting path is searched among the edges ranked 1 and 2, and the i^{th} augmenting path is searched among edges of rank up to i. Indeed, if i paths exist in G, the certificate property guarantees that they exist in the union of the first i forests, so the restricted search does not terminate inappropriately. Because the i^{th} path is searched in a restricted network with less than $i.n$ edges, the total cost is $O(\lambda^2.n)$, which is usually smaller than $\lambda.m$.

Testing global connectivity. Algorithms for computing 2- or 3-connected components may profit from execution on a 2- or 3-connectivity certificate [14]. The certificate can be computed in $O(m)$ time and messages, and guarantees that the subsequent connectivity algorithm has to consider only $O(n)$ edges.

13.3 Termination Detection

A distributed algorithm terminates when it reaches a global state (configuration) in which no event of the algorithm is applicable. However, such a terminal configuration does not imply that each node is in a terminal state, that is, a (local) state from which no events are applicable, as is illustrated by Algorithm 13.5. Each node awaits the arrival of messages in a receiving state, and reacts to their arrival by sending some (possibly zero) messages. While a node always returns to a receiving state (hence not explicitly terminated) the computation as a whole halts when all nodes are simultaneously in this state and no messages are in transit.

This section discusses techniques to make termination explicit by distributively detecting that the program has reached a terminal configuration. A description of the problem is given in Section 13.3.1, and we discuss two classes of solutions in Sections 13.3.2 and 13.3.3.

13.3.1 Problem Definition

The description of the termination detection problem abstracts away from the purpose and operations of the computation in question, but concentrates on the aspects relevant for termination. A node is assumed to be in either an *active* or a *passive* state, where in an active state the node may send messages and in a passive state it may not. (In Alg. 13.5 a node receiving a message immediately sends the

```
var   state : (act, pas) ;

Su: { state = act }
      send [mes]

Ru: { A message [mes] arrives at u }
      receive [mes] ; state := act

Iu: { state = act }
      state := pas
```

Algorithm 13.8 Steps of distributed computation (node u).

resulting message and becomes receiving (passive) again. Here we model a slightly more general situation where a node may already receive while still processing previous messages.) The transition from active to passive may occur spontaneously (namely, when the active node finishes its current activities), but a passive node can *only* be awakened by receiving a message. The operation is modelled by the transitions in Alg. 13.8; again, the actual computation as well as the content of the exchanged message are abstracted away from.

Receiving messages is impossible if no messages are in transit, and sending messages is impossible if all nodes are passive; becoming passive is clearly also impossible in this case, and hence termination of Alg. 13.8 occurs when *simultaneously* all nodes are passive and all channels are empty.

A *termination detection algorithm* is added to a distributed computation and requires to make termination explicit. Detection requires executing some extra statements with the operations of the computation, as well as exchanging some extra messages for the detection purpose only. (These additional *control* message do not render a passive node active, of course.) Correctness requires that (1) if the computation terminates, this is detected within finite time thereafter (liveness) and (2) termination is not detected prematurely (safety).

The detection algorithms roughly fall in two categories. *Tracing* algorithms follow the computation flow by tracing active nodes along the message chains that activated them, and call termination when all traced activity has ceased. *Probe* algorithms rely on global (coordinated) scans of the network state and call termination when no activity is found. The distinction can be compared to that between reference counting and mark-and-sweep type garbage collectors [30].

13.3.2 Tracing Algorithms

A tracing algorithm relies on knowledge of the set of initially active nodes, because all activity of the computation originates from these nodes by message chains. Dijkstra and Scholten's algorithm [8] assumes that initially exactly one node is active; we call this node the *root* node.

```
var   state : (act, pas)  init if u=u0 then act else pas ;
      cc    : integer     init 0 ;
      fat   : node        init if u=u0 then u else undef ;

Su: { state = act }
      send [mes] ; cc := cc + 1

Ru: { A message [mes] from v arrives at u }
      receive [mes] ; state := act ;
      if  fat = undef then fat := v
                      else send [sig] to v

Iu: { state = act }
      state := pas

Au: { A message [sig] arrives at u }
      receive [sig] ; cc := cc – 1

Tu: { cc = 0 and state = pas and fat != undef }
      if  fat = u (* Root node! *)
         then Detect
         else   send [sig] to fat ; fat := undef
```

Algorithm 13.9 Dijkstra and Scholten's algorithm (node u).

Global description: Computation tree. The detection algorithm maintains, during the distributed computation, a *computation tree* T, whose vertices are nodes of the network and messages in transit; the root node is the root of T. Steps of the computation trigger updates in the tree structure aimed at preserving the crucial property of T:

> at any time, *all active nodes* as well as *all* `[mes]` *in transit* are vertices of T.

In addition, control messages and passive nodes may be in T, but their presence serves the maintenance of the tree rather than the correctness of the algorithm directly. In view of this property, termination can be concluded if the root node is passive and has no children. Indeed, the root node having no children implies T contains only the root, so no `[mes]` are in transit and no node other than the root is active; if the root is also passive we have termination.

Detailed description. Variable fat_u is *undef* if u is not in the tree, points to u itself if u is the root node, and points to u's father if u is a non-root tree node; cc_u counts the children of u in T. When active node u sends a `[mes]`, this message becomes a child of u hence cc_u is incremented (action $\mathbf{S}_u$ in Alg. 13.9). When u is activated (action $\mathbf{R}_u$) its membership of T must be ensured and this can be done by assuming the sender of the `[mes]` as its father; the father *cc* is unaltered as u replaces the `[mes]` as a child. If u is already in the tree, the `[mes]` is removed

```
var la  : string      init infty ;
    cc  : integer     init 0 ;
    fat : node        init undef ;

For the initiator only:
    la := u ; fat := u ;
    forall x in Neigh
        do begin send [path,la] to x ; cc := cc + 1 end

Upon arrival of a [path,rho] message from v:
    receive [path,rho] from v ;
    if  fat = undef then fat := v
                    else send [sig] to v ;
    if  rho.u < la then
        begin la := rho.u ;
              forall x in Neigh s.t. x not in la
                    do begin send [path,la] to x ; cc := cc + 1 end
        end

Au: { A message [sig] arrives at u }
    receive [sig] ; cc := cc - 1

Tu: { cc = 0 and fat != undef }
    if  fat = u (* Root node! *)
        then Detect: construction completed
        else send [sig] to fat ; fat := undef
```

Algorithm 13.10 Relaxation with Termination Detection (node u).

from the tree and a `[sig]` message is sent to its father to decrease the *cc*. Observe that a passive node remains in the tree if it has children, and a childless node remains in the tree if it is active; only if a passive, childless node is in the tree the withdrawal action $\mathbf{T}_u$ takes place. A non-root node sends a `[sig]` to its father so as to decrement the latter's *cc*, while the root node calls termination in this case.

Correctness, variations, discussion. It is far from trivial to firmly establish that the algorithm is correct and operates as described above, even under the most exotic scenarios of the computation and its timing. The basic techniques (invariant properties and variant functions) and their application to this algorithm are discussed in [27, Sec. 8.1] but are outside the scope of this chapter. Actually, the termination detection problem and the publication of several incorrect solutions strongly motivated research in verification techniques for distributed algorithms.

The algorithm can be applied to computations like Alg. 13.5, where the active state is not explicit. The resulting fast DFS algorithm with termination detection is shown as Alg. 13.10. The requirement that only one node initiates the computation was relaxed by Shavit and Francez [22]; in their algorithm each initiator of the computation traces a subset of the activity, and one round of global communication

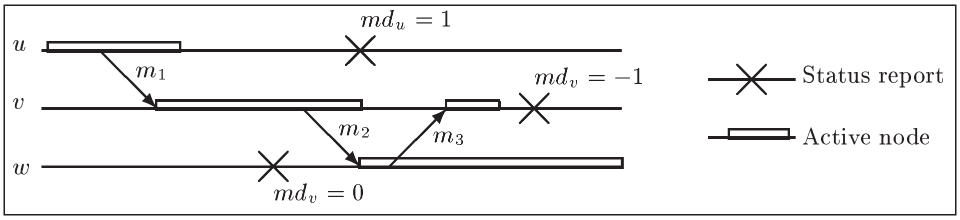

Figure 13.3 Compensated behind-the-back activation.

is used to determine that all traced activity has ceased.

The number of exchanged control messages equals the number of messages exchanged by the underlying computation, and this was shown to be optimal in the worst case. If a computation is started from a single node and the number of exchanged messages is relatively small (linear in n or m, say), the Dijkstra–Scholten algorithm is the termination detector of choice.

13.3.3 Probe Algorithms

Probe algorithms repeatedly scan the entire network for active nodes and computation messages; they are based on the principle laid out by Dijkstra, Feijen, and Van Gasteren [7]. For simplicity of explanation we shall assume a special node (the *controller*) to coordinate detection; the controller exchanges status reports with all nodes.

In order to establish the absence of computation messages, each node maintains a *message deficit*, being the number of messages it has sent so far minus the number of messages it received so far. At any time, the number of messages in transit equals the sum of all deficits, hence empty channels mean zero deficit sum. In reply to a request (`[req]` message) from the coordinator, each node sends a status report (`[stat,` m`,` c`]` message), but defers sending it until it is passive.

It is tempting to believe that, because the nodes were passive when sending the report, the controller can detect termination if it receives status reports from all nodes and the deficits add to zero. However, unsafety results from the status reports being produced at different times, as is illustrated in the space-time diagram of Figure 13.3. Node w was activated "behind-the-back" of the controller, but the activating message m_2 causes no negative deficit because the deficit was compensated for by receiving m_3 from w! Message m_2 *crosses* the probe because it was sent before the status report of its sender, but received after the status report of its receiver, and m_3 is said to cross the probe *backwards*.

Taking the status reports can be coordinated so as to prevent any message from crossing the probe backwards, which would render the algorithm safe; the status reports would then form a *consistent snapshot* cf. [5]. It is easier however, to detect the possibility of any compensated behind-the-back activation; to this end, each node also includes in its status report, whether any `[mes]` message was

```
var  state    : (act, pas) ;
     md       : int       init 0 ;
     rec      : bool      init false ;

Su:  { state = act }
     send [mes] ; md := md + 1

Ru:  { A message [mes] arrives at u }
     receive [mes] ; state := act ;
     rec := true ; md := md – 1

Iu:  { state = act }
     state := pas

Au:  { state = pas and a [req] message has arrived. }
     send [stat,md,rec] to controller ; rec := false

Code for the controller:
     repeat t := false ; s := 0 ;
            forall u do send [req] to u ;
            forall u do
                  begin receive [stat,m,r] ;
                        t := (t and r) ; s := s+m
                  end
     until ( t = false and s = 0 ) ;
     Detect termination
```

Algorithm 13.11 Probe based termination detection (node u).

received since sending the previous report. If this is the case, termination is not concluded; thus the receipt of a compensating backward message prevents detection; the resulting algorithm is shown as Alg. 13.11.

Variations, complexity, discussion. The various probe based algorithms differ considerably, mainly in their treatment of in-transit messages, and the collection of the status reports [25, 26]. Instead of *counting* messages as we have shown, acknowledgements or time-outs can be used.

Instead of direct communication with the controller as in Alg. 13.11, probe propagation through a Hamiltonian Cycle, or the Echo algorithm can be used for status communication. To implement the latter possibility, the controller acts as the initiator in the Echo algorithm. Status reports are sent upward in the constructed spanning tree in an accumulated fashion, i.e., each node reports the sum over all md_u and the conjunction over all rec_u of the nodes u in its subtree.

Instead of having an additional controller, one of the nodes of the computation will perform the controller task in addition to the computation proper. In this way it is not necessary to add either nodes or channels to the network solely for the purpose of detection.

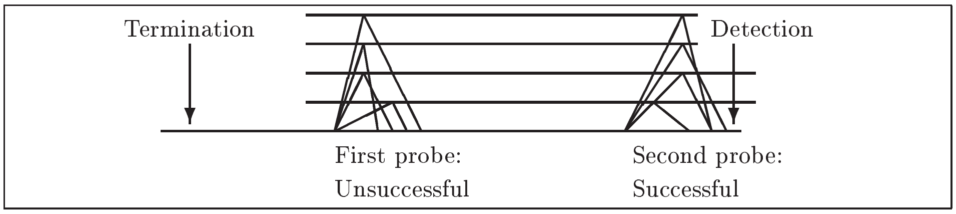

Figure 13.4 The detection delay.

Probe algorithms are the detectors of choice in computations that exchange a lot of messages, especially if many are exchanged in parallel. The reason is that probe algorithms exchange a fixed number of control messages *per probe*, independent of the number of basic messages. A good balance between detection overhead and detection delay can be achieved by starting probes under the control of a timer (as in [20].) Assume a fixed delay of Δ is introduced between the end of an unsuccessful probe and the start of the next one, and that the duration of the probe is small compared to Δ. After termination occurs, some of the nodes may have $rec = true$ so that the first probe started after termination fails to detect. The next probe finds all nodes with $rec = false$ and thus termination is detected after at most 2Δ delay.

13.4 Distributed Arc Consistency and CSP

To demonstrate the application of the distributed algorithm techniques to distributed AI problems, we shall now study the distributed Arc Consistency algorithm DisAC4; see also [20]. The Constraint Satisfaction Problem and consistency filters were also discussed in Chapter 4.

The Constraint Satisfaction Problem (CSP) and Arc Consistency (AC) are defined in Section 13.4.1, and the sequential AC4 algorithm is outlined in Section 13.4.2. We then consider a resource distributed model, where the resources for checking the consistency of a variable are located at a particular node. Section 13.4.3 gives the resulting algorithm, where each node is assumed responsible for one model variable, and Section 13.4.4 discusses termination detection for this version. An alternative computational model, a multiprocessor computer, where each node holds a subset of the variables, is considered in Section 13.4.5. Section 13.4.6 discusses how the distributed AC algorithm can be extended to be used in distributed backtracking CSP algorithms.

13.4.1 Constraint Satisfaction and Arc Consistency

A Constraint Satisfaction Problem is defined by a set of *variables* $Z = \{x_1, \ldots, x_n\}$, where x_i must be assigned a value v_i from a *domain* D_i but subject to *constraints*. The constraints are a collection of binary predicates C_{ij} where $C_{ij}(v, w)$ indicates if assigning v to x_i is legitimate w.r.t. C_{ij} if value w is assigned to x_j. A *solution* to the problem is an assignment that is simultaneously legitimate for all constraints, or, equivalently, in which no constraint is violated. It is usually assumed that constraints are symmetric (that is, $C_{ij}(v, w) = C_{ji}(w, v)$), but symmetry is not used in the algorithms of this section.

Finding a solution is computationally hard (the problem is NP complete) and generally involves testing all or many possible assignments. The size of each domain is assumed finite (in order to express complexities we assume a uniform upper bound $|D_i| \leq a$), but the number of possibilities is still exponential in n.

Arc Consistency is a polynomial technique that may help to reduce the search space considerably; it deletes a value from a domain if some constraint is seen to be unfulfillable with this value. More specifically, consider constraint C_{ij} and assume that for some $v \in D_i$ there is *no* $w \in D_j$ for which $C_{ij}(v, w)$ is true. As C_{ij} can not be fulfilled with $x_i = v$, the value v in D_i is redundant and can be eliminated for further consideration. This elimination may lead to other values becoming redundant in turn. A problem is called *arc consistent* if it has no redundant domain values. The *Arc Consistency* problem is to restrict all domains in a constraint satisfaction problem, so as to make the problem arc consistent but without eliminating possible solutions.

Formally, given domains D_1 through D_n, the Arc Consistency problem requires to find D'_1 through D'_n such that:

1. The domains are restricted: $D'_i \subseteq D_i$.
2. The restricted problem is arc consistent: No D'_i contains a redundant value.
3. The output is maximally arc consistent: if sets D''_i with $D'_i \subseteq D''_i \subseteq D_i$ are arc consistent, then $D''_i = D'_i$ for each i.

The third requirement implies that no solutions are eliminated: if $(v_1, \ldots, v_n)$ is a satisfying assignment, then $v_i \in D'_i$ for each i.

Usually not every pair of variables has a non-trivial constraint (a constraint different from *true*). The problem is modelled as a directed graph where the variables are the nodes, and there is an edge from x_i to x_j if C_{ji} is non-trivial. Let $Succ_i$ denote the successors and $Pred_i$ the predecessors of i in this graph and m the number of edges.

13.4.2 The AC4 Algorithm

Mohr and Henderson [18] proposed the following data structures and algorithm for detecting redundant values; see Alg. 13.12. For each x_i, and each $v \in D_i$, an array of counters is maintained, where the counter $cnt[i, v, j]$ exists for each j for which a constraint C_{ij} exists. The counter $cnt[i, v, j]$ expresses the number of values $w \in D_j$ for which $D_{ij}(v, w)$ is true. When some counter $cnt[i, v, j]$ equals zero, the value v is redundant and is removed from D_i. As a result, $cnt[j, w, i]$ should be decremented for all j, w such that $C_{ji}(w, v)$ and to this end the pair (i, v) is queued for later processing. In this processing it is not necessary to evaluate the

```
(* Initialize counters and support structures *)
forall Cij
      do forall v in Di
              do forall w in Dj
                      do if Cij(v,w)
                          then begin cnt[i,v,j] +:= 1 ;
                                      Insert( Supp[j,w], <i,v> )
                               end ;

(* Check for initially redundant values *)
forall Cij
      do forall v in Di
              do if cnt[i,v,j] = 0
                  then begin Enque (Q,<i,v>) ; Delete(Di, v) end ;


(* Main loop *)
while not Empty(Q)
      do begin Deque(Q, <j,w>) ;
         forall <i,v> in Supp[j,w]
              do if v in Di
                  then begin cnt[i,v,j] := cnt[i,v,j] - 1 ;
                             if cnt[i,v,j]=0
                             then begin Enque (Q,<i,v>) ;
                                        Delete( Di, v)
                                  end
                      end
         end
```

Algorithm 13.12 Sequential AC4.

C_{ji} predicate again, because all relevant information is stored in additional support data structures *Supp*$[j, w]$. The set *Supp*$[j, w]$ contains all pairs $\langle i, v\rangle$ for which $C_{ij}(v, w)$ is *true*, or, equivalently, for which w is counted in *cnt*$[i, v, j]$. The main loop of Alg. 13.12 shows how the relevant counters are decremented and how this may make other values redundant in turn.

The size of the *cnt* arrays is at most $m.a$ integers because (i, j) ranges over edges of the graph. The size of the support structure is larger because for each constraint C_{ij}, all values in D_j may support all values in D_i, in which case the *Supp* lists together have $m.a^2$ pairs. As each pair (i, v) is queued at most once, the queue never holds more than $n.a$ pairs. Thus, the storage complexity of AC4 is dominated by the support structures.

Initialization of the data structures costs $m.a^2$ time, the initial check for redundant values takes $m.a$ time, and the main loop may again take $m.a^2$ time. The resulting $O(m.a^2)$ time complexity is optimal for Arc Consistency [18].

13.4.3 The Distributed AC4 Algorithm

In this subsection we shall describe a distributed implementation of the AC4 algorithm, first assuming that there is one computing node for each variable. Thus, node i maintains the domain D_i and holds the resources for evaluating C_{ij}; this makes node i the place of choice to maintain $cnt[i, v, j]$ as well. Neighbouring nodes will communicate the elimination of nodes in order to enforce decrementing the counters.

We shall now discuss the storage of the support structures. One possibility is to store $Supp[j, w]$ in node j and have j send a message to node i for each $\langle i, v\rangle$ found in $Supp[j, w]$. However, if node j sends just one message to node i when w is eliminated, node i must still evaluate C_{ij} to find out for which v $cnt[i, v, j]$ must be decremented. If j sends a list of values v for which this is the case, the communication complexity becomes very high.

Another possibility is to split $Supp[j, w]$ over the various neighbors: the pairs $\langle i, v\rangle$ are stored in node i. When node j eliminates w it will inform node i with a single message, and on receipt of this message node i must consider all its pairs of $Supp[j, w]$. This possibility is chosen in [20].

However, we observe that the support structure can be eliminated completely without significantly increasing the computational complexity of the algorithm. Indeed, for each j, w the set $Supp[j, w]$ is read just *at most* once, namely, when j is eliminated from D_j. Our distributed implementation therefore uses a different decrement policy. When w is eliminated from D_j, rather than enumerating a stored set $Supp[j, w]$, we will test for each i, v if $C_{ij}(v, w)$ is true, and, if so, decrement $cnt[i, v, j]$,

The queue of the sequential algorithm is distributed over all nodes as the receive queues (RQ) and send queues (SQ). Whenever node i detects v to be redundant, v is placed in the local send queue SQ_i. An independent subprocess $\mathbf{S}_i$ is responsible for taking all values out of this queue and informs the neighbors by sending a `[remove,` v`]` message. Incoming messages are buffered in the receive queue; an independent subprocess $\mathbf{R}_i$ inserts all received values in this queue, and the worker process $\mathbf{W}_i$ reads its input from this queue.

The elimination of support data structures reduces the storage requirements: Algorithm 13.13 stores an array of a counters $cnt_i[*, j]$ (in node i) for each constraint C_{ij}, hence the overall storage requirement is $O(m.a)$.

The initialization requires $O(m.a^2)$ time (as does the sequential algorithm) and exchanges no messages. (If δ_i is the in-degree of node i, the computation for node i is $O(\delta_i.a^2)$.) To this end we assume that node i knows the initial domain D_j, denoted as D_j^0, and counts, for each v, the number of supporters in this initial domain.

To assess the communication and computation cost of the processing phase, first observe that each value $w \in D_j^0$ is deleted from D_j and queued in SQ_j *at most once* because of the test in procedure *Redundant*. Consequently, each arc in the constraint graph carries at most a messages, to a total message complexity of $O(m.a)$. Each

```
var  D          init Di   (* Domain *)
     cnt[v,j]             (* Count support *)
     SQ,RQ                (* Send and Receive queue *)

procedure Redundant (v):
     if v in D then
        begin Delete(D, v) ; Enque(SQ, v) end

Initialization (for each node):
     forall j in Pred
          do forall v in D
                 do begin cnt[v,j] := 0 ;
                          forall w in Dj^0
                             (* Node i knows the initial set Dj *)
                             do  if Cij(v,w)
                                   then cnt[v,j] := cnt[v,j] + 1 ;
                          if  cnt[v,j] = 0
                             then Redundant(v)
                    end

Wi:  { Receive queue RQ is not empty }
     Deque (RQ, <j,w> ) ;
     forall v in D
          do if Cij(v,w)
             then begin cnt[v,j] := cnt[v,j] - 1 ;
                        if cnt[v,j] = 0
                        then Redundant (v)
                  end

Ri:  { receive [remove,w] from node j }
     Enque (RQ, <j,w> )

Si:  { SQ is not empty }
     Deque ( SQ, v) ;
     forall j in Succ do send [remove,v] to j
```

Algorithm 13.13 Distributed AC4 Algorithm (node i).

value received is enqueued for later processing, and this processing (action $\mathbf{W}_i$) consists of a loop over (at most) a values in D_i. The local computation cost is therefore bounded by $O(m.a^2)$ steps, hence the initialization phase still dominates the computation.

The distributed time complexity is $O(n.a)$; indeed, at most this many values are eliminated altogether, and the redundancy of some value is detected at most one time unit after its last supporter was eliminated.

13.4.4 Termination Detection

Termination of Alg. 13.13 is implicit, because after the elimination of all redundant values the nodes will be in a receiving state, ready to receive and process further `[remove, *]` messages. Fortunately, application of the results of Section 13.3 is straightforward. Define $pas(i)$ to be the following predicate:

$$SQ_i \text{ is empty} \wedge RQ_i \text{ is empty} \wedge \text{ initialization is completed in node } i$$

We observe the following.

1. *If $pas(i)$ holds, it can be falsified only by the receipt of a* `[remove, w]` *message.* After initialization, the only steps for node i are $\mathbf{W}_i$, $\mathbf{S}_i$, and $\mathbf{R}_i$, but processing and sending are not possible when the receive and send queue are empty. So only receipt is possible in this case, and will place a value in the receive queue, thereby falsifying $pas(i)$.
2. *If $pas(i)$ holds, node i cannot send a* `[remove, *]` *message.* The empty send queue disables the send action.
3. *If simultaneously for all i $pas(i)$ holds and no channel contains a* `[remove, *]` *message, the algorithm has terminated.* In this case, no processing or sending is possible because all queues are empty, and no receiving is possible because no messages are in transit.

Thus the assumptions for the termination detection problem are satisfied, and we can apply the algorithms of Section 13.3 to make termination explicit. The tracing algorithm (Alg. 13.3.2) is not appropriate here. First, it requires that there is exactly one initiator, which is not the case in Alg. 13.13 (generalizations to more initiators exist, though). The main reason is the overhead of control messages; tracing algorithms double the communication, while probe algorithms can have a much lower communication overhead if the distributed computation exchanges a lot of messages in parallel.

Thus, the Distributed AC4 algorithm should be combined with a probe based termination detection algorithm, such as Alg. 13.11. We shall not give the combined algorithm here.

On termination of the distributed AC4 algorithm, the remaining domains D_i are maximally arc consistent and two special situations deserve our attention.

1. **Contradiction:** *On termination, some D_i is empty.*
   Clearly, the product space is also empty, and because no solution to the problem is eliminated by the Arc Consistency algorithm, this condition implies that there exists no assignment satisfying all constraints.
2. **Solution:** *On termination, each D_i is reduced to a singleton $\{v_i\}$.*
   In this case the product space contains just a single assignment, namely $(x_1, \ldots, x_n) = (v_1, \ldots, v_n)$. Because the domains are arc consistent, this assignment is easily seen to satisfy all constraints. Indeed, consider constraint

C_{ji} and observe that, because v_i was not removed from D_i, there is at least one w in D_j for which $C_{ji}(w, v_i)$ is true. But D_i is the singleton $\{v_j\}$, so $C_{ji}(v_j, v_i)$ is true.

Evaluating these conditions can easily be done by augmenting the termination detection algorithm; in addition to reporting the rec_i and md_i information, node i states if D_i is a singleton, and if D_i is empty.

13.4.5 Partitioning for Multiprocessor Computers

We have so far assumed that there is a given, one-to-one correspondence between nodes and variables; a natural assumption if the resources for checking consistency are distributed and expensive to reallocate. Other applications may allow to freely allocate variables of the problem to processing nodes, for example, when a multiprocessor machine is used to solve a CSP (with all resources at hand).

We first discuss the execution of Algorithm 13.13 in this case, especially if more than one variable is assigned to any machine node. Node u maintains the administration for a collection $Z_u \subset Z$ and will execute all computations of Alg. 13.13 for the relevant variables, with only two twists that are not completely trivial. First, if node i sends a message to node j while i and j are in the same machine, no message is sent but the eliminated value is placed in the queue locally. Second, a machine can use a single receive queue, rather than a separate one for each of the variables it holds.

Thus the execution of the Arc Consistency itself is not very complicated, but the interesting question is to find a good allocation of variables over nodes. This distribution should have a favourable processor load, and need as little communication as possible. Fortunately, as a result of the analysis in the previous subsection, the load and communication of a distribution can be computed.

Let node i of the Arc Consistency Problem be allocated to processor $p(i)$ of the machine. As node i of the problem requires $O(\delta_i.a^2)$ work, the total load of processor p is $\sum_{i:p(i)=p} \delta_i.a^2$. As $O(a)$ messages are exchanged through each edge of the problem, the total amount of communication will be $O(a).|\{ij \in E : p(i) \neq p(j)\}|$. Minimising load and communication (over all allocations) is NP-hard, so an approximation algorithm is needed; see for example the work by Lo [17].

13.4.6 Distributed Constraint Satisfaction Algorithm

We shall now briefly discuss how Distributed Arc Consistency can be used in distributed solutions for Constraint Satisfaction Problems. A CSP is usually solved by backtracking, where parts of the solution space are eliminated from search by hypothesis generation. A hypothesis for variable x_i specifies a subset of the domain D_i and restricts the search to tuples for which x_i is in the subset. The current problem instance is narrowed down with the additional restriction that x_i is in the subset, yielding a new problem instance. More generally, a hypothesis can itself be a binary predicate assuming a constraint on combinations of x_i and x_j values.

If **solution** occurs in the restricted search space, the problem is solved and the found tuple is the solution. (It satisfies all the original constraints plus the current collection of hypotheses.) If **contradiction** is found, a backtracking step is taken: the hypothesis is replaced by its negation because the hypothesis is found to be inconsistent with the problem (including earlier hypotheses) and search is continued. If neither of these situations occurs, a next hypothesis is generated to narrow down the search space further.

The evaluation of the problem instances uses Arc Consistency: after each generation of a hypothesis or its replacement by its negation, the domains are further restricted by the Arc Consistency algorithm. We have seen that the AC algorithm reduces the domains to the maximally arc consistent subsets, and allows to conclude if **solution** or **contradiction** occurs.

A distributed CSP solver alternates hypothesis generation, hypothesis evaluation (by means of arc consistency), and hypothesis elimination (backtracking) in a coordinated way. To decide what hypothesis to generate, we assume that each node can locally evaluate the attractiveness of hypotheses it can generate. For example, generating a hypothesis concerning a variable with 20 possible values may be less attractive than one concerning a variable with 2 possible values. After termination of each arc consistency phase, the controller coordinates a global search for the node with the most attractive hypothesis. This does not require that the controller has access to all information or even that it can communicate with each node directly. In the next section we show (in the context of a graph processing algorithm) how such an evaluation is possible in a network of arbitrary topology using broadcasts and convergecast over a spanning tree.

More detailed description. When detecting termination of the arc consistency phase, the controller also evaluates if **solution** or **contradiction** occurs and informs the nodes.

If the search space is still too large, all nodes stack the current value of their domain D_i and the support structures $cnt_i[v, j]$. They evaluate the attractiveness of any hypothesis they can generate, and report the most attractive one. The convergecast allows the coordinator to find the most attractive hypothesis, and informs the node that submitted this hypothesis. This hypothesis is added to the constraints, after which Arc Consistency is started again.

In case of **solution**, the computed assignment is the output of the problem and the whole algorithm is terminated.

In case of **contradiction**, a backtrack step is taken. All nodes restore the previous values of D_i and $cnt_i[v, j]$ and in addition, the node that generated the most recently added hypothesis replaces it by its negation. After this, arc consistency is started again.

```
C := empty ;
while V != empty
      do begin if there is v in V with deg(v) = 1
                 then remove v from G
                 else  begin K := { v in V | indeg(v) <= 1 } ;
                             v := node of highest degree in K ;
                             C := C + {v} ;
                             remove v from G
                       end
         end
```

Algorithm 13.14 Suermondt and Cooper Loop Cutset.

13.5 Distributed Graph Processing

We shall demonstrate various techniques for distributed processing of the network topology by a distributed algorithm that has the topology as the input graph. The example worked out is the computation of a loop cutset in a Belief Network, which is a necessary preprocessing stage for the application of loop cutset conditioning in these networks. The aim of this section is to show how a sequential algorithm (by Suermondt and Cooper [23]) can be modified for distributed execution.

13.5.1 The Problem: Loop Cutset

A *Belief Network* is a directed acyclic graph in which the nodes represent various hypotheses and the arcs represent known statistical dependencies. Let $\vec{G} = (V, \vec{E})$ denote the directed graph, and $G = (V, E)$ the underlying undirected graph. The algorithms for updating the probability distribution of the hypotheses assume that G is free of cycles, and hence to apply these algorithms, cycles must be eliminated. A vertex in an undirected cycle is called a *pit* if both of its adjacent cycle arcs are incoming, and we require each cycle to be broken by the removal of at least one non-pit vertex.

Definition 13.3
A *loop cutset* is a subset $C \subseteq V$ such that for each cycle in G, C contains at least one node of the cycle that is not a pit of that cycle.

Algorithm of Suermondt and Cooper. For efficiency reasons, the cutset should be small, but computation of an optimal cutset is NP hard. The best-known heuristic for computing small cutsets (Suermondt and Cooper [23]) includes vertices in C one by one, trying to choose vertices that cut as many loops as possible. This is done by choosing a vertex with maximal degree, but to avoid cutting a cycle by removal of a pit, the chosen node must have in-degree zero or one. Because nodes of degree one are never part of a cycle, these nodes are removed (repeatedly) before searching for a cut-node; see Alg 13.14.

13.5.2 Distributed Execution of the Algorithm

The distributed algorithm does not represent the cut set in any central place; instead, at the end each node will know whether it is itself a cutnode or not. Algorithm 13.14 is simulated by two alternating phases, each under control of a coordinating node, which is the root of a spanning tree in the network. The spanning tree is used for control purposes, and an edge of the network can be part of it regardless whether it was already eliminated by the Suermondt/Cooper algorithm.

A *leaf trim* phase removes as many degree-one nodes as possible, and repeatedly; that is, if the removal of a node causes the degree of another node to drop to one, the latter is removed in the same phase. A *cut node search* is initiated when there are no more leaves, and searches the network for the highest degree node (with in-degree zero or one). When identified, the cut node becomes the new controller; a *shift controller* phase moves the root of the spanning tree to this node and hands control to the next leave trim phase. This phase has no counterpart in Alg. 13.14, and neither has the initial phase that constructs the control spanning tree.

Part one: Variables and leaf trim. Algorithm 13.15 shows the variables and constants used by the node u. The constants In_u and Out_u represent the incoming and outgoing neighbors of u in the graph; in the algorithms, x and y will range over neighbors of u, i.e., over $In_u \cup Out_u$.

To construct and maintain the control tree, each adjacent edge ux has a *link control status* $lcs_u[x]$ with the following meaning. The initial status is *basic*; when x is a child or the father of u the status is *son* or *fat*; and when the edge was rejected for the spanning tree, its status is *frond*.

The removal of edges and nodes by the Suermondt/Cooper algorithm is represented by the *link activity status* and *node activity status* $las_u[x]$ and nas_u. Initially the link is active (status is *yes*) but upon removal of x or u, $las_u[x]$ becomes *no*. The nodes are also initially active ($nas_u = yes$), but they can be removed either as a leaf or as a cut node, and nas_u becomes either *noncut* or *cut*.

The variables $mydeg_u$, $bestdeg_u$, and $bestbranch_u$ are used to determine the next cut node; $mydeg_u$ is the degree of u, $bestdeg_u$ the highest degree in u's subtree, and $bestbranch_u$ points to the location in the tree where the highest degree is found.

Algorithm 13.15 also presents the procedures for removal of leaves. The *TrimTest* procedure verifies if u has degree one, and if so, u becomes *noncut*; a `[remove]` message is sent to the only neighbour to inform it of the removal, and the procedure terminates after receipt of an acknowledgement `[sig]`. Receipt of the `[remove]` message causes the carrying edge to be non-active ($las = no$), and the node performs

```
cons In                                 (* Incoming neighbors *) ;
     Out                                (* Outgoing neighbors *) ;

var  lcs[x]          init basic         (* Link control status *) ;
     las[x]          init yes           (* Link activity status *) ;
     nas             init yes           (* Node activity status *) ;
     mydeg                              (* Compute degree of u *) ;
     bestdeg                            (* Highest degree in subtree *) ;
     bestbranch                         (* Point to best degree *) ;

procedure TrimTest:
     if | { x : las[x] = yes } | = 1
     then begin x := neighbour s.t. las[x] = yes ;
                nas := noncut ; las[x] := no ;
                send [remove] to x ;
                receive [sig] or [remove] from x
                          (* Optimisation, see text *)
          end

Upon receipt of [remove] from y:
     las[y] := no ; TrimTest ; send [sig] to y
```

Algorithm 13.15 Variables and Leaf Trim.

TrimTest itself. If a `[remove]` message is sent as a result, the replying `[sig]` message is deferred until a reply was received, according to the Dijkstra/Scholten principle. A slight twist is the possibility to receive a `[remove]` message instead of a `[sig]`; this will happen where two nodes (v and w) connected by a single edge remain at some point in the execution. Both nodes call *TrimTest* and decide to remove themselves and send a `[remove]` message over the edge. Rather than having both nodes reply to the other's message with a `[sig]`, each one treats the received message as the reply, thus saving the two extra messages. *I do not know how many messages are saved* in this way, but with this modification it is possible to compute the overall number of messages easily; see Section 13.5.3.

Part two: Control tree construction. The initial control tree is constructed by executing the echo algorithm from the initiator; this is shown in Alg 13.16. Procedure *ConstructSubtree* sends `[construct, 0]` messages through all *basic* edges and awaits the receipt of a `[construct,` i`]` message. The construct messages of the exploration stage have $i = 0$, while the replies to the father have $i = 1$; thus upon receipt of the message, the edge is recognised as either *son* or *frond*. Upon completion of the subtree a message (with $i = 1$ of course) is returned to the father.

Nodes run *TrimTest* in parallel with the construction of the subtree and await its return before replying to the father. Consequently, when the construction terminates at the initiator, the first round of leaf elimination was completed, and the search for the node of highest degree is initiated by calling the procedure

```
procedure ConstructSubtree:
      forall x s.t. lcs[x] = basic
           do send [construct,0] to x ;
      while exists x : lcs[x] = basic
           do begin receive [construct,i] from y ;
                     if i=0  then lcs[y] := frond
                             else lcs[y] := son
              end

The initiator starts the algorithm:
      pardo ConstructSubtree & TrimTest odrap ;
      InitSearchCutnode

The others, upon arrival of the first [construct,i] message:
              (* i=0 in the first message, because the
              first message is certainly NOT a reply. *)
      receive [construct,0] from x ; lcs[x] := fat ;
      pardo ConstructSubtree & TrimTest odrap ;
      send [construct,1] to x
```

Algorithm 13.16 Construction of control spanning tree.

InitSearchCutnode.

Part three: Search for cut node. The procedure *NodeSearch*, called in node u, computes the highest node degree in the subtree of u (with the restriction, of course, that only nodes with in-degree zero or one are taken into account). This procedure computes the degree of u itself and initiates a recursive computation in the subtrees by sending `[search]` messages to the sons of u. The procedure terminates only after receipt of a `[bestis,` d`]` message from each child; the order in which these messages arrive is not relevant. While processing the replies, u maintains the highest degree seen in the variable $bestdeg_u$ and $bestbranch_u$ points to either u itself or to the subtree reporting the highest degree.

This computation and the exchange of `[search]` and `[bestis,` d`]` messages over a spanning tree are a typical example of the *broadcast/convergecast* mechanism. By changing the local computation, the same mechanism can be used to compute other functions, such as summation or conjuntion and disjunction, as the application requires.

The coordinator of the round initiates the search by calling *InitSearchCutnode*, and all other nodes become involved upon receipt of a `[search]` message (from their father necessarily). In the latter case, after completion of *NodeSearch* the result value is sent to the father in a `[bestis,` d`]` message, where d is the computed degree ($bestdeg_u$). When the *NodeSearch* procedure terminates in the coordinator, the stored value is the overall highest degree. A value 0 at this point indicates that there are no nodes left with in-degree bounded by 1; actually a termination condition for the algorithm, as any non-empty directed acyclic graph has such

```
procedure NodeSearch:
    if    | { x in In : las[x] =yes } | <= 1
          then mydeg := | { x : las[x] =yes } |
          else mydeg := 0 ;
    bestdeg := mydeg ; bestbranch := u ;
    forall x s.t. lcs[x] = son
          do send [search] to x ;
    forall x s.t. lcs[x] = son
          do (* in order of message arrival !! *)
             begin receive [bestis,d] from x ;
                   if d > bestdeg
                     then begin bestdeg := d ;
                                bestbranch := x
                          end
             end

procedure InitSearchCutnode:
    NodeSearch ;
    if bestdeg = 0 then Terminate
                   else ChangeRoot

Upon receipt of [search] from x:
                   (* x is the father *)
    NodeSearch ;
    send [bestis,bestdeg] to x
```

Algorithm 13.17 Search for cut node.

nodes. If *NodeSearch* leads to a positive value, this is the maximal node degree, and a node of this degree can be chosen as the next cut node. To pass control to such a node, the current coordinator calls the procedure *ChangeRoot*.

Part four: Controller shift. Because the newly selected cutnode is at the center of the leaf-trim activity of the next round, we prefer to make it the new controller.

After the execution of *NodeSearch*, each node u has the pointer $bestbranch_u$ pointing to the highest degree node in the subtree of u. The procedure *ChangeRoot* is called only in nodes for which the subtree contains the globally highest degree. Indeed, the first call to *ChangeRoot* occurs in the controller (after completion of *NodeSearch*) and the subtree of the controller contains the entire network.

We now consider the procedure *ChangeRoot*; if $bestbranch_u = u$, then the maximum over u's subtree occurs at u, and because this maximum equals the global maximum, node u itself is chosen as the next cut node. Otherwise a `[changeroot]` message is sent to the son that reported the highest degree (pointed by $bestbranch_u$) because this subtree must contain the globally maximal degree. The direction of all control tree edges through which the `[changeroot]` message is forwarded is reversed so that the new controller becomes the root of this tree.

Finally we discuss the removal of the cut node from the network. The node

```
procedure ChangeRoot:
      if bestbranch = u
      then begin nas := cut ;
                  (* Coordinate next round *)
                  TrimFromNeighbors ;
                  InitSearchCutnode
            end
      else  begin lcs[bestbranch] := fat ;
                  send [changeroot] to bestbranch
            end

Upon receipt of [changeroot] from x:
      lcs[x] := son ; ChangeRoot

procedure TrimFromNeighbors:
      forall x s.t. las[x] = yes
           do send [remove] to x ;
      forall x s.t. las[x] = yes
           do (* In order of arrival of the messages *)
              begin receive [sig] or [remove] from x ;
                    las[x] := no
              end
```

Algorithm 13.18 Controller shift.

becomes a cut node (nas_u := *cut*) and informs its neighbors about its removal by calling *TrimFromNeighbors*. If removing one of the edges decreases the degree of a neighbour to 1, trimming of this new leaf is performed immediately, and termination of the whole procedure is detected as before.

Observe that *before* removal of the cut node there were no leaves (as a result of the previous trimming round), and only the neighbors of the cut node decrement their degree. Consequently, if there are any leaves at this point, they are contained in the neighbors of the cut node, and hence *TrimTest* need only be initiated in these neighbors.

After termination of this trimming round the controller initiates the search for the next cut node by calling *InitSearchCutnode*.

13.5.3 Complexity and Conclusions

To evaluate the complexity of the distributed algorithm, we introduce some parameters; n and m are the number of nodes and edges of G as usual, let s be the size of the computed cutset, and d the diameter of the control tree (worst case: $n - 1$). We then observe that in all procedures of the algorithm, at most a constant amount of work is associated with receiving or sending a message. Thus, the computation complexity of the algorithm is asymptotically equal to the number of messages exchanged by the algorithm. As remarked before, this is usually the case in distributed graph algorithms.

For the communication complexity we consider how many messages of each type are exchanged. For the construction of the control tree, two `[construct, `i`]` messages are sent through each edge of the graph to a total of $2m$ messages. Each edge is deactivated exactly once at the expense of two messages, so the total amount of `[remove]`/`[sig]` messages is also $2m$. The evaluation of the highest degree node requires the exchange of one `[search]` and one `[bestis, ...]` message through each edge of the control tree, which is $2(n-1)$ messages. This evaluation is performed $s+1$ times (the last evaluation yields 0 but is used to detect the end of the algorithm), so the overall number of `[search]` and `[bestis, `d`]` messages is $2(s+1)(n-1)$. Finally, the execution shifts the controller s times, which requires `[changeroot]` messages to be sent through a path in the control tree; the total number of `[changeroot]` messages is bounded by $s.d$. We thus see that $2m + 2m + 2(s+1)(n-1) + s.d$ messages are exchanged, which is about $4m + 2s.n$.

When evaluating the amount of time used by the algorithm we must realize that we have no guarantee of any actual parallelism occurring in the trimming of leaves. If we *ignore* leaf trimming, the construction of the control tree takes at most $2d$ time, a search for a cut node takes at most $2d$ time, and changing the root to the new cut node takes at most d time. These procedures together take $(3s+4).d$ time, but their progress can be delayed when nodes wait for leaf trimming to terminate. However, in the worst case all trimming is done sequentially and the exchange of `[remove]`/`[sig]` messages takes 2 time units per node, so the other procedures are delayed at most $(n-s).2$ time units, and the overall time complexity is bounded by $(3s+4).d + 2(n-s)$.

Our example of a distributed graph processing algorithm was taken from the Artificial Intelligence domain, namely loop cutset computation. Other graph algorithms can be treated in a similar way to yield distributed versions; known examples include Shortest Path [27, Sec. 4.2], Minimum Spanning Tree [10], Maximum Flow [32], Connectivity problems [14].

13.6 Conclusions

This chapter gives an overview of the most important techniques of distributed algorithm design for Distributed Artificial Intelligence applications. Important issues in this domain are the distributed control of computations, and the distributed processing of the network graph.

We have seen two important control paradigms. Termination detection is necessary to observe when some subcomputation has ended, and a new phase of the application can start. Examples included termination of arc consistency in distributed Constraint Satisfaction, and the leaf trimming sub-phase of Suermondt and Cooper's loop cutset algorithm. Distributed coordination can be issued by a controller using broadcast and convergecast over a spanning tree. Such a tree can be constructed using the echo algorithm, and can be used to broadcast computation states, to convergecast maximal values or sums, and the root of the tree can move. All these techniques were used in Suermondt and Cooper's algorithm, and are applicable to the distributed CSP algorithm outlined in Section 13.4.6. The interested reader is referred to [27] to read about more paradigms, such as leader election, control for anonymous networks, snapshots, synchronous algorithms; I consider them of lesser importance for the AI community.

Distributed graph processing is based on sequential techniques for the same problem, and distributed graph exploration is an important step. We have seen several depth-first search algorithms, and studied an algorithm for connectivity certificates. Breadth-first search is notoriously difficult to implement in distributed algorithms.

We have not addressed any issues related to failure and recovery of nodes; fault tolerance is an important area in distributed algorithms research, but the results are not easily transferred to the Artificial Intelligence application domain.

13.7 Exercises

1. *[Level 2]* Prove that the time complexity of a distributed algorithm is bounded by its message complexity.
2. *[Level 1]* Prove that the total length of the $2n-2$ lists exchanged by Algorithm 13.5 is between n^2-1 and $\frac{3}{2}n(n-1)$.
3. *[Level 1]* How does the Relaxation Algorithm (Alg. 13.5) prevent the formation of paths that are not simple?
4. *[Level 1]* Prove that the time complexity of the Echo algorithm (Alg. 13.6) is $O(D/\alpha)$ if message delay is not only upper bounded by one time unit, but also lower bounded by α time units ($0 < \alpha \leq 1$).
5. *[Level 2]* Prove that the time complexity of the Echo algorithm (Alg. 13.6) is $O(D.\beta)$ ifthe ratio between the delays, suffered by two messages sent through the same link (even in different directions) is bounded by a constant β.
6. *[Level 2]* The *detection delay* of a termination detection algorithm is the maximum time that can elapse between termination and its detection. Prove that the detection delay of the Dijkstra–Scholten algorithm is $\Theta(n)$. Prove that $\Omega(D)$ is a lower bound on the detection delay.
7. *[Level 2]* Algorithm 13.11 is not exactly fault-tolerant: it will already fail in case of a *single* error where one node erroneously increments its *md* counter without sending a message. Describe what happens in this case, and argue why this behaviour is inevitable for termination detection algorithms.
8. *[Level 4]* Develop termination detection algorithms that can work even if nodes may crash. You will find out that detecting termination in spite of t possible crashes requires that a passive node can only be activated by receiving at least $t+1$ activation messages.

9. *[Level 3]* Work out all the details and give a complete algorithm for the distributed Constraint Satisfaction Problem.
10. *[Level 2]* Write a distributed algorithm to test if the network graph is bipartite (2-colorable) and uses $O(m)$ messages and $O(n)$ time.
11. *[Level 3]* Write a distributed algorithm to construct a Maximal Independent Set (MIS) in the network graph.
12. *[Level 3]* Develop a distributed algorithm that works on a directed acyclic graph and computes a topological sort. At the end, each node knows its sequence number in the topological sort.
13. *[Level 4]* The Distributed Cutset Algorithm (Alg. 13.15–13.18) can be improved. A change of a node's degree during a leaf-trim round occurs only if it is a neighbor of a node trimmed in that round. Consequently, the global search for the highest-degree node (Alg. 13.17) can be replaced by a contest between neighbors of trimmed leaves and the previously highest-degree nodes. Develop an improved algorithm along these lines and analyse its complexity (messages and time).

13.8 References

1. Yehuda Afek and Moty Ricklin. Sparser: A paradigm for running distributed algorithms. In Adrian Segall and Shmuel Zaks, editors, *6th Int. Workshop on Distributed Algorithms*, volume 647 of *Lecture Notes in Computer Science*, pages 1–10, Haifa, November 1992. Springer Verlag.
2. Baruch Awerbuch. A new distributed depth-first search algorithm. *Information Processing Letters*, 20:147–150, 1985.
3. Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Symp. on Theory of Computing*, pages 230–240, May 1987.
4. Baruch Awerbuch and David Peleg. Routing with polynomial communication-space trade-off. *SIAM J. Discr. Math.*, 5(2):151–162, May 1992.
5. K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
6. Israel Cidon. Yet another distributed depth-first search algorithm. *Information Processing Letters*, 26:301–305, January 1988.
7. Edsger W. Dijkstra, Wim H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.
8. Edsger W. Dijkstra and Carel S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.
9. Shimon Even, Gene Itkis, and Sergio Rajsbaum. On mixed connectivity certificates. In Paul Spirakis, editor, *European Symposium on Algorithms*, volume

979 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 1995.

10. Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5:67–77, 1983.
11. Juan A. Garay, Shay Kutten, and David Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Symp. on Theory of Computing*, pages 659–668, 1993.
12. Jean-Michel Hélary, Aomar Maddi, and Michel Raynal. Calcul distribué d'un extremum et du routage associé dans un réseau quelconque. Technical Report 516, INRIA, Rennes, April 1986.
13. Lisa Higham and Teresa Przytycka. A simple, efficient algorithm for maximum finding on rings. In André Schiper, editor, *7th Int. Workshop on Distributed Algorithms*, volume 725 of *Lecture Notes in Computer Science*, pages 249–263. Springer Verlag, September 1993.
14. Esther Jennings. *Distributed Graph Connectivity Algorithms*. PhD thesis, Dept of Elec. Eng., Luleå Un. (Sw.), Sept. 22, 1997.
15. Nathan Linial. Distributive graph algorithms: Global solutions from local data. In *Foundations of Computer Science*, pages 331–335. IEEE, 1987.
16. Igor Litovsky, Yves Métivier, and Wiesław Zielonka. On the recognition of families of graphs with local computations. *Information and Computation*, 118(1):110–119, April 1995.
17. Virginia Mary Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. on Computers*, 37(11):1384–1397, November 1988.
18. R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artif. Intell.*, 28:225–233, 1986.
19. Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583–596, 1992.
20. Thang Nguyen and Yves Deville. A distributed arc-consistency algorithm. Technical report, Dépt Informatique, Univ. Cath. de Louvain, 1348 Louvain-la-Neuve, Belgium, September 1995.
21. Pierre Rosenstiehl, J. R. Fiksel, and A. Holliger. Intelligent graphs: Networks of finite automata capable of solving graph problems. In R. C. Read, editor, *Graph Theory and Computing*, pages 219–265. Academic Press, 1972.
22. Nir Shavit and Nissim Francez. A new approach to the detection of locally indicative stability. In Laurent Kott, editor, *Int. Colloq. on Automata, Languages, and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 344–358. Springer Verlag, 1986.
23. H. J. Suermondt and G. F. Cooper. Probabilistic inference in multiply connected belief networks using loop cutsets. *Int. J. of Approximate Reasoning*, 4:283–306, 1990.
24. Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.
25. Gerard Tel. Distributed infimum approximation. Technical Report RUU–CS–86–12, Dept of Computer Science, Utrecht Univ., 1986. URL `http://www.cs.ruu.nl/~gerard/liter/dia.dvi`.
26. Gerard Tel. Total algorithms. *Algorithms Review*, 1(1):13–42, January 1990.
27. Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press,

Cambridge, U.K., 1994.

28. Gerard Tel. Network orientation. *Int. Journal on Foundations of Computer Science*, 5(1):23–57, March 1994.
29. Gerard Tel. Linear election in hypercubes. *Parallel Processing Letters*, 5(3):357–366, 1995.
30. Gerard Tel and Friedemann Mattern. The derivation of termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.
31. Linda C. van der Gaag. Bayesian belief networks: Odds and ends. *The Computer Journal*, 39(2):97–113, 1996.
32. Bram Verweij. Distributed edge depletion for maximum flows. Master's thesis, Dept of Computer Science, Utrecht Univ., July 1996.