MIT/LCS/TM-453

COST-SENSITIVE ANALYSIS OF COMMUNICATION PROTOCOLS

Baruch Awerbuch
Alan Baratz
David Peleg

June 1991

Cost-Sensitive Analysis of Communication Protocols

Baruch Awerbuch [*] Alan Baratz [†] David Peleg [‡]

June 13, 1991

Abstract

This paper introduces the notion of cost-sensitive communication complexity and exemplifies it on the following basic communication problems: computing a global function, network synchronization, clock synchronization, controlling protocols' worst-case execution, connected components, spanning tree, etc., constructing a minimum spanning tree, constructing a shortest path tree.

[*]Department of Mathematics and Laboratory for Computer Science, M.I.T., Cambridge, MA 02139. ARPA: baruch@theory.lcs.mit.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, DARPA contract N00014-89-J-1988, and a special grant from IBM. Part of the work was done while visiting IBM T.J. Watson Research Center.

[†]IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

[‡]Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. BITNET: peleg@wisdom. Supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Haas Career Development Award. Part of the work was done while visiting MIT and IBM T.J. Watson Research Center.

1 Introduction

1.1 Motivation

Traffic load is one of the major factors affecting the behavior of a communication network. This fact is well recognized, and is the reason why most models for communication networks and most algorithms for routing, traffic analysis etc. model the network using a *weight function* on the edges, capturing this factor. In this model, the weight of an edge reflects the estimated delay for a message transmitted on this edge, and thus also the cost for using this edge. The significance of the load factor has also motivated the intense study of efficient methods for performing basic network tasks such as computing shortest paths and constructing minimum weight spanning trees (with length / weight defined with respect to the weight function).

However, in most of the previous work on *distributed algorithms* for these and other tasks, the design and analysis of the algorithms themselves completely disregards this weight function. That is, transmission over all the edges is assumed to be equally costly and completed within the same time bound. Such assumptions are made even when the task performed by the algorithm is directly related to the edge costs, and the algorithm has to be executed over the same network, and thus suffer the same delays. This seems to contradict the very purpose towards which the tasks are performed. It is sometimes argued that it is not crucial to take the weights into account when considering such "network service" algorithms, since these algorithms occupy only a thin slice of the network's bandwidth. Nonetheless, it is clear that an algorithm that can do well in that respect is preferable to one that ignores the issue.

This paper proposes an approach enabling us to take traffic loads into account in the design of distributed algorithms. This issue is addressed by introducing *cost-sensitive* complexity measures for analysis of distributed protocols. We consider weighted analogs for both communication and time complexity. We then examine a host of basic network problems, such as connectivity, computing global functions, network synchronization, controlling the worst-case execution of protocols, and constructing minimum spanning trees and shortest path trees. For each of these problems we seek to establish some lower bounds and propose some efficient algorithms with respect to the new complexity measures.

We feel that the approach proposed in this paper may serve as a basis for a more accurate account of the behavior of distributed algorithms in communication networks.

1.2 The model

We consider the standard model of (static) asynchronous communication networks. We consider a communication graph $G = (V, E, w)$, where a weight $w(e)$ is associated with each (undirected) edge of the network. We denote $n = |V|$, $m = |E|$. We also denote by W the maximal weight $w(e)$ of a network edge, $W = \max_{(u,v)\in E} w(u, v)$. We make the assumption that $W = poly(n)$, and thus $\log W = O(\log n)$. For any subgraph $G' = (V', E', w)$ of G, let $w(G')$ denote the total weight of G', i.e., $w(G') = \sum_{e\in E'} w(e)$. Let $dist(u, v, G')$ be the weighted distance from u to v in G', i.e., the minimum of $w(p)$ over all paths p from u to v in G', and let $Path(u, v, G')$ denote some arbitrary path achieving this minimum. Let $Diam(G')$ denote the *diameter* of G', i.e., $\max_{u,v\in V'} dist(u, v, G')$. Given a tree T and two vertices x, y in it, denote by $Path(x, y, T)$ the path from x to y in T.

Next let us define some basic graph notation. For a vertex $v \in V$, let

$$Rad(v, G) \;=\; \max_{w\in V}(dist_G(v, w)).$$

Given a set of vertices $S \subseteq V$, let $G(S)$ denote the subgraph induced by S in G. A *cluster* is a subset of vertices $S \subseteq V$ such that $G(S)$ is connected. The *radius* of a cluster S is denoted $Rad(S) = \min_{v\in S} Rad(v, G(S))$. A *cover* is a collection of clusters $\mathcal{S} = \{S_1, \ldots, S_m\}$ such that $\bigcup_i S_i = V$. Given a collection of clusters $\mathcal{S}$, let $Rad(\mathcal{S}) = \max_i Rad(S_i)$. For every vertex $v \in V$, let $deg_{\mathcal{S}}(v)$ denote the degree of v in the hypergraph $(V, \mathcal{S})$, i.e., the number of occurrences of v in clusters $S \in \mathcal{S}$. The *maximum degree* of a cover $\mathcal{S}$ is defined as $\Delta(\mathcal{S}) = \max_{v\in V} deg_{\mathcal{S}}(v)$.

Given two covers $\mathcal{S} = \{S_1, \ldots, S_m\}$ and $\mathcal{T} = \{T_1, \ldots, T_k\}$, we say that $\mathcal{T}$ *subsumes* $\mathcal{S}$ if for every $S_i \in \mathcal{S}$ there exists a $T_j \in \mathcal{T}$ such that $S_i \subseteq T_j$.

In the sequel we make use of the following theorem of [AP91].

Theorem 1.1 [AP91] Given a graph $G = (V, E)$, $|V| = n$, an initial cover $\mathcal{S}$ and an integer $k \geq 1$, it is possible to construct a cover $\mathcal{T}$ that satisfies the following properties:

(1) $\mathcal{T}$ subsumes $\mathcal{S}$,

(2) $Rad(\mathcal{T}) \leq (2k-1)Rad(\mathcal{S})$, and

(3) $\Delta(\mathcal{T}) = O(k|\mathcal{S}|^{1/k})$.

∎

1.3 The complexity measures

This paper introduces *weighted* complexity measures analogous to the traditional time and communication measures. We define the *cost* of transmitting a message over an edge e as $w(e)$. The *communication complexity* of a protocol π, denoted c_π, is the sum of all transmission costs of all messages sent during the execution of π. The *time complexity* of the protocol π, denoted t_π, is the maximal physical time it takes π to complete its execution, assuming that the delay on an edge e varies between 0 and $w(e)$. The classical complexity measures correspond to the case where $w(e) = 1$ for all $e \in E$.

Traditionally, communication protocols are evaluated in terms of E, V, D, which denote, respectively, the number of edges, the number of vertices, and the unweighted (hop based) diameter of the network. It turns out that it is convenient to evaluate the weighted complexity of protocols using the "weighted analogs" of E, V, D, denoted by $\mathcal{E}, \mathcal{V}, \mathcal{D}$, which are defined as follows:

$$\begin{aligned}
\mathcal{E} &= w(G) \left(= \sum_{e \in E} w(e) \right) \\
\mathcal{V} &= w(T) \text{ where } T \text{ is an MST of } G \\
\mathcal{D} &= Diam(G)
\end{aligned}$$

The analogy between these parameters and their unweighted counterparts is manifested in the fact that $\mathcal{E}$ equals the total cost of transmitting a single message over all the edges of the network, $\mathcal{V}$ is the minimal cost of reaching (or, disseminating a message to) all vertices, and $\mathcal{D}$ is the maximal cost of transmitting a message between a pair of network nodes.

In the sequel we express the complexity of our algorithms in terms of $\mathcal{E}, \mathcal{V}$ and $\mathcal{D}$. This gives results that are conveniently similar in appearance to the results of the unweighted case, as follows from the statement of results in the following subsection.

1.4 Problems and results

1.4.1 Global function computation

The problem: We are concerned here with computing global functions in a network. We assume that the structure of the network is known to all the vertices (including the edge weights). The only unknowns are the values of the n arguments of the function, which are initially stored at different vertices of the network, one at each vertex. The outputs must be be produced at all the vertices.

We restrict ourselves to the family of functions called *symmetric compact* in [GS86].

Global function computation		
	Communication	Time
Upper bound	$O(\mathcal{V})$	$O(\mathcal{D})$
Lower bound	$\Omega(\mathcal{V})$	$\Omega(\mathcal{D})$

Figure 1: Lower and upper bounds for global function computation.

The functions $f_n : X^n \rightarrow X$ in this family are symmetric (i.e., any two arguments can be switched) and compact, in the sense that the contribution of any subset of arguments can be represented in "compact form" by a string of size $\log_2 |X|$. The latter condition is formalized by assuming that there exists a function $g : X^2 \rightarrow X$ such that for any $k < n$, $f(x_1, x_2 \ldots x_n) = g(f_k(x_1, x_2 \ldots x_k), f_{n-k}(x_{k+1}, x_{k+2} \ldots x_n))$.

Computing such functions is quite a basic task in the area of network protocols. Many functions belong to this family, e.g. maximum, sum, basic boolean functions (XOR, AND, OR). Many other tasks, e.g. broadcasting a message from a given node to the rest of the network, termination detection, global synchronization, etc. can be represented as computing a symmetric compact function. A similar class of functions is considered in [ALSY88].

The results: We show that the computation of global functions requires $\Theta(\mathcal{V})$ messages and $\Theta(\mathcal{D})$ time.

The upper bound is derived as follows. Define a spanning tree as *shallow-light tree* (SLT) if its diameter is $O(\mathcal{D})$ and its weight is $O(\mathcal{V})$. We then show that SLT trees are effectively constructible, which implies that computing the value of our global function can be performed (optimally) with $O(\mathcal{V})$ messages and $O(\mathcal{D})$ time.

We are also concerned with efficient distributed constructions of SLT trees, or, in short, SLT algorithms. We present a specific SLT algorithm that requires $O(\mathcal{V} \cdot n^2)$ communication and $O(\mathcal{D} \cdot n^2)$ time.

1.4.2 Clock Synchronization

Problem: The purpose of the clock synchronization is to generate at each node a sequence of pulses, such that pulse p at a node is generated *after* (in the "causal" sense [Lam78]) all neighbors generate pulse $p-1$.

As argued by Even and Raijsbaum [ER90], the relevant complexity measure here is the "pulse delay", which is the maximal time delay in between two successive pulses at a node.

Let us denote $d = \max_{(u,v)\in E} dist(u, v)$, i.e., d is largest distance between neighbors in the network. Clearly $d \leq W$, and the problem is interesting when $d \ll W$. A lower bound of $\Omega(d)$, and an upper bound of $O(W)$ are derived in [ER90]. (It is worth pointing out that the main emphasis of [ER90] is on somewhat different "directed" version of this problem.)

Results: In this paper, we show that one can achieve a pulse delay of $O(d \cdot \log^2 n)$, i.e. leave a gap of $\log^2 n$ between the lower and upper bounds. This result relies heavily on a number of existing techniques, like the "Network Partition" of [AP91], and the "Synchronizer γ" of [Awe85a].

1.4.3 Network Synchronization

The problem: Asynchronous algorithms are in many cases substantially inferior in terms of their complexity to corresponding synchronous algorithms, and their design and analysis are more complicated. This motivates the development of a general simulation technique, known as the *synchronizer*, that allows users to write their algorithms as if they are run in a synchronous network. Implicitly, such techniques were proposed already in [Jaf80] and [Gal82]. The first explicit statement of the problem was given in [Awe85a], and better constructions for various cases were given in [PU89, AP90a]. Our goal is to extend the concept of the synchronizer to the weighted case and provide an appropriate construction.

On a conceptual level, the synchronizer (as well as the *controller*, described in following sections) is a protocol transformer, transforming a protocol π into a protocol ϕ that is equivalent to π in some sense but enjoys some additional desirable properties. Recall that c_π and t_π denote the communication and time complexity of the protocol π, and similarly for c_ϕ and t_ϕ. Our purpose is to guarantee that the transformation maintains c_ϕ and t_ϕ small compared to c_π and t_π.

The synchronizer can be viewed as a way to remove variations from link delays in an asynchronous network. In the "unweighted" case, this means that we want to "force" all link delays to be exactly 1. In the "weighted" case, the most natural and most useful generalization of this concept is to force the delay on each link e to be exactly $w(e)$. In a sense, the synchronizer enables to simulate a "weighted" synchronous network $G(V, E, w)$ with each link e having a delay of exactly $w(e)$ by a "weighted" asynchronous network $G(V, E, w)$. Such simulations may be useful for various applications, for which the absence of variations in edge delays significantly simplifies the tasks in hand, e.g., shortest paths [Awe89], constructing routing tables [ABLP89], and others. However, in addition to simplifying protocol design and analysis, synchronizers actually lead to complexity improvements for concrete algorithms.

For example, the algorithm SPT_{synch} derived via a synchronizer (presented in Subsection 9.1), is the best known shortest path algorithm for certain values of $\mathcal{V}, \mathcal{D}, \mathcal{E}$.

We define the *amortized* costs of a synchronizer ξ (i.e., the overhead per pulse) in communication and time as follows.

$$\mathcal{C}_p(\xi) = \frac{c_\phi - c_\pi}{t_\pi}$$

$$\mathcal{T}_p(\xi) = \frac{t_\phi}{t_\pi}$$

At first sight, the clock synchronization problem from Subsection 1.4.2 seems to resemble the problem of simulating an "unweighted" synchronous network $G'(V, E)$ (with all link delays being exactly 1) by a "weighted" asynchronous network $G(V, E, w)$. The main difference is in the fact that the only goal of the network synchronizer is to simulate a particular protocol, whereas the purpose of the clock synchronizer is to generate pulses. In general, it would be ineffective to use clock synchronizers for network synchronization, and vice versa. Even though the methods that we use to handle both problems have certain techniques in common, the differences are quite substantial.

The results: We construct a synchronizer γ_w, which is an analog of synchronizer γ of [Awe85a], such that for any fixed parameter k,

$$\mathcal{C}_p(\gamma_w) = O(kn \cdot \log n)$$

$$\mathcal{T}_p(\gamma_w) = O(\log_k n \cdot \log n)$$

1.4.4 Controllers

Problem: The controller [AAPS87] is a protocol transformer transforming a protocol π into a protocol ϕ that is equivalent to π in terms of its input-output relation on a static network, but is more "robust" than π in the sense that it has "reasonable" complexity even if it operates on "wrong" data.

Results: In the unweighted case, [AAPS87] presents a controller guaranteeing $t_\phi = c_\phi = O(c_\pi \cdot \log^2 c_\pi)$. We show that the same bounds hold for the weighted case as well.

1.4.5 Connected components, spanning tree

The problems: The problems considered here are finding connected components and constructing a (not necessarily minimum) spanning tree [Seg83, AGPV89]. These problems are

Connectivity		
	Communication	Time
DFS	$O(\mathcal{E})$	$O(\mathcal{E})$
$\mathtt{CON}_{flood}$	$O(\mathcal{E})$	$O(\mathcal{D})$
$\mathtt{CON}_{hybrid}$	$O(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$	$O(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$
Lower bound	$\Omega(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$	$\Omega(\mathcal{D})$

Figure 2: Our Connectivity algorithms.

equivalent to each other.

The results: We show that performing any of the above tasks requires $\Theta(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$ communication by providing matching upper and lower bounds. To be more precise, we prove that

1. For every distributed connectivity algorithm A and for any n there exists a family of n vertex graphs G on which A requires communication complexity $\Omega(n \cdot \mathcal{V})$ and a family of n vertex graphs G on which A requires communication complexity $\Omega(\mathcal{E})$.

2. There is a distributed connectivity algorithm with communication complexity $O(\min\{\mathcal{E}, n \cdot \mathcal{V}\})$ on any graph G.

1.4.6 Constructing minimum spanning trees

Problem: The *minimum spanning tree (MST)* of the graph G is a tree of minimum weight spanning G.

Results: We develop a number of MST algorithms, based on modifications of the algorithms of [GHS83, Awe87].

1. An algorithm with communication complexity $O(\min\{\mathcal{E} + \mathcal{V} \cdot \log n,\ n \cdot \mathcal{V}\})$.

2. An algorithm with communication complexity $O(\mathcal{E} \cdot \log n \log \mathcal{V})$ and time complexity $O(\mathcal{D} \cdot n \cdot \log n \cdot \log \mathcal{V})$.

Minimum Spanning Trees (MST)		
Algorithm	Communication	Time
$\mathtt{MST}_{ghs}$	$O(\mathcal{E}+\mathcal{V}\cdot\log n)$	$O(\mathcal{E}+\mathcal{V}\cdot\log n)$
$\mathtt{MST}_{centr}$	$O(n\cdot\mathcal{V})$	$O(n\cdot Diam(MST)$
$\mathtt{MST}_{fast}$	$O(\mathcal{E}\cdot\log n\log\mathcal{V})$	$O(\mathcal{D}\cdot n\cdot\log n\cdot\log\mathcal{V})$.
$\mathtt{MST}_{hybrid}$	$O(\min\{\mathcal{E},n\cdot\mathcal{V}\cdot\log n\})$	$O(\min\{\mathcal{E},n\cdot\mathcal{V}\cdot\log n\})$
Lower bound	$\Omega(\min\{\mathcal{E},n\cdot\mathcal{V}\})$	$\Omega(\mathcal{D})$

Figure 3: Our MST algorithms.

Shortest Path Trees (SPT)		
Algorithm	Communication	Time
$\mathtt{SPT}_{centr}$	$O(w(SPT)\cdot n)=O(n^2\cdot\mathcal{V})$	$O(\mathcal{D}\cdot n)$
$\mathtt{SPT}_{recur}$	$O(\mathcal{E}^{1+\epsilon})$	$O(\mathcal{D}^{1+\epsilon})$
$\mathtt{SPT}_{synch}$	$O(\mathcal{E}+\mathcal{D}\cdot kn\cdot\log n)$	$\mathcal{D}\cdot\log_k n\log n$
$\mathtt{SPT}_{hybrid}$	$O(\min\{\mathcal{E}+\mathcal{D}\cdot kn\cdot\log n),\mathcal{E}^{1+\epsilon}\})$	$O(\mathcal{D}^{1+\epsilon})$
Lower bound	$\Omega(\min\{\mathcal{E},n\cdot\mathcal{V}\})$	$\Omega(\mathcal{D})$

Figure 4: Our SPT algorithms.

1.4.7 Constructing shortest path trees

Problem: The *shortest paths tree (SPT)* of the graph G with respect to a source vertex $s\in V$ is a tree defined by the collection of shortest paths from s to all other vertices in G.

Results: We develop a number of SPT algorithms:

1. An algorithm with communication complexity $O(\mathcal{E}^{1+\epsilon})$ and time complexity $O(\mathcal{D}^{1+\epsilon})$. This is analogous to the result of [Awe89], which achieves same result for the unweighted case.

2. An algorithm with communication complexity $O(\mathcal{E}+\mathcal{D}\cdot kn\cdot\log n)$ and time complexity $O(\mathcal{D}\cdot\log_k n\log n)$.

1.5 Structure of this paper

The paper proceeds as follows. Section 2 gives tight upper and lower bounds on the computation of global functions. Section 3 contains clock synchronization algorithms. In Section 4, we give upper and lower bounds for network synchronizers. Section 5 deals with controller algorithms. Section 6 discusses basic algorithmic techniques for network problems such as broadcast, depth first search and construction of minimum spanning trees and shortest path trees. Section 7 discusses the problems of broadcast and constructing connected components and spanning trees. Finally, Sections 8 and 8 describe efficient algorithms for constructing minimum spanning trees and shortest path trees, respectively.

2 Optimal computation of global functions

2.1 The lower bound

Theorem 2.1 The computation of global symmetric compact functions requires $\Omega(\mathcal{V})$ communication and $\Omega(\mathcal{D})$ time.

Proof: Suppose that the value of the function has been computed at the vertex v. Since the value of a global function depends on the value of all of its arguments, there must be some information flow from each of the vertices to v. Thus the subgraph $G'(V, E')$, defined by the set of edges E' traversed by messages of the protocol, must contain a path from v to any other vertex in V, i.e., it must contain some spanning tree of G.

Observe that the distance $dist(u, v, G')$ from v to any other vertex $u \in V$ is a lower bound on time complexity of the protocol. Picking a pair of vertices u, v realizing $\mathcal{D}$ (i.e., maximizing the distance $dist(u, v, G)$) and noting that $dist(u, v, G') \geq dist(u, v, G) = \mathcal{D}$, we get that $\mathcal{D}$ is a lower bound on the time complexity of the protocol.

Furthermore, the total weight of the edges of G', $w(G')$, is a lower bound on the communication complexity of the computation. Now, since G' contains a spanning tree of G, its total weight satisfies $w(G') \geq \mathcal{V}$. Thus $\mathcal{V}$ lower bounds the communication complexity of the protocol. ▮

2.2 The upper bound

It is easy to see that given a spanning tree T for the network, a global function can be computed with communication complexity $w(T)$ and time complexity $Diam(T)$. Clearly, any shortest path tree T_S has small depth, namely $Diam(T_S) = O(\mathcal{D})$, but its weight may be as big as $w(T_S) = \Omega(n \cdot \mathcal{V})$ [BKJ83]. Analogously, any minimum spanning tree T_M has small weight, namely $w(T_M) = \mathcal{V}$; but its depth may be as high as $Diam(T_M) = \Omega(n \cdot \mathcal{D})$

[BKJ83]. Thus in [BKJ83, Jaf85] it is advocated to approach such problems by attempting to construct a tree approximating both a shortest-path tree and a minimum-weight spanning tree.

Recall that a spanning tree is *shallow-light tree* (SLT) if its diameter is $O(\mathcal{D})$ and its weight is $O(\mathcal{V})$. Such trees minimize simultaneously both weight and depth; existence of such tree would imply that in any graph, one can compute global functions with communication complexity $O(\mathcal{V})$ and $O(\mathcal{D})$ time. However, it is not clear that such trees exist. In the next subsection we establish

Theorem 2.2 Every graph has a shallow-light spanning tree.

Corollary 2.3 The computation of global symmetric compact functions can be performed with communication complexity $O(\mathcal{V})$ and $O(\mathcal{D})$ time. ▮

The shallow-light tree algorithm

We next provide an algorithm (hereafter referred to as the SLT algorithm) for constructing an SLT for an arbitrary graph, thus proving Theorem 2.2.

1. Construct an MST T_M and an SPT T_S for G, rooted at an arbitrary vertex v_0.

2. Traverse T_M in a depth-first search (DFS) fashion, starting from v_0. We think of the DFS as carried out by a "token", representing the algorithm's center of activity. Observe that in the tour taken by this token through the tree according to the specifications of the DFS, each tree edge is traversed exactly twice. Define the "mileage" of the DFS token at a given time to be the number of steps (forward and backward tree edge traversals) up to this time. Denote by $\nu(i)$ $(0 \le i \le 2(n-1))$ the location of the DFS token at the time its mileage is exactly i. For example, $\nu(0) = \nu(2n-2) = v_0$, where v_0 is the source of the DFS.

3. Construct the "line-version" L of T_M, which is a (weighted) path graph containing vertices $0, 1, \ldots, 2n-2$. A vertex i on the path corresponds to $\nu(i)$. We assign each edge $e = (i, i+1)$ on the line L the weight of the corresponding edge $(\nu(i), \nu(i+1))$ in the graph G. Observe that neighboring vertices on the line L correspond to neighbors in T_M. Observe that the total weight of the line is at most twice the total weight of the MST T_M, i.e., $w(L) \le 2\mathcal{V}$.

4. Fix a parameter $q > 0$. Construct "break-points" B_i on L by scanning it from left to right according to the following rules.

```
Construct a minimum spanning tree $T_M$ for $G$.
Construct a shortest path tree $T_S$ for $G$.
Construct $L$ based on $T_M$ as described above.
Assign each edge $e$ of $L$ the same weight as $\nu(e)$ in $G$.
$E' \leftarrow T_M$
$X \leftarrow 0$; $Y \leftarrow 0$
repeat
    repeat $Y \leftarrow Y + 1$
    until $dist(X, Y, L) > q \cdot dist(X, Y, T_S)$
        $E' \leftarrow E' \bigcup Path(X, Y, T_S)$
        $X \leftarrow Y$
until $Y = n$
Construct a shortest path tree $T$ in $G' = (V, E')$
output $T$
```

Figure 5: The SLT algorithm

(a) Break-point B_1 is vertex 0 on the line L.

(b) Break-point B_{i+1} is the first point to the right of B_i such that $dist(B_i, B_{i+1}, L) > q \cdot dist(\nu(B_i), \nu(B_{i+1}), T_S)$, meaning that the distance from B_i to B_{i+1} in T_M exceeds that in T_S by a factor of at least q.

5. Create a subgraph G' of G by taking T_M and adding $Path(\nu(B_i), \nu(B_{i+1}), T_S)$ for all break-points B_i, $i > 1$.

6. Construct a shortest path tree T rooted at v_0 in the resulting graph G'.

7. Output T.

The algorithm is presented formally in Figure 5. In the algorithm, T denotes the set of edges selected to the shallow-light tree and X, Y are pointers to nodes on the line L. An example for a possible execution of the algorithm is given in Figure 6.

2.3 Analysis

Lemma 2.4 The tree T constructed by the algorithm satisfies $w(T) \leq (1 + \frac{2}{q})\mathcal{V}$.

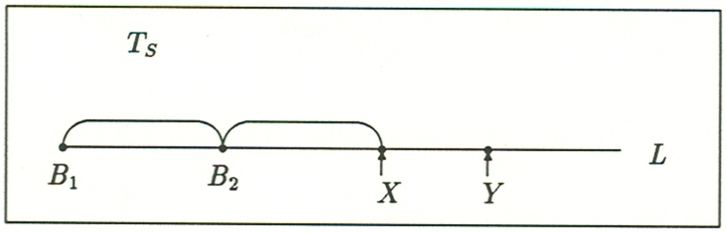

Figure 6: An example run of the SLT algorithm

Proof: The tree T is a subgraph of G', created by adding the paths $Path(\nu(B_i), \nu(B_{i+1}), T_S)$, for $i \geq 1$, to T_M. Therefore

$$w(T) \leq w(G') = w(T_M) + \sum_{i \geq 1} w(Path(\nu(B_i), \nu(B_{i+1}), T_S)).$$

But by choice of the breakpoints B_i,

$$w(Path(\nu(B_i), \nu(B_{i+1}), T_S)) = dist(\nu(B_i), \nu(B_{i+1}), T_S) < \frac{1}{q} \cdot dist(B_i, B_{i+1}, L),$$

hence

$$\sum_{i \geq 1} w(Path(\nu(B_i), \nu(B_{i+1}), T_S)) < \frac{1}{q} \sum_{i \geq 1} dist(B_i, B_{i+1}, L) \leq \frac{1}{q} \cdot w(L) \leq \frac{1}{q} \cdot 2\mathcal{V},$$

and thus $w(T) \leq (1 + \frac{2}{q})\mathcal{V}$. ▮

Lemma 2.5 The tree T constructed by the algorithm satisfies $Diam(T) \leq (q+1)\mathcal{D}$.

Proof: Consider an arbitrary vertex $x \in V$. We need to show that its depth in T is at most $(q+1)\mathcal{D}$. For that it suffices to bound $dist(v_0, x, G')$. Let j denote the point corresponding to x on the line L. Suppose that $B_\ell \leq j < B_{\ell+1}$, i.e., j occurs on the line L between B_ℓ and $B_{\ell+1}$ for some ℓ. Since $Path(\nu(B_i), \nu(B_{i+1}), T_S)$ is included in G' for every $1 \leq i \leq \ell - 1$, it follows that the entire path $Path(\nu(B_1), \nu(B_\ell), T_S)$ connecting the nodes corresponding to B_1 and B_ℓ in T_S is included in G', and hence

$$dist(v_0, \nu(B_\ell), G') \leq \mathcal{D}.$$

If $j = B_\ell$ then we are done. Otherwise, by the choice of $B_{\ell+1}$,

$$dist(B_\ell, j, L) \leq q \cdot dist(\nu(B_\ell), x, T_S).$$

Put together, we get that $dist(v_0, x, G') \leq (q+1)\mathcal{D}$. ▮

Corollary 2.6 The tree T constructed by the algorithm is an SLT.

2.4 Distributed construction of shallow-light trees

Theorem 2.7 There is a distributed algorithm for constructing an SLT requiring $O(\mathcal{V} \cdot n^2)$ communication and $O(\mathcal{D} \cdot n^2)$ time.

Proof: By Subsection 6.3, the MST T_M can be constructed using Algorithm $\mathtt{MST}_{centr}$ with $O(n \cdot \mathcal{V})$ communication and $O(n^2 \cdot \mathcal{D})$ time.

Now, the rest involves stretching the MST into a line. By Fact 6.3, the total weight of the MST is at most $n - 1$ times the diameter, or, $\mathcal{V} \leq (n-1)\mathcal{D}$. Thus the time and communication of the main body of the SLT algorithm are both $O(n^2 \cdot \mathcal{D})$.

Finally, we need to compute one more SPT in order to get the final tree T out of the subgraph G'. This is done using Algorithm $\mathtt{SPT}_{centr}$ of Subsection 6.4, and costs us additional $O(\mathcal{D} \cdot n)$ time and $O(n^2 \cdot \mathcal{V})$ communication.

Overall, the algorithm invests $O(\mathcal{D} \cdot n^2)$ time and $O(\mathcal{V} \cdot n^2)$ communication. ▮

3 Clock synchronization

In this section we describe three methods of clock synchronization, called synchronizer α^*, β^* and γ^*. These are modifications of synchronizers α, β and γ of [Awe85a].

3.1 Clock synchronizer α^*

As pointed out in [ER90], the most natural approach to clock synchronization is to use the following synchronization mechanism, called *synchronizer* α^*.

Pulse generation: whenever a node generates pulse p, is send messages to all neighbors, and when it receives messages of pulse p from all neighbors, it generates pulse $p+1$.

This method clearly requires time proportional to the highest edge weight, namely $O(W)$. Our goal is to approach the lower bound, which is $O(d)$ (recall that d is largest distance between neighbors).

The naive way to improve the delay is to construct a shortest path $Path(u, v)$ for all $(u, v) \in E$ and to communicate with each neighbor over such path. The problem with this method is that a particular edge may belong to many paths (up to E), and thus the resulting congestion will slow down the communication time by the corresponding factor (up to E).

3.2 Clock synchronizer β^*

In order to minimize congestion, we may try the following method, called *synchronizer β^**.

Preprocessing: We construct a spanning tree T of the network, and select a "leader" to be the root of this tree.

Pulse generation: Information about the completion of the current pulse is gathered up the tree by means of a communication pattern referred to as *convergecast*, which is started at the leaves of the tree and terminates at the root. Namely, whenever a node learns that it is done with this pulse and all its descendants in the tree are done with it as well, it reports this fact to its parent. Thus within finite time after the execution of the pulse, the leader eventually learns that all the nodes in the network are done. At that time it broadcasts a message along the tree, notifying all the nodes that they may generate a new pulse.

The time complexity of Synchronizer β^* is $\Omega(\mathcal{D})$, because the entire convergecast and broadcast process is performed along a spanning tree, whose depth is at least the diameter of the network.

3.3 Clock synchronizer γ^*

Our final synchronizer, called *synchronizer γ^**, combines synchronizer γ of [Awe85a]) with the network partitions of [AP91, AP90b].

Definition 3.1 Given an n-vertex weighted graph $G(V, E, w)$, a *tree edge-cover* for G is a collection $\mathcal{M}$ of trees, such that

1. every edge of G is shared by at most $O(\log n)$ trees of $\mathcal{M}$,
2. the depth of each tree in $\mathcal{M}$ is at most $O(\log n \cdot d)$, and
3. for each edge, there exists at least one tree containing both endpoints.

Lemma 3.2 For every n-vertex weighted graph $G(V, E, w)$, it is possible to construct a *tree edge-cover.*

Proof: The desired collection of trees can be constructed as follows. Apply Thm. 1.1 to the graph G, with the initial cover taken to be $\mathcal{S} = \{Path(u, v, G) \mid (u, v) \in E\}$, and the parameter $k = \log n$. The tree edge-cover is now constructed by selecting a shortest path spanning tree for each of the clusters of $\mathcal{T}$. The desired properties are guaranteed by the theorem (noting, in particular, that $Rad(\mathcal{S}) \leq d$ and $|\mathcal{S}| = n$, and therefore the output cover $\mathcal{T}$ satisfies $Rad(\mathcal{T}) = O(d \log n)$ and $\Delta(\mathcal{T}) = O(\log n)$). ▮

Preprocessing: Construct a tree edge-cover for G. Inside each tree, a *leader* is chosen to coordinate the operations of tree. We call two trees *neighboring* if they share a node.

Pulse generation: The process is performed in two phases. In the first phase, Synchronizer β is applied separately in each tree. Whenever the leader of a tree learns that its tree is done, it reports this fact to all the nodes in the tree which relay it to the leaders of all the neighboring trees. Now, the nodes of the tree enter the second phase, in which they wait until all the neighboring trees are known to be done and then generate the next pulse (as if Synchronizer α^* is applied among trees). More details will be given in the full paper.

Complexity: The "congestion" caused by the fact that messages of different trees cross the same edge, adds at most an $O(\log n)$ multiplicative factor to the time overhead. Since the height of each tree is $O(d \log n)$, it follows that the time to simulate one pulse is $O(d \cdot \log^2 n)$.

4 Network synchronizers

4.1 Construction outline

The synchronizers discussed in this section operate by generating sequences of "clock-pulses" at each vertex of the network, satisfying the following property: pulse p is generated at a vertex only after it receives all the messages of the synchronous algorithm that arrive at that vertex prior to pulse p. This property ensures that the network behaves as a synchronous one from the point of view of the particular synchronous algorithm.

The problem arising with synchronizer design is that a vertex cannot know which messages were sent to it by its neighbors and there are no bounds on edge delays. Thus, the above property cannot be achieved simply by waiting "enough time" before generating the next pulse, as may be possible in a network with bounded delays. However, it may be achieved if additional messages are sent for the purpose of synchronization.

In the unweighted synchronizers of [Awe85a], incoming links are "cleaned" from transient messages in between any two consecutive pulses, similar to the clock synchronizers in Section

3. In our (weighted) case, this would be very inefficient since cleaning the links requires time proportional to the maximal link weight $W \gg 1$, which would therefore dictate the multiplicative overhead of the synchronization. The idea for overcoming this problem is that links of high weight should be cleaned less frequently, thus enabling to amortize the cost of cleaning them over longer time intervals.

Our synchronizer, denoted γ_w, is also a modification of synchronizer γ of [Awe85a]. Synchronizer γ is a combination of the two simple synchronizers α and β, which are, in fact, generalizations of the techniques of [Gal82]. Synchronizer α is efficient in terms of time but wasteful in communication, while synchronizer β is efficient in communication but wasteful in time. However, we manage to combine these synchronizers in such a way that the resulting synchronizer is efficient *both* in time and communication. Before describing these synchronizers, we introduce the concept of *safety* for the weighted case.

Definition 4.1 A message sent from a vertex v to one of its neighbors u over the edge $e = (v, u)$ at pulse q is said to *affect* a later pulse p if $q + w(e) \leq p$. A vertex v is said to be *safe* with respect to pulse p if each affecting message of the synchronous algorithm sent by v at earlier pulses has already arrived to its destination.

Each vertex eventually becomes safe w.r.t. a pulse p some time after sending all of its messages from earlier pulses. If we require that an acknowledgment is sent back whenever a message of the algorithm is received from a neighbor, then each vertex may detect that it is safe w.r.t. pulse p whenever all its affecting messages have been acknowledged. Observe that the acknowledgments do not increase the asymptotic communication complexity, and each vertex learns that it is safe w.r.t. pulse p within constant time after it executed its pulse $p - 1$.

A new pulse p may be generated at a vertex whenever it is guaranteed that no affecting message sent at the previous pulses of the synchronous algorithm may arrive at that vertex in the future. Certainly, this is the case whenever all the neighbors of that vertex are known to be safe w.r.t. pulse p. It remains to find a way to deliver this information to each vertex with small communication and time costs.

We need to state a number of definitions first.

Definition 4.2 Given a synchronous protocol π running on a synchronous weighted network $G(V, E, w)$, we say that π is *in synch* with G if π transmits a message on edge e only at times that are divisible by $w(e)$.

Definition 4.3 A weighted network $G(V, E, w)$ is said to be *normalized* if all weights $w(e)$ are powers of 2.

Informally, our solution proceeds according to the following plan.

1. Design a synchronizer for normalized networks and protocols that are in synch with the networks on which they are run.

2. Show that one can transform an arbitrary synchronous protocol π and synchronous network G, so that the above assumptions are satisfied, without significantly increasing the complexities.

These two steps are described in the following two subsections.

4.2 Synchronizer γ_w

We assume now that the weights of all network edges are powers of 2, and messages are sent on an edge of weight 2^i only at times divisible by 2^i.

Let $\delta = \log W$. We define a collection of sub-networks $\{G_i(V, E_i) \mid 0 \leq i \leq \delta\}$, by defining E_i to be the set of edges whose weights are divisible by 2^i. (Note that an edge e with weight $w(e) = 2^j$ occurs in all graphs G_i for $j \geq i$.)

The idea is that pulses divisible by 2^i are handled by a so-called synchronizer γ_i, which is exactly synchronizer γ of [Awe85a], applied to the graph G_i. The synchronizer γ_i treats pulse $p \cdot 2^i$ as "super-pulse" p. It guarantees that super-pulse p is executed only *after* all messages sent along edges in E_i at super-pulse $(p-1)$ have arrived.

A vertex has to satisfy all δ synchronizers in order to proceed with a pulse. More specifically, consider a pulse $p = 2^j \cdot (2r+1)$, i.e., such that 2^j is the maximal power of 2 dividing p. Then pulse p is postponed until super-pulse $(2r+1) \cdot 2^{j-i}$ of synchronizer γ_i is executed. For example, pulse $24 = 3 \cdot 2^3$ is completed only after the synchronizers γ_0, γ_1, γ_2, and γ_3 are done carryng their pulses 24, 12, 6 and 3, respectively.

Lemma 4.4 Synchronizer γ_w is correct.

Proof Sketch: We need to show that under synchronizer γ_w, a vertex v generates pulse p only after receiving all messages it would receive by pulse p were the protocol executed on a synchronous network. This follows from the fact that the set of messages it would get by pulse p, i.e., the set of messages affecting this pulse, includes messages sent on edges belonging to G_i sent at pulse $p - 2^i$, and the arrival of these messages is guaranteed by synchronizer γ_i. ▌

4.3 Designing the protocol transformation

In order to justify the assumptions of the previous subsection we need to prove the following claim.

Lemma 4.5 Given a synchronous protocol π running on a synchronous weighted network $G(V, E, w)$, there exist a synchronous protocol π' and a synchronous network $G'(V, E, w')$ with the following properties:

1. G' is normalized.
2. The protocol π' is in synch with G'.
3. The output of π' on G' is identical to the output of π on G.
4. The time and communication complexities of a run of π' on G' are at most twice higher than the complexities of the corresponding run of π on G.

The lemma is proved throughout the rest of this subsection. We proceed as follows. Consider a message M transmitted by π on the edge $e = (u, v)$ with weight $w(e) = w$. For this message we define the following quantities.

S_M: the time by which M is sent by u.

R_M: the time by which M is received by v.

P_M: the *processing time* of M at v, which is the first time (at v) by which the contents of that message *might* actually be used, i.e., by which the vertex program at v behaves differently depending on the contents of M or the fact that M has not been sent.

Clearly, $S_M + w = R_M \leq P_M$. However, without loss of generality, we can assume $P_M = R_M$ for the protocol π.

Step 1: Transform π into a synchronous protocol π' slowed down by a factor of 4, i.e., where events that happen at time t at π now happen at time $t' = 4t$ at π'. That is, for any message M, the sending, receiving, and processing times S'_M, R'_M, P'_M in π' are shifted as follows.

$$\begin{aligned} S'_M &= 4S_M \\ R'_M &= S'_M + w = 4S_M + w \\ P'_M &= 4R_M = 4(S_M + 4w) \end{aligned}$$

Observe, however, that edge delays are not stretched by a factor of 4, and hence the arrival time R'_M of M in π' clearly precedes its processing time P'_M. This is not a problem, since the message can be kept in an edge buffer and be effectively ignored until time $t' + 4w$. This implies that the artificial increase in the delay of edge e from w to any value below $4w$ is not going to affect the protocol. This explains the next step.

Definition 4.6 Let $power(w)$ denote the smallest power of 2 that is larger than or equal to w, i.e., $power(w) = 2^{\lceil \log w \rceil}$.

Observe that $w \leq power(w) < 2w$.

Step 2: Instead of running π' on G, run π' on the network $\tilde{G}(V, E, \tilde{w})$, where $\tilde{w}(e) = power(w(e))$. The sending, receiving, and processing times $\tilde{S}'_M, \tilde{R}'_M, \tilde{P}'_M$ in π' on $\tilde{G}$ are

$$\begin{aligned} \tilde{S}'_M &= S'_M \\ \tilde{R}'_M &= S'_M + power(w) \\ \tilde{P}'_M &= P'_M \end{aligned}$$

Next, it is necessary to guarantee that messages are sent on e only at times divisible by $power(w)$.

Definition 4.7 Define $next_w(t)$ as the first time after t that is divisible by $power(w)$.

Observe that $t \leq next_w(t) \leq t + (w - 1)$.

Step 3: Modify π' to obtain a new protocol π'', running on $\tilde{G}$, where for the above message M, the sending, receiving, and processing times S''_M, R''_M, P''_M in π'' all shifted as

$$\begin{aligned} S''_M &= next_w(\tilde{S}'_M) \\ R''_M &= S'_M + power(w) \\ P''_M &= P'_M \end{aligned}$$

The main difference between π' and π'' is that the transmission of M in π'' may be delayed by at most w. This does not cause a problem, because the message M is being ignored until time $P''_M = 4w + S''_M$ at the receiving end; thus $P''_M \geq R''_M = S''_M + power(w)$ still holds.

4.4 Complexity

Lemma 4.8 The synchronizer γ_w described above has the following complexities:

$$\begin{aligned} \mathcal{C}_p(\gamma_w) &= O(k \cdot n \cdot \log W) = O(k \cdot n \cdot \log n) \\ \mathcal{T}_p(\gamma_w) &= O(\log_k n \cdot \log W) = O(\log_k n \cdot \log n) \end{aligned}$$

Proof: Synchronizer γ_i is invoked on the graph G_i once every 2^i time units. This costs us $O(2^i \cdot n \cdot k)$ in communication and $O(2^i \cdot \log_k n)$ time. This waste is amortized over 2^i time units, and then summed over all $0 \leq i \leq \log W$ graphs G_i. ▮

5 Controllers

Controllers are applicable in situations where one suspects the possibility that errors in the input data, or processor faults, may cause a given protocol π to diverge away from its specification. The danger is that while the protocol may have firm bounds c_π and t_π on its communication and time complexity under normal circumstances (i.e., for correct executions), it may waste valuable resources in an uncontrolled fashion once it diverges at some processors in the network. The task of a controller is to transform the protocol π into a "controlled" protocol ϕ whose semantics is identical to that of the original π under correct input but whose complexities are reasonably bounded even on incorrect input.

We consider here the same model as in [AAPS87]. The protocol starts at a certain vertex, called the *initiator*, and vertices enter the protocol as a result of receiving a message of the protocol. This model is referred to as "diffusing computation" by Dijkstra and Scholten [DS80]. It is worth mentioning that it is easy to extend the case of a single initiator to the general case of multiple initiators.

Suppose that at the time a vertex receives the first message of the protocol, it marks the edge over which the message has been received. It is easy to see that the collection of marked edges forms, at all times, a dynamically growing tree rooted at the initiator vertex. This tree is called the *execution tree* of the protocol.

Our purpose is to control the growth of this tree, namely, to guarantee that not too many messages are sent during the protocol, without affecting the "correct" executions of the protocol.

Towards this goal, the MAIN CONTROLLER of [AAPS87] views every message sent by the protocol as consuming one unit of some abstract "resource". The protocol must authorize every single consumption of the resource. That is, a vertex that wants to consume a resource unit (i.e. send a message) must first send a special "request" up on the execution tree, and then wait to get a special "grant" message, before the resource may actually be consumed.

The most naive (but inefficient) way of controlling the amount of resource units consumed in a growing tree is to request the root to authorize the consumption of each resource unit. That is, for each resource unit consumed, a request has to go up on the execution tree towards the root. Once this request reaches the root, the root increases its "permit counter" by 1. In case this counter is still less than or equal to some "threshold", it sends back a permit, which authorizes the consumption of the appropriate resource. Otherwise, if the counter has exceeded the "threshold", execution is suspended and the execution tree stops growing.

In our case, we need to set this threshold to the value of c_π, which is the complexity of π in a correct execution. Thus the naive controller above will not interfere with correct executions; its only effect is to stop executions that are obviously incorrect.

The more efficient algorithm of [AAPS87] is similar to the naive controller above in that the resource requests are propagated up to the root and the permits are distributed from the root. The algorithm of [AAPS87] takes advantage of the fact that a single message can represent a number of resource requests or a number of permits. The main idea is to aggregate a large number of requests/permits together, represent them by a single message, and allow this message to travel up the tree for a distance proportional to this number. Permits are kept not only at the root, but also at intermediate vertices. The root keeps an *approximate* permit counter, so that the actual number of resource units consumed is at most twice the value of this counter. For this reason, even though the root threshold remains the same as in the naive controller above, the guaranteed upper bound on the number of protocol messages sent will be twice greater than in the case of the naive controller, namely $2c_\pi$. Again, the algorithm does not interfere with correct executions of the protocol.

It is shown in [AAPS87] that the communication overhead of the authorization mechanism (namely, permit/grant messages) results in at most $\log^2 c_\pi$ control messages traversing a given edge of the execution tree. It follows that the total overhead of control messages, as well the total complexity of the resulting protocol ϕ is $c_\phi = O(c_\pi \log^2 c_\pi)$.

When running this protocol on a "weighted network", we consider a transmission of a message on an edge w as a request to consume $w(e)$ units of the resource. Essentially, this is equivalent to running the same algorithm as in [AAPS87] on the "unweighted" version $\tilde{G} = (\tilde{V}, \tilde{E}, \tilde{w})$ of the network, where an edge e is substituted by a path containing w_e edges and $w_e - 1$ "dummy vertices", and weight $\tilde{w}(e) = 1$ for all edges e. Since the results of [AAPS87] do not depend on the type of resource being consumed, we deduce:

Corollary 5.1 The CONTROLLER protocol given in [AAPS87] transforms an arbitrary protocol π into an equivalent "controlled" protocol ϕ whose (weighted) communication and time complexities are $c_\phi = O(c_\pi \log^2 c_\pi)$ and $t_\phi = O(c_\pi \log^2 c_\pi)$, respectively.

6 Basic algorithmic techniques

In this section we briefly describe several standard network algorithms and state their complexities in the weighted setting. These algorithms will be used as basic components in the more involved algorithms to be presented later.

6.1 The flooding algorithm $\texttt{CON}_{flood}$

The goal of the flooding algorithm $\texttt{CON}_{flood}$ (cf. [Seg83]) is to broadcast a message throughout the network. This is done as follows. Each vertex that receives the message for the first time forwards it further to all its neighbors. Future arrivals of the message are ignored.

Fact 6.1 Algorithm $\texttt{CON}_{flood}$ has communication complexity $O(\mathcal{E})$ and time complexity $O(\mathcal{D})$.

6.2 The depth-first search algorithm DFS

The goal of the DFS algorithm `DFS` (cf. [Eve79, Awe85b]) is to traverse the network in depth-first order. In order to make use of this algorithm as a component in the algorithms described later, it is necessary to modify it as follows. At any time, the algorithm maintains estimates of the total cost of all the edges traversed so far. Such estimates are kept both at the center of the activity and at the root, and are called the *root estimate* EST_R and the *center estimate* EST_C, respectively. The estimates are updated as follows.

1. Each time an edge is traversed, its weight is added to the center estimate EST_C.

2. The root estimate EST_R is updated only whenever the center of activity is about to traverse an edge that will cause the center estimate EST_C to double compared to the current value of EST_R. The update is done via a message from the center of activity to the root, which sets EST_R to be the new value of EST_C.

Thus, the center estimate is the total sum of edge weights for all edges traversed so far, while the root estimate is a lower bound on the total edge weight for all the edges traversed so far, plus the next edge to be traversed. Moreover, this lower bound is within a factor of two of the real value.

Observe that going to the root whenever the estimate is doubled can at most double the communication complexity, as this complexity can be viewed as the sum of a geometric progression. This establishes the following fact.

Fact 6.2 Algorithm DFS has communication complexity $O(\mathcal{E})$ and time complexity $O(\mathcal{E})$.

6.3 The full-information minimum spanning tree algorithm MST$_{centr}$

The full-information MST algorithm MST$_{centr}$ is similar in structure to Prim's MST algorithm (cf. [Eve79]). This algorithm proceeds in stages, each selecting the minimum-weight edge connecting a vertex in the tree with a vertex outside the tree, and adding this edge to the tree. The algorithm terminates when the tree spans all the vertices. The resulting tree is an MST of the network. It remains to show how to implement each stage.

Throughout the execution, the invariant maintained by the algorithm is that each vertex in the tree knows the structure of the whole tree. To preserve this invariant, whenever a new vertex is added to the tree, its name is broadcast on the tree, so it reaches all other vertices. This requires only one message on each tree edge for each new vertex added.

As in algorithm DFS, we maintain at the root an estimate on the total weight of all tree edges, called the *root estimate.* Observe that in this algorithm, the root estimate is precise, as the root knows the structure of the whole tree.

In order to analyze the algorithm we need the following fact, which gives an upper bound on the diameter of an MST.

Fact 6.3 For any minimum spanning tree T of G, $Diam(T) \leq \mathcal{V} \leq (n-1)\mathcal{D}$.

Proof: The first inequality is trivial. For the second, consider an edge $e = (v_1, v_2) \in T$. By definition of the MST, this edge induces a partition of V into two subsets of vertices, $V = V_1 \bigcup V_2$, such that $v_1 \in V_1$, $v_2 \in V_2$ and e is the minimum weight edge connecting a vertex in V_1 with a vertex in V_2. It follows that e is a shortest path from v_1 to v_2, since any other path from v_1 to v_2 contains at least one edge with one endpoint in V_1 and the other endpoint in V_2. Thus $w(e) \leq Diam(G) = \mathcal{D}$, and hence

$$\mathcal{V} = w(T) = \sum_{e \in T} w(e) \leq (n-1)\mathcal{D}.$$

▮

Each phase of the algorithm MST$_{centr}$ constructing a tree T requires $O(\mathcal{V})$ communication and $O(Diam(T))$ time. There are exactly $n-1$ phases. Consequently we have

Corollary 6.4 Algorithm MST$_{centr}$ has communication complexity $O(n \cdot \mathcal{V})$ and time complexity $O(\min\{n\mathcal{V},\ n^2\mathcal{D}\})$.

6.4 The full-information shortest path tree algorithm $\mathtt{SPT}_{centr}$

The full-information SPT algorithm $\mathtt{SPT}_{centr}$ is in fact a distributed implementation of Dijsktra's algorithm (cf. [Eve79, Gal82]), computing a shortest path tree rooted at a source s. This algorithm is very similar to the MST algorithm $\mathtt{MST}_{centr}$ described above. The algorithm proceeds in phases, each adding one more vertex to the tree. The tree vertices know the structure of the whole tree.

The vertices outside the tree are labeled. The label of a vertex x is the minimum, over all its neighbors y in the tree, of $dist(s,y)+w(y,x)$. In each phase, the vertex with minimum label is added to the tree. Once this vertex is chosen, its name is broadcast over the whole tree.

Again, we need a basic fact giving an upper bound on the total weight of a shortest path tree in order to analyze the algorithm.

Fact 6.5 For any vertex s and any shortest path tree T for s in G, $w(T) \le (n-1)\mathcal{V}$.

Proof: Consider an edge $e=(u,v)\in T$. Let T' be an MST of G, and let P denote $Path(u,v,T')$, the path connecting u with v in T'. By the definition of an SPT,

$$w(e) \le w(P) \le w(T') = \mathcal{V}.$$

Thus,

$$w(T) = \sum_{e\in T} w(e) \le (n-1)\mathcal{V}.$$

∎

Each phase of the algorithm $\mathtt{SPT}_{centr}$ constructing a tree T requires $O(w(T))$ communication and $O(\mathcal{D})$ time. There are exactly $n-1$ phases. Consequently we have

Corollary 6.6 Algorithm $\mathtt{SPT}_{centr}$ has time complexity $O(n\mathcal{D})$ and communication complexity $O(n^2\mathcal{V})$.

7 Connected components and spanning tree construction

In this section, we prove matching upper and lower bounds on the communication complexity of performing the tasks of finding connected components and constructing a spanning tree.

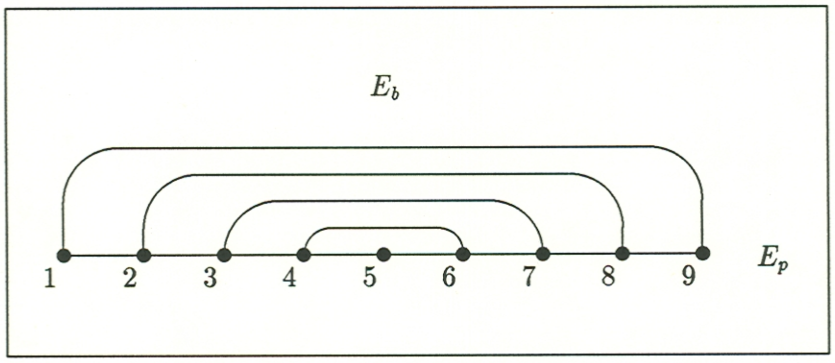

Figure 7: The graph G_9

7.1 Lower bounds

Let us first point out that an $\Omega(\mathcal{E})$ lower bound on communication is given in [AGPV89] for the case where all edge weights are unity. In the rest of this subsection, we prove an $\Omega(n \cdot \mathcal{V})$ lower bound on the communication complexity.

Consider the family of graphs $G_n = (V, E, w)$ defined as follows. $V = \{1, \ldots, n\}$. The set of edges is composed of two subsets, $E = E_p \cup E_b$, where the first subset creates a *path*, $E_p = \{(i, i+1) \mid 1 \leq i \leq n-1\}$, and the second subset consists of *bypassing edges*, $E_b = \{(i, n+1-i) \mid 1 \leq i \leq n/2\}$. The weights are defined as

$$w(e) = \begin{cases} X, & e \in E_p, \\ X^4, & e \in E_b, \end{cases}$$

where X is some large value, say $X \geq n$. Figure 7 depicts the graph G_n for $n = 9$.

Note that the MST for G is the subgraph (V, E_p) based on the path alone, so $\mathcal{V} = nX$.

We make some assumptions similar to those of [AGPV89] regarding the model. In particular, we assume that the only operation one can do with ID's is comparisons; this can be extended also to general operations in case the ID's are allowed to be sufficiently large.

In addition, let us formulate the following concepts. Note that every vertex in our graphs has two or three neighbors. Let us assume that each vertex maintains its own id in register R_I and the id's of its neighbors in input registers R_1, R_2 and R_3 (if needed). The only things that can be done with these id's is comparing them to each other and to other id's. The vertex does not distinguish between the registers. In particular, it cannot tell which register contains the id of its neighbor along the bypassing edge. However, in order to enable us to speak about this particular register, let us refer to it also as the *bypassing register*, R_B.

Messages may include vertex id's, as well as information about the relationships between id's, e.g., "the contents of register R_1 at vertex id ϕ is the id ϕ'." A vertex i may know any other vertex j only by its id, $\phi(j)$. When we say "vertex i obtains the contents ϕ' of register R_k of vertex j" we mean that at some stage of the run, i learns the fact that register R_k at the vertex whose id is $\phi(j)$ contains the id ϕ'. This can happen either by i being itself j or by its getting a message with that statement. Similarly, when saying "vertex i obtains the id of vertex j" we mean that at some stage of the run, the id $\phi(j)$ becomes available to i, i.e., either i is itself j or it gets a message containing $\phi(j)$ (as the id of a vertex, i.e., as the contents of register R_I of some vertex).

Let A be a deterministic algorithm that succeeds in computing a spanning tree on every input graph and whose communication complexity is $f(n) = o(n^4)$. In particular, this means that there exists a constant n_0 such that for every $n > n_0$, the algorithm A completes the construction of tree on G_n with communication cost less than n^4. Clearly, then, the algorithm does not send any messages over any bypassing edge in these graphs, since using such an edge immediately incurs a cost of n^4. Henceforce we restrict attention to graphs G_n for $n > n_0$.

Our proof is based on the following lemma.

Lemma 7.1 In the execution of A on G_n, for every $1 \leq i \leq n/2$ there is some vertex j such that one of the following two events must happen.

1. j receives both the id of i, $\phi(i)$, and the id stored in $(n+1-i)$'s bypassing register R_B.
2. j receives both the id of $n+1-i$, $\phi(n+1-i)$, and the id stored in i's bypassing register R_B.

Proof: Suppose that there is some $1 \leq i \leq n/2$ for whom the lemma does not hold. Consider the graph G_n^i obtained from G_n by adding two vertices v, w and replacing the edge $(i, n+1-i)$ with the two edges (i, v) and $(n+1-i, w)$, both with weight X^4. Figure 8 depicts the graph G_n^i for $n = 9$ and $i = 3$.

Select the id assignment $\phi(k) = 2k$ for $k \in V$, and the id's $\phi(v) = 2(n+1-i) - 1$ and $\phi(w) = 2i - 1$.

Consider the run of A on the graph G_n^i. We claim that the executions of A on G_n and G_n^i are similar. This is proved inductively on the length of the runs, noting that as long as no messages are sent over the edges connecting u and w to the rest of the graph, any comparison made by any vertex has the same result except a comparison of the id of i to the contents of the bypassing register R_B of $n+1-i$, or a comparison of the id of $n+1-i$ to the contents of the bypassing register R_B of i. Since no vertex holds both these values,

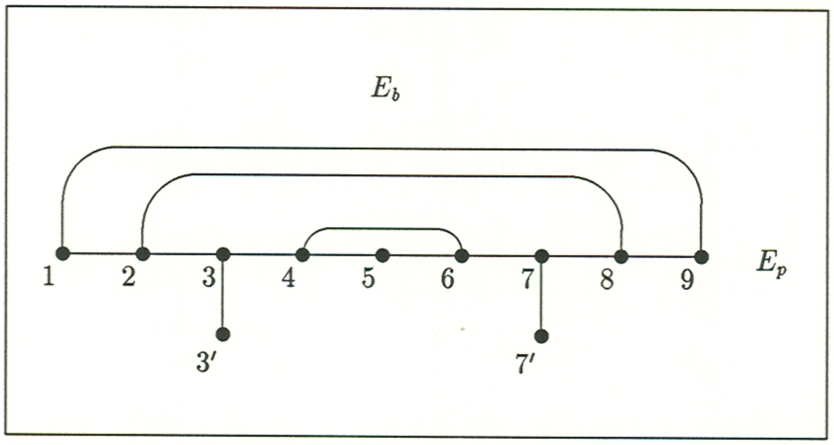

Figure 8: The graph G_9^3

the runs will remain similar. This implies that in G_n^i, the vertices v and w will not join the spanning tree, contradicting the assumption that A operates correctly on every graph. ▮

Lemma 7.2 Algorithm A requires $\Omega(n\mathcal{V})$ messages. ▮

Proof: Consider the execution of A on G_n and pick some $1 \leq i \leq n/2$. It follows from the previous lemma that messages containing the names i or $n+1-i$ were passed over at least $n+1-2i$ edges during the run. Thus the message complexity of A is at least

$$X\sum_{i=1}^{n/2}(n+1-2i) \;\geq\; n^2X/4 \;=\; \Omega(n\mathcal{V}).$$

▮

7.2 An upper bound

Claim 7.3 Algorithm $\texttt{CON}_{hybrid}$ (presented below) requires $O(\min\{\mathcal{E}, n\cdot\mathcal{V}\})$ messages.

We now describe Algorithm $\texttt{CON}_{hybrid}$, whose communication complexity is the minimum between those of the algorithms DFS and $\texttt{MST}_{centr}$ presented above. In effect, algorithm $\texttt{CON}_{hybrid}$ runs algorithms DFS and $\texttt{MST}_{centr}$ in parallel. The idea is that the root vertex "controls" both algorithms and suspends the more expensive one. Towards this goal, the root maintains the variables W_a and W_b which are the root estimates of algorithms DFS and $\texttt{MST}_{centr}$, respectively. In addition, the root maintains the variable *Permit* which takes values $\{\texttt{DFS}, \texttt{MST}_{centr}\}$ and is updated whenever W_a or W_a are updated at the root. As a rule,

$$Permit = \begin{cases} \texttt{DFS}, & W_a < W_b, \\ \texttt{MST}_{centr}, & \text{otherwise.} \end{cases}$$

When $Permit$ = DFS, algorithm DFS is running and algorithm $\mathtt{CON}_{centr}$ is suspended, and vice versa.

Observe that it is easy to suspend either of the two algorithms, as algorithm DFS (respectively, $\mathtt{MST}_{centr}$) needs to be suspended only when W_a (resp., W_b) is increased. At that moment, the center of activity of algorithm DFS (resp., $\mathtt{MST}_{centr}$) is located at the root, so the algorithm can be suspended by requesting that the center of activity stays at the root.

Since at any time the root estimates are within a factor of two of the actual communication costs of both algorithms, and since only the algorithm with the smaller estimate is enabled at any given time, the total complexity of algorithm $\mathtt{CON}_{hybrid}$ cannot exceed the complexity of the cheaper of the two algorithms by more than a factor of four.

8 Fast minimum spanning tree algorithms

This section proceeds as follows. In order to explain the more complex algorithms $\mathtt{MST}_{fast}$ and $\mathtt{MST}_{hybrid}$, we start with a description of the algorithm of [GHS83], referred to as "algorithm $\mathtt{MST}_{ghs}$", and show that it has communication complexity $O(\mathcal{E}+\mathcal{V}\cdot\log n)$ and time complexity $O(\mathcal{E}+\mathcal{V}\cdot\log n)$. Then, we develop our new algorithms:

- An algorithm $\mathtt{MST}_{hybrid}$ with communication complexity $O(\min\{\mathcal{E}+\mathcal{V}\log n, n\cdot\mathcal{V}\})$.
- An algorithm $\mathtt{MST}_{fast}$ with communication complexity $O(\mathcal{E}\log\mathcal{V}\log n)$ and time complexity $O(Diam(MST)\log\mathcal{V}\log n) = O(n\mathcal{D}\log\mathcal{V}\log n)$.

8.1 Algorithm $\mathtt{MST}_{ghs}$

The algorithm consists of two stages, namely, a *wake-up* stage followed by a *work* stage. The wake-up stage of [GHS83] is based on flooding a "wake-up" message through the network. The work stage can be thought as a modification of the following simple algorithm. The algorithm consists of $\log n$ phases. At each phase, we have a number of "fragments", i.e., subtrees of the MST, that try to merge, in parallel, into larger fragments. Towards this goal, each fragment performs the following steps:

1. The name of the fragment is broadcast from the root to all the vertices.
2. Each vertex scans, serially and in decreasing order of weights, all edges that have not been scanned so far, until an edge outgoing from the fragment is found. The name of that edge and its weight are reported to the parent.

3. Each vertex collects reports from its children containing the name of the edge chosen by the subtree rooted at the child. Among those, the minimum weight edge is selected and propagated to the parent. The vertex also marks its edge to the child in whose subtree the selected edge is located.

4. When the root selects its outgoing edge, the path from the selected edge to the root has been marked. Now, the root of the tree is moved along this path, and the fragment is hooked onto another fragment along the outgoing edge.

It is easy to implement the above algorithm if the phases of different fragments are synchronized. However, synchronization requires scanning all the edges. The algorithm of [GHS83] manages to accomplish its task without synchronization, thus saving communication. For more details, see [GHS83].

Lemma 8.1 Algorithm $\mathtt{MST}_{ghs}$ requires $O(\mathcal{E} + \mathcal{V} \cdot \log n)$ communication and $O(\mathcal{E} + \mathcal{V} \cdot \log n)$ time.

Proof: The wake-up stage obviously takes $O(\mathcal{E})$ communication and $O(\mathcal{D})$ time. Throughout the algorithm, each non-tree edge is scanned at most twice, and each tree edge is scanned $\log n$ times. It follows that the communication complexity is $O(\mathcal{E} + \mathcal{V} \cdot \log n)$. The time complexity is naturally bounded by the communication complexity. ▮

Unfortunately, this algorithm does not exhibit any parallelism; its time complexity could be almost as high as its communication complexity. This is due to the fact that the edges scanned in a given phases may be much heavier that the weight of the MST itself. Thus, edge scanning contributes a term of $\mathcal{E}$ to the time complexity. The communication over MST contributes additional $O(Diam(MST) \cdot \log n) = O(\mathcal{V} \cdot \log n)$ time.

8.2 Algorithm $\mathtt{MST}_{hybrid}$

The "hybrid" MST algorithm, called $\mathtt{MST}_{hybrid}$, is obtained according to the following plan:

1. Modify the wake-up phase of algorithm $\mathtt{MST}_{ghs}$ so that instead of flooding, wake-up is performed via the `DFS` algorithm of the previous section.

2. Combine the resulting algorithm with algorithm $\mathtt{MST}_{centr}$, as in Section 7.2.

The idea of the first step is that the protocol becomes "controlled", i.e., the root is aware of the communication wasted so far. This makes it easy to perform the next step, where we achieve the minimum between the two algorithms.

Corollary 8.2 Algorithm $\texttt{MST}_{hybrid}$ has communication complexity $O(\min\{\mathcal{E} + \mathcal{V}\log n, n\mathcal{V}\})$.

8.3 Algorithm $\texttt{MST}_{fast}$

The algorithm $\texttt{MST}_{fast}$ is a modification of algorithm $\texttt{MST}_{ghs}$. The idea is to reduce the time it takes to scan very heavy edges that obviously do not belong to the MST. Also, we wish to avoid the time-consuming process of scanning the edges serially. Towards this goal, we modify the process of selecting an outgoing edge of a fragment as follows.

In order to avoid the scanning of heavy edges, the root makes a "guess" for the weight of the outgoing edge. Initially, this guess is 1. If the guess is too low, then the process of searching for an outgoing edge fails. In this case, the root doubles its guess and repeats the search. This continues until the search succeeds.

In order to achieve concurrency in the edge scanning process, the vertices scan all the edges that are below the value guessed by the root *in parallel*. This guarantees that the time of the search is upper-bounded by $O(Diam(MST))$.

Corollary 8.3 Algorithm $\texttt{MST}_{fast}$ requires $O(\mathcal{E} \cdot \log n \log \mathcal{V})$ communication and $O(Diam(MST) \log \mathcal{V} \log n) = O(n\mathcal{D} \log \mathcal{V} \log n)$ time.

9 Fast shortest path tree algorithms

9.1 Algorithm $\texttt{SPT}_{synch}$

Algorithm $\texttt{SPT}_{synch}$ is the fastest SPT algorithm we know of. It is obtained by combining the synchronizer of Section 4 with the synchronous SPT algorithm.

Observe that the synchronous SPT algorithm runs in $O(\mathcal{D})$ time and has $O(\mathcal{E})$ communication complexity (again, making the assumption that a message sent on edge e requires precisely $w(e)$ time). The synchronizer adds $O(kn \log n)$ communication overhead and $O(\log_k n \log n)$ time units for each of the $\mathcal{D}$ time units of the original synchronous protocol. We thus have

Corollary 9.1 The algorithm $\texttt{SPT}_{synch}$ requires $O(\mathcal{E} + \mathcal{D} \cdot kn \cdot \log n)$ in communication and $O(\mathcal{D} \cdot \log_k n \cdot \log n)$ time.

9.2 Algorithm SPT_{recur}

Observe that a "weighted" network $G = (V, E, w)$ can be reduced to a BFS problem on an "unweighted" network $\tilde{G} = (\tilde{V}, \tilde{E}, \tilde{w})$ where an edge e is substituted by a path containing w_e edges and $w_e - 1$ "dummy vertices", and weight $\tilde{w}(e) = 1$ for all edges e. Thus, we can construct an SPT of the original graph by running the BFS algorithm as in [Awe89] on the "unweighted" version $\tilde{G} = (\tilde{V}, \tilde{E}, \tilde{w})$ of the network.

The BFS algorithm of [Awe89] is based an a very simple BFS algorithm, referred to in the sequel as the DIJKSTRA Algorithm, due to its resemblance to Dijkstra's shortest path algorithm (cf. [Gal82]) and Dijkstra-Sholten distributed termination detection procedure [DS80]. This algorithm is similar to Algorithm SPT_{centr} of Section 6, with the only difference being that the names of vertices that join the tree do not have to be broadcast along the tree.

The algorithm maintains a tree rooted at the source vertex. Initially, the tree is empty. Upon termination of the algorithm, the tree is the desired BFS tree. Throughout the algorithm, the tree can only grow, and at any time it is a subtree of the final BFS tree. The algorithm operates in successive iterations, each adding one more BFS layer to the tree. At the beginning of a given iteration l, the tree contains all vertices in layers less than or equal to $l - 1$. Upon termination of iteration l, the tree is extended to layer l as well.

The complexities of this algorithm are $O(d \cdot n + E)$ messages and $O(d \cdot D)$ time, where d is the number of layers being processed. Indeed, there are d iterations and in each of them synchronization is performed over the BFS tree, which requires $O(n)$ messages and $O(D)$ time. In addition, one exploration message is sent over each edge once in each direction. Obviously, the performance of the algorithm degrades as the number d of layers to be processed increases.

The idea behind [Awe89] is to reduce the problem where a large number of layers needs to be processed to a problem with a small number of layers. If the network has diameter D, we can conceptually "slice" the network into $\frac{D}{d}$ "strips" of length d, and process those strips sequentially (see Figure 9).

In [Awe89], an efficient reduction strategy is proposed. It is proved in [Awe89] that, in the unweighted case, $O(D^{\epsilon})$ time units are spent on each BFS layer, and $O(E^{\epsilon})$ messages traverse each network edge. In our (weighted) case, we have $\mathcal{D}$ layers, and $\mathcal{E}$ edges in the "unweighted" version of our network. Thus, we have

Corollary 9.2 The weighted communication and time complexities of the resulting protocol SPT_{recur} are $O(\mathcal{E}^{1+\epsilon})$ and $O(\mathcal{D}^{1+\epsilon})$, respectively.

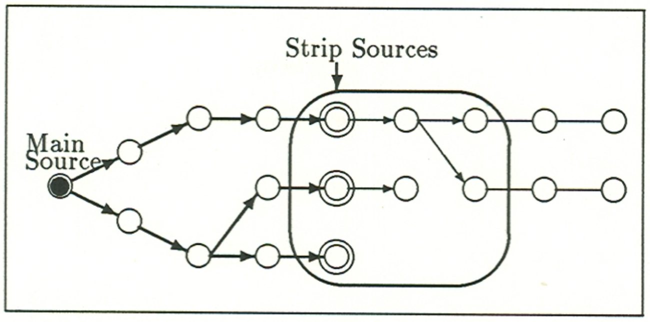

Figure 9: Strip Method.

9.3 The hybrid SPT algorithm SPT_{hybrid}

As before, it is possible to combine the two SPT algorithms SPT_{synch} and SPT_{recur} and obtain an algorithm SPT_{hybrid} that is as efficient in communication as either one of the two. This is done in a manner similar to design of the hybrid MST algorithm MST_{hybrid} in Section 8.2.

Corollary 9.3 The weighted communication and time complexities of the resulting protocol SPT_{hybrid} are given by $O(\min\{\mathcal{E}^{1+\epsilon}, \mathcal{E} + n \cdot \mathcal{D}\})$.

Acknowledgments

We warmly thank Oded Goldreich for his illuminating comments on a previous draft.

References

[AAPS87] Yehuda Afek, Baruch Awerbuch, Serge A. Plotkin, and Michael Saks. Local management of a global resource in a communication network. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, October 1987.

[ABLP89] Baruch Awerbuch, Amotz Bar-Noy, Nati Linial, and David Peleg. Compact distributed data structures for adaptive network routing. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 230–240, May 1989.

[AGPV89] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A tradeoff between information and communication in broadcast protocols. *J. of the ACM*, 1989.

[ALSY88] Y. Afek, G.M. Landau, B. Schieber, and M. Yung. The power of multimedia: combining point-to-point and multiaccess networks. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 90–104, Toronto, Canada, August 1988.

[AP90a] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 514–522, 1990.

[AP90b] Baruch Awerbuch and David Peleg. Sparse partitions. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pages 503–513, 1990.

[AP91] Baruch Awerbuch and David Peleg. Routing with polynomial communication - space trade-off. *SIAM J. on Discr. Math.*, 1991. To appear. Also in Technical Memo TM-411, MIT, Sept. 1989.

[Awe85a] Baruch Awerbuch. Complexity of network synchronization. *J. of the ACM*, 32(4):804–823, October 1985.

[Awe85b] Baruch Awerbuch. A new distributed depth-first-search algorithm. *Info. Process. Letters*, 20:147–150, April 1985.

[Awe87] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the* 19^{th} *Annual ACM Symposium on Theory of Computing*, pages 230–240, May 1987.

[Awe89] Baruch Awerbuch. Distributed shortest paths algorithms. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 230–240, May 1989.

[BKJ83] K. Bharath-Kumar and J. M. Jaffe. Routing to multiple-destinations in computer networks. *IEEE Trans. on Commun.*, pages 343–351, March 1983.

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Info. Process. Letters*, 11(1):1–4, August 1980.

[ER90] Shimon Even and Sergio Rijsbaum. The use of a synchronizer yields maximum computation rate in distributed networks. In *Proc. 22nd ACM Symp. on Theory of Computing.* ACM SIGACT, ACM, May 1990.

[Eve79] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[Gal82] Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, MIT, Lab. for Information and Decision Systems, January 1982.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

[GS86] O. Goldreich and L. Shrira. The effects of link failures on computations in asynchronous rings. In *Proc. 5th ACM Symp. on Principles of Distributed Computing*, pages 174–186. ACM, August 1986.

[Jaf80] Jeffrey Jaffe. Using signalling messages instead of clocks. Unpublished manuscript., 1980.

[Jaf85] Jeffrey M. Jaffe. Distributed multi-destination routing: the constraints of local information. *SIAM Journal on Computing*, 14:875–888, 1985.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[PU89] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. on Comput.*, 18(2):740–747, 1989.

[Seg83] Adrian Segall. Distributed network protocols. *IEEE Trans. on Info. Theory*, IT-29(1):23–35, January 1983. Some details in technical report of same name, MIT Lab. for Info. and Decision Syst., LIDS-P-1015; Technion Dept. EE, Publ. 414, July 1981.