# Compact Distributed Data Structures for Adaptive Routing

Baruch Awerbuch †
*Dept. of Mathematics and Lab. for Computer Science*
*MIT, Cambridge, MA 02138*

Amotz Bar-Noy ¶
*Computer Science Department, Stanford University*
*Stanford, CA 94305*

Nathan Linial
*IBM Almaden Research Center 650 Harry Road, San Jose, CA 95120*
*and Computer Science Department, Hebrew University, Jerusalem*

David Peleg §
*Department of Applied Mathematics*
*The Weizmann Institute, Rehovot 76100, Israel*

In designing a routing scheme for a communication network it is desirable to use as short as possible paths for routing messages, while keeping the routing information stored in the processors' local memory as succinct as possible. The efficiency of a routing scheme is measured in terms of its *stretch factor*—the maximum ratio between the cost of a route computed by the scheme and that of a cheapest path connecting the same pair of vertices.

This paper presents a family of adaptive routing schemes for general networks. The *hierarchical schemes* $HS_k$ (for every fixed $k \geq 1$) guarantee a stretch factor of $O(k^2 \cdot 3^k)$ and require storing at most $O(kn^{2/k} \log n)$ bits of routing information *per vertex*. The new important features, that make the schemes appropriate for adaptive use, are

- applicability to networks with arbitrary edge costs;
- name-independence, i.e., usage of original names;
- a balanced distribution of the memory;
- an efficient on-line distributed preprocessing.

† Supported by Air Force contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.
¶ Supported in part by a Weizmann fellowship and by contract ONR N00014-88-K-0166.
§ Part of this work was carried out while this author was visiting Stanford University. Supported in part by a Weizmann fellowship, by contract ONR N00014-88-K-0166 and by a grant of Stanford Center for Integrated Systems.

Copyright © 1989, Stichting Mathematisch Centrum, Amsterdam
Printed in the Netherlands

## 1. INTRODUCTION

### 1.1. Background

A central activity of any computer network is the passing of messages among the processors. This activity is performed by a routing subsystem, consisting of a collection of message forwarding mechanisms and information tables, whose quality is pivotal to the overall performance of the network. It is therefore natural that the design of efficient routing schemes was the subject of much study over the last two decades.

When designing a routing strategy for a network it is clearly desirable to be able to route messages with small communication cost. The cost of routing a message is simply the sum of the costs of the transmissions performed during the routing. The *route efficiency* of a routing scheme is formulated in terms of its *stretch factor*—the maximum ratio (over all possible origin-destination pairs) between the communication cost of routing a message from an origin to a destination using the scheme and the cheapest possible cost for passing a message from origin to destination.

At the same time, the space used for the routing tables is also a significant consideration. There are a number of reasons to minimize the memory requirements of a routing scheme. The task of routing is usually performed by a special-purpose processor (an 'IMP' in the ISO terminology [12,15]) which may have limited resources. Furthermore, it is usually desirable that the routing tables be kept in fast memory (e.g., a 'cache'), in order to expedite message traffic. Also, we do not want memory requirements to grow fast with the size of the network, since it means that the incorporation of new nodes to the network requires adding hardware to all the nodes in the network. It is therefore interesting to search for routing schemes that involve small communication cost and have low space requirements at the individual vertices.

The problem of routing with small memory is analogous to the problem of designing *compact* signs at highway exits, which enable drivers to find their way, even if they do not have any information about geography of the area. Since signs do not contain full map of the area, drivers will sometimes make mistakes, i.e. take wrong exits; however drivers should be capable to 'learn' from the mistakes and eventually find a way out. Intuitively, it appears plausible that the 'larger' the signs (i.e. memory overhead), the less time and gas is wasted (i.e. communication overhead).

Let us look at two extreme examples. The *direct* routing scheme in an $n$-processor network is constructed by specifying, at each node $v$, a set of $n-1$ pointers, one pointer for each possible destination node $x \neq v$. Each such pointer points to some neighbour $w$ of $v$, to which $v$ will forward a message destined to $x$. The message is forwarded along those pointers until it eventually arrives at its destination. Clearly, it is advantageous to set up the pointers with respect to a fixed destination $x$ in such a way that they form a tree of shortest paths from the node $x$, based on the edge costs. Then the communication cost of such a routing (measured in terms of the sum of the costs of all message transmissions) is optimal, i.e., the stretch factor is 1. The disadvantage

of the direct routing scheme is that each node has to maintain a very large ($\Omega(n)$ bit) routing table.

At the other extreme lies the *flooding* routing scheme, in which instead of forwarding a message along a shortest path, the origin simply floods (broadcasts) it through the whole network. Clearly, this scheme requires no memory overhead. On the other hand, the communication cost of such scheme may be significantly higher than optimal, since instead of using just one link, we may be using a lot of (possibly expensive) links. Thus, the stretch factor is unbounded.

The natural question which arises here is whether one can design a routing scheme which combines low memory requirements and small communication cost.

*1.2. Adaptive vs. static schemes*

In classifying the various types of routing schemes one usually distinguishes between those based on *adaptive* and *static* policies. While static routing schemes are simpler to design and maintain, it is commonly accepted that adaptiveness is crucial to the efficient operation of any 'store and forward' communication network.

The most significant feature of an adaptive routing scheme is its ability to sense changes in the traffic distribution and the load conditions throughout the network, and modify the routes accordingly, so that messages in transition avoid congested or disconnected areas of the network. Typically, the routing subsystem of the network is required to perform periodic updates in the routing scheme. Such an update operation involves two main steps. First, it is necessary to collect some information about the network state, like processor and link operational status, current queue loads and expected traffic loads. The collected data is used to compute the new *edge costs*. The cost associated with a link reflects the estimated link delay for a message transmitted over that link. The next step involves deciding on the new routes and setting up the information tables accordingly. In this paper we do not concern ourselves with precisely how the link costs are determined. Rather, we concentrate on the second step of setting up a routing scheme, with respect to the parameters of route efficiency and memory requirements discussed earlier.

The adaptive approach imposes several inherent requirements on the routing schemes. These essential requirements are sometimes hard to achieve. One may list four main properties characterizing an adaptive scheme.

*Arbitrary edge costs.* The entire adaptive approach revolves on the ability to compute and attach *varying* costs to the edges. Consequently, an inherent requirement is that the routing schemes be able to handle arbitrary edge costs (as well as arbitrary network topologies). An additional desirable property one should strive to achieve is that the complexity of the routing scheme does not depend on the range of the costs (i.e., that the routing algorithm is 'purely combinatorial').

*Name-independence.* Many proposed routing strategies have the property that the initial design has to determine not only the routes but also the labels used for addressing the vertices. In such strategies, the addressing label of a node encrypts partial information needed for computing routing paths towards it. (Of course, these labels cannot be too large.) (In the extreme, if one allows a node's name to contain an explicit description of the shortest paths leading to it from all other nodes, then the routing problem becomes trivial, but labels become prohibitively large.)

To get convinced that modifying user names by appending appropriately chosen addressing labels makes the problem much easier, consider the (somewhat similar) telephone system and observe that it is easy (and cheap) to get the phone number of an old friend if you know the exact city in which your friend lives, since it takes one call to appropriate telephone directory. On the other hand, it is a difficult (and expensive) task without this knowledge, since to locate the city where your friend lives, you may end up calling many directories all around the country.

The approach of using addressing labels is reasonable for static routing, where the routes and the labels are fixed once and for all. However, it is obviously inappropriate for adaptive schemes, since it would require changing the addresses of nodes each time the routes are re-computed. Clearly, it is essential that routing-related system activities be transparent to the user, and in particular, the addresses specified by a user in order to describe the destination of its messages should be fixed and independent of the actual routes. (In fact, it is preferable to allow each vertex to choose its own address.)

A viable approach is to allow the scheme to employ (changeable) routing labels internally, but use original (fixed) node names for addressing by users. This requires the routing algorithm to be able to extract the necessary routing labels on the basis of the original name. Clearly, the naive approach of storing 'translation' tables at the nodes requires $\Omega(n)$ memory bits at each node even if the addressing labels are short, and immediately defeats the very essence of memory-efficient routing. Thus a more sophisticated retrieval mechanism is required.

*Balanced memory.* In many routing schemes, different vertices play different roles and require different amounts of space. For instance, some nodes are designated as *communication centers* and are required to store more information than others. Other nodes may just happen to be the crossing point of many routes which are maintained by the scheme. The roles are assigned to nodes on the basis of graph-theoretic considerations and depend crucially on the edge costs. Nevertheless, those schemes guarantee that the total (and thus the average) memory requirements are small.

Such variability in space requirements is again reasonable for static routing, as some computers may have more memory than others, and it is possible to tailor the scheme to the availability of appropriate resources at specific nodes, and designate in advance some nodes to play the role of communication centers. However, in an adaptive setting the routes change dynamically, and in

principle it is possible for any node to play any role. This forces *every* node to have sufficient memory for performing the *most demanding* role in the scheme, rendering the bound on average space meaningless. Thus, it is necessary to ensure *balanced* memory requirements, i.e., to guarantee a bound on the *worst-case* (rather than average) memory requirements of each node. Such bound is important even in major nodes where memory is not a problem, since as mentioned earlier, the special-purpose routing processor (the 'IMP') may have limited resources and small fast memory.

*On-line preprocessing.* In real systems it is common practice to perform very frequent updates of the routing tables. (For example, in ARPA the routes are recomputed about every second.) The routing scheme has to accommodate for an efficient performance in this respect. One may consider a centralized algorithm which collects all the information about the network into a single computer, runs a sequential algorithm and then notifies all network nodes what routes were selected and which data structures should be maintained. However, this approach is resource-consuming, and it is preferable to have a preprocessing algorithm which is distributed and space efficient. In particular, one should take into account the fact that some nodes have limited amounts of memory, so it is desirable that the preprocessing algorithm obeys the same space constraints imposed on the size of routing tables in the individual nodes. (This may be of secondary importance when the update phase is allowed access to auxiliary, slower memory.)

### *1.3. Existing work*

The problem was first raised in [5]. The solution given there and in several consequent papers [6,8,11] applies only to some special networks. Optimal or near-optimal routing strategies were designed for various topologies like trees [10], rings, complete networks and grids [13,14], series-parallel networks, outerplanar, $k$-outerplanar and planar networks [3,4]. (By 'optimal' we mean here stretch factor 1 and a total memory requirement of $O(n \log n)$ bits in an $n$-processor network.)

In [9] the problem is dealt for general networks. Rather than designing a scheme only for some fixed stretch factor, the method presented in [9] is parameterized, and applies to the entire range of possible stretch factors. The construction yields an almost optimal behaviour, as implied from a lower bound given in [9] on the space requirement of any scheme with a given stretch factor. Specifically, the hierarchical routing schemes of [9] guarantee a stretch factor of $12k+3$ while requiring a total of $O(n^{(k+1)/k})$ bits of routing information in the network (for every fixed $k \geqslant 1$).

Unfortunately, the routing strategy of [9] lacks all four properties required from an adaptive routing scheme. It deals only with unit-cost edges (while the construction of [5] and the separator-based strategies of [3], for instance, apply also to the case of networks with costs on the edges). It is name-dependent, since the scheme has to be allowed to fix the addressing labels for all vertices. (This problem exists also in virtually all previous works.) Local memory is not

balanced; the method bounds the *total* space requirements of the scheme, but not the *individual* memory requirements of each node, which may reach $\Omega(n^{(k+1)/k})$ bits. (This problem exists also in some previous works, e.g., [3,4,10].) Finally, its preprocessing algorithm is a centralized (polynomial time) one. (Again, this issue was not considered in previous work as well.)

Two previous schemes of ours [1,2] have tackled these problems. Each of those schemes succeeds in achieving three of the four desirable properties but leaves one out (balanced memory and name-independence, respectively). The current scheme is based on ingredients taken from these two schemes combined with some additional new ideas.

### *1.4. Contributions of this paper*

This paper suggests a novel approach to the problem of routing with small space. In contrast to previous works, it enables to simultaneously achieve *all* the adaptiveness properties mentioned above. We present a family of *hierarchical schemes* $HS_k$, for every $\log n \geqslant k \geqslant 1$, which use $O(k \cdot \log n \cdot n^{2/k})$ bits of memory per vertex and guarantee a stretch of $O(k^2 \cdot 3^k)$. Note that these complexities do not depend on the range of allowed edge costs, i.e. the algorithm is 'purely combinatorial'. We also have an efficient distributed preprocessing algorithm for setting up the tables, that uses space which is bounded by the same bounds per vertex.

Our approach is based on constructing a *hierarchy* of partitions in the network, and using this hierarchy for routing. In each level, the graph is partitioned into clusters, each managed by a center, or 'pivot' node. Messages are transferred to their destinations using these pivots. This indirect forwarding enables us to reduce the memory requirements needed for routing, since one has to define routing paths only for cluster pivots, and not for all the nodes. On the other hand, it increases communication cost, since messages need not, in general, be moving along shortest paths towards their destination. With appropriate partition of the network into clusters we guarantee that both overheads are low.

The particular construction method described here differs from that of [9] in several important ways. To begin with, the two methods make use of inherently different *hierarchical designs*. In [9], the scheme is composed of a *collection* of *independent* schemes. Each scheme $\mathcal{RES}_i$, $i \geqslant 1$, is based on a partition of the network into clusters of radius $2^i$. Transmitting a message from an origin to a destination is based on a 'trial and error' process involving repeated trials, each attempting to route the message using *one* of the individual scheme. In case of failure, the message is returned to the origin, which then tries the next scheme in the hierarchy. The route of each individual scheme is itself a complex path, composed of three segments: from the origin to its cluster leader in that partition, from this leader to the leader of the destination and finally to the destination. In contrast, the routing process described here is conceptually much simpler. A message is passed from the origin to the destination in a single try, and no retries are needed. The path consists of two main segments: first, from the origin to its pivot on the

appropriate level (via a chain of lower-level pivots), and then to the destination itself. The hierarchical organization is thus utilized *internally* within the (single) routing scheme.

A second important difference is in the clustering method. The clusters described in [9] are based on *radius* constraints, whereas the clustering structure proposed here is based on *size* constraints. This difference is responsible for the fact that the new method is capable of handling arbitrary edge costs and that individual memory requirements can be bounded.

Finally, the proposed scheme employs a novel type of distributed data structure that enables it to store routing information (such as routing labels) compactly in a balanced fashion among the nodes of the network, and guarantees efficient data retrieval.

The rest of the paper is organized as follows. The next Section 2 contains necessary definitions. In Section 3 we outline the hierarchical routing scheme. In Section 4 we introduce some technical preliminaries. In the following Section 5 we fill in the missing details of the scheme, describing the various routing tools and components. Section 6 gives the complexity analysis for the combined scheme. Finally, in Section 7 we give a distributed preprocessing algorithm that initializes the schemes $HS_k$ .

## 2. Definition of the problem

### *2.1. The network model*

We consider the standard model of a point-to-point communication network, described by an undirected graph $G=(V,E)$, $V=\{1,...,n\}$. The vertices $V$ represent the processors of the network and the edges $E$ represent bidirectional communication channels between the vertices. A vertex may communicate directly only with its neighbours, and messages between nonadjacent vertices are sent along some path connecting them in the network.

We assume existence of a *weight* function $w: E \rightarrow \Re^+$, assigning an arbitrary positive weight $w(e)$ to each edge $e \in E$. Also, there exists a *name* function *name*: $V \rightarrow U$, which assigns to each node $v \in V$, an *arbitrary* name *name*$(v)$ from some universe $U$ of names. We sometime abuse notation, referring to *name*$(v)$ simply by $v$.

We assume a *synchronous* network model in the sense that all nodes have access to a global clock.

A message sent upon time $\tau$ from node $v$ to node $u$ arrives at $u$ strictly *after* time $\lceil\tau\rceil + w(v,u) - 1$, but no later than $\lceil\tau\rceil + w(v,u)$. Intuitively, this means that messages can be sent only at integer times. (This is why arrival times depend on $\lceil\tau\rceil$, rather than $\tau$, and edge delays $\delta(e)$ may fluctuate as $w(e) - \epsilon \leqslant \delta(e) \leqslant w(e)$, for all $\epsilon < 1$, i.e. the delay of $e$ is *at most* $w(e)$.)

Since we allow message transmissions at times which are not integers, the algorithm is driven by messages and by the global clock. Messages can be sent either in response to arriving messages, or in response to clock pulses, which occur at integer times.

For two vertices $u,w$ in a graph $G$ let $dist_G(u,w)$ denote the (weighted) length

of a shortest path in $G$ between those vertices, i.e. the cost of the cheapest path connecting them, where the cost of a path $(e_1,...,e_s)$ is $\Sigma_{1 \leq i \leq s} w(e_i)$. For two sets of vertices $U, W$ in $G$, let $dist_G(U, W) = \min\{dist_G(u,w) | u \in U, w \in W\}$. The degree (number of neighbours) of each vertex $v \in V$ is denoted by $deg_G(v)$. (We sometimes omit the subscript $G$ where no confusion arises.)

*2.2. Routing schemes*

A *routing scheme RS* for the network $G = (V,E)$ consists of two procedures, a *preprocessing* protocol and a *delivery* protocol. The preprocessing protocol performs certain preprocessing in the network, by constructing some distributed data-structures.

The delivery protocol can be invoked at any *origin* node and be required to deliver a certain message to some *destination* node, which is specified by its *name*. The protocol delivers the message from the origin to the destination via a sequence of message transmissions, which depends on the particular data structures constructed by the preprocessing protocol.

*2.3. Complexities of routing schemes*

It is convenient to define a *character* as $\log n$ bits, and to count communication and space in terms of characters. We assume that messages sent and variables maintained at nodes contain a constant number of characters.

We now give precise definitions for our complexity measures for stretch and memory. The *communication cost* of transmitting a message over edge $e$ is the weight $w(e)$ of that edge. The *communication cost* of a *protocol* is the sum of the communication costs of all message transmissions performed during the protocol. Let $C(RS,u,v)$ denote the communication cost of the delivery protocol when invoked at an origin $u$, with respect to a destination $v$ and an $O(1)$-character message, i.e., the total communication cost of all message transmissions associated with the delivery of the message. Given a routing scheme $RS$ for an $n$-processor network $G = (V,E)$, we say that $RS$ *stretches* the path from $u$ to $v$ by $\frac{C(RS,u,v)}{dist(u,v)}$.

We define the *stretch factor* of the scheme $RS$ to be

$$STRETCH(RS) = \max_{u,v \in V} \left\{ \frac{|C(RS,u,v)|}{dist(u,v)} \right\}. \quad (1)$$

*Comment.* $STRETCH(RS)$ is essentially the 'normalized communication complexity' of the routing scheme, i.e., the ratio of the communication cost to the optimal communication cost.

The memory requirement of a protocol is the maximum amount of memory characters used by the protocol in any single processor in the network. We define the *memory requirement* of a routing scheme $RS$, $MEMORY(RS)$, as the maximum between the memory requirements of the delivery protocol and the preprocessing protocols of $RS$.

## 3. OUTLINE OF THE SCHEME

Let us start with an overview of the hierarchical paradigm and the structures it uses in the network. We base our schemes $HS_k$ on a hierarchy consisting of $k+1$ levels, $0 \leq i \leq k$. In each level $i$ we select a subset $P_i$ of the vertices to serve as post centers, or 'pivots'. Each pivot is responsible for getting messages from vertices in its zone and forwarding them. The sets of *pivots* satisfy $P_k \subset P_{k-1} \subset \cdots \subset P_1 \subset P_0 = V$. Further, for every $0 \leq i \leq k$, $|P_i|$ is of size about $n^{(k-i)/k}$. Hence the higher the level, the fewer pivots there are, and the larger the subnetworks (or 'zones') controlled by them.

Each pivot has two types of zones: an *in-zone*, consisting of those vertices which selected it as their 'post-office', and thus will forward their messages to it, and an *out-zone*, consisting of those vertices to which $p$ knows how to forward a message locally. These zones are not necessarily the same; in fact, typically the out-zone of a pivot is much larger (by about $n^{1/k}$ times) than its in-zone. Moreover, the in-zones of the various pivots on a given level are disjoint, while their out-zones largely overlap.

The routing process proceeds as follows. Suppose a vertex $w$ wishes to send a message to a destination vertex $u$. Then $w$ (which is always a pivot at level 0) checks whether it is possible to deliver the message locally to $u$. This succeeds if $u$ happens to be in $w$'s out-zone. Otherwise, $w$ identifies this fact and forwards the message to its pivot on the next level, say $p_1$. This pivot has higher chances of succeeding locally since it controls a larger out-zone. In case the destination $u$ is not in $p_1$'s out-zone too, the message gets forwarded by $p_1$ to its pivot in the next level, and so on. The process may repeat itself until the message reaches a pivot having $u$ in its out-zone. In particular, a pivot in the highest level is guaranteed to succeed, since its out-zone consists of the entire network.

The routing mechanism used for forwarding a message 'upwards' toward a pivot $v$ from an origin in its in-zone is based on an *in-tree* (referred to as $IT(v)$) rooted at $v$ and spanning the appropriate zone. Similarly, the converse routing mechanism (handling messages 'downwards' from the pivot $v$ to a destination in its out-zone) is based on an *out-tree* (referred to as $OT(v)$). This algorithm is naturally more complex than the one used for climbing upward towards the root.

The routing algorithm is described in Figure 1. The organizational structure and the corresponding route from an origin to a destination is portrayed in Figure 5.

```
j ← -1                                    /* level in the hierarchy climbed so far */
do while Origin ≠ Destination and j < k                          /* no success */
  j ← j + 1                                      /* go next level of hierarchy */
  Call DOWNWARD ROUTING (OT_j(Origin))
                                     /* trying to reach Destination on OT_j(x) */
  Call UPWARD ROUTING (IT_{j+1}(Origin))         /* x proceeds to next pivot */
end
```

FIGURE 1. Hierarchical Routing Algorithm

*Comment.* Intuitively, the scheme of [2] operates in 'reverse' direction to that described here; that is the hierarchical organization there is based on pivots associated with the *destinations*, and the basic hierarchical process involves *distributing* messages *downwards* in the pivot chain towards their destinations. In contrast, in the scheme presented here the hierarchical organization of the pivots is based on associating pivots with the *origins* and *collecting* messages from their origins *upward* in the pivot chain.

While the general paradigm is conceptually very simple, some care is needed in designing the pivot selection and assignment, constructing the in- and out-trees and the data structures maintained in them and specifying the search and forwarding algorithms in order to guarantee the adaptive properties and the requirements on the memory and the stretch. The following sections furnish the details of the scheme.

## 4. Technical preliminaries

### 4.1. The concept of neighbourhoods

Our schemes are based on a notion of *neighbourhood* which is defined by *volume* rather than radius (as in [9]). The *neighbourhood* of a vertex $v \in V$ with respect to a specific set of destinations $S \subseteq V$ and a parameter $1 \leq j \leq n$, is a collection of $j$ of the nearest vertices to $v$ from the set $S$. More precisely, order the vertices of $S$ by increasing distance from $v$, breaking ties by increasing vertex names. Hence $x \prec_v y$ if either $dist(x,v) < dist(y,v)$ or $dist(x,v) = dist(y,v)$ and $x < y$. Then $N(v,j,S)$ contains the first $j$ vertices in $S$ according to the order $\prec_v$. When $S = V$ we sometimes omit the third parameter and write simply $N(v,j)$.

The *radius* of the neighbourhood $N(v,j,S)$ is defined as

$$r(v,j,S) = \max_{x \in N(v,j,S)} dist(v,x).$$

The properties we need regarding neighbourhoods are the following.

Lemma 4.1.

1. *For every vertex* $w \in N(v,j,S)$, $dist(v,w) \leq r(v,j,S)$.
2. *For every vertex* $w \in S - N(v,j,S)$, $dist(v,w) \geq r(v,j,S)$.
3. *If* $w \in N(v,j,S)$ *and* $x$ *occurs on some shortest path connecting* $v$ *and* $w$ *then also* $w \in N(x,j,S)$.
4. *For every set* $S \subseteq V$, *vertices* $u,w \in V$ *and integer* $1 \leq j \leq n$, $r(u,j,S) \leq r(w,j,S) + dist(u,w)$.

Proof of 3. By contradiction. Assume that $w \notin N(x,j,S)$. We claim that for every $z \in N(x,j,S)$, also $z \in N(v,j,S)$. In order to prove this it suffices to show that every $z \in N(x,j,S)$ satisfies $Z \prec_v w$, since $w$ is included in $N(v,j,S)$.

Consider some $z \in N(x,j,S)$. By the triangle inequality $dist(v,z) \leq dist(v,x) + dist(x,z)$. By Lemma 4.1(1), $dist(x,z) \leq dist(x,w)$. Since $x$ is on a shortest path from $v$ to $w$, $dist(v,w) = dist(v,x) + dist(x,w)$. Put together, $dist(v,z) \leq dist(v,w)$.

There are two cases to consider. If $dist(v,z) < dist(v,w)$ then the claim is immediate. Now suppose $dist(v,z) = dist(v,w)$. Then necessarily $dist(x,z) = dist(x,w)$ too. Since $w \notin N(x,j,S)$ and $z \in N(x,j,S)$, by definition $z \prec_x w$, so $z < w$. Therefore also $z \prec_v w$.

It follows from our claim that $N(x,j,S) \subseteq N(v,j,S)$ and since both are of size $j$, $N(x,j,S) = N(v,j,S)$. But $w \in N(v,j,S) - N(x,j,S)$; a contradiction. □

PROOF OF 4. By contradiction, relying on the fact that $|N(u,j,S)| = |N(w,j,S)| = j$. □

*4.2. Covers*

The pivot selection process has to guarantee that pivots are well-distributed and properly 'cover' the neighbourhoods in the network. We now give some basic facts concerning the concept of covers.

Consider a collection $\mathcal{H}$ of subsets of size $s$ of a set $B$, so that each element of $B$ appears in at least one set. A set of elements $M \subseteq B$ is said to *cover* those sets of $\mathcal{H}$ that contain at least one element of $M$. A *cover* of $\mathcal{H}$ is an $M \subseteq B$ covering all sets in $\mathcal{H}$. A *fractional cover* for $\mathcal{H}$ is a system of nonnegative real weights $\{t_v | v \in B\}$ such that $\Sigma_{x \in S} t_x \geq 1$ for every set $S \in \mathcal{H}$. Let $\tau^* = \min \Sigma_{x \in B} t_x$ where the minimum is taken over all fractional covers.

Since the weight system assigning $t_v = 1/s$ to every vertex $v$ is a fractional cover, we get

$$\tau^* \leq |B|/s.$$

Consider the following two simple procedures for creating a cover for $\mathcal{H}$.

**Algorithm 4.2 (Greedy Cover).** Start with $M = \varnothing$ and iteratively add an element that increases the number of sets covered by $M$ as much as possible. Stop when $M$ becomes a cover. This $M$ is called a *greedy cover* for $\mathcal{H}$.

For the construction of our routing schemes we rely on the following Lemma of Lovász [7].

LEMMA 4.3. *Let $M$ be a greedy cover for $\mathcal{H}$. Then*

$$|M| < (\log|\mathcal{H}| + 1)\tau^* \leq \frac{(\log|\mathcal{H}| + 1)|B|}{s}.$$

**Algorithm 4.4 (Randomized Cover).** Randomly select each element $v \in B$ into the set $M$ with probability $\frac{c \ln|\mathcal{H}|}{s}$, where $c > 1$ is a constant.

LEMMA 4.5. *Let $M$ be a set constructed by the randomized algorithm above. Then, with probability $1 - O(|\mathcal{H}|^{1-c})$, $M$ is a cover for $\mathcal{H}$ and $|M| \leq \frac{2c|B|\ln|\mathcal{H}|}{s}$.*

## 5. Details of the scheme

### 5.1. Pivot selection

The pivots are constructed as follows. Let $P_0 = V$, i.e. the set $P_0$ of pivots of level 0 is simply the set of all the vertices. Fix $m = n^{1/k}$. The pivots of level $P_{i+1}$, for $0 \leq i \leq k-1$, are selected so as to have the covering property with respect to the neighbourhoods $N(v,m,P_i)$ of pivots $v \in P_i$. This process is shown in Figure 2.

```
P_0 ← V                                     /* first pivot level */
For i = 0 to k − 1 do                       /* for all pivots levels */
  𝒳 ← {N(v,m,P_i) | v ∈ P_i}               /* new collection of sets */
  P_{i+1} ← COVER (𝒳)        /* i + 1'st level of pivots; P_{i+1} ⊆ P_i */
od;
```

FIGURE 2. Construction of pivot sets

A vertex $v$ is a *j-pivot* if it belongs to $P_j$ (hence also to $P_i$ for $0 \leq i < j$) but not to $P_{j+1}$ (hence also not to $P_i$ for $j+1 < i \leq k$). With every vertex $v$ we associate a unique pivot $p_i(v)$, referred to as the *i-post* of $v$, in every level $0 \leq i \leq k$. This $i$-post is selected as follows. First, $p_0(v) = v$ is set for every $v \in V$. Suppose that $v$ is a $j$-pivot. For $0 \leq i \leq j+1$, $p_i(v)$ is taken to be the smallest pivot in $P_i$ according to $\prec_v$. For $j+2 \leq i \leq k$ we define $p_i(v)$ recursively by setting $p_i(v) = p_i(p_{i-1}(v))$.

Observe that the above construction guarantees that for $0 \leq i \leq j$, $p_i(v) = v$, and that $p_{j+1}(v) \in N(v,m,P_j)$.

### 5.2. In-trees and upward routing

For each level $0 \leq i \leq k$ and for each pivot $v \in P_i$, let the *in-zone* of a pivot $v \in P_i$ on level $i$, $IZ_i(v)$, be the collection of vertices $u$ which chose $v$ as their $i$-post (i.e., such that $p_i(u) = v$). Denote by $IT_i(v)$ a shortest path tree in $G$ connecting $v$ to the vertices in $IZ_i(v)$. Each vertex in $IT_i(v)$ maintains a pointer upwards to its parent in the tree. This makes routing towards the root trivial.

Note that, a-priori, this tree might contain also vertices not in $IZ_i(v)$. However, in order to analyze the memory requirements of the scheme we need to prove that the in-trees of the various pivots in a given level $P_i$ are all disjoint, hence each covers precisely its zone. This is done in the following technical lemma.

LEMMA 5.1. *For every $0 \leq i \leq k$ and for every $u_1, u_2 \in P_i$, $IT_i(u_1)$ and $IT_i(u_2)$ are vertex-disjoint.*

PROOF. By contradiction. Assume that $IT_i(u_1)$ and $IT_i(u_2)$ intersect in some vertex $x$ for some $u_1, u_2 \in P_i$. Then for $j = 1,2$ there is a vertex $w_j \in IZ_i(u_j)$ such that $x$ occurs on a shortest path connecting $w_j$ to $u_j$. Without loss of general-

ity, $x$ is the first intersection of the paths, tracing the paths from the endpoints $u_1$ and $u_2$. See Figure 3.

By the rules of the construction, since $p_i(w_1)$ was set to $u_1$, necessarily $u_1 \prec_{w_1} u_2$. (Recall that $a \prec_c b$ if either $dist(a,c) < dist(b,c)$ or $dist(a,c) = dist(b,c)$ and $a < b$). Since $dist(u_1,w_1) = dist(u_1,x) + dist(x,w_1)$ and $dist(u_2,w_1) \leq dist(u_2,x) + dist(x,w_1)$ it follows that $u_1 \prec_{w_2} u_2$. Since $dist(u_2,w_2) = dist(u_2,x) + dist(x,w_2)$ and $dist(u_1,w_2) \leq dist(u_1,x) + dist(x,w_2)$, it follows that $u_1 \prec_{w_2} u_2$. Hence $u_1$ have been chosen as $p_i(w_2)$; a contradiction. □

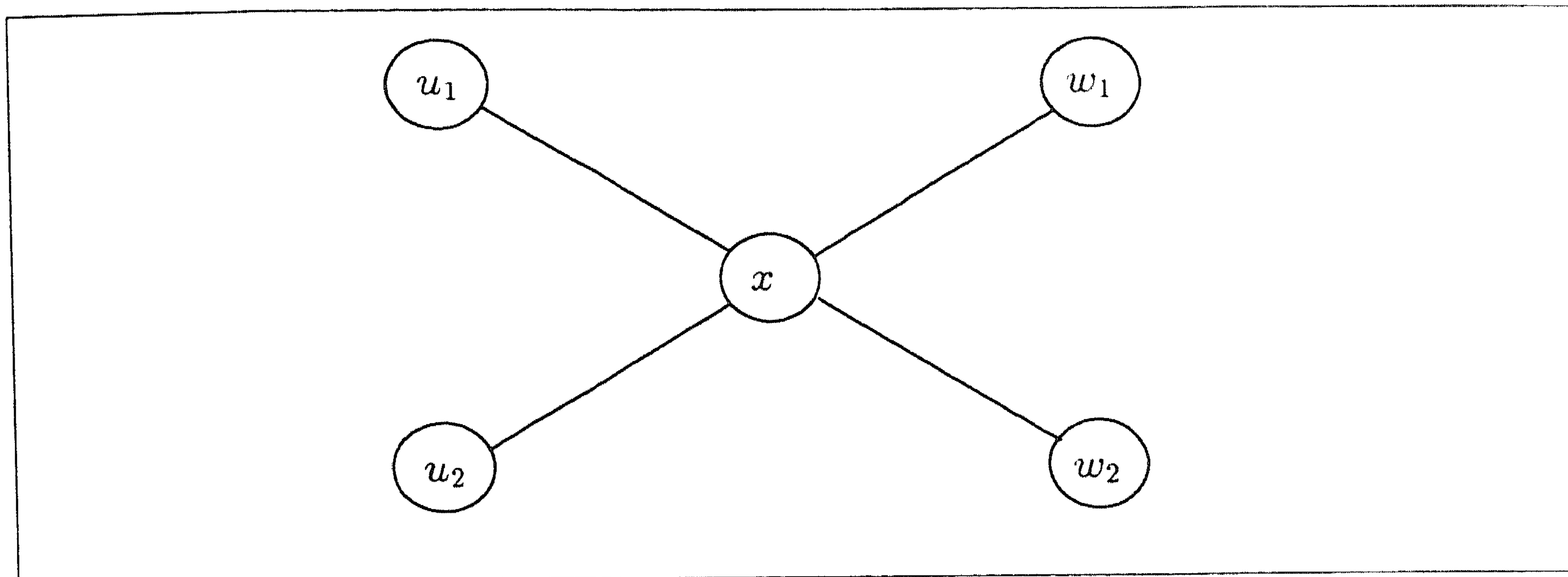


FIGURE 3. Node $x$ belongs to both $IT_i(u_1)$ and $IT_i(u_2)$

*Complexity.* Routing a message from a vertex $u \in IZ_i(v)$ to its $i$-post $v$ is done on the shortest path between them, so its communication complexity is $dist(u,v)$. By the disjointness of the in-trees (Lemma 5.1) the amount of memory stored in each vertex for this component is $O(1)$ characters per level, or $O(k)$ characters overall.

### 5.3. *Out-trees*

Define the *out-zone* of a pivot $v \in P_i$ on level $i$, $OZ_i(v)$, as follows: For $0 \leq i \leq k-1$, let

$$OZ_i(v) = \{u \mid v \in N(u,m,P_i),\ dist(u,v) \leq \tfrac{1}{2} dist(v,p_{i+1}(v))\}. \tag{2}$$

For $i = k$ let $OZ_k(v) = V$.

The apparently counter-intuitive restriction on the depth of the out-trees is crucial for the analysis of the resulting stretch in Lemma 6.2. The idea is to bound the cost of an *unsuccessful* search on an out-tree.

Denote by $OT_i(v)$ the shortest path tree in $G$ connecting $v$ to the vertices in $OZ_i(v)$. Again, a-priori this tree might contain also vertices not in $OZ_i(v)$. However, as an immediate application of Lemma 4.1(3) we get that this is not the case.

LEMMA 5.2. *For every* $0 \leq i \leq k$ *and* $v \in P_i$, *the vertex set of* $OT_i(v)$ *is exactly* $OZ_i(v)$.

COROLLARY 5.3. *For every $0 \leqslant i \leqslant k$ and $u \in V$, the number of different out-trees on the i-th level in which u participates is at most m.*

PROOF. The vertex $u$ might occur in at most $m$ different out-zones $OZ_i(v)$. □

The routing procedure from the root downwards on the $OT$ trees is quite complex, and the rest of the section is dedicated to describing it.

*5.4. Basic downward-routing*

In this sub-section we consider a simplified version of the downward routing problem, in which we are allowed to assign specially selected names to nodes. This is a basic subtask in many existing routing strategies.

*Problem statement.* Given is a directed tree $T=(V',E')$ rooted at a vertex $r \in V'$ (typically some special spanning tree of some cluster $V'$ in the network) and an instance of the routing problem, where the origin is the root, and the destination is not necessarily in the tree. The names of nodes in the tree can be chosen by the routing scheme. If the destination is not in the tree then the message is to return to the root.

This subproblem was treated in previous papers using a simple scheme called the *interval routing scheme* and denoted $ITR(G',r)$ or $ITR(V',r)$ [1,2,9,10]. Using it in the adaptive setting poses some new technical problems, whose solutions are the subject of this section.

Let us first supply some definitions. The *depth* of a node $v$ in the tree $T$, denoted $depth_T(v)$, is the weighted distance $dist_T(r,v)$ from the root to $v$. The depth of the tree $T$, $depth(T)$, is the maximum depth of a node in $T$.

We now give an overview of (a variant of) the ITR scheme. It is constructed as follows.

1. Assign the vertices $v \in V'$ a DFS (pre-order) numbering $DFS(v)$ according to $T$.
2. Store at a node $u$ its DFS number and the DFS numbers $DFS(w)$ of each of its children $w$ in the tree.

Routing a message from the root $r$ to a vertex $v \in V'$ (assuming that $r$ knows the value $DFS(v)$) involves propagating the message from each intermediate vertex $u$ with children $w_1,...,w_k$ (ordered by increasing DFS numbering) to the child $w_i$ such that $DFS(w_i) \leqslant DFS(v) < DFS(w_{i+1})$ (setting $DFS(w_{k+1})$ to $\infty$).

*Complexity.* Consider the complexity of the $ITR$ scheme on a tree of size $t$. The amount of memory stored in a node $v$ is $O(deg_T(v))$ characters, since a node needs to maintain a data item for each of its outgoing edges, in order to identify the intervals. As for communication overhead, consider separately two cases. If the destination is outside the tree then the cost is at most $2 \cdot depth(T)$, since we traverse a path from the root to one of the leaves and back. If the destination $v$ is in the tree then we traverse the path from the root to $v$, and

the cost is $depth_T(v)$. (Observe that if $T$ is an out-tree $OT(r)$ in the graph $G$, the resulting path is of length $dist(r,v)$, since out-trees are shortest-path trees.)

The ITR scheme has two major problems in an adaptive setting.

- The scheme requires using special routing labels; in order for the root to forward a message to a destination in the tree it needs to know its DFS number. This interferes with the *name-independence* requirement.
- The memory stored at a node depends (linearly) on its degree, and thus may be as high as $\Omega(n)$ in the worst case. This interferes with the *balanced-memory* requirement.

The two problems are tackled as follows. In order to avoid the need to know the DFS labels in advance by the origins, the scheme uses a distributed data structure enabling the root to retrieve the DFS label of a node using its original name as a key. The main technical difficulty stems from the need to guarantee that if the destination occurs in the tree, we pay for the search no more than the distance from the root to the destination. This prohibits a solution spreading the data arbitrarily over the tree, since then the search for a nearby destination may cost as much as the depth of the tree.

The second problem is handled by arguing that one can embed into any tree a tree of 'small' degrees, without paying too high a price in memory and without increasing the depth of the tree too much.

The next three subsections develop the necessary tools for solving these two problems and present the combined solution.

### 5.5. *Tree dictionary search*

*Problem statement.* Consider a function $F: X \rightarrow Y$ on some ordered domain $X=\{x_1 < \cdots < x_{|X|}\}$. That is, $F$ is a list of $|X|$ pairs $F=\{(x_1,y_1),\ldots(x_{|X|}, y_{|X|})\}$. (For $x \notin X$ we set $F(x)=$ *undefined*.) Also, consider a rooted tree $T$ of size $t$ with root $r$. Our task is to store any function $F$ on any tree $T$, as above, and be able to support searches from the root. That is, if $r$ is given an arbitrary argument $x$ (not necessarily in $X$) then the search returns $F(x)$ (including *undefined*, whenever $x \notin X$).

The problem is solved using a distributed data structure called the *Tree Dictionary*. Define the *load* of the function as $L=\lceil |X|/n \rceil$. Let $DFS(v)$ denote DFS number of a node $v$ in the tree $T$. Each node is required to store $L$ values of the function $F$, by increasing DFS order. That is, the node $u$ with $DFS(u)=j$ stores the pairs $(x_i,y_i) \in F$ for $(j-1)L+1 \leqslant i \leqslant jL$. Also, define $Lowest(v)$ for each node $v$ as the minimal $x$ whose value is stored at $v$ (i.e., if $DFS(v)=j$ then $Lowest(v)=x_{(j-1)L+1}$). The variable $Lowest(u)$ is maintained at the node $u$ as well as at its parent in the tree.

The process of computing value $F(x)$ of an argument $x$ is almost identical to the one used in the DFS routing scheme. Namely, the search message is propagated from each intermediate vertex $u$ with children $w_1,\ldots,w_k$ (ordered by increasing DFS numbering) to the child $w_i$ such that $Lowest(w_i) \leqslant x < Lowest(w_{i+1})$ (setting $Lowest(w_{k+1})=\infty$).

*Complexity.* Tree dictionary search requires $O(L + deg_T(v))$ characters at a node $v$ and $O(depth(T))$ communication cost.

*5.6. Stratified tree dictionary*

Now, we combine the ITR scheme with a tree dictionary to obtain a downward routing scheme for trees.

*Problem statement.* Same as in Subsection 5.4, except that the names of nodes in the tree are *given,* and not chosen by the scheme.

The first step is to assign a DFS numbering to the tree $T$, and construct the DFS routing schemes $ITR$ for $T$. Thus, for all $u \in T$, once $DFS(u)$ is known, $u$ can be reached. Ultimately, we would like to apply this $ITR$ scheme to the tree $T$. As mentioned before, we need to overcome the problem of storing (and retrieving) the DFS numbers required for using the scheme. Note that we cannot use a single tree dictionary for storing the DFS labels. This is because, as said earlier, we might waste too much on searching a nearby destination. We thus need to stratify the stored data and store it in a succession of larger and larger trees.

For any $m^q > t > 0$, and any tree $T$ of size $t$, consider the $q$ subtrees $T_i$, $0 \leq i \leq q$, so that $T_i$ contains the $m^i$ nodes closest to the root ($T_q = T$). Observe that $T_0 = \{r\}$ and $T_i \subset T_{i+1}$ and that if $v \notin T_i$ then $depth_T(v) \geq depth(T_i)$.

The algorithm itself proceeds as follows. We construct several Tree Dictionary Search schemes $TDS_i$ one for each tree $T_i$. Each of these schemes stores a portion of the DFS function. Specifically, the scheme $TDS_i$ stores the subset of $DFS$ over the domain $X_i = \{v | v \in T_i\}$ at the nodes of the tree $T_{i-1}$. Put another way, if $u \in T_i$ then the value $DFS(u)$ will be stored on the tree $T_{i-1}$. Now, we simply go through the whole hierarchy, until we find $DFS(u)$ or declare that $u$ is not in the tree $T$. The algorithm is formally presented in Figure 4.

```
i ← −1                                             /* level in the hierarchy */
DFS_number ← undefined       /* DFS number of destination unknown */
do while DFS_number = undefined and i ≤ k
                          /* until Destination found or all levels exhausted */
  i ← i + 1                          /* go to the next level of the hierarchy */
  DFS_number ← DICTIONARY TREE SEARCH(T_i(Origin))
                                 /* search for Destination address in T_i */
end                                                /* end of search for u */
if DFS_number ≠ undefined then          /* DFS number of Destination has
                                                              been found */
   Call ITR (T_{i+1}, DFS_number)
                  /* use ITR routing to deliver message to the destination */
```

FIGURE 4. Downward Routing Algorithm

*Complexity*. Here $L = \frac{|T_i|}{|T_{i-1}|} \leqslant n^{1/k}$. Thus the total memory requirements of maintaining tree $T_i$ at a node $v$ are $O(deg_{T_i}(v) + n^{1/k})$ characters, and overall we need $O(k(n^{1/k} + deg_T(v)))$ characters at a node $v$.

As for the routes obtained by this algorithm, we may end up searching through each of the $k$ trees. More specifically, suppose first that the destination $u$ is in $T$, hence $u \in T_i$ but $u \notin T_{i-1}$ for some $0 \leqslant i \leqslant k$. In that case we pay $\Sigma_{0 \leqslant j < i} depth(T_j)$ for the searches plus $depth_T(u)$ for the path on the ITR scheme, and overall at most $(k+1) depth_T(u)$. Now suppose $u \notin T$. Then we pay $\Sigma_{0 \leqslant j \leqslant k} depth(T_j) \leqslant (k+1) depth(T)$ for the searches.

*5.7. Controlling tree degrees*

We now get rid of the dependency on the degrees using the following theorem.

THEOREM 5.4. *For any rooted tree T with maximal degree d, there exists an embedded tree S on the same set of nodes and with the same root, but with a different set of edges, so that*

1. *the maximal degree of S is* $2n^{1/k}$;
2. *an edge of S is a path of length at most two in T*;
3. $depth_S(v) \leqslant (2k-1) depth_T(v)$ *for every node v*.

PROOF. Denote $m = n^{1/k}$. Consider a node $u_0$ in the tree $T$ with degree $d_0$ and children $u_1, \ldots, u_{d_0}$. Here, we assume that the children are ordered in increasing order of depth.

For every $m \leqslant a \leqslant d_0$, make $u_b$ the parent of $u_a$ in the tree $S$ where $b = \lfloor a/m \rfloor - 1$.

Every node in the new tree $S$ will have at most $2m$ children, including up to $m$ of its closest children and up to $m$ of its siblings in the original tree $T$.

It is easy to verify that requirements (2) and (3) are guaranteed as well. □

We construct for the given tree $T$ an embedded tree $S$ as in the theorem, and apply the downward routing algorithm to the embedded tree $S$, which is of maximal degree $O(n^{1/k})$.

Sending a message over an edge $(x,y)$ of the simulated tree $S$ requires the origin $x$ to specify the path represented by this edge in the real tree $T$. In practical terms, though, this contributes only a constant increase in the header size of messages, by property (2) of the theorem.

*Complexity*. We finally consider the complexity of the resulting downward routing algorithm. By property (1) of the theorem, the memory requirements of the tree dictionary and the ITR scheme for $S$ are $O(k \cdot n^{1/k})$ characters per vertex (noting that the degrees of the subtrees $S_i$ are bounded by $O(n^{1/k})$ as well).

The length of the resulting path is stretched by $O(k)$ by property (3) of the

theorem, hence we now pay up to $O(k^2 \cdot depth_T(u))$ for $u \in T$ and $O(k^2 \cdot depth(T))$ for $u \notin T$.

## 6. Complexity analysis

Our final task is to analyze the overall complexity of the hierarchical scheme $HS_k$.

### 6.1. *Memory requirements*

In terms of memory we need to sum, for each node, the memory requirements of maintaining all the in- and out-trees in which it participates, on all $k+1$ levels of the hierarchy.

Lemma 6.1. *The memory requirements of a vertex in $HS_k$ are $O(kn^{2/k})$ characters.*

Proof. By the discussion of Subsection 5.2 each vertex $v$ participates in at most one in-tree component in each level, so the memory it uses for the in-trees in all levels is at most $O(k)$ characters. In addition, by Corollary 5.3 $v$ participates in up to $m$ out-trees at each level but the last. This involves $O(kn^{2/k})$ characters of memory. For the last level the cost is $O(k|P_k|n^{1/k})$ characters. By Lemma 4.3 $|P_i| \leq |P_{i-1}| \log |P_{i-1}|/m$, so since $\log n = o(n^{1/k^2})$ for sufficiently large $n$, $|P_k| \leq m$. It follows that $O(kn^{2/k})$ characters are used by each vertex for downward routing on the out-trees. Altogether, $MEMORY(HS_k) = O(kn^{2/k})$. □

### 6.2. *Stretch factor*

The crux of our analysis is in estimating the length of the combined route traversed by a message according to the scheme (including the cost of dictionary searches) and thus bound the total stretch. Here is where we make use of the particular choice of pivots, the properties of covers and the definition of out-zones.

Lemma 6.2. *For $0 \leq i \leq k-1$, if $u \notin OZ_i(p_i(w))$ then*

1. $dist(p_{i+1}(w), p_i(w)) \leq 2dist(u, p_i(w))$ *and*
2. $dist(u, p_{i+1}(w)) \leq 3dist(u, p_i(w))$.

Proof. Let $y = p_i(w)$ and $z = p_{i+1}(w)$ (see Figure 5), and let $P = P_i$. There are two cases to consider in proving the first claim. If $y \in N(u,m,P)$ then the claim follows straightforwardly from the definition of out-trees. So suppose $y \notin N(u,m,P)$. By Lemma 4.1(1,2) this implies $r(u,m,P) \leq dist(u,y)$. Since $z \in N(y,m,P)$, see Subsection 5.1, $r(y,m,P) \geq dist(y,z)$. By Lemma 4.1(4), $r(y,m,P) \leq r(u,m,P) + dist(u,y)$. Put together, we get that $dist(y,z) \leq 2dist(u,y)$, which completes the proof of the first claim.

The second claim follows since by the triangle inequality, $dist(u,z) \leq dist(u,y) + dist(y,z)$. □

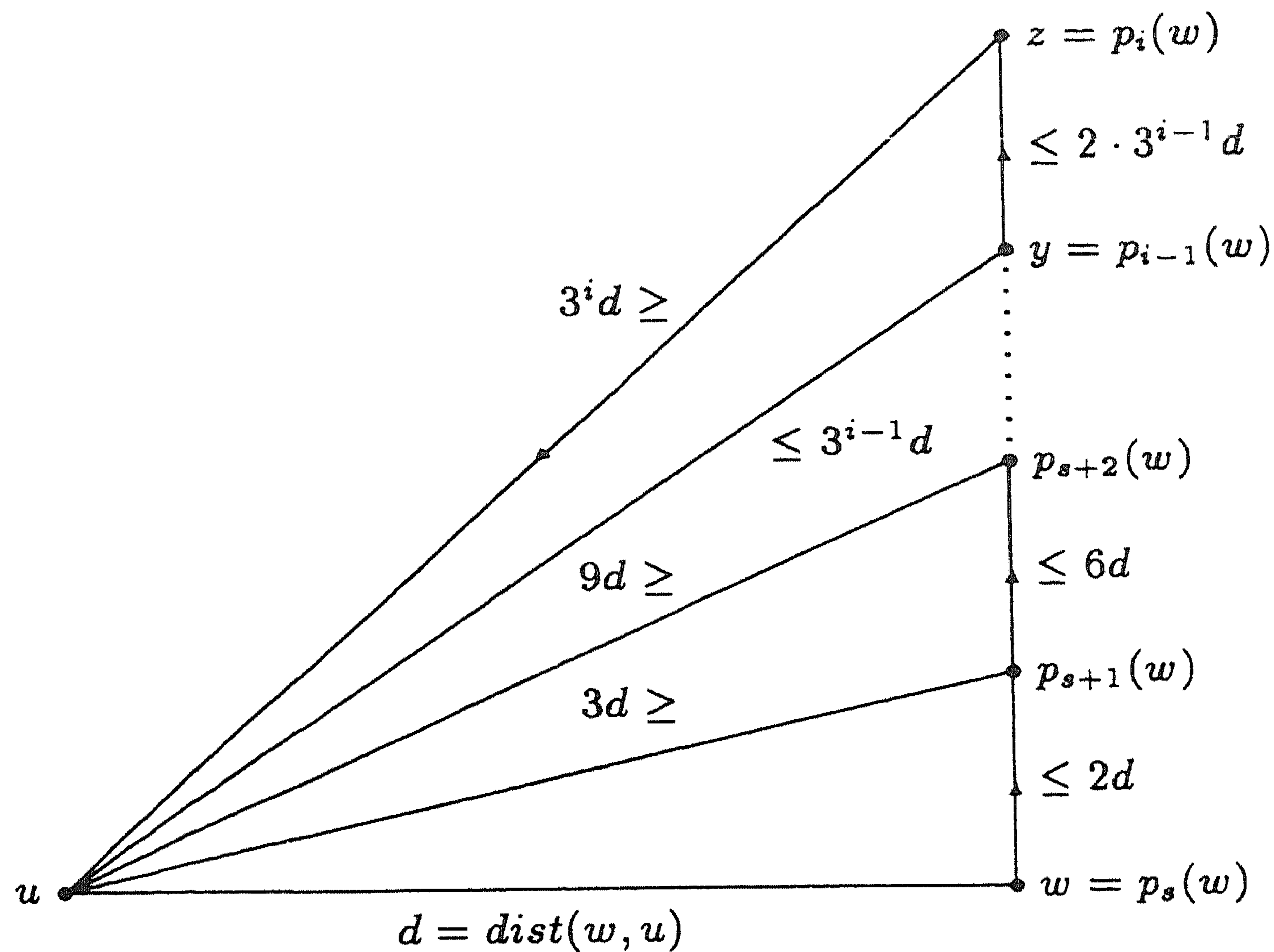


FIGURE 5. Schematic representation of the routing process from the origin $w$ to the destination $u$. The message is forwarded to the first pivot $p_i(w)$ that is capable of reaching $u$. Observe that the distance from $p_i(w)$ to $u$ does not grow too fast.

As a result, we get the global distance relationships depicted in Figure 5 and summarized in the following Corollary.

COROLLARY 6.3. *For every* $0 \leqslant j < i$ *and* $u \notin OZ_j(p_j(w))$

1. $dist(p_{i-1}(w), p_i(w)) \leqslant 2 \cdot 3^{i-1} \cdot dist(u,w)$ *and*
2. $dist(u, p_i(w)) \leqslant 3^i \cdot dist(u,w)$.

LEMMA 6.4. *If* $u \in OZ_i(p_i(w))$ *and* $u \notin OZ_j(p_j(w))$ *for every* $0 \leqslant j < i$ *then the cost of routing a message from* $w$ *to* $u$ *using* $HS_k$ *satisfies* $C(HS_k, w, u) \leqslant k^2(2 \cdot 3^i - 1) dist(w,u)$.

PROOF. The route constructed by the scheme consists of $i+1$ segments. Each of the first $i$ segments starts with a local search for $u$ using the tree dictionary, followed by a climb up the in-tree $IT_j(p_j(w))$ to the next pivot. The cost of the search on level $j$ is $O(k^2 depth(OT_j(p_j(w))))$. By the construction of out-trees, $depth(OT_j(p_j(w))) \leqslant 1/2 dist(p_j(w), p_{j+1}(w))$. The movement from the pivot $p_j(w)$ to $p_{j+1}(w)$ is done (using the in-trees) along shortest paths, and its length is therefore exactly $dist(p_j(w), p_{j+1}(w))$. Thus the entire spending in the $j$-th level is $O(k^2 \cdot dist(p_j(w), p_{j+1}(w)))$. On the $i$-th level itself we forward the message to $u$ by downward routing (i.e., successful tree dictionary search followed by $ITR$

routing) paying, by similar arguments, an overall of $O(k^2 \cdot depth_{OT_i(p_i(w))}(u)) = O(k^2 \cdot dist(p_i(w),u))$.

Overall we get that

$$C(HS_k,w,u) \leq k^2 \Big( \sum_{1 \leq j \leq i} dist(p_{j-1}(w),p_j(w)) + dist(p_i(w),u) \Big)$$

$$\leq k^2 \Big(3^i + 2 \sum_{1 \leq j \leq i} 3^{j-1}\Big) dist(u,w) \leq k^2(2 \cdot 3^i - 1) dist(u,w). \qquad \square$$

COROLLARY 6.5. *The stretch factor of the routing scheme $HS_k$ satisfies*

$$STRETCH(HS_k) = O(k^2 3^k).$$

THEOREM 6.6. *For every n-vertex network G and for every $k \geq 1$ it is possible to construct a hierarchical balanced scheme $HS_k$ with*

1. $MEMORY(HS_k) = O(kn^{2/k})$, *and*
2. $STRETCH(HS_k, G) = O(k^2 3^k)$.

## 7. Distributed preprocessing with bounded space

### *7.1. Overview*

In this section we describe a distributed preprocessing algorithm that initializes the schemes $HS_k$. We present a Monte-Carlo version of this algorithm, i.e., a version in which the algorithm may make a mistake with small probability. It is easy to modify our algorithm to be a Las-Vegas algorithm, by detecting the fact that there is no pivot in a certain neighbourhood; in this case the algorithm is restarted. Its space overhead is $O(kn^{1/k} + k^2)$ characters.

Our algorithm employs the following four procedures.

1. *Initialization* procedure: appoints all the pivots and triggers all other procedures.
2. *Forwarding-Construction* procedure: constructs the parent pointers of the forwarding schemes and the zones.
3. *DFS-Numbering* procedure: simultaneously operates on the spanning trees of all the zones, assigning DFS numbering to the nodes of the tree for the *ITR* schemes, as well as constructing children pointers in the zones. Upon termination of this procedure, the *Post-Update* procedure below is executed.
4. *Post-Update* procedure: specifies for each vertex its posts.

Each of those procedures has local variables, and output variables. The local variables are recognized only inside the procedure, while the output of the procedure is written in output variables. Initially, all the variables are initialized to *nil* or 0 as appropriate.

*7.2. Initialization procedure*

*7.2.1. Outline.* The procedure first randomly selects the pivots for each level, and then invokes the other procedures. At each node, procedure outputs variables $p_i$, so that $p_i = self$ if the node is chosen as a pivot of level $i$ and $p_i = nil$. The selection process is very simple. Every vertex is a pivot at level 0. Any pivot at level $i-1$ becomes a pivot at level $i$ with probability $c \log n/m$; here $c > 1$ is a constant. This selection rule guarantees that, on average, $|P_i| = |P_{i-1}| \log n/m$, and, moreover, there exists a pivot in every neighbourhood $N(v,m,P_{i-1})$ with probability $1 - O(n^{1-c})$. Observe that selection of pivots as above requires no communication.

Upon termination of selection process, each node triggers the *Forwarding-Construction* procedure, and, after 1 clock pulse, it triggers *DFS-Numbering* procedure.

See formal presentation in Appendix, section A.1.

*7.3. Forwarding-Construction procedure*

*7.3.1. Basic strategy.* Our purpose is to construct a forwarding routing scheme $FR(P_{i-1},m)$. The essence of this construction is specifying, for each node $v$, pointers $FE(w)$, pointing to neighbours of the node, for nodes in $N(v,m,P_{i-1})$, namely $m$ pivots of level $i-1$, which are closest according to the $\prec_v$ ordering. Additional task of this procedure is to construct parent pointers $Parent_i$ for the zones of level $i$.

The algorithm below takes advantage of the fact that network is synchronous and that delays are very closely related to costs of the edges. Thus, delay of a message sent over a communication path $p$ is 'essentially' the cost $cost(p)$ of that path; to be more precise it varies in between $cost(p)$ and $cost(p-1)$.

The node $v$ will keep a list $List_i$, which will contain pairs $(u,w)$, such that the node $v$ thinks that $u \in N(v,m,P_{i-1})$, and $w = FE_i(u)$. Also, it will keep a variable $Parent_i$, which is initialized to $nil$.

Each node $u$, which is a pivot of level $i-1$, will initiate a broadcast process, which propagates message $FORWARD_i(u)$ through the whole network. Once a node $v$ which receives such message $FORWARD_i(u)$ from $w$, it checks whether $Parent_i = nil$; if so it sets $Parent_i := w$. Also, it checks whether $List_i$ contains already a pair $(u,\bar{w})$. If no such pair exists, it means that message from $u$ has been received for the first time. Then, the node $v$ adds pair $(u,w)$ to $List_i$, and propagates this message to all neighbours.

In fact, node $v$ will propagate only $FORWARD_i(u)$ messages for nodes $u \in N(v,m,P_{i-1})$, and discards messages of nodes outside of $N(v,m,P_{i-1})$. Formally, for nodes $u \in N(v,m,P_{i-1})$, we define $t_i(v,u)$ be the time at which node $v$ receives $FORWARD_i(u)$ message for the first time. It is defined as $\infty$ if such message never arrives.

Under the strategy described above, for all nodes $u \in N(v,m,P_{i-1})$, $\rceil t_i(v,u) \lceil = dist(v,u)$. This fact is proved by induction on the length of a shor-

test path from $v$ to $u$, using Lemma 4.1(3) as induction step (the base of induction is trivial).

The strategy described above makes an assumption that upon receipt of $FORWARD_i(u)$ message, node $v$ can determine whether $u \in N(v,m,P_{i-1})$. This is not the case. Below, we describe the modification in the algorithm needed to get around this difficulty.

*7.3.2. Technical details.* Denote $r = r(N(v,m,P_{i-1}))$ (radius of $N(v,m,P_{i-1})$). Let us divide the set $P_{i-1}$ into the following four classes:

1) Class $A$ contains all elements $u \in P_{i-1}$ such that $dist(u,v) < r$.
2) Class $B$ containing the smallest $m - |A|$ elements $u \in P_{i-1}$ such that $dist(u,v) = r$. This class is empty if $|A| = m$.
3) Class $C$ containing the all elements $u \in P_{i-1}$ such that $dist(u,v) = r$, which are not in class $C$.
4) Class $D$ contains all elements $u \in P_{i-1}$ such that $dist(u,v) > r$.

Clearly, $N(v,m,P_{i-1})$ is union of classes $A$ and $B$.

The invariant of the algorithm is that by pulse $q$, $List_i$ contains all pairs $(u,w)$ such that for all nodes $u \in N(v,m,P_{i-1})$, with $dist(u,v) \leqslant q$. Under that invariant, observe that:

1) At clock pulse $r-1$, $List_i$ contains all elements corresponding to class $A$ and no other elements. Thus, $|List_i| = |A| < m$ at this pulse.
2) Let $S$ be set of nodes $u$ such that in between clock pulse $r$, and clock pulse $m$, node $v$ receives for the first time $FORWARDING_i(u)$ message. Then, $S$ is union of classes $B$ and $C$. Also, nodes in class $B$ are $m - |A|$ smallest node names among nodes in set $S$, and nodes in class $C$ whose names the node heard in between pulses $r-1$ and $r$ are the remaining nodes in class $S$.
3) At clock pulse $r$, $List_i$ contains all elements corresponding to class $A$ and $B$ and no other elements. Thus, $|List_i| = |A| + |B| = m$ at this pulse.

This enables to recognize $FORWARD_i(u)$ messages of nodes $u$ in class $D$, since upon their first arrival $|List_i| = m$. Node $v$ can recognize $FORWARD_i(u)$ messages of nodes $u$ in class $A$ as upon their first arrival $|List_i| < m$. The only problem is to figure out how to distinguish between nodes in classes $B$ and $C$, since their *FORWARD* message arrive (for the first time) in between pulses $r-1$ and $r$ in *arbitrary* order, i.e. messages of nodes from $B$ *do not* necessarily enter before nodes in $C$.

What we need to do during this time interval is to delete dynamically nodes from $C$, as we insert nodes from $B$. Towards that goal, each node maintains a list $New_i$, which contains all new entries that node has heard since last pulse, which are 'candidates' to enter into $List_i$. Upon each clock pulse, the $New_i$ list is added to $List_i$ list, and then emptied.

In general, whenever a node receives $FORWARD_i(u)$ message from a neighbour $w$, it checks whether either $List_i$ *or* $New_i$ contains a pair $(u,\overline{w})$, or $|List_i| = m$. If either one of those is true, the message is discarded. Else, the node inserts the pair $(u,w)$ into $New_i$ list. Next, the node checks whether $|New_i| + |List_i| > m$. If this is indeed the case, then it means that

$|New_i|+|List_i|=m+1$ as a result of the last addition into $New_i$. Then, the node deduces that it is now in between pulses $r$ and $r-1$, and thus one of the entries in $New_i$ corresponds to a node in class $C$, which should be purged. Clearly, this entry is the pair $(\overline{u},\overline{w})$ corresponding to maximal name $\overline{u}$ among all entries in $New_i$. Thus, this entry is deleted from $New_i$, thus reducing the sum of sizes of $New_i$ and $List_i$ back to $m$.

Even though the number $r$ is not known to node $v$ it can easily identify clock pulse $r$ as the first clock pulse such that by that time $|List_i|=m$ at $v$.

See formal presentation in Appendix, section A.2.

*7.4. DFS-numbering and zone-construction*

In this section we set up (for each level $i$) the level $i$ zones, and the $ITR_i$ schemes in each zone. Recall that zones have been determined by $Parent_i$ pointers computed in *Forwarding* $-$ *Construction* procedure of the previous subsection. This procedure will, in addition, compute the children pointers in each zone, which are of course uniquely determined from *Parent* pointers.

For each pivot $q$ of level $i$, we consider the tree $T_i(q)$, which defines the zone $Z_i(q)$ of level $i$ around $i$.

We assign a DFS (pre-order) number to each vertex $v$, and maintain counters $Lowest_i$ and $Highest_i$ as the lowest and highest pre-order DFS numbers in the sub-tree rooted at $v$. The interval $int(v)=(Lowest_i,Highest_i)$ contains the names of all nodes in $v$'s sub-tree of level $i$. (Recall that $Lowest_i$ is the DFS number of $v$ itself.) Also, for each child $u$ of $v$, we maintain parameters $(Lowest_i(u),Highest_i(u))$, where $Highest_i(u)$ equals $Highest_i$ of node $u$, and $Lowest_i(u)$ equals $Lowest_i$ of node $u$.

The DFS proceeds by forwarding a token, which represents the 'center of activity' along network edges. An interesting property of the DFS procedure is that initially the node does not know its children in the zone tree, but knows its parent, since the Forwarding-Construction procedure above only specifies parents. Only upon termination of the exploration process from a given node will that node know all its children. Thus, the token is going to traverse edges of the tree as well as other edges outgoing from the tree to nodes outside of the tree.

Another interesting property is that, in fact, node $q$ starts DFS procedure without waiting for the process of propagating of *FORWARD* messages to terminate, i.e., when DFS is started, the construction of tree $T_i(q)$ is not complete yet. In fact, as seen from the code in Section 5.2, DFS is started one pulse after transmission of *FORWARD* messages. This guarantees that the 'wave' of *FORWARD* messages will precede the token by at least on pulse.

That is, once token moves from $u$ to $v$ in DFS, then $v$ already knows whether $u$ is its *Parent*. If this is the case, then $v$ executes DFS with respect to itself. Otherwise, the DFS is immediately returned back to $u$, and $u$ looks for other unexplored edges. If there are no more unexplored edges, then DFS terminates and returns to the parent of $u$.

Transmission of a token is performed by sending a $VISIT(p)$ message, where $p$ is the highest DFS number in the sub-tree rooted at the node which

sends the token. Returning a token is performed by sending a $RETREAT(q)$ message, where $q$ is the highest DFS number in the sub-tree rooted at the node which returns the token. The only exception to the rule is the case in which node $v$ returns token to $u$, and $u$ is not the *Parent* of $v$, i.e. $u$ sent the token to $v$ 'by mistake'. In this case, the value of parameter $q$ is *nil*.

See formal presentation in Appendix, section A.3.

### *7.5. Post-Update procedure*

Observe that whenever an $(i-1)$-pivot $u$, i.e., a node which is a pivot of level $i-1$ but not of level $i$, learns about its post of level $j>i-1$, it has to pass on this information to all the lower-level pivots in its sub-tree. For that purpose, it broadcasts *POST* messages over its zone. The lower-level pivots in that zone will rebroadcast that message over their zones, etc., until it will reach all the lower-level pivots.

See formal presentation in Appendix, section A.4.

### *7.6. Complexity of the algorithm*

It takes $km$ characters to maintain all lists $List_i$, and $New_i$. Also, it takes $k^2$ characters to maintain $Closest_j$, $k$ characters for $p_i$, $Closest_i$, $Parent_i$, $Children_i$. Overall it amounts to $O(k^2+n^{1/k}k)$ characters, each one of $\log n$ bits.

## APPENDIX: FORMAL DESCRIPTION OF THE ALGORITHMS

### *A.1. Initialization*

**Output variables**
  $p_i$ (pivot of the node on level $i$, initially set to *nil*.)
**Local variables**
  *Coin* (the value of the coin flip; receives the values *head* and *tail*.)
**The procedure**
  See Figure 6.

```
Set i:=0 and Coin:=head
While i≤k and Coin=head do:
   Set p_i:=self /* you are a pivot of level i */
   Trigger procedure Forwarding-Construction(i)
   i:=i+1
   Flip Coin so that it comes head with probability
   c log n/m for some constant c>1.
End while
   j:=i−1 /* you are a j-pivot */
   wait one clock pulse
For i=1 to j do:
   Trigger DFS −Numbering(i)
End For
```

FIGURE 6. Initialization procedure

### *A.2. Forwarding-Construction*

**Data Structures**
We assume the existence of data structures that support the following operations on dynamic sets $S$ whose members are ordered pairs $(a,b)$, which are stored according to key $a$. For any two different tuples $(a_1,b_1)$, $(a_2,b_2)$ in $S$, $a_1 \neq a_2$.

*Member*$(S,a)$: returns *true* if there exists $c$ such that $(a,c) \in S$; else returns *false*.

*Insert*$(S,(a,b))$: adds $(a,b)$ to $S$.

*Delete*$(S,a)$ : deletes element $(a,b)$ from $S$, if there is any.

*ExtractMax*$(S)$: returns the tuple $(a,c) \in S$ with the largest key $a$ among all elements of $S$.

*DeleteMax*$(S)$: abbreviation for *Delete*$(S,$*ExtractMax*$(S))$.

*ExtractMin*$(S)$: returns the tuple $(a,c) \in S$ with the smallest key $a$ among all elements of $S$.

*size*$(S)$: returns the number of elements in $S$.

**Output variables** $FE_i(w)$ (the forwarding edge to $w$, kept for each $w$ in $List_i$).
  $Closest_i$ (the closest pivot at level $i$ to $v$).

*Parent*$_i$: the parent of $v$ in the zone of at level $i$, defined by *Closest*$_i$.

**Local variables**

*List*$_i$: Dynamic set, containing list of edges $(w, FE_i(w))$ for each $w \in N(v,m,P_i)$. Initially, $List_i = \varnothing$.

*New*$_i$: Dynamic set, containing list of elements which are candidates to enter *List*$_i$. Their names have not been transmitted yet. Initially, $New_i = \varnothing$.

**The procedure**

See Figure 7.

```
Upon triggering the procedure (at a pivot v∈P_i):
  send message FORWARD_i(v) to all neighbours w.
Upon receiving FORWARD_i(u) message from neighbour w:
  if Parent_i=nil then Parent_i:=w
  if Member(List_i,u)=false and Member(New_i,u)=false and
     |List_i|<m then Insert(New_i,(u,w)) /* insert tentatively */
     if size(List_i)+size(New_i)>m
     then DeleteMax(New_i) /* handle overflow */
Upon Clock pulse:
  if size(List_i)=0 and size(New_i)>0 then
     (u,w):=ExtractMin(New_i)
     Closest_i:=u /* closest pivot of level i */
     Parent_i:=w /* choosing the parent in the zone of v */
  for all neighbours w, and all (u,w)∈New_i,
     FE_i(u):=w
     send message FORWARD_i(u) on the link to w
     Insert(List_i,(u,v)),
     Delete(New_i,u).
```

FIGURE 7. *Forwarding* $-$ *Construction*$(i)$ for a node $v$

### *A.3. DFS-numbering and zones construction*

**Output variables**

*Children*$_i$: the set of children of $v$ in the zone of at level $i$.

*Lowest*$_i$: DFS (pre-order) number of the node in the DFS of level $i$.

*Lowest*$_i(w)$: an estimate of *Lowest*$_i$ of a neighbour $w$.

*Highest*$_i$=: the maximal DFS number assigned to a vertex in the sub-tree rooted at $v$.

*Highest*$_i(w)$=: an estimate of *Highest*$_i$ of a neighbour $w$.

**Local variables**

*Unvisited*$_i$: the set of unvisited nodes in the DFS of level $i$. Initially, *Unvisited*$_i$ contains all the neighbours of $v$.

**The procedure**

See Figure 8.

```
Upon triggering the procedure (at the root):
  Lowest_i: = 1,
  Highest_i: = 1,
  Call Procedure DFS_i.
Procedure DFS_i:
  If Unvisited_i ≠ ∅ then
    pick u ∈ Unvisited_i
    delete u from Unvisited_i;
    Lowest_i(u): = Highest_i + 1
    send VISIT_i(Highest_i) to v;
  Else
    if Parent_i ≠ nil, then
      send Retreat_i(Highest_i) to Parent_i;
    else (Parent_i = nil)
      Trigger Procedure Post − Update(i)
Upon receiving VISIT_i(p) from u:
  if u ≠ Parent_i then send RETREAT(nil) to u
  else
    Lowest_i: = p + 1;
    Highest_i: = p + 1;
  Call DFS_i.
Upon receiving RETREAT_i(p) from u:
  if p ≠ nil then
    add u to Children_i
    Highest_i(u): = p,
    Highest_i: = p,
  else /* p = nil */
    Lowest_i(u): = nil
    Highest_i(u): = nil
  Call DFS_i.
```

FIGURE 8. *DFS − Numbering*$(i)$ for a node *self*

### *A.4. Post-Update*

**Output variables**

$p_i$: pivot of the node at level $i$. Initialization guarantees that $p_i = self$ for nodes which are pivots of level $i$.

**The procedure**

See Figure 9.

```
Upon triggering the procedure:
  for all j > i, for which p_j ≠ nil,
    for each node z in Children_i, send message POST_i(j,p_j) to z.
Upon receiving POST_i(j,u) from w = Parent_i:
  for each node z in Children_i, send message POST_i(j,u) to z.
  if p_{i-1} = self, then
    p_j := u,
    for each node z in Children_{i-1},
    send message POST_{i-1}(j,u) to z.
```

FIGURE 9. *Post* $-$ *Update*$(i)$ (for a node *self*)

ACKNOWLEDGEMENTS

We thank Mike Saks and Noga Alon for helpful discussions and criticism of earlier versions of this paper.

REFERENCES

1. BARUCH AWERBUCH, AMOTZ BAR-NOY, NATI LINIAL, DAVID PELEG (1988). *Improved Routing Strategies with Succinct Tables,* Technical Report MIT/LCS/TM-354, Massachusetts Institute of Technology.
2. BARUCH AWERBUCH, AMOTZ BAR-NOY, NATI LINIAL, DAVID PELEG (1988). *Memory-Balanced Routing Strategies,* Technical Report MIT/LCS/TM-369, Massachusetts Institute of Technology.
3. GREG N. FREDERICKSON, RAVI JANARDAN (1986). Seperator-based strategies for efficient message routing. *27th Annual Symposium on Foundations of Computer Science, Toronto, Ontario, Canada,* 428-437, IEEE.
4. GREG N. FREDERICKSON, RAVI JANARDAN (1988). Designing networks with compact routing tables. *Algorithmica 3,* 171-190.
5. L. KLEINROCK, F. KAMOUN (1977). Hierarchical routing for large networks; performance evaluation and optimization. *Computer Networks 1,* 155-174.
6. L. KLEINROCK, F. KAMOUN (1980). Optimal clustering structures for hierarchical topological design of large computer networks. *Computer Networks 10,* 221-248.
7. LÁSZLÓ LOVÁSZ (1975). On the ratio of optimal integral and fractional covers. *Discrete Mathematics 13,* 383-390.
8. R. PERLMAN (1982). Hierarchical networks and the subnetwork partition problem. *5th Conference on System Sciences.*
9. DAVID PELEG, ELI UPFAL (1988). A tradeoff between size and efficiency for routing tables. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing,* 43-52, ACM.
10. N. SANTORO, R. KHATIB (1985). Labelling and implicit routing in networks. *The Computer Journal 28,* 5-8.
11. C.A. SUNSHINE (1982). Addressing problems in multi-network systems. *IEEE INFOCOM,* IEEE.

12. A. TANENBAUM (1981). *Computer Networks,* Prentice Hall.
13. J. VAN LEEUWEN, R.B. TAN (1982). Routing with compact routing tables. G. ROZENBERG, A. SALOMAA (eds.). *The Book of L,* 259-273, Springer-Verlag, New York.
14. J. VAN LEEUWEN, R.B. TAN (1987). Interval routing. *The Computer Journal 30,* 298-307.
15. H. ZIMMERMAN (1980). OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Trans. Comm. 28,* 425-432.