Compact routing schemes

Mikkel Thorup
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932, USA
mthorup@research.att.com

Uri Zwick *
School of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel
zwick@post.tau.ac.il

ABSTRACT

We describe several compact routing schemes for general weighted undirected networks. Our schemes are simple and easy to implement. The routing tables stored at the nodes of the network are all very small. The headers attached to the routed messages, including the name of the destination, are extremely short. The routing decision at each node takes *constant* time. Yet, the *stretch* of these routing schemes, i.e., the worst ratio between the cost of the path on which a packet is routed and the cost of the cheapest path from source to destination, is a small constant. Our schemes achieve a near-optimal tradeoff between the size of the routing tables used and the resulting stretch. More specifically, we obtain:

1. A routing scheme that uses only $\tilde{O}(n^{1/2})$ bits of memory at each node of an n-node network that has stretch 3. The space is *optimal*, up to logarithmic factors, in the sense that every routing scheme with stretch < 3 must use, on some networks, routing tables of total size $\Omega(n^2)$, and every routing scheme with stretch < 5 must use, on some networks, routing tables of total size $\Omega(n^{3/2})$. The headers used are only $(1 + o(1)) \log_2 n$-bit long and each routing decision takes *constant* time. A variant of this scheme with $\lceil \log_2 n \rceil$-bit headers makes routing decisions in $O(\log \log n)$ time.

2. Also, for every integer $k > 2$, a general *handshaking* based routing scheme that uses $\tilde{O}(n^{1/k})$ bits of memory at each node that has stretch $2k - 1$. A conjecture of Erdős from 1963, settled for $k = 3, 5$, implies that the routing tables are of near-optimal size relative to the stretch. The handshaking is similar in spirit to a DNS lookup in TCP/IP. Headers are $o(\log^2 n)$ bits long and each routing decision takes *constant* time. Without handshaking, the stretch of the scheme increases to $4k - 5$.

One ingredient used to obtain the routing schemes mentioned above, may be of independent practical and theoretical interest:

3. A shortest path routing scheme for *trees* of arbitrary degree and diameter that assigns each vertex of an n-node tree a $(1 + o(1)) \log_2 n$-bit label. Given the label of a source node and the label of a destination it is possible to compute, in *constant* time, the port number of the edge from the source that heads in the direction of the destination.

The general scheme for $k > 2$ also uses a clustering technique introduced recently by the authors. The clusters obtained using this technique induce a sparse and low stretch *tree cover* of the network. This essentially reduces routing in general networks into routing problems in trees that could be solved using the above technique.

*Work supported in part by the **Israel Science Foundation** founded by The Israel Academy of Sciences and Humanities.

1. INTRODUCTION

Routing is one of the basic tasks that a distributed network of processors must be able to perform. A *routing scheme* is a mechanism that can deliver packets of information from any node of the network to any other node of the network. Here, we consider undirected weighted networks. We aim at routing along short paths.

More specifically, a routing scheme is a distributed algorithm. Each processor in the network has a routing daemon running on it. This daemon receives packets of information and has to decide whether these packets have already reached their destination, and if not, how to forward them towards their destination. Each packet of information has a *header* attached to it. This header contains the destination of the packet, and in some cases, some additional information that can be used to guide the routing of this message towards its destination. Each routing daemon has a local *routing table* at its disposal. It has to decide, based on this table and on the packet header, whether to pass the packet to its host, or whether to forward the packet to one of its neighbors in the network. In either case, it has to determine the relevant *port number*. The *stretch* of a routing scheme is the worst ratio between the length (or cost) of a path on which a message is routed and the length (or cost) of the shortest (or cheapest) path in the network from the source to the destination.

Most of our routing schemes are *labeling* schemes that rename, or label, the vertices. The header of a packet is then simply the label of its destination. Some of our schemes use a more complicated *handshaking* process to choose the header. How and where the source finds the label of a desired destination is not the issue of this paper. Typically, many packets would be sent from a source to a destination using the *same* header. For us, the essential point is to make sure that the router at each node can quickly forward packets based on their headers. For our most compact headers/labels, we assume that we can choose the port numbers of the edges incident to a node. Otherwise, we loose a $O(\log n/ \log \log n)$ factor in size of the headers.

The design of efficient routing schemes is a well studied subject. For a general overview of this area, with many references, we refer the reader to Peleg [22]. There are two extreme solutions to the routing problem. The first is to store a complete routing table at each node of the network. This table specifies, for any destination, the link on which packets to that destination should be forwarded. Packets could then be routed along shortest paths of the network. The obvious drawback of this solution is that, in the worst case, each node of the network would need a table of size $\Omega(n)$. In the other extreme, of source directed routing, each packet carries in its header a complete description of the path along which it should be routed. Packets could again be routed along shortest paths, but the headers attached to them may need to be of size $\Omega(n)$. Both of these solutions do not scale well. It is desirable, therefore, to find trade-off schemes with substantially smaller routing tables, yet having headers of only logarithmic size. (Note that $\lceil \log_2 n \rceil$ bits in the header are required just to specify the destination.) To obtain routing schemes for general graphs that use $o(n)$ of memory at each node and only a small number of bits in each header, we have to abandon the requirement that packets are always routed on shortest paths, and settle instead for the requirement that packets are routed on paths with relatively small stretch.

The first tradeoff between the size of the routing tables and the stretch of the resulting routing scheme, for general network topologies, was obtained by Peleg and Upfal [23]. However, they only dealt with *unweighted* networks and with the total size of the routing tables at all the nodes of the network. Weighted networks were first considered by Awerbuch *et al.* [4] who obtained, for every integer $k \geq 1$, a routing scheme that uses only $\tilde{O}(n^{1/k})$ space at each vertex, but has a huge stretch factor of $O(k^2 9^k)$. A better tradeoff was then obtained by Awerbuch and Peleg [5]; using tables of size $\tilde{O}(n^{1/k})$ they obtain stretch $O(k^2)$. A stretch 3 scheme that uses local routing tables of size $\tilde{O}(n^{2/3})$ was obtained by Cowen [8], and a stretch 5 scheme that uses local routing tables of size $\tilde{O}(n^{1/2})$ was obtained by Eilam *et al.* [11]. These results are summarized in Table 1.

We substantially improve these results. We present, in particular, a simple stretch 3 routing scheme that uses only $\tilde{O}(n^{1/2})$ memory at each node of the network. This relation is essentially best possible: as mentioned, every routing scheme with stretch < 3 must use $\Omega(n^2)$ total space on some n-node networks, and hence at least $\Omega(n)$ at some node. (See Fraigniaud and Gavoille [14], Gavoille and Pérennès [20], and Gavoille and Gengler [19]. The claim also follows easily from the lower bounds given by us in [25].) Furthermore, any routing scheme of stretch < 5 must use a total space $\Omega(n^{3/2})$ on some n-node network, and hence at least $\Omega(n^{1/2})$ at some node. (This follows again from [25].) The scheme is a labeling scheme using $(1 + o(1)) \log_2 n$-bit labels. Each routing decision takes *constant* time. A variant of this scheme, based on techniques of Eilam *et al.* [11], uses consecutive $\lceil \log_2 n \rceil$-bit labels, but routing decisions would then take non-constant time.

We also present a sequence of routing schemes that exhibit a probably optimal trade-off between stretch and routing table size. These schemes assume the existence of a *handshaking* mechanism by which the source u and the destination v agree on an $o(\log^2 n)$-bit header that is then attached to all packets sent from u to v. The scheme uses labels of size $o(k \log^2 n)$. Moreover, to perform the handshaking, u sends $O(\log n)$ bits of information to v, and then v sends $O(\log n)$ bits of information back to u. (This is similar to DNS lookup in TCP/IP.) The overhead resulting from this handshaking is not expected to be large as, typically, a long stream of packets are to be sent from a given source to a given destination. Furthermore, the header obtained may be *cached* and reused.

Authors	Stretch	Table size
Cowen [8]	3	$\tilde{O}(n^{2/3})$
Eilam, Gavoille, Peleg [11]	5	$\tilde{O}(n^{1/2})$
Awerbuch, Peleg [5]	$O(k^2)$	$\tilde{O}(kn^{1/k})$
Awerbuch *et al.* [4]	$O(k^2 9^k)$	$\tilde{O}(kn^{1/k})$

Table 1: Previously available routing schemes

Stretch	Table size	Handshaking required?
3	$\tilde{O}(n^{1/2})$	no
5	$\tilde{O}(n^{1/3})$	yes
7	$\tilde{O}(n^{1/3})$	no
$2k-1$	$\tilde{O}(kn^{1/k})$	yes
$4k-5$	$\tilde{O}(kn^{1/k})$	no

Table 2: Our new routing schemes

We note that some sort of lookup mechanism may be needed, in any case, in conjunction with any routing scheme that, like our schemes, *renames*, or assigns labels, to the nodes of the network. We also present routing schemes that do *not* require handshaking, at a price of a slightly increased stretch. These schemes could be used to implement handshaking processes.

For any integer parameter $k > 2$, our handshaking based routing scheme uses routing tables of size $\tilde{O}(n^{1/k})$ and has stretch $2k-1$. Erdős [12] conjectured, in 1963, that for every $k \geq 1$, there is an n-vertex graph with $\Omega(n^{1+1/k})$ edges whose *girth* is at least $2k+2$. (The conjecture is known to hold for $k = 1, 2, 3, 5$.) If such graphs do exist, then every routing scheme of stretch $< 2k+1$ would have to use tables of total size $\Omega(n^{1+1/k})$ on at least one subgraph of such a graph. (This again follows easily from [25]. Details would appear in the full version of the paper.) A variant of the above scheme avoids the handshaking at the cost of an increased stretch of $4k-5$.

A summary of our results appears in Table 2. Our routing schemes, with or without handshaking, substantially improve all previously available results. Furthermore, our results are near-optimal in many respects.

An ingredient used in the above mentioned routing schemes, that we believe is interesting in its own right, both from a theoretical and a practical viewpoint, is a new routing scheme for *trees of arbitrary degree*. Santoro and Khatib [24] introduced the notion of *interval routing* using which shortest paths routing on tree can be achieved by storing at each node u of the tree a local routing table of size $O(\deg(u))$, where $\deg(u)$ is the degree of u. (For more on interval routing, see Fraigniaud and Gavoille [15] and Gavoille [17].) Cowen [8] (see also Gavoille [17]) describes a simple way of reducing the size of the routing table used in each node to $O(\min\{\deg(u), n^{1/2}\})$.

Our new tree routing scheme assigns to each node in an n-node tree a $(1 + o(1)) \log_2 n$-bit label. Given the label of a source node, and the label of the destination, and no additional information, it is possible to compute, in *constant* time, the port number of the link that heads from the source towards the destination. Thus, the routing table of a node is simply its label, and the header of a packet used in this scheme is simply the label of its destination. Simple variants of this scheme are a expected to be of practical value.

Our routing scheme for general graphs are based on *tree covers* which are family of induced trees such that for each pair of vertices, there is a tree in the family containing a low-stretch path between them. We can then apply our tree routing schemes within each tree. The use of tree covers in routing is, in itself, standard (see, e.g., Awerbuch and Peleg [5], Eilam *et al.* [11], Cowen [8] and Peleg [22, Chapter 15]). For our general trade-offs with $k > 2$, we can essentially just use the tree covers recently introduced by us in [25]. However, for our stretch 3 result, we need a new simple, but powerful, *recursive sampling technique* for selecting the tree covers.

The rest of this paper is organized as follows. We begin in the next section by describing our new routing scheme for trees. The results presented will be used in the subsequent sections. In Section 3 we then describe our stretch 3 scheme for general graphs. In Section 4 we describe our schemes for general stretches.

2. ROUTING IN TREES

In this section we describe an extremely efficient routing scheme for trees. Each vertex in an n-vertex graph is assigned a $(1 + o(1))\log_2 n$-bit label. This label is the only information stored at the vertex. No additional routing tables are required. This label also serves as the header attached to messages sent to that vertex. We describe this routing scheme in a gradual fashion. In Section 2.1 we describe a scheme that uses labels of size $O(\log^2 n)$. The size of the labels is reduced to $O(\log n)$ in Section 2.2 and then to $(1 + o(1))\log_2 n$ in Section 2.3. An important feature of all there variants is that each routing decision takes *constant* time using only elementary standard operations. The first two schemes are expected to be of practical value. A somewhat similar scheme was found, independently, by Gavoille [18]. (But, to the best of our knowledge, it does not work with $(1 + o(1))\log_2 n$-bit labels.)

Before describing our schemes, we recall the standard interval routing technique as it applies to trees (Santoro and Khatib [24], van Leeuwen and Tan [28]): We root the tree arbitrarily and perform a depth first enumeration of the vertices. We identify each vertex with its depth first search number. For each vertex w, let f_w be the largest descendant of w. Then, a vertex v is a descendant of w if and only $v \in [w, f_w]$. A packet destined for v that arrives to w is routed as follows: If $w = v$, the packet has reached its destination. If $v \notin [w, f_w]$, the packet is sent to the parent of w, using the parent pointer of w. Otherwise, a predecessor search among the children $w_1, ..., w_d$ of w is performed. If w_i is the last child smaller than or equal to v, the packet is forwarded to w_i.

The routing table used at a vertex w is of size $O(\deg(w))$, where $\deg(w)$ is the degree of w. (Each element in this table is $\lceil\log_2 n\rceil$-bit long.) Thus, very large routing tables are needed in high degree vertices. Furthermore, the routing decision at such high degree vertices would take *non-constant* time, as it requires a predecessor search (see Beame and Fich [6]). We note that the degree distribution of many networks in the real world was found to have heavy tails (see, e.g., Faloutsos *et al.* [13]), meaning that some nodes have comparatively high degrees. A natural example is the star topology with a single center of degree $n-1$. The routing schemes described here are vastly superior. In our theoretically strongest scheme, each vertex, irrespective of its degree, only needs to store its $(1+o(1))\log_2 n$-bit label. Yet, each routing decision takes just constant time.

2.1 A compact scheme

We begin by describing, for every integer $b > 1$, a routing scheme that uses routing tables consisting of $O(b)$ words, and labels (and therefore headers) consisting of $O(\log_b n)$ words. Each word here is $O(\log n)$-bit long, where n is the number of vertices in the tree. This scheme works with *any* assignment of port numbers. We later present a tuned version with $b = 2$ that could be useful in practice, e.g., for cable networks.

The *weight* s_v of a vertex v is the number of its descendants in the tree. (A vertex is considered to be a descendant of itself.) A child v' of a vertex v is said to be *heavy* if $s_{v'} \geq s_v/b$, and *light* otherwise. In other words, the child v' is heavy if a fraction of at least $1/b$ of the descendants of v are also descendants of v'. Obviously, each vertex can have at most $b-1$ heavy children. For convenience, we define r, the root of the tree, to be heavy. The *light level* ℓ_v of a vertex v is defined as the number of light vertices on the path from r to v, including v if it is light.

We again enumerate the vertices of the tree in depth first order, where all the light children of a vertex are visited before its heavy children, if any. As before, we identify a vertex v with the number assigned to it, and let f_v be the largest descendant of v. We let h_v be the first heavy child of v, if it exists, or $f_v + 1$ otherwise. We let H_v be an array containing in its first element the number of heavy children that v has, and in its subsequent elements all the heavy children of v. Finally, we let P_v be an array containing in its first element $P_v[0]$ the port number corresponding the the edge from v to its parent, and then the port numbers corresponding to the edges from v to its heavy children. The routing information stored at v consists of (v, f_v, h_v, H_v, P_v), requiring a total of at most $O(b)$ words.

Each time an edge from a vertex to one of its light children is descended, the number of descendants in the corresponding subtree decreases by a factor of at least b. Thus, the light level ℓ_v of every vertex v is at most $\log_b n$. Let $\langle v_0, v_1, \ldots, v_k\rangle$, where $r = v_0$ and $v_k = v$, be the path from the root of the tree to v, and let i_j, for $1 \leq j \leq \ell_v$ be the index of the j-th light vertex on the path. We let

$$L_v = (\mathit{port}(v_{i_1-1}, v_{i_1}), \mathit{port}(v_{i_2-1}, v_{i_2}), \ldots, \mathit{port}(v_{i_{\ell_v}-1}, v_{\ell_v})).$$

In other words, L_v is an array of at most $\log_b n$ words containing the port numbers corresponding to the edges leading to the light vertices on the path from r to v. We then let $\mathit{label}(v) = (v, L_v)$ be the label of v. A packet addressed to v would carry $\mathit{label}(v)$ at its header.

The routing algorithm should now be obvious. Suppose that a packet with the header (v, L_v) arrives at w. If $w = v$, we are done. Otherwise, we check whether $v \in [w, f_w]$. If not, then v is not a descendant of w and the packet is forwarded to the parent of w using port $P_v[0]$. Next, we check whether $v \in [h_w, f_w]$. If so, then v is a descendant of a heavy child of w. We find the appropriate heavy child by searching H_w, and then obtain the corresponding port number from P_w. Otherwise, v is a descendant of a light child, in which case we use the port number given in $L_v[\ell_w]$. (We assume that the indices of L_v start from 0.)

Fine tuning Assuming that *routing speed* is the most critical consideration, we suggest the following concrete implementation of the scheme just described with $b = 2$. We suppose that each router has a special link used for transferring packets of information to its host. We let $L_v[\ell_v]$ be the port number corresponding to the link from v to its host. With this arrangement we would not have to check separately whether a packet has reached its destination. Also, as $b = 2$, the array P_v now contains only two elements: $P_v[0]$ is the port number of the link from v to its parent, and $P_v[1]$ is the port number of the link from v to its heavy child, if there is one. When a message with header (v, L_v) arrives at w, the router can find the port number on which the packet should be forwarded using the following simple C expression (see [21]):

```
((v>=w && v< h) ? L[l] : P[v>=h && v<=f]),
```

where $\mathtt{v} = v$ and $\mathtt{L} = L_v$ are taken from the header, and $\mathtt{w} = w$, $\mathtt{l} = \ell_w$, $\mathtt{f} = f_w$, $\mathtt{h} = h_w$ and $\mathtt{P} = P_v$ are stored locally at the router. It can hardly get any faster!

2.2 A more compact scheme

As mentioned, the above scheme works with *any* assignment of $O(\log n)$-bit port numbers. It is not difficult to see that if the assignment of port numbers is arbitrary, and each vertex is only allowed to store $O(b)$ port numbers in its local routing table, then some headers must contain $\Omega(\log_b n)$ port numbers. In that sense, the scheme presented is optimal. However, if we are allowed to assign the port numbers ourselves, then the size of the labels, and therefore headers, used by the scheme of Section 2.1 can be easily reduced to $3\log_2 n$. The details are given, due to lack of space, in Appendix A. We note that the resulting scheme is still simple and practical.

2.3 An extremely compact scheme

Let T be a tree. We say that the assignment of port numbers to the edges of T is *canonical* if it obtained in the following way: Let v be a vertex of T whose parent, if it exists, is v', and whose children, arranged in non-increasing order of weight are $v_1, v_2, \ldots, v_d$, i.e., $s_{v_1} \geq s_{v_2} \geq \ldots \geq s_{v_d}$. Then $port(v, v') = 0$ and $port(v, v_i) = i$, for $1 \leq i \leq d$. (There may be several canonical assignments.) In this section we obtain the following two results:

THEOREM 2.1. *Let T be an n-vertex tree with a* canonical *assignment of port numbers. Then, it is possible to assign every vertex v of T a $(1 + o(1))\log_2 n$-bit label $label(v)$ such that given $label(u)$ and $label(v)$, and nothing else, it is possible to determine, in constant time, the port number, at u, of the first edge on the path in T from u to v.*

THEOREM 2.2. *Let T be an n-vertex tree with an* arbitrary *assignment of port numbers. Then, it is possible to assign every vertex v of T a $o(\log^2 n)$-bit label $label(v)$ such that given $label(u)$ and $label(v)$, and nothing else, it is possible to determine, in constant time, the port number, at u, of the first edge on the path in T from u to v.*

This also provides a solution to a problem of Abiteboul *et al.* [1] in which it is only required, given $label(u)$ and $label(v)$, to determine whether u is an ancestor of v. They describe a solution that uses $(3/2 + o(1))\log_2 n$-bit labels. A solution to this more restricted problem was found, independently, and slightly earlier, by Alstrup [3].

Each of our labels is composed of a sequence $s_1, s_2, \ldots, s_k$ of variable length strings of total length $(1 + o(1))\log_2 n$. To form the label we cannot just concatenate these strings, as we would not know how to separate them. We need, therefore, a way of replacing each string s_i by a string $\mathrm{code}(s_i)$, whose length is only slightly longer than s_i, such that given the concatenation of $\mathrm{code}(s_1)$, $\mathrm{code}(s_2), \ldots, \mathrm{code}(s_k)$, we could efficiently reconstruct the original sequence.

We start, therefore, with a coding lemma. Let $s \in \{0,1\}^*$ be a bit string. Let $|s|$ be the length of s. If $s_1, s_2 \in \{0,1\}^*$, we let $s_1.s_2$, or sometimes just $s_1 s_2$, be the concatenation of s_1 and s_2. We let ε denote the empty string.

LEMMA 2.3. *There is an efficient encoding scheme* $\mathrm{code}: \{0,1\}^* \to \{0,1\}^*$ *that satisfies: (i) The set $\{\mathrm{code}(s) \mid s \in \{0,1\}^*\}$ is an infinite suffix free code. (ii) For every $s \in \{0,1\}^*$ we have $|\mathrm{code}(s)| \leq |s| + O(\log(|s|+2))$. (iii) Given a machine word that contains the string $\mathrm{code}(s)$ in its least significant bits, it is possible to extract s and $|s|$ using a constant number of standard operations. It is also possible to remove $\mathrm{code}(s)$ from this word.*

The proof of Lemma 2.3 is given, due to lack of space, in Appendix B. The proof uses standard coding techniques. The novelty here is in the fact that the decoding can be done in *constant* time. It follows easily from the Kraft inequality (see Cover and Thomas [7]), that this encoding is almost optimal. No uniquely decipherable encoding, for example, can have $|\mathrm{code}(s)| \leq |s| + \log|s|$, for every sufficiently long string s. If i is an integer, we let $\mathrm{bin}(i)$ be the bit string containing the binary representation of i. We let $\mathrm{code}(i) = \mathrm{code}(\mathrm{bin}(i))$.

We let $\prec$ be a total ordering on bit strings defined as follows: For any strings $s, t_1, t_2 \in \{0,1\}^*$ we have $s0t_1 \prec s \prec s1t_2$. This ordering has the following natural interpretation. Imagine all the binary strings arranged in an infinite binary tree in which the root is labeled ε, and in which the left child of a vertex labeled s is labeled $s0$, and its right child labeled $s1$. The meaning of this order should now suggest itself. For two strings $s_1, s_2 \in \{0,1\}^*$ we have $s_1 \prec s_2$, if s_1 is "to the left of" s_2 in this tree. It is not difficult to check that given two machine words that contain $\mathrm{code}(s_1)$ and $\mathrm{code}(s_2)$, it is possible to determine, using a constant number of standard operations, whether $s_1 \prec s_2$.

LEMMA 2.4. *Let $p_1, p_2, \ldots, p_k$ be a sequence of numbers satisfying $p_i > 0$, for $1 \leq i \leq k$, and $\sum_{i=1}^k p_i = 1$. Then, there exist bit strings $s_1 \prec s_2 \prec \ldots \prec s_k$ such that $|s_i| \leq \log_2 \frac{1}{p_i}$, for $1 \leq i \leq k$.*

PROOF. We prove the claim by induction on k. If $k = 1$, then $p_1 = 1$, and we simply let $s_1 = \varepsilon$. Then, $|s_1| = \log_2 1 = 0$, as required. Otherwise, let $1 \leq r \leq k$ be such that $\sum_{i=0}^{r-1} p_i < \frac{1}{2}$ but $\sum_{i=0}^{r} p_i \geq \frac{1}{2}$. Let $q_0 = \sum_{i=1}^{r-1} p_i$ and $q_1 = \sum_{i=r+1}^{k} p_i$. Clearly $0 \leq q_0, q_1 \leq \frac{1}{2}$. By the induction hypothesis, we can construct $s'_1 \prec s'_2 \prec \cdots \prec s'_{r-1}$ such that $|s'_i| \leq \log_2 \frac{q_0}{p_i}$, for $1 \leq i < r$, and $s''_{r+1} \prec s''_{r+2} \prec \cdots \prec s''_k$ such that $|s''_i| \leq \log_2 \frac{q_1}{p_i}$, for $r < i \leq k$. Let $s_i = 0s'_i$, for $1 \leq i < r$, and $s_i = 1s''_i$, for $r < i \leq k$. Clearly, $s_1 \prec s_2 \prec \ldots \prec s_k$. Also, $|s_i| \leq \log_2 \frac{q_0}{p_i} + 1 \leq \log_2 \frac{1}{p_i}$, for $1 \leq i < r$, as $q_0 \leq \frac{1}{2}$. Similarly, $|s_i| \leq \log_2 \frac{1}{p_i}$, for $r < i \leq k$, as required. $\square$

We now describe the labeling scheme whose existence is asserted in Theorem 2.1. Let b be a parameter. (We would later set $b = \lceil \sqrt{\log_2 n}\, \rceil$.) Recall that the weight s_v of a vertex v is the number of descendant of v. We say, this time, that a vertex v is *heavy* if $s_v \geq n/b$, and *light* otherwise. Let T_h be the subtree of T spanning the heavy vertices. It is easy to see that the number of leaves of T_h is at most b. We break T_h into a collection of paths in the following way: If v has more than one child in T_h, then we remove the edges connecting it to all its children. This partitions T_h into at most $2b - 1$ *heavy paths*. Some of these paths may be single vertices. By definition, if $\langle \bar{v}_1, \bar{v}_2, \ldots, \bar{v}_m \rangle$ is a heavy path, and $\bar{v}_1$ is the vertex on this path closest to the root, then $\bar{v}_{i+1}$ is the only heavy child of $\bar{v}_i$, and therefore the heaviest child of $\bar{v}_i$. Thus $port(\bar{v}_i, \bar{v}_{i+1}) = 1$. Furthermore, if u is in a heavy path P while v is in a different heavy path P', then the path from u to v in the tree passes through one of the endpoints of P. This process is depicted in Figure 1. The filled vertices there are heavy, the rest are light. The bold edges are part of heavy paths.

If v is a vertex of T and $v_1, v_2, \ldots, v_d$ are its light children, we let $t_v = s_{v_1} + s_{v_2} + \ldots + s_{v_d}$ be the *light weight* of v. (Note that

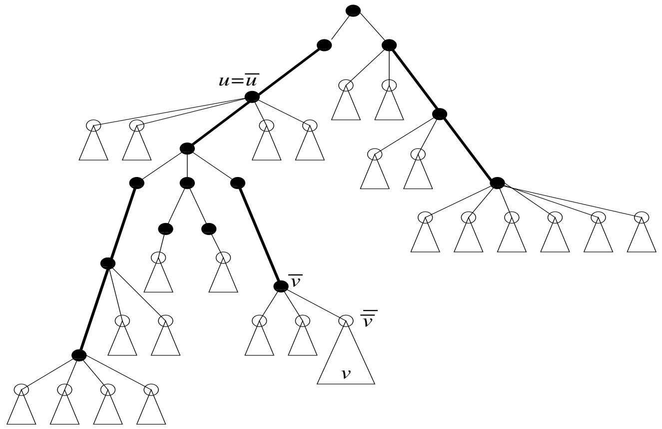

Figure 1: Decomposing a tree.

descendants of light vertices are also light.) The light weight of a path $P = \langle \bar{v}_1, \bar{v}_2, \ldots, \bar{v}_m \rangle$ is the sum of the light weights of the vertices on that path, i.e., $t_P = t_{\bar{v}_1} + \ldots + t_{\bar{v}_m}$. Let $P_1, P_2, \ldots, P_\ell$, where $1 \le \ell \le 2b - 1$, be the heavy paths into which T_h is decomposed, arranged in non-increasing order of light weight, i.e., $t_{P_1} \ge t_{P_2} \ge \ldots \ge t_{P_k}$.

Let v be a vertex of T. Let $\bar{v}$ be the last heavy vertex on the path from r, the root of the tree, to v. If v is not heavy then $\bar{v} \ne v$ and we let $\bar{\bar{v}}$ be the child of $\bar{v}$ that is an ancestor of v. (Note that $\bar{\bar{v}}$ is light.) Let $P_i = \langle \bar{v}_1, \bar{v}_2, \ldots, \bar{v}_m \rangle$ be the heavy path containing $\bar{v}$. For $1 \le r \le m$, let $p_r = t_{\bar{v}_r}/t_{P_i}$, i.e., the relative contribution of $\bar{v}_r$ to the light weight of P_i. Let $s_1 \prec s_2 \prec \ldots \prec s_m$ be the strings corresponding to $p_1, p_2, \ldots, p_m$ as per Lemma 2.4. Let $1 \le j \le n$ be such that and assume that $\bar{v} = \bar{v}_j$.

The label of a vertex v is composed of two components, an *identity label*, denoted $\mathrm{ID}(v) = \mathrm{ID}(T, v)$, and a 'mini' routing table, denoted $\mathrm{RT}(v)$. (We call it a mini routing table at its size is $o(\log n)$.) The identity label $\mathrm{ID}(v)$ is composed of the following items: (i) The binary representation of i, the index of the heavy path containing $\bar{v}$. (ii) The string s_j corresponding to $\bar{v} = \bar{v}_j$ in P_i. (iii) The port number of the edge leading from $\bar{v}$ to $\bar{\bar{v}}$, or an indication that $v = \bar{v}$. (iv) If $v \ne \bar{v}$, the identifying label of v in the subtree $T_{\bar{\bar{v}}}$, the subtree of T rooted at $\bar{\bar{v}}$, defined recursively. More formally,

$$\mathrm{ID}(T, v) \;=\; \begin{cases} \mathrm{ID}(T_{\bar{\bar{v}}}, v).\mathrm{code}(i).\mathrm{code}(s_j).\mathrm{code}(\mathit{port}(\bar{v}, \bar{\bar{v}})) & \text{if } v \ne \bar{v}, \\ \mathrm{code}(i).\mathrm{code}(s_j) & \text{otherwise.} \end{cases}$$

It is easy to check that $\mathrm{ID}(v)$ uniquely identifies v, i.e., $\mathrm{ID}(v) \ne \mathrm{ID}(u)$, whenever $v \ne u$.

The mini routing table $\mathrm{RT}(v)$ is constructed as follows. Consider the last iteration of the recursive process defining $\mathrm{ID}(v)$, i.e., the iteration in which v becomes heavy. Let v' be the root of the tree considered at that iteration, and let $P_1, P_2, \ldots, P_\ell$ be the heavy paths into which $(T_{v'})_h$ is decomposed, again in non-increasing order of light weight. Recall that $\ell \le 2b - 1$. Assume that v is contained in $P_i = \langle v_1, v_2, \ldots, v_m \rangle$. Note that for any $j \ne i$, the same edge from v should be used for all destinations in P_j. Now $\mathrm{RT}(v)$ is the table of ℓ ($\lfloor \log_2 b \rfloor + 1$)-bit numbers where for $j \ne i$, $\mathrm{RT}(v)[j]$ is the port number of the edge from v leading to P_j and $\mathrm{RT}(v)[i] = \perp$. Since the numbers have the same length, the entry $\mathrm{RT}(v)[j]$ is easily found in constant time. Note that if $v \ne v_m$, then all the port numbers in this table are either 0 or 1. The number of bits needed to code $\mathrm{RT}(v)$, including b and $\lfloor \log_2 b \rfloor + 1$, is $O(b \log b)$. Finally, we define

$$\mathit{label}(v) = \mathrm{code}(\mathrm{ID}(v)).\mathrm{code}(\mathrm{RT}(v)).\mathrm{code}(\mathrm{pnt}(v)) \;,$$

where $\mathrm{pnt}(v)$ is the length of the first three code words in $\mathrm{ID}(v)$. Note that these code words make up $\mathrm{ID}(T_{\bar{\bar{v}}}, v)$, where $\bar{\bar{v}}$ is the root of the subtree in which v becomes heavy. This pointer enables us to extract $\mathrm{ID}(T_{\bar{\bar{v}}}, v)$ is constant time. Theorem 2.1 would now follow from the following two lemmas:

LEMMA 2.5. *Let T is an n-vertex tree. Then, for every vertex v of T we have $|\mathit{label}(v)| = (1 + o(1)) \log_2 n$.*

PROOF. The identifier $\mathrm{ID}(v)$ is composed of the concatenation of the encodings of $3k$ bits $c_1, c_2, \ldots, c_{3k}$, where $k \le \log_b n$ is the number of iterations of the recursive process defining $\mathrm{ID}(v)$. Thus, $|\mathrm{ID}(v)| = \sum_{i=1}^{3k} |\mathrm{code}(c_i)| = \sum_{i=1}^{3k} |c_i| + O(\sum_{i=1}^{3k} \log(|c_i| + 2))$. As $b = \lceil \sqrt{\log_2 n}\, \rceil$, we get that $k = O(\log n / \log\log n)$. We shall show that $\sum_{i=1}^{3k} |c_i| \le \log_2 n + O(k)$. It would then follow easily, using the convexity of $\log x$, that $\sum_{i=1}^{3k} \log(|c_i| + 2)) = O(\log n \cdot \log\log\log n / \log\log n)$. To see that $\sum_{i=1}^{3k} |c_i| \le \log_2 n + O(k)$, we show that $|c_1| + |c_2| + |c_3| = \log_2 b + O(1)$. The same would hold for the other $k - 1$ triples and the claim would follow. Let $\bar{v}$ and $\bar{\bar{v}}$ and the heavy paths $P_1, P_2, \ldots, P_\ell$ be as in the construction of $\mathrm{ID}(v)$. Suppose $\bar{v}$ is in P_i and that the string corresponding to it is s. Let $p = \mathit{port}(\bar{v}, \bar{\bar{v}})$. Then, $c_1 = \mathrm{bin}(i)$, $c_2 = s$, and $c_3 = \mathrm{bin}(p)$. By the ordering of the heavy paths, we have $t_{P_i} \le n/i$. By the choice of s, we have $|s| \le \log_2 \frac{t_{P_i}}{t_{\bar{v}}}$. Finally, as the assignment of port numbers is canonical, we have $t_{\bar{\bar{v}}} \le t_{\bar{v}}/p$. Thus,

$$\begin{aligned} |c_1| + |c_2| + |c_3| &\le (\log_2 \tfrac{n}{t_{P_i}} + 1) + \log_2 \tfrac{t_{P_i}}{t_{\bar{v}}} + (\log_2 \tfrac{t_{\bar{v}}}{t_{\bar{\bar{v}}}} + 1) \\ &\le \; \log_2 \tfrac{n}{t_{\bar{\bar{v}}}} + 2 \;\le\; \log_2 b + 2 \;, \end{aligned}$$

as required. Thus,

$$|\mathrm{ID}(v)| \le \log_2 n + O(\log n \cdot \log\log\log n / \log\log n) \;.$$

As $|\mathrm{code}(\mathrm{pnt}(v))| = O(\log\log n)$, and $|\mathrm{RT}(v)| = O(b \log b)$, we also have $|\mathit{label}(v)| \le \log_2 n + O(\frac{\log n \cdot \log\log\log n}{\log\log n})$. $\square$

LEMMA 2.6. *Given $\mathit{label}(u)$ and $\mathit{label}(v)$, it is possible to determine, in constant time, the port number, at u, of the first edge on the path in T from u to v.*

PROOF. If $\mathit{label}(u) = \mathit{label}(v)$, we are done. Otherwise, we extract $\mathrm{ID}(u)$, $\mathrm{RT}(u)$ and $\mathrm{pnt}(u)$ from $\mathit{label}(u)$, and $\mathrm{ID}(v)$ from $\mathit{label}(v)$. Let T' be the subtree containing u in the last iteration of the process defining $\mathrm{ID}(u)$. (Refer again to Figure 1.) Then, by the definition of $\mathrm{ID}(u)$, we have $\mathrm{ID}(u) = \mathrm{ID}(T', u).\mathrm{ID}'(u)$, for some string $\mathrm{ID}'(u)$. Using $\mathrm{pnt}(u)$, we can extract $\mathrm{ID}(T', u)$ and $\mathrm{ID}'(u)$, in constant time. If it easy to see that if u is an ancestor of v, then $\mathrm{ID}'(u)$ is a *suffix* of $\mathrm{ID}(v)$, and then $\mathrm{ID}(v) = \mathrm{ID}(T', v).\mathrm{ID}'(u)$. If it is not, we use port 0 for the parent pointer. Otherwise, we can extract from $\mathrm{ID}(T', u)$ the pair (i_u, s_u), where i_u is the index of the heavy path to which u belongs, and s_u is the string corresponding to u in this path. Let $\bar{v}$ be the last heavy child on the path from the root of T' to v. We can also extract from $\mathrm{ID}(T', v)$ the corresponding pair (i_v, s_v), or triple (i_v, s_v, p_v). Here a triple means that v is light in T', and then p_v is the port leading from $\bar{v}$ to v

If $i_v \ne i_u$, we just use port number $\mathrm{RT}(u)[i_v]$. If $i_v = i_u$ and $s_u = s_v$, we know $\bar{v} = u$, and we use port number p_v. Finally, if $i_v = i_u$ and $s_u \ne s_v$, u is an ancestor of v if and only if $s_u \prec s_v$. If it is, the port number is 1, i.e. the port number of the edge

```
algorithm center(G, s)
A ← ∅ ; W ← V ;
while W ≠ ∅ do
{
    A ← A ∪ sample(W, s) ;
    C(w) ← {v ∈ V | δ(w, v) < δ(A, v)}, for every w ∈ V;
    W ← { w ∈ V | |C(w)| > 4n/s } ;
}
return A ;
```

Figure 2: Choosing a set of centers.

connecting u to its heaviest child; otherwise we use port 0 for the parent pointer. $\square$

This completes the proof of Theorem 2.1. The proof of Theorem 2.2 is similar and would appear in the full version.

3. STRETCH 3

We now present a stretch 3 routing scheme that uses only $\tilde{O}(n^{1/2})$ bits of memory at each vertex of the network. Our scheme improves upon a result of Cowen [8] that uses $\tilde{O}(n^{3/2})$ bits of memory at each vertex.

3.1 Centers and clusters

Let $G = (V, E)$ be an undirected graph with positive edge weights assigned to its edges. We let $n = |V|$ and $m = |E|$. If $u, v \in V$, we let $\delta(u, v)$ denote the (weighted) distance between u and v in the graph. Let $A \subseteq V$ be a subset of vertices referred to as *centers* (or *landmarks* in the terminology of Cowen [8]). We let $\delta(A, v) = \min\{\delta(u, v) \mid u \in A\}$. For every $w \in V$, we let

$$C_A(w) = \{ v \in V \mid \delta(w, v) < \delta(A, v) \}$$

be the *cluster* of w with respect to the set A. Note that if $w \in A$ then $C_A(w) = \emptyset$, as if $w \in A$ then $\delta(w, v) \geq \delta(A, v)$, for every $v \in V$. Also note, that $C_A(w) \cap A = \emptyset$, for every $w \in V$, as if $v \in A$ then $\delta(w, v) \geq \delta(A, v) = 0$. Clusters belonging to different vertices are not necessarily disjoint. Finally, let $\mathit{cent}_A(v)$ denote a vertex from A nearest to v. We occasionally use $\mathit{cent}(v)$ and $C(w)$ instead of $C_A(w)$ and $\mathit{cent}_A(v)$, when the set A is clear from the context.

Dor *et al.* [10] suggest the following way of getting a stretch 3 path from u to v: If $v \in C_A(u)$, use a shortest path from u to v. Otherwise, take a shortest path from u to $\mathit{cent}_A(v)$, and then a shortest path from there to v. Using symmetry and the triangle inequality we get, if $v \notin C_A(u)$, that $\delta(u, v) \geq \delta(A, v) = \delta(\mathit{cent}_A(v), v)$, and hence

$$\begin{aligned} &\delta(u, \mathit{cent}_A(v)) + \delta(\mathit{cent}_A(v), v) \\ \leq\ &\delta(u, v) + 2\delta(\mathit{cent}_A(v), v) \ \leq\ 3\delta(u, v) . \end{aligned}$$

Based on this, Cowen [8] showed that any set of centers $A \subseteq V$ gives rise to a routing scheme with stretch 3, where the size of the routing table at each vertex $w \in V$ is $O(|A| + |C_A(w)|)$. It is easy to pick the set A so that the average size of the clusters $C_A(w)$ is $n/|A|$. However, we want *all* routing tables to be of small, and hence *all* clusters need to be small.

Cowen [8] describes a way of efficiently finding, in every weighted undirected graph, a set $A \subseteq V$ of size $\tilde{O}(n^{2/3})$ such that $|C_A(w)| = \tilde{O}(n^{2/3})$, for every $w \in V$. As a result, she obtains a stretch 3 routing scheme that uses only $\tilde{O}(n^{2/3})$ memory at each vertex. We improve Cowen's construction by showing that every weighted undirected graph has a set of centers $A \subseteq V$ such that $|A| = \tilde{O}(n^{1/2})$ and $|C_A(w)| = \tilde{O}(n^{1/2})$ for every $w \in V$.

It is shown in [25] that if we just pick A randomly of size s, we get an expected average cluster size of n/s. Our contribution here is the formulation and analysis of a recursive sampling algorithm that brings the size of *all* clusters down to $\tilde{O}(n/s)$.

Our recursive sampling algorithm **center**(G, s) is presented in Figure 2. It receives a weighted undirected graph $G = (V, E)$ and a parameter $1 \leq s \leq n$. We shall show that the algorithm returns a set $A \subseteq V$ of *expected* size $O(s \log n)$ such that all the clusters $C_A(w)$, for $w \in V$ are of size at most $4n/s$.

The algorithm **center**(G, s) uses a subroutine **sample**(W, s) that receives a set W and returns a random subset of W obtained by selecting each element, independently, with probability $s/|W|$. If $|W| \leq s$, then **sample**(W, s) returns the set W itself. The expected size of the sample, if $|W| \geq s$, is therefore s. Algorithm **center**(G, s) begins by letting $A \leftarrow \emptyset$ and $W \leftarrow V$. As long as W is not empty, the algorithm chooses a random sample containing, on average, s elements from W, and adds it to the set A. It then recomputes the clusters $C(w) = C_A(w)$, for every $w \in V$, and redefines W to be the set of vertices $w \in V$ whose cluster $C(w)$ is too large, i.e., of size greater than $4n/s$. A random sample is then chosen from this set, and so on. We claim:

THEOREM 3.1. *The expected size of the set A returned by algorithm* **center**(G, s) *is at most $2s \log n$. For every $w \in V$ we then have $|C_A(w)| \leq 4n/s$.*

The proof of Theorem 3.1 is facilitated by the introduction of the following definition. The *bunch* $B_A(v)$ of a vertex $v \in V$ with respect to the set A is defined as follows:

$$B_A(v) = \{ w \in V \mid \delta(w, v) < \delta(A, v) \} .$$

It is easy to see that $w \in B_A(v)$ if and only if $v \in C_A(w)$. Thus, bunches, in some sense, are the *inverses* of the clusters. It is easy to see that if $A' \subseteq A$, then for every $w \in V$ we have $C_A(w) \subseteq C_{A'}(w)$, and that for every $v \in V$ we have $B_A(v) \subseteq B_{A'}(v)$. The proof of Theorem 3.1 relies on the following lemma:

LEMMA 3.2. *Let $W \subseteq V$, let $1 \leq s \leq n$, and let $A' \leftarrow$* **sample**(W, s)*, i.e., A' is obtained by including each element of W, independently, with probability $s/|W|$, or $A' = W$ if $|W| \leq s$. Then, for every $v \in V$, we have $E[|B_{A'}(v) \cap W|] \leq |W|/s$.*

PROOF. Let $v \in V$. If $|W| \leq s$, then $A' = W$ and $B_{A'}(v) \cap W = \emptyset$. Otherwise, let $w_1, w_2, \ldots, w_\ell$ be the elements of W in non-decreasing order of distance from v. If $w_i \in A$, then $B_{A'}(v) \cap W \subseteq \{w_1, w_2, \ldots, w_{i-1}\}$. Thus, the size of $B_{A'}(v) \cap W$ is stochastically dominated by the index of the first element in this sequence that belongs to A'. This index is a geometric random variable with success probability $s/|W|$, so its expectation is $|W|/s$. $\square$

We now turn to the proof of Theorem 3.1

PROOF. (**of Theorem 3.1**) The termination condition of the while loop ensures that $|C_A(w)| \leq 4n/s$, for every $w \in V$. All that remains, therefore, is to show that the while loop does indeed terminate, and to bound the expected size of the set A. We show that the expected number of iterations performed is at most $2 \log n$. In each iteration, we add to A a sample of expected size s. The expected size of A at the end of the process is therefore at most $2s\log n$, as required. To show that the expected number of iterations is at most $2\log n$, we show that in each iteration, with a probability of at least $1/2$, the size of W is decreased by a factor of at least 2.

Let W_i be the set W at the beginning of the i-th iteration. Let $A' \leftarrow$ **sample**(W_i, s) be set of elements added to A during this iteration. By Lemma 3.2, $E[B_A(v)\cap W_i] \le E[|B_{A'}(v)\cap W_i|] \le |W_i|/s$, for every $v\in V$. Thus

$$E[\sum_{w\in W_i} |C_A(w)|] \;=\; E[\sum_{v\in V} |B_A(w)\cap W_i|] \le \frac{n|W_i|}{s} \;.$$

By Markov's inequality, with a probability of at least $1/2$, we have $\sum_{w\in W_i} |C_A(w)| \le \frac{2n|W_i|}{s}$. We call iterations in which this happens *successful* iterations. (We can check whether this condition holds and choose a new sample $A' \leftarrow$ **sample**(W_i, s) if it does not. This would reduce the size of A by a constant factor but would slightly complicate the algorithm.)

Let W_{i+1} be the set of elements whose clusters, at the end of the i-th iteration, are of size at least $4n/s$. As the size of a cluster cannot increase when elements are added to A, we must have $W_{i+1}\subseteq W_i$. If the i-th iteration is successful, then

$$\frac{4n|W_{i+1}|}{s} \le \sum_{w\in W_i} |C_A(w)| \le \frac{2n|W_i|}{s} \;,$$

and thus $|W_{i+1}| \le |W_i|/2$. This completes the proof. □

Using some simple techniques from [25], it is possible to implement algorithm **sample**(G, s) so that its expected running time would be $O(mn\log n/s)$, where m is the number of edges of G. It is also possible to *derandomize* it, though increasing the running time to $O(mn)$. We omit the details from this extended abstract.

By calling **sample**(G, s) with $s = (n/\log n)^{1/2}$, we get the following corollary:

COROLLARY 3.3. *Let $G = (V, E)$ be a weighted undirected graph. It is possible, in $O(m\sqrt{n\log n})$ expected time, or $O(mn)$ worst-case time, to find a set A such that $|A| \le 2(n\log n)^{1/2}$ and $|C_A(w)| \le 4(n\log n)^{1/2}$, for every $w\in V$.*

3.2 The routing scheme

The stretch 3 routing scheme with local routing tables of size at most $O((n\log n)^{1/2})$ is obtained by using the set of centers A of size $O((n\log n)^{1/2})$ that induces clusters of size $O((n\log n)^{1/2})$ (see Corollary 3.3) in conjunction with Cowen's stretch 3 routing scheme [8]. Below, we review Cowen's routing scheme, based on our set A, and later we show how the header size can be reduced from $O(\log n)$ bits to $(1+o(1))\log_2 n$ bits if we are free to rename vertices and port numbers.

Recall that for every $v\in V$, $cent(v)\in A$ is a vertex satisfying $\delta(cent(v), v) = \delta(A, v)$. For every two vertices $u, v\in V$, let $port(u, v)$ be the port corresponding to the first edge on a shortest path from u to v. We assume that ports have consecutive numbers $0, 1, 2, \ldots, \deg(u)-1$, hence that they can be described using at most $\lceil \log_2 n\rceil$ bits. Each vertex $v\in V$ is assigned a label $label(v) = (v, cent(v), port(cent(v), v))$. As stated, each label is $\lceil 3\log n\rceil$-bit long. We improve on this later.

At each vertex $w\in V$ we keep a *2-level hash table* TAB_w, that holds for each $v\in A\cup C(w)$ the pair $(v, port(w, v))$. The size of this table is $O(|A| + |C(w)|)$ and for each $v\in A\cup C(w)$, the pair $(v, port(w, v))$ can be located in *worst-case* constant time. The implementation of such 2-level hash tables is described in Fredman *et al.* [16]. A deterministic construction of such hash tables is presented by Alon and Naor [2]. We shall denote by $\mathrm{TAB}_w(v)$ a probe into this table that returns $port(w, v)$, if $v\in A\cup C(w)$, and returns $\perp$ (i.e., not found), if $v\notin A\cup C(w)$.

A packet sent to destination v would carry $label(v)$ in its header. This header will *never* be changed. The routing decision at each vertex is then extremely simple. Suppose that a packet with $label(v)$ at its header reaches a vertex $w\in V$. (Initially $w = u$, where u is the source of the message.) If $w = v$ then the packet reached its destination. Otherwise, if $v\in A\cup C(w)$, then the packet is directly routed towards v via $port(w, v)$. This port can be obtained, in constant time, by accessing TAB_w. Otherwise, we extract $cent(v)$ from $label(v)$ and check whether $w = cent(v)$, i.e., whether w is the center closest v. If $w = cent(v)$, we again route directly towards v using $port(w, v) = port(cent(v), v)$. This port number is also extracted from $label(v)$. Finally, if none of the previous cases applies, the packet is routed towards $cent(v)$ on $port(v, cent(v))$. Since $cent(v)\in A$, this port number can be obtained, in constant time, by accessing TAB_w.

The correctness of the above algorithm is proved in [8]. We want to show that when a packet stops moving along a shortest path towards $cent(v)$, it will keep moving on a shortest path towards v. This follows from two observations: (1) if $v\in C(w)$, we follow an edge (w, w') on a shortest path towards v, but then $\delta(w', v) < \delta(w, v) \le \delta(A, v)$, so $v\in C(w')$, and we would continue directly towards v. (2) when we use $port(cent(v), v)$ from $cent(v)$ to get to a vertex w', we have $\delta(w', v) < \delta(cent(v), v) = \delta(A, v)$.

3.3 Reducing the header size

To reduce the header size to $(1+o(1))\log_2 n$, we do as follows. For every center $w\in A$, we let $N(w)$ be the set of vertices for whom w is the closest center. The centers are enumerated in a non-increasing order of how many vertices they are centers for. Now, the ports of w are enumerated in non-increasing order of the number of vertices from $N(w)$ they lead to. Let $N(w, i)$ be the set of vertices reached via port i. The vertices in $N(w, i)$ are now enumerated locally in any order. The above scheme assigns three indices to each vertex. These three indices identify the vertex uniquely. We will use the variable length encoding from Lemma 2.3 for each of these indices.

The enumeration orders imply that the number assigned to w is at most $n/|N(w)|$, and that port i leads to at most $|N(w)|/|N(w, i)|$ vertices, each of which has a local number below $|N(w, i)|$. Hence the total size is

$$\begin{aligned}(1+o(1))(\log_2(n/|N(w)|) + \log_2(|N(w)|/|N(w,i)|)\\ + \log_2(|N(w,i)|)) = (1+o(1))\log_2 n\;.\end{aligned}$$

Alternatively, we can use the technique of Eilam *et al.* [11] to get labels in the range $\{0, 1, \ldots, n-1\}$. We enumerate one set $N(w)$ at the time, in order of decreasing size of $|N(w)|$. The routing decision is then a predecessor search in a universe of size $O(n)$, and can be done in $O(\log\log n)$ using van Emde Boas's data structure [26],[27]. Summing up, we have shown

THEOREM 3.4. *The routing scheme defined always routes packages on simple paths of stretch at most 3, using routing tables of size $O(\sqrt{n\log n})$ and labels of size $(1+o(1))\log_2 n$. Each routing decision made between source and destination takes constant time. We can also use consecutive labels getting routing decisions in $O(\log\log n)$ time.*

4. STRETCH $2K-1$ USING HANDSHAKING

We start be giving a routing oriented formulation of some of our recent results from [25]. We then combine these results with our new tree routing schemes to obtain new routing schemes for general graphs.

In [25] we describe an efficient algorithm that, given an integer parameter $k \geq 1$, produces a tree cover, i.e., a family of induced subtrees, such that each vertex v is contained in $O(n^{1/k} \log^{1-1/k} n)$ trees and such that each pair of vertices is connected by a stretch $2k-1$ path in one of these trees. From a routing perspective, the challenge would be to identify this tree. We could then route using the techniques of Section 2.3.

Each tree of the cover is a shortest path tree from some source to the nodes it spans. No two trees have the same source. Each vertex v stores, in a hash table, the set B_v of its sources, i.e., the sources of the trees containing v.

In addition, each vertex v has a list cent_v of k special sources, called its *centers*. (In the notation of [25], $\mathit{cent}_v[0] = v$ and $\mathit{cent}_v[i] = p_{k-i}(v)$.) For any pair of vertices (u, v), let $i(u, v)$ be the smallest i such that $\mathit{cent}_v[i] \in B_u$, and let $w(u, v) = \mathit{cent}_v[i(u, v)]$. It follows easily from the results of [25], that

$$\begin{array}{c} i(u,v) \leq i(v,u) \quad \Rightarrow \\ \delta(u, w(u,v)) + \delta(w(u,v), v) \;\leq\; (2k-1)\delta(u,v) \end{array} \tag{1}$$

Thus $w(u, v)$ is a source of a tree with a u-v path of the desired stretch. In Appendix C we show that, in any case:

$$\delta(u, w(u,v)) + \delta(w(u,v), v) \;\leq\; (4k-3)\delta(u,v) \tag{2}$$

We let cent_v be part of the label of v. Then, with access to the label of v, u can always compute $w(u, v)$, thus giving it, by (2), an immediate source for stretch $4k-3$ routing. However, to get stretch $2k-1$ routing, using (1), u has to initiate a *handshaking* process in which it asks v for $i(v, u)$ and $w(v, u)$, both of which v can compute using u's label. The handshaking could be done using the stretch $4k-3$ scheme. Since we typically want to send a stream of packets from u to v, the overhead of the handshaking would usually be negligible.

We still need to describe, more precisely, how to perform the routing in the selected tree. We assign the vertices of each tree of the tree cover $o(\log^2 n)$-bit labels, as described in Theorem 2.2. Note that we cannot specialize the port numbers because each link may be contained in many trees. If $w \in B_v$, let $\mathit{tree\text{-}label}_v(w)$ be the label of v in the tree whose source is w. We can store $\mathit{tree\text{-}label}_v$ in connection with the hash table for B_v. Also, we add a list $\mathit{cent\text{-}tree\text{-}label}_v$ of length k to the label of v with $\mathit{cent\text{-}tree\text{-}label}_v[i] = \mathit{tree\text{-}label}_v(\mathit{cent}_v[i])$.

Now, the header attached to packets sent from u to v would be of the form $(w, \mathit{tree\text{-}label}_v(w))$, where w is the source of the tree in which we want to route. When the packet arrives a vertex x, it can compute the appropriate port number using $\mathit{tree\text{-}label}_v(w)$ and $\mathit{tree\text{-}label}_x(w)$, the latter of which is available locally.

For our stretch $4k-3$ scheme, u just computes the header

$$(w(u,v), \mathit{cent\text{-}tree\text{-}label}_v[i(u,v])$$

and sends the packet. For our stretch $2k-1$, we use the handshaking described in full in Figure 3. The $4k-3$ scheme can be combined with ideas from the previous section giving a quite similar $4k-5$ scheme. Details are given, due to lack of space, in Appendix C. We conclude:

THEOREM 4.1. *Let $k > 1$ be integer. The above routing schemes use $\tilde{O}(n^{1/k})$-bit space at the routers, $o(k \log^2 n)$-bit labels and $o(\log^2 n)$-bit headers. They route directly with stretch $4k-5$. Moreover, after one round of handshaking, in which only $o(\log^2 n)$ bits are exchanged, they route with stretch $2k-1$.*

`Protocol` **handshaking**

1. Vertex u computes $i(u, v)$ and $w(u, v)$, and sends them to v.
2. Upon receipt, v computes $i(v, u)$ and $w(v, u)$, and sets $w \leftarrow w(v, u)$ if $i(v, u) < i(u, v)$, and $w \leftarrow w(u, v)$, otherwise. Now v sends w and $\mathit{tree\text{-}label}_v(w)$ to u.
3. Upon receipt, u creates the header $(v, \mathit{tree\text{-}label}_v(w))$ and sends the packets.

Figure 3: One round handshaking.

5. CONCLUDING REMARKS AND OPEN PROBLEMS

We presented very efficient routing schemes for trees and for general graphs. Perhaps the most interesting open problem remaining is the following: is there a stretch $2k-1$ routing scheme with routing tables of size $\tilde{O}(n^{1/k})$ and headers of size $O(\log n)$ that does not use handshaking?

6. REFERENCES

[1] S. Abiteboul, T. Milo, and H. Kaplan. Compact labeling schemes for ancestor queries. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms, Washington, D.C.*, pages 547–556, 2001.

[2] N. Alon and M. Naor. Derandomization, witnesses for boolean matrix multiplication, and construction of perfect hash functions. *Algorithmica*, 16:434–449, 1996.

[3] S. Alstrup. Personal communication, SODA 2001.

[4] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *Journal of Algorithms*, 11(3):307–341, 1990.

[5] B. Awerbuch and D. Peleg. Routing with polynomial communication-space trade-off. *SIAM Journal on Discrete Mathematics*, 5(2):151–162, 1992.

[6] P. Beame and F. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31th Annual ACM Symposium on Theory of Computing, Atlanta, Georgia*, pages 295–304, 1999.

[7] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley, 1991.

[8] L. Cowen. Compact routing with minimum stretch. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, Maryland*, pages 255–260, 1999. Journal version to appear in Journal of Algorithms.

[9] S. Deering and R. Hinden. Internet protocol, version 6 (IPv6), specification. Network Working Group, Request for Comments: 2460, `ftp://ftp.ipv6.org/pub/rfc/rfc2460.txt`, December 1998.

[10] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.

[11] T. Eilam, C. Gavoille, and D. Peleg. Compact routing schemes with low stretch factor. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing, Puerto Vallarta, Mexico*, pages 11–20, 1998.

[12] P. Erdős. Extremal problems in graph theory. In *Theory of Graphs and its Applications (Proc. Sympos. Smolenice, 1963)*, pages 29–36. Publ. House Czechoslovak Acad. Sci., Prague, 1964.
[13] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proc. ACM SIGCOMM'99: Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 251–262, 1999.
[14] P. Fraigniaud and C. Gavoille. Memory requirements for univesal routing schemes. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, Ontario, Canada*, pages 223–230, 1995.
[15] P. Fraigniaud and C. Gavoille. Interval routing schemes. *Algorithmica*, 21(2):155–182, 1998.
[16] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.
[17] C. Gavoille. A survey on interval routing. *Theoretical Computer Science*, 245(2):217–253, 2000.
[18] C. Gavoille. Personal communication at SODA, 2001.
[19] C. Gavoille and M. Gengler. Space-efficiency of routing schemes of stretch factor three. *Journal of Parallel and Distributed Computing*, 2000. To appear.
[20] C. Gavoille and S. Pérennès. Memory requirements for routing in distributed networks. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania*, pages 125–133, 1996.
[21] B. Kernighan and D. Ritchie. *The C programming language*. Prentice Hall, second edition, 1988.
[22] D. Peleg. *Distributed computing – A locality-sensitive approach*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000.
[23] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, 1989.
[24] N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 28(1):5–8, 1985.
[25] M. Thorup and U. Zwick. Approximate distance oracles. In *Proceedings of the 33th Annual ACM Symposium on Theory of Computing, Crete, Greece*, 2001. To appear.
[26] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.
[27] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.
[28] J. van Leeuwen and R. Tan. Computer networks with compact routing tables. In G. Rozemberg and A. Salomaa, editors, *The book of L*, pages 259–273. Springer-Verlag, 1986.

APPENDIX

A. A VERY COMPACT, YET PRACTICAL, ROUTING SCHEME FOR TREES

We show now that if we are allowed to choose the port numbers of the links, then the size of the headers can be reduced from $O(\log n)$ words, i.e., $O(\log^2 n)$ bits, to only $O(\log n)$ bits.

We slightly change now the definition of heavy vertices. The *weight* s_v of a vertex v is the number of its descendants in the tree. Each non-leaf vertex would now have a single heavy child which is its child with the highest weight, ties broken arbitrarily. If v is a non-leaf vertex, we let v' be its heavy child, and $v_0, v_1, \ldots, v_{d-1}$ be its light children in decreasing (or rather non-increasing) order of weight, i.e., $s_{v'} \geq s_{v_0} \geq \cdots \geq s_{v_{d-1}}$. It is easy to see that $s_{v_i} \leq s_v/(i+2)$, for $0 \leq i < d$. Assign the edge (v, v_i), for $0 \leq i < d$, port number i, and assign the edge (v, v') port number d.

As in the previous subsection, let ℓ_v be the light level of v, and let $L_v = (q_1, q_2, \ldots, q_{\ell-1})$ be the port numbers of the edges leading to the light vertices on the path from r to v. It is easy to see that we now have $\Pi_{i=0}^{\ell-1}(q_i+2) \leq n$. Instead of storing each port number q in a separate word, we now use only $\lfloor \log_2 q \rfloor + 1$ bits, or a single bit if $q = 0$, and concatenate all these bit strings. Thus, for example, the sequence $(2, 0, 5, 3)$ would yield the string `11'101'0'10`. The quotes are, of course, not part of this sequence and were added for illustration purposes only. Also note that we have reversed the order of the elements, this would come out handy later. Instead of the quotes, we use a *mask*. Each '`1`' in this mask would mark the end of a string representing a number. Thus, the mask corresponding to our string above would be `10'100'1'10`. (Again, without the quotes.) To each vertex v we therefore attach a bit string L_v and a masking bit string M_v. The length of each one of them is

$$\sum_{i=1}^{\ell-1}(\lfloor \log_2 q_i \rfloor + 1) \;\leq\; \alpha \sum_{i=1}^{\ell-1} \log_2(q_i+2) \;\leq\; \alpha \log_2 n \;,$$

where $\alpha = \max_q(\lfloor \log_2 q \rfloor + 1)/\log_2(q+2)$. It is not difficult to check that $\alpha = 4/\log_2 10 \simeq 1.20412$, where the maximum is attained at $q = 8$. We let $\mathit{label}(v) = (v, L_v, M_v)$ be the new label attached to v. Thus, instead of $O(\log n)$ words, we now need only one $(\log_2 n)$-bit word and two $(1.21 \log_2 n)$-bit words. Each one of these would, most likely, fit into one machine word, and if not into two.

When a packet with header (v, L_v, M_v) reaches w, a vertex of light level ℓ_w, we may have to extract the ℓ_w-th number coded in L_v. We only have to do that, however, when v is a descendant of w, in which case we know that the first $\ell_w - 1$ numbers coded in L_v are exactly the same as those in L_w. We store with w the total length k_w of these $\ell_w - 1$ numbers. Using a well known programming trick, we can now easily extract the right number. The only change needed in the code given in the previous subsection is replacing `L[l]` by

```
(L>>k)&((M>>k)^((M>>k)-1) ,
```

where `L` $= L_v$ and `M` $= M_v$ are extracted from the header, and `k` $= k_w$ is a new piece of information stored locally at w. The resulting code is still extremely fast, and the savings in the size of the headers is substantial.

The combined length of L_v and M_v can be reduced to only $2\log_2 n$ bits by noticing that the most significant bits of the strings used in L_v are redundant for $q \geq 2$. (In our example above, we could have used the bit string `1'01'0'0`, together with the original mask `10'100'1'10`.) The C expression can be easily adapted to this change but would be somewhat slower. In the next subsection, we will take things to an extreme, showing that the length of the labels could be reduced to $(1 + o(1))\log_2 n$, but this seems to require additional ideas.

From a practical perspective, using $3.4 \log_2 n$ bits for the header should be easily supported in hardware. The next generation IP protocol [9] suggests 128-bit IP addresses, meaning that the next generation routers will be tuned for this large headers. Our scheme will then work for trees with more than 2^{37} nodes, which is plenty for any foreseeable future.

B. ENCODING VARIABLE LENGTH STRINGS

PROOF. **(of Lemma 2.3)** Given an integer $i \geq 0$, we let $\text{bin}(i)$ be the bit string containing the binary representation of i, i.e., $\text{bin}(0) = 0$, $\text{bin}(1) = 1$, $\text{bin}(2) = 10$, etc. We let $\ell(i) = |\text{bin}(i)|$, for $i \geq 0$. It is not difficult to check that $\ell(0) = 1$, and $|\ell(i)| = \lfloor \log_2 i \rfloor + 1$, for $i > 0$. If $k \geq \ell(i)$, we let $\text{bin}(k, i) = 0^{k-\ell(i)}\text{bin}(i)$, i.e., the binary representation of i padded with leading 0's, if necessary, to make a string of length k. We also let $m(i) = 2^{\lceil \log_2 \ell(i) \rceil}$, for $i \geq 0$. Note that $\ell(i) \leq m(i) \leq 2\ell(i)$, for every $i \geq 0$. For brevity, we let $||s|| = m(|s|)$. We then define

$$\text{code}(s) = s \,.\, \text{bin}(||s||, |s|) \,.\, \text{bin}(||s||, ||s||) \ .$$

In other words, $\text{code}(s)$ is the concatenation of three strings: (i) the string s itself; (ii) the binary representation of $|s|$, the length of s; and (iii) the binary representation of $||s||$, the number of bits in the binary representation of the length of $|s|$, rounded up to the next power of 2. The definition of $\text{code}(s)$ includes padding that is used to speed up the decoding. From the information theoretic point of view, we could have used the more compact definition $\text{code}'(s) = s \,.\, \text{bin}(\ell(|s|), |s|) \,.\, \text{bin}(||s||)$.

It is easy to see that $|\text{code}(s)| = |s| + 2m(|s|) = |s| + 2 \cdot 2^{\lceil \log_2(\lfloor \log_2 |s| \rfloor + 1) \rceil} \leq |s| + 4(\log_2 |s| + 1)$, for $|s| \geq 1$, and $|\text{code}(\varepsilon)| = |0100| = 4$. (Note that $\ell(0) = 1$ and $m(0) = 2$.)

Suppose now that we have a bit string $t = s'.\text{code}(s)$, where s' is some other bit string. To extract s, we note that $||s||$ is a power of 2. Thus it is easily identified by finding the first 1 in t. We can now ignore the first $||s||$ bits of the string (including the $\ell(||s||)$ bits used to represent $||s||$). The next $||s||$ bits then contain $|s|$, the length of s. Finally, having $|s|$, we can extract s, and know exactly where $\text{code}(s)$ ends.

Suppose that x is a machine word containing the string $t.\text{code}(s)$ in its least significant portion. Then `a`$= ||s||$, `b`$= |s|$ and `s`$= s$ could be obtained using the following short sequence of C instructions:

```
#define suffix(x,i) (x & ((2<<i)-1))

a=x&-x; x=x>>a; b=suffix(x,a);
x=x>>a; s=suffix(x,b); x=x>>b;
```

Also note that after this sequence we have `x`$= t$. □

C. STRETCH $4K - 3$ AND $4K - 5$ WITHOUT HANDSHAKING

We start by presenting a proof of equation (2) on which the stretch $4k - 3$ routing scheme is based. We then sketch the details of a stretch $4k - 5$ routing scheme.

LEMMA C.1. *Given $(u, v) \in V^2$, let i be the least index such that $\mathit{cent}_v[i] \in B_u$. Then $\delta(u, \mathit{cent}_v[i]) + \delta(\mathit{cent}_v[i], w) \leq (4k - 3)\delta(u, v)$.*

PROOF. In the notation of [25], we have $\mathit{cent}_x[i] = p_{k-i}(x)$. From the basic definitions in [25], we shall need for all $u, v \in V$ and $i \leq k - 1$:

- if $\mathit{cent}_v[i] \notin B_u$, then $\delta(u, \mathit{cent}_u[i + 1]) \leq \delta(u, \mathit{cent}_v[i])$.
- $\delta(u, \mathit{cent}_u[i]) \leq \delta(u, \mathit{cent}_v[i])$.
- $\mathit{cent}_v[k - 1] \in B_u$, so $i \leq k - 1$.

Let $\Delta = \delta(u, v)$. By induction, we will now prove

$$\delta(v, \mathit{cent}_v[i]) \ \leq \ 2i\Delta \qquad (3)$$

$$\delta(u, \mathit{cent}_v[i]) \ \leq \ (2i + 1)\Delta \qquad (4)$$

Since $i \leq k - 1$, (3) and (4) imply the statement of the lemma.

For the base case with $i = 0$, we have $\mathit{cent}_v[0] = v$, so (3) follows. Also, (3) implies (4) by triangle inequality for all i. Hence, assuming (3) and (4) for i, we just need to prove (3) for $i + 1$.

The condition for considering $i + 1$ is that $\mathit{cent}_v[i] \notin B(u)$. But then $\delta(u, \mathit{cent}_u[i + 1]) \leq \delta(u, \mathit{cent}_v[i])$ Hence,

$$\begin{aligned} \delta(v, \mathit{cent}_v[i + 1]) &\leq \delta(v, \mathit{cent}_u[i + 1]) \leq \Delta + \delta(u, \mathit{cent}_u[i]) \\ &\leq \ \Delta + \delta(u, \mathit{cent}_v[i]) \leq (2i + 2)\Delta \ , \end{aligned}$$

as desired. The last inequality followed inductively from (4). □

In order to get the $4k - 5$ bound, we have to combine with techniques from Section 3. In [25], there is a subset A_{k-1} of the vertices, including each vertex independently with probability $p = 1/n^{1/k}$, hence with A_{k-1} of expected size $n^{1-1/k}$. If we instead apply the recursive sampling technique from Section 3 with $s = n^{1-1/k}$, we get what corresponds to a superset A^*_{k-1} of A_{k-1} of expected size $O(n^{1-1/k} \log n)$. Except for a log-factor in space, the extension of A_{k-1} to A^*_{k-1} does not affect the analysis of [25].

However, by Theorem 3.1, we now have $|C_{A^*_{k-1}}(u)| = O(n^{1/k})$ for each vertex u. We can therefore, using hashing, store $C_u = C_{A^*_{k-1}}(u)$ with each vertex u. The tree cover produced by the algorithm in [25] will contain a shortest path tree sourced in u, spanning at least C_u, and possibly more if $u \in A^*_{k-1}$. For each destination $v \in C_u$, we just want to route, with stretch 1, using the shortest path in this tree, and hence, with u, we store $\mathit{tree\text{-}label}_v(u)$ for each $v \in C_u$.

The stretch $4k - 5$ algorithm is now a simple augmentation of the stretch $4k - 3$ algorithm from Section 4. When u wants to send packets to v, if $v \in C_u$, we use header $(u, \mathit{tree\text{-}label}_v(u))$; otherwise, if $v \notin C_u$, we use $(w(u, v), \mathit{cent\text{-}tree\text{-}label}_v[i(u, v)])$ exactly as in Section 4.

The claimed stretch of $4k - 5$ follows directly from the lemma below.

LEMMA C.2. *Given $(u, v) \in V^2$ with $v \notin C_u$. Let i be the least index such that $\mathit{cent}_v[i] \in B_u$. Then*

$$\delta(u, \mathit{cent}_v[i]) + \delta(\mathit{cent}_v[i], w) \leq (4k - 5)\delta(u, v) \ .$$

PROOF. With the exception of the base case, the proof is identical to that of Lemma C.1. From the definitions in [25], the condition $v \notin C_u$ implies that $\delta(v, \mathit{cent}_v[1]) \leq \Delta = \delta(u, v)$, whereas in Lemma C.1, we had $\delta(v, \mathit{cent}_v[1]) \leq 2\Delta$. In the induction, this means that we save Δ in both (3) and (4), and hence we save 2Δ in the final bound, reducing $(4k - 3)\Delta$ to $(4k - 5)\Delta$. □