Originally published in *Algorithms for Sensor and Ad Hoc Networks*, volume 4621 of *LNCS*, pages 187–202. Springer, 2007.

Compact Routing

Michael Dom

Institut für Informatik, Friedrich-Schiller-Universität Jena,
Ernst-Abbe-Platz 2, D-07743 Jena, Germany.
dom@minet.uni-jena.de

1 Introduction

In a distributed network, the delivery of messages between its nodes is a basic task, and a mechanism is required that is able to deliver packages of data from any node of the network to any other node. To this end, usually a distributed algorithm runs on every node of the network, which properly forwards incoming packages, employing some information that is stored in the local memory of the node.

In sensor and ad-hoc networks, the resources of the nodes are typically strongly limited—in particular, small memory sizes are common. However, the number of nodes in these networks can be very high, such that it is normally too expensive to use $\Omega(n)$ bits of local memory in each node: Storing at each node a routing table that contains for every destination node one entry of routing information is not possible. Therefore, distributed algorithms and data representations are required that allow to use only $o(n)$ bits of local memory, typically at the cost of increasing the lengths of the routing paths. *Compact routing* is the field of research that deals with the theoretical basics of this problem. It analyzes how "good" routing algorithms can be when the available resources, such as memory and computational power, are limited, and provides the fundamentals for memory-efficient routing protocols.

The model that is used assumes that a distributed algorithm, the so called *routing scheme*, runs at each node of the network and decides for any incoming data package to which immediate neighbor of the node the package shall be forwarded in order to make it reach its destination.

The network is usually considered as an undirected and edge-weighted graph with labeled nodes in which every edge has a number—the *port number*—at each of its endpoints. Each data package has a *header* which contains information about the destination of the package—for example the label of the destination node—and possibly additional information that is used during the routing. A node that receives a data package has to check whether the node itself is the destination of the package. If this is not the case, it has to send it immediately to another node in such a way that the package eventually reaches its destination.

The decision process which determines in which direction a data package has to be forwarded can be regarded as a *routing function*, which takes as input the current node, the port number of the edge through which the data package has reached the node, and the header of the data package, and which computes two outputs: the port number of the edge through which the package has to be sent and, if the model of the network allows to change headers of data packages during the routing, a new header for the package. A routing scheme is the implementation of a routing function.

Most routing schemes need some information, which is stored in local *routing tables*, and these tables have to be initialized before any data packages can be routed. Hence, in order to run a distributed network, a *routing strategy* is needed, which takes care of all tasks associated with routing and which consists of

– a global preprocessing algorithm, which initializes the local data structures of all nodes and which—if this is allowed by the model of the network—assigns labels to the nodes and port numbers to the edges (otherwise, the node labels and port numbers, respectively, are part of the input graph and cannot be changed), and
– a distributed algorithm, called the routing scheme, which implements an adequate routing function.

Consider now the following straightforward routing strategy: In the preprocessing phase, the shortest paths between all pairs of nodes are computed, and at each node v of the network a routing table is stored which contains for each node w of the network the port number of the edge leading from v to the next node on the shortest path from v to w. If a data package whose header contains the label of a node w as destination address arrives at a node v, the routing algorithm at v searches in the local routing table of v for the entry belonging to w and sends the package through the edge determined by the port number found in the table. This routing strategy needs no rewritable package headers and it routes every data package on the shortest path to its destination.

However, each such routing table needs $n \log(\deg(v))$ bits of local memory space in every node v, where n is the number of nodes in the network and $\deg(v)$ is the number of edges incident to v. In a large network, such a demand for memory can be too expensive, and more intricate routing strategies are needed. A routing strategy (and also its routing scheme) is called *compact* if, for a network consisting of n nodes, it needs only $o(n)$ bits of local information at each node (i.e., it needs *less* than $c \cdot n$ bits of local memory at each node for every constant c if n is big enough).

In order to save memory space, one often accepts that packages are routed towards their destination not on the shortest possible path. Therefore, besides the memory consumption, one of the most important performance measurements of a routing strategy is its *stretch*, which is defined as the maximum ratio, taken over all node pairs (v, w), between the length of the path a data package between v and w is routed and the length of the shortest path between v and w in the network. The trade-off between the memory space that is needed and the maximum stretch guaranteed by a routing strategy is a very extensively studied topic in the area of routing in networks.

Of course, there are also other properties of routing strategies that have to be optimized and that are considered in literature. The most common performance measurements for routing strategies are

– the memory space needed in each node (called *local memory*),
– the total memory space used by all nodes (called *total memory*),
– the stretch,
– the sizes of the addresses and package headers,
– the time needed to compute the routing function (called *routing time* or *latency*), and
– the time needed for the preprocessing.

Note that the memory consumption of a routing strategy cannot be considered without also considering the sizes of the addresses and package headers, because if routing strategies are allowed to assign arbitrary addresses to the nodes, the addresses can be used for storing routing information.

In this chapter, we will first give an overview over some important results and then describe three exemplary routing strategies.

2 Definitions

A network is modeled as an undirected, connected graph $G = (V, E)$ with $n := |V|$, $m := |E|$ and a weight function $r : E \to \mathbb{R}^+$ representing distances between the nodes that are connected by edges. (Sometimes networks are also modeled as unweighted graphs, i.e., $\forall e \in E : r(e) = 1$.) The nodes of the graph are labeled with numbers or, more generally, with arbitrary data types that serve as *addresses*. We denote such an address of a node v with $a(v)$. Moreover, there exists a numbering $p : \{(v, w) \mid \{v, w\} \in E\} \to \mathbb{N}$ representing the *port numbers*: The edge $\{v, w\}$ is labeled with $p(v, w)$ at its endpoint v and with $p(w, v)$ at its endpoint w. The node addresses and port numbers are either fixed (and, therefore, part of the input for the preprocessing algorithm of the routing strategy) or can be assigned arbitrarily by the routing strategy, which can possibly lead to more efficient routing schemes. In order to prevent port numbers from being abused for storing hidden routing information, the port numbers of the edges incident to a node v must lie between 1 and $\deg(v)$, where $\deg(v)$ stands for the number of neighbors of v, i.e., $\forall \{v, w\} \in E : 1 \le p(v, w) \le \deg(v)$.

To every data package a *header* is attached containing some information (e.g., the address of the destination node), which is used by the routing scheme. Some routing schemes need rewritable headers, that is, the headers can be modified by the nodes that route the data packages.

A *shortest path* between two nodes v and w is a path where the sum of the edge weights is minimum; this sum is denoted as the *distance* $d(v, w)$ between v and w. The *stretch* of a routing scheme is defined as

$$\max_{v,w \in V} \frac{\sum_{e \in P(v,w)} r(e)}{d(v, w)},$$

where $P(v, w)$ is the edge set traversed by a package that is routed by the routing scheme from v to w.

While some routing strategies can only provide routing schemes for special classes of graphs, a *universal* routing strategy works on every arbitrary graph.

A *direct* routing function depends only on the address of the destination node (and, of course, on the current node)—this makes direct routing schemes (i.e., routing schemes implementing direct routing functions) quite simple. A direct routing scheme does not change the package header (which only contains the address of the destination node), which implies that it has to route any package on a loop-free path (i.e., a path that passes each of its nodes exactly once). A routing scheme with stretch 1 is called a *shortest path routing scheme.*

Depending on whether a routing strategy is allowed to assign addresses to the nodes and port numbers to the edges, the following concepts are distinguished: In the case of *labeled routing*, the node addresses can be chosen arbitrarily by the routing strategy, such that routing information can be stored in the node addresses, whereas *name-independent routing* handles with node addresses that are fixed—mostly numbers between 1 and n. In the *fixed port model*, the port numbers are fixed before the labels of the nodes are given. In the *designer port model* port numbers from 1 to $\deg(v)$ can be assigned by the routing strategy arbitrarily to the edges incident to every node v.

A *node coloring* for a graph is a mapping $c : V \to \{1, \ldots, b\}$ where b stands for the number of the colors that are used.

With log we denote the logarithm to the base 2; the notation $\tilde{O}()$, similar to $O()$, omits constant and poly-logarithmic factors. We generally omit rounding notations $\lceil \ldots \rceil$ and $\lfloor \ldots \rfloor$, e.g., we write $\log n$ instead of $\lceil \log n \rceil$ or $\lfloor \log n \rfloor$.

3 Overview

In this section we will give an overview over a selection of results concerning routing with low memory consumption; this overview will contain fundamental past works as well as up to date results. While in Sect. 3.1 we will consider universal routing strategies, in Sect. 3.2 we will turn our attention to routing strategies that work only on special graph classes, for example on trees. Some of the presented results are summarized in Tables 1 and 2.

3.1 Universal Routing Strategies

Universal routing strategies are able to route in any network, without any restriction on the topology. For routing on shortest paths it is optimal with respect to memory consumption to simply store a routing table in each node with one entry for each destination node [26]. However, by routing on almost-shortest paths the memory consumption can be reduced; this issue was first raised by Kleinrock and Kamoun [27]. For overviews see [21, 22, 24, 31, 40].

Labeled Routing. Labeled routing strategies are allowed to arbitrarily assign addresses to the nodes. Using thereby addresses that contain routing information often results in a smaller memory consumption compared to name-independent routing, where the node names are given and cannot be changed.

Universal routing strategies				
Labeled routing				
Port model	**Stretch**	**Addr. size**	**Local memory**	**Remarks**
designer port	5	$\log n$	$O(\sqrt{n}(\log n)^{3/2})$ [17]	
fixed port	3	$3 \log n$	$O(n^{2/3}(\log n)^{4/3})$ [16]	
designer port	3	$(1 + o(1)) \log n$	$O(\sqrt{n \log n})$ [36, 35]	
designer port	$4k - 5$	$o(k(\log n)^2)$	$\tilde{O}(n^{1/k})$ [36, 35]	str. $2k - 1$ w. handsh.
Name-independent routing				
Port model	**Stretch**	**Local memory**		**Remarks**
fixed-port	5	$\tilde{O}(\sqrt{n})$ [9]		
fixed-port	3	$O(\sqrt{n}(\log n)^3 / \log \log n)$ [6]		
Labeled routing in trees				
Port model	**Stretch**	**Addr. size+local memory**		**Remarks**
designer-port	1	$(1 + o(1)) \log n$ [35]		
fixed-port	1	$O((\log n)^2 / \log \log n)$ [18, 35]		

Table 1. Performance data of several routing strategies.

Interval Routing. An often used way to save memory space is to assign the node labels in such a way that, at each node, packages for several consecutive destination addresses can be routed through the same edge—this technique is called *interval routing.* In such interval routing schemes typically each edge of a node serves as output port for a set of consecutive destination addresses, i.e., for an interval of addresses. A typical routing table then consists of a mapping from address intervals to port numbers. This idea was introduced by Santoro and Khatib [33]; the term interval routing was introduced by van Leeuwen and Tan [38] who showed that for every network there is a (not necessarily shortest path) interval routing scheme that uses every edge of the network. Characterizations of networks that have shortest path interval routing schemes can be found in [20, 21, 33, 39]. There are also interval routing schemes using more than one address interval per port [39]; the *compactness* of a network is then defined as the maximum number of intervals, taken over all nodes, that have to be mapped to the same output port [21]. For an overview about interval routing schemes see [21, 24].

Upper Bounds for Labeled Routing. A labeled routing strategy for the fixed port model using addresses of size $O((\log n)^2)$ and headers of size $O(\log n)$ was presented by Peleg and Upfal [32]; their strategy needs a total memory space of $O(k^3 n^{1+1/k} \log n)$ and guarantees a stretch of $s = 12k + 3$ for $k \geq 1$. However, it neither gives a guarantee for the local memory consumption nor is it able to handle weighted edges. The strategy is based on a technique called *hierarchical routing*—the idea here is to cover the network with several levels, that is, with subgraphs of increasing radii. A package is first sent on the lowest level; if it does not reach its destination because it is out of the radius the package will bounce back to the sender, which tries to send it on a higher level.

A much more simple scheme of Cowen [16] uses some selected nodes as "landmarks" which results in a direct routing scheme with a stretch of three in the fixed port model; it has addresses and headers of size $3 \log n$ bit and needs a local memory of size $O(n^{2/3}(\log n)^{4/3})$.

By using the technique of interval routing, the memory demand can be further reduced to $O(\sqrt{n}(\log n)^{3/2})$ while attaining a maximum stretch of five and an average stretch of three [17]; this scheme for the designer port model uses numbers from 1 to n as addresses and takes $O(\log n)$ routing time.

An improved stretch-three strategy in the designer port model was given by Thorup and Zwick [35, 36]; their scheme uses size-$(1 + o(1)) \log n$ addresses and headers and needs $O(\sqrt{n \log n})$ bits of local memory; the routing time is constant. A variation of the algorithm has $(\log n)$-bit headers and takes $O(\log \log n)$ time. Thorup and Zwick also generalize their result: Stretch $4k - 5$ is possible with $o(k(\log n)^2)$-bit addresses, $o((\log n)^2)$-bit headers and $\tilde{O}(n^{1/k})$ bits of local memory. If handshaking—that is, communication between nodes in order to get routing information—is allowed, the stretch can be reduced to $2k - 1$.

Lower Bounds for Labeled Routing. Gavoille and Pérennès [26] showed that every shortest path routing scheme needs $\Omega(n \log(\deg(v)))$ bits of local memory per node v if addresses of size $O(\log n)$ are used, independent of the header size of the packages and even in the designer port model. This implies that there is no better universal strategy for routing on shortest paths than simply storing a routing table in each node with one entry for every destination node.

For routing on non-shortest paths, Peleg and Upfal [32] showed that every universal strategy of a stretch factor $\Theta(s)$ requires a total memory space of $\Omega(n^{1+1/\Omega(s)})$ bits (a more detailed look at the hidden constants shows a lower bound of $\Omega(n^{1+1/(2s+4)})$ bits total memory for a stretch of $s \geq 1$ [22]).

Eilam et al. [17] proved that even in the designer port model there is no loop-free universal strategy (in particular, no strategy with a direct routing scheme) with addresses chosen from $\{1, \ldots, n\}$ that uses a local memory space smaller than $c\sqrt{n}$ bits on every network for a constant $c = \pi\sqrt{2/3}/\ln 2$. This holds for almost every family of graphs, even for trees, and, therefore, for every stretch (note that every non-shortest path between two nodes in a tree has to go through at least one node twice). This explains why many of the strategies mentioned in this chapter (for example [6, 12]) have to rewrite the headers of the packages at least once during the routing.

Another lower bound for universal routing strategies using integers from 1 to n as addresses involves a total memory space of $\Omega(n^2)$ bits (i.e., $\Omega(n)$ bits local memory in at least one node) for a stretch smaller than three [23] even in the designer port model; this result implies that there is no compact universal routing strategy with a stretch smaller than three.

Name-Independent Routing. In the case of name-independent routing the node addresses are part of the input network and cannot be changed in the preprocessing phase of the routing strategy.

Universal routing strategies			
Labeled routing with node addresses from $\{1, \ldots, n\}$			
Port model	**Stretch**	**Local memory**	**Remarks**
designer port	< 3	$\Omega(n)$ [23]	
designer port	1	$\Omega(n \log n)$ [26]	
designer port	every	$> \pi\sqrt{2/3}/\ln 2 \cdot \sqrt{n}$ [17]	bound for loop-free strategies
Name-independent routing			
Port model	**Stretch**	**Local memory**	**Remarks**
designer port	< 5	$\Omega(\sqrt{n})$ [35]	
fixed port	$< 2k+1$	$\Omega((n \log n)^{1/k})$ [4]	for any integer $k \geq 1$
Labeled routing in trees			
Port model	**Stretch**	**Addr. size+local memory**	**Remarks**
fixed port	1	$\Omega((\log n)^2/\log\log n)$ [19]	

Table 2. Some lower bounds for the demand of local memory.

Upper Bounds for Name-Independent Routing. The first to distinguish between labeled and name-independent routing were Awerbuch et al. [10] who presented a name-independent routing strategy that uses the technique of hierarchical routing and needs $O(kn^{2/k} \log n)$ bits of local memory per node and attains a stretch factor of $O(k^2 3^k)$ for any integer $k \geq 1$. A series of improvements [2, 11, 12] resulted in a stretch-five routing strategy using $\tilde{O}(\sqrt{n})$ bits of memory per node by Arias et al. [9] and a stretch-three routing strategy using rewritable headers of size $O((\log n)^2/\log\log n)$ and $O(\sqrt{n}(\log n)^3/\log\log n)$ bits of memory per node by Abraham et al. [6]. We will describe the algorithm of Abraham et al. in Section 4.

Lower Bounds for Name-Independent Routing. Obviously, all lower bounds given for the case of labeled routing also hold for the case of name-independent routing, but there are also stronger bounds.

One of them can be seen by considering the complete bipartite graph $K_{n/2,n/2}$: Every universal name-independent routing strategy in the fixed port model with a stretch smaller than three has to use a local memory of $\Omega(n \log n)$ bits [6]. This result was generalized by Abraham et al. [4], who proved a lower bound of $\Omega((n \log n)^{1/k})$ bits of local memory for any stretch stretch smaller than $2k+1$ for any integer $k \geq 1$. Moreover, it is known [35] that there is no universal name-independent routing strategy with stretch smaller than five using $o(\sqrt{n})$ bits of local memory.

By considering the center node of a star, one can see that there is no universal loop-free name-independent routing strategy that uses $o(n)$ bits of memory in every node [6].

Originally published in *Algorithms for Sensor and Ad Hoc Networks*, volume 4621 of *LNCS*, pages 187–202. Springer, 2007.

3.2 Special Graph Classes

There is a large portion of work on special families of graphs, especially for trees. Routing strategies for trees are also used as subroutines in some universal routing strategies (see Sect 4.3).

Trees. The first interval routing scheme for trees using $O(\deg(v)\log n)$ bits of memory in each node v was presented by Santoro and Khatib [33]. Gavoille [21] showed that labeled routing with $3.71\sqrt{n}$ bits local memory per node is possible in the designer port model, although with exponential routing time. If addresses of size $5\log n + o(\log n)$ are allowed, there is a direct routing scheme in the designer port model that needs $3\log n + O(\log\log n)$ bits per node and has constant routing time and a preprocessing time of $O(n\log n)$ [18]. It is even possible to reduce the address size to $2.8\log n$ bit while increasing the routing time to $n^{O(1)}$ and the time for preprocessing to $n^{O(1)}$ [18]. A similar result was achieved by Thorup and Zwick [35] whose routing scheme uses addresses and local memory of size $(1 + o(1))\log n$ bit and has a constant routing time.

For the fixed port model, Peleg [30] presented a direct labeled routing scheme for trees that needs addresses and local memory of size $O((\log n)^2)$ and whose routing time is $O(\log n)$. This scheme is based on a technique called *distance labeling*: This labeling of the nodes allows, given the addresses of two nodes, to compute their distance (see also [14, 15, 25]). Fraigniaud and Gavoille [18] as well as Thorup and Zwick [35] improved this result by giving direct routing schemes with addresses, headers and local memory of size $O((\log n)^2/\log\log n)$; their routing time is constant and the preprocessing time is $O(n\log n)$. Complementing this result, Fraigniaud and Gavoille [19] showed that for every shortest path routing scheme for trees in the fixed port model the sum of the address size and the local memory size is $\Omega((\log n)^2/\log\log n)$ for some node in some tree. Moreover, it is known [18] that any shortest path routing scheme with addresses from $\{1,\ldots,n + o(\sqrt{n}/\log n)\}$ requires a local memory of size $n - o(n)$ bit in the fixed port model and a local memory of size $c\sqrt{n} - o(n)$ bit for some constant c in the designer port model, which points out how even a small a change of the header size has a big impact on the size of the local memory (note that by attaching the routing information of a node into its address, it is even possible to create routing strategies that need no local memory but that require bigger addresses).

Further Graph Classes. Other graph classes are considered as well; a large list of literature is given, e.g., by Abraham et al. [3]. For planar graphs, the best known (labeled) shortest path routing scheme is from Lu [29], who uses addresses from 1 to n and $7.181n + o(n)$ bits of local memory in the designer port model. No lower bound for the memory size of shortest path routing schemes on planar graphs is known. For a stretch greater than one, Thorup [34] presented a stretch-$(1 + \epsilon)$ labeled routing scheme that uses addresses and routing tables of size $O((1/\epsilon)(\log n)^2)$. Abraham et al. [3] gave a name-independent routing

Originally published in *Algorithms for Sensor and Ad Hoc Networks*, volume 4621 of *LNCS*, pages 187–202. Springer, 2007.

strategy for unweighted graphs excluding any fixed minor (which is, therefore, applicable on planar graphs) in the fixed port model. They use headers of size $O((\log n)^2/\log\log n)$ and need a local memory of size $\tilde{O}(1)$.

Other results regard, e.g., routing on euclidean metrices [7], routing in growth bounded networks [8], routing in graphs with low doubling dimension [1, 28], routing in power law graphs [13], or routing in interval graphs and circular arc graphs [15].

4 Algorithms

In this section we describe three routing strategies. First, in Sect. 4.1, we present a straightforward labeled routing strategy for trees as an easy to understand example for interval routing. This strategy needs $O(\deg(v)\log n)$ bits of memory in each node v. The memory requirement is improved to $6\log n$ bits of local memory in Sect. 4.2, where the idea of interval routing is combined with that of moving part of the routing information from the local memories into the node addresses. Finally, we sketch a state of the art name-independent routing strategy for arbitrary graphs in Sect. 4.3 which needs $O(\sqrt{n}(\log n)^3/\log\log n)$ bits of local memory per node and guarantees a stretch of three. Although this routing strategy is name-independent, it uses as a subroutine a compact labeled routing strategy for trees, for example the one of Sect. 4.2.

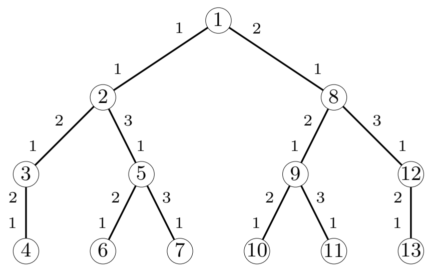
4.1 An Interval Routing Scheme for Trees

The direct labeled routing strategy of Santoro and Khatib [33] for trees in the fixed port model uses the technique of interval routing and needs $O(\deg(v)\log n)$ bits of local memory in each node v. The strategy is based on a depth first search (DFS) numbering of the nodes, and because of its simplicity we will use it as an example for interval routing on trees.

In the preprocessing phase of the strategy, the tree is rooted at an arbitrary node, and the nodes are labeled via a depth first search (DFS). Thus, for each node v with address $a(v)$, the addresses of the nodes in the subtree rooted at v form a complete interval, that is, if t is the number of the nodes in this subtree, then the addresses of the nodes in the subtree are exactly the numbers from $a(v)$ to $a(v)+t-1$. Now the following information is stored at each node v (see Fig. 1 for an example):

– its address $a(v)$,
– the highest address occurring in the subtree rooted at v, denoted with f_v,
– the port number of the edge leading to the parent of v, and
– a table with one entry (a_i, p_i) for each child v_i of v, where a_i is the highest node address occurring in the subtree rooted at v_i and p_i is the port number $p(v, v_i)$ of the edge leading from v to v_i.

The header of a data package contains nothing but the address of its destination node. A node v that has to route a package with destination address $a(w)$ performs the following steps:



Local memory of node 2	
Own address	2
f_2 (= highest addr. in subtree)	7
Port to parent	1
Port to subtree	(4, 2)
Port to subtree	(7, 3)

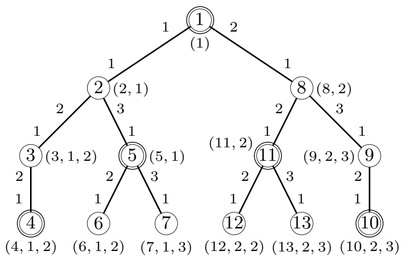
Fig. 1. Example for the interval routing strategy for trees of Sect 4.1. The picture on the left shows a tree whose edges are labeled with port numbers. The routing strategy performs a DFS to determine the node addresses. The table on the right shows the local memory of node 2.

1. If $a(w) = a(v)$: The package has reached its destination. Stop.
2. If $a(w) < a(v)$ or $a(w) > f_v$: The destination is not a descendant of v. Send the package to the parent of v and stop.
3. Otherwise, the destination node lies in a subtree rooted at a child of v. Search in the local memory the entry (a_i, p_i) with the smallest $a_i \geq a(w)$ and send the package through the port numbered with p_i.

Clearly this strategy uses $O(\deg(v) \log n)$ bits local memory at each node v: three entries of size at most $\log n$ bit and $\deg(v) - 1$ entries of size $\log(n) + \log(\deg(v))$ bit. Although the total memory over all nodes is only $O(n \log n)$ bits, such a local memory requirement can be impractical when the node degrees are not distributed equally over the network and there are nodes with very high degree. The routing time also depends on the degree of the current node, but can be bounded from above by $O(\log \log n)$ [37].

4.2 An Improved Labeled Routing Scheme for Trees

Thorup and Zwick [35] introduced a direct labeled routing scheme for trees in the fixed port model that uses only $6 \log n$ bits of local memory in each node. The main idea is to enlarge the addresses of the nodes by storing some routing information in the addresses. In contrast to the algorithm of Sect. 4.1, the local memory of a node u here does not contain the port numbers of all edges leading to child nodes of u, but only the port number of at most one edge leading to a child node that has a "big" number of successors. The intuition is that then the depths of the subtrees rooted at the other children v of u cannot exceed a certain value. More specifically, the depths of these subtrees are bounded from above by $\log n$ (we will prove this bound later), and, therefore, one can afford for every successor w of such a child v to put the port numbers used on the path from u to w into the address of w. Altogether, the scheme uses addresses (and headers) of size $(\log n)^2 + \log n$ bit (compared to $(\log n)$-bit addresses in the scheme introduced in Sect. 4.1). The routing takes only constant time in each node.



Local memory of node 2	
dfs_2 (= own index)	2
f_2 (= highest index in subtree)	7
h_2 (= index of heavy child)	5
ℓ_2 (= #light nodes on path from root)	1
$P_2[0]$ (= port to parent)	1
$P_2[1]$ (= port to heavy child)	3

Fig. 2. Example for the improved routing strategy for trees of Sect 4.2. The picture on the left shows a tree whose edges are labeled with port numbers. The routing strategy performs a DFS—visiting always the light children before the heavy one—to determine the indices of the nodes. The addresses of the nodes are written in brackets; heavy nodes are drawn with two bordering circles. The table on the right shows the local memory of node 2.

We will now have a closer look at the details. Like in the strategy of Sect. 4.1, the strategy here also uses a DFS labeling for the nodes, but unlike in Sect. 4.1, the numbers obtained by this traversal are not directly used as addresses. We call such a number an *index*, and we denote the index of a node v with $\mathrm{dfs}(v)$. Before performing this DFS, the nodes of the tree are partitioned (also by a DFS) into *heavy* and *light* nodes: A node v is called heavy if the subtree rooted at v contains at least half of the nodes of the subtree rooted at the parent of v. Nodes that are not heavy are called light; the root is defined to be heavy. The DFS that determines the indices of the nodes is performed in such a way that at each node the light children are visited before the heavy child (clearly every node has at most one heavy child). The following information is stored at each node v:

- the index dfs_v of v,
- the highest address occurring in the subtree rooted at v, denoted with f_v,
- if v has a heavy child, the index of the heavy child, denoted with h_v; otherwise $h_v = f_v + 1$,
- the number of light nodes (including v itself if v is light) lying on the path from the root to v, called the *light level* of v and denoted with ℓ_v,
- the port number of the edge to the parent of v, denoted with $P_v[0]$, and
- the port number of the edge leading to the heavy child of v, denoted with $P_v[1]$ (if v has no heavy child, $P_v[1]$ contains an arbitrary entry).

As mentioned before, some routing information is stored in the addresses of the nodes: The address $a(v)$ of a node v consists of an array $(\mathrm{dfs}_v, L_{v,1}, L_{v,2}, \dots, L_{v,l_v})$, where $L_{v,i}$ denotes the port number $p(x, y)$ with y being the i-th light node on the path from the root to v and x being the parent of y. Fig. 2 shows an example for the addresses and the local memory used by the scheme.

The routing scheme at a node v performs the following steps to route a package with destination address $(\mathrm{dfs}_w, L_{w,1}, L_{w,2}, \dots, L_{w,\ell_w})$:

Originally published in *Algorithms for Sensor and Ad Hoc Networks*, volume 4621 of *LNCS*, pages 187–202. Springer, 2007.

1. If $\mathrm{dfs}_w = \mathrm{dfs}_v$: The package has reached its destination. Stop.
2. If $\mathrm{dfs}_w < \mathrm{dfs}_v$ or $\mathrm{dfs}_w > f_v$: The destination is not a descendant of v. Send the package through the edge labeled with $P_v[0]$ to the parent of v and stop.
3. If $\mathrm{dfs}_w \geq h_v$: The destination is a node in the subtree rooted at the heavy child of v (because the heavy child is the last child visited by the DFS). Send the package through the edge labeled with $P_v[1]$ to the heavy child of v and stop.
4. Otherwise, the destination must be a node in a subtree rooted at a light child of v. Send the package through the edge labeled with L_{w,ℓ_v+1} to the light child of v who lies on the path from v to the destination node (the port number L_{w,ℓ_v+1} can be extracted from the destination address).

Clearly every package that is routed by this algorithm reaches its destination. Let us discuss the performance—in particular, the memory consumption and address size—of this strategy. The information stored in each node consists of at most $\log n$ bits for each of the entries dfs_v, f_v, h_v, ℓ_v, $P_v[0]$, and $P_v[1]$, which is altogether $6 \log n$ bits of local memory per node.

To determine the size of the addresses and headers, it suffices to give a bound for ℓ_v, because clearly the sizes of the addresses are bounded from above by $(1 + \ell_v) \log n$ bit. Recall the definition of a light node: The subtree rooted at a light node v contains less than half of the nodes of the subtree rooted at the parent of v. Hence, if a node v is a descendant of a node u and $\ell_v = \ell_u + 1$, the subtree rooted at v contains less than half of the nodes of the subtree rooted at u. Therefore, the light level ℓ_v of each node v can be at most $\log n$. Altogether, the addresses have a size of $(\log n)^2 + \log n$ bit as claimed. Note that the routing can be performed in constant time and that it is independent from the numbering of the ports (i.e., it accords to the fixed port model).

By a slight modification the size of the addresses can be further reduced [35]. To this end, the definition of heavy (and light) nodes is modified: A node v is called *heavy* if the subtree rooted at v contains at least $1/b$ (instead of one half) of the nodes of the subtree rooted at the parent of v for a fixed integer $b \geq 3$. The information stored locally now contains not only one entry $P_v[1]$ with the port number of the edge to one heavy child, but one entry for each of the at most $b - 1$ heavy children. Moreover, there is a new field $H_v[0]$ containing the number of heavy children, and the field h_v is replaced by new fields $H_v[i], 1 \leq i < b$ containing the indices of all heavy children of v. This information allows to route every package again in constant time. The modified strategy needs $(2b + 3) \log n$ bits of local memory, but has addresses consisting of only $(1 + \log_b n) \log n$ bits.

With much more effort, Thorup and Zwick [35] and Fraigniaud and Gavoille [18] could also give labeled routing strategies for trees in the fixed port model that need only $O((\log n)^2 / \log \log n)$-bit addresses.

4.3 A Universal Compact Name-Independent Routing Scheme

The algorithms presented in Sect. 4.1 and Sect. 4.2 have been labeled shortest path routing strategies for trees. Now we will use such a strategy as a subroutine for a universal name-independent routing strategy. The algorithm of Abraham et al. [6] that we are going to describe provides a stretch of three for arbitrary networks in the fixed port model and needs only $O(\sqrt{n}(\log n)^3/\log\log n)$ bits of local memory and $O((\log n)^2/\log\log n)$-bit rewritable headers. The preprocessing can be done in polynomial time, and the routing time is constant.

The main idea behind the routing strategy is to cover the network with several single source shortest paths spanning trees and to store in every node of the network for each of these trees an $O((\log n)^2/\log\log n)$-bit routing table according to one of the labeled routing strategies mentioned in Sect. 4.2 [18, 35]. A data package will then be routed along one of these spanning trees. In order to guarantee a stretch of three, only $\sqrt{n}\log n$ spanning trees are necessary, and, consequently, the local memory of each node seemingly needs to store only $\sqrt{n}\log n$ routing tables, each of size $O((\log n)^2/\log\log n)$ bit. Unfortunately, the compact shortest path routing scheme used for the routing in the spanning trees is a labeled routing scheme. This means that the routing tables stored in the local memories of the nodes are useless if one does not additionally know the node addresses that are used by the labeled routing strategy and that differ from the node addresses of the input network for the name-independent routing scheme.

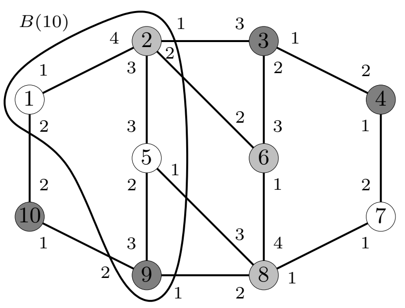
In order to overcome this difficulty, the routing strategy combines several ingredients. First of all, we define the vicinity $B(v)$ of a node v as the set of the $b\sqrt{n}\log n$ nodes closest to v (ties are broken by choosing nodes with small addresses first) with b being a fixed constant large enough. The second ingredient is a node coloring $c : \{1,\ldots,n\} \to \{1,\ldots,\sqrt{n}\}$ with the following two properties: At most $2\sqrt{n}$ nodes are colored with the same color, and for each node v there is at least one node from each color in its vicinity $B(v)$. (Note that, in contrast to the colorings occurring in many common graph problems, it is not required that two adjacent nodes have different colors.) A random coloring would satisfy these two properties with high probability, but the coloring can also be performed deterministically in polynomial time [6].

The third ingredient is a hash function $h : \{1,\ldots,n\} \to \{1,\ldots,\sqrt{n}\}$ that maps node addresses to colors such that at most $O(\sqrt{n}\log n)$ addresses are mapped onto the same color. Such a hashing can be computed in constant time [6], for example by extracting $(1/2)\log n$ bits from the node addresses.

For the sake of a clear presentation of the main ideas, we will present a slightly simplified version of the algorithm using only the first two ingredients; the purpose of the hash function will be explained at the end of the section. Moreover, we will denote the colors of the nodes with color names instead of numbers.

After the network is colored such that each node v has a color $c(v)$, w.l.o.g. the color *red* is designated as a special color, and the following steps are performed: For each node w whose color $c(w)$ is red, a spanning tree T_w is constructed that connects the node w to all other nodes of the network on shortest paths. On this tree T_w, we run the preprocessing algorithm of the labeled routing strategy mentioned in Sect. 4.2, which computes for each node v of the network an address

Originally published in *Algorithms for Sensor and Ad Hoc Networks*, volume 4621 of *LNCS*, pages 187–202. Springer, 2007.



Local memory of node 10	
Own address	10
$\mu(T_2, 10)$ (= local mem. of node 10 in T_2)	
…	…
…	…
$\mu(T_6, 10)$ (= local mem. of node 10 in T_6)	
…	…
…	…
$\mu(T_8, 10)$ (= local mem. of node 10 in T_8)	
…	…
…	…
Port to node $1 \in B(10)$	2
Port to node $2 \in B(10)$	2
Port to node $5 \in B(10)$	1
Port to node $9 \in B(10)$	1
Red node in $B(3)$	2
$\lambda(T_2, 3)$ (= addr. of node 3 in T_2)	…
Red node in $B(4)$	6
$\lambda(T_6, 4)$ (= addr. of node 4 in T_6)	…

Fig. 3. Example for the universal name-independent routing strategy of Sect 4.3. The picture on the left shows a network with node labels and port numbers. We assume that the vicinity $B(v)$ of each node v contains four nodes and that the network is colored with three colors in the preprocessing phase. The red color is displayed as light grey (nodes 2, 6, and 8 are red); the four nodes 1, 2, 5, and 9 are the vicinity of node 10. The table on the right shows the local memory of node 10.

and a routing table for routing in T_w. We denote this address with $\lambda(T_w, v)$ and the routing table with $\mu(T_w, v)$.

Now the following informations are stored in each node v:

– its address $a(v)$,
– for each red node $w \in V$: the routing table $\mu(T_w, v)$,
– for each node $w \in B(v)$: the port number of the edge leading on the shortest path from v to w, and
– for each node $w \in V$ that has the same color as v (i.e., $c(w) = c(v)$): the number of a red node $u \in B(w)$ and the address $\lambda(T_u, w)$ of w in the spanning tree T_u of u.

Fig. 3 shows an example for a network and the resulting local memory of one of its nodes.

To route a data package with destination address $a(w)$, the routing scheme running at a node v performs the following steps:

1. If $a(w) = a(v)$: The package has reached its destination. Stop.
2. If $w \in B(v)$: Send the package to the next node on the shortest path to w, using the port number stored in the local memory. Stop.

3. If $c(w) = c(v)$: Overwrite the header of the data package with the number of a red node $u \in B(w)$ and the address $\lambda(T_u, w)$ (both values are stored in the local memory of v) and send the package according to $\mu(T_u, w)$. Stop. (Each subsequent node x can now use its routing table $\mu(T_u, x)$ and route the package on the tree T_u.)
4. Otherwise: Send the package to a node $u \in B(v)$ with $c(u) = c(w)$. Then u can look up the adequate red node and start routing the package on the corresponding spanning tree.

Let us shortly discuss the stretch of this scheme. In the worst case, the node v sends a package with destination w to a node $u_1 \in B(v)$ with $c(u_1) = c(w)$, and from there the package is routed on a spanning tree T_{u_2} with $u_2 \in B(w)$ to w.

Assume the case that on every shortest path from v to w there is a node $x \notin B(v) \cup B(w)$. If we denote with $b(v)$ the "radius" of $B(v)$, that is, $b(v) = \max_{x \in B(v)} d(v, x)$, then it clearly holds that $b(v) + b(w) \leq d(v, w)$. The length of the path that is tracked by the package is bounded from above by

$$\begin{aligned}
&d(v, u_1) + d(u_1, u_2) + d(u_2, w) \\
\leq\ &b(v) + d(u_1, u_2) + b(w) \\
\leq\ &b(v) + (d(u_1, v) + d(v, w) + d(w, u_2)) + b(w) \\
\leq\ &b(v) + (b(v) + d(v, w) + b(w)) + b(w) \\
\leq\ &3d(v, w)
\end{aligned}$$

For the other case (i.e., there is a shortest path from v to w that contains only nodes from $B(v) \cup B(w)$) we cannot prove a stretch of three due to our simplification. In the original algorithm, not only the spanning trees consisting of shortest paths to the red nodes are used, but also spanning trees consisting of shortest paths to all nodes $w \in B(v)$ for every node v. A data package from v to w can then be routed along a path that is composed of two such spanning trees.

Our second simplification concerns the following question: How can a node v that has to route a package with destination $w \notin B(v)$ determine the color of the node w? To overcome this problem, the original algorithm uses the mentioned hash function h. Instead of storing in the local memory of a node v informations for each node w with $c(w) = c(v)$, it stores informations for each node w with $h(w) = c(v)$. During the routing, a node v then does not have to compute the color $c(w)$ of a destination node w, but only its hash value $h(w)$.

5 Chapter Notes

In Section 3 we have given an overview over some of the most important results and their sometimes subtle differences. For other overviews see [21, 22, 24, 31, 40]. The algorithms considered in Section 4 have been originally presented by Santoro and Khatib [33], Thorup and Zwick [35], and Abraham et al. [6], respectively. Recent work includes, e.g., results of Abraham et al. [1, 3–5], Arias et al. [9], Brady and Cowen [14, 15, 13], and Konjevod et al. [28].

References

1. I. Abraham, C. Gavoille, A. V. Goldberg, and D. Malkhi. Routing in networks with low doubling dimension. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS'06)*, page 75. IEEE Computer Society, 2006.
2. I. Abraham, C. Gavoille, and D. Malkhi. Routing with improved communication-space trade-off. In *Proceedings of the 18th International Symposium on Distributed Computing (DISC'04)*, volume 3274 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 2004.
3. I. Abraham, C. Gavoille, and D. Malkhi. Compact routing for graphs excluding a fixed minor. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *Lecture Notes in Computer Science*, pages 442–456. Springer-Verlag, 2005.
4. I. Abraham, C. Gavoille, and D. Malkhi. On space-stretch trade-offs: lower bounds. In *Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'06)*, pages 207–216. ACM Press, 2006.
5. I. Abraham, C. Gavoille, and D. Malkhi. On space-stretch trade-offs: upper bounds. In *Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'06)*, pages 217–224. ACM Press, 2006.
6. I. Abraham, C. Gavoille, D. Malkhi, N. Nisan, and M. Thorup. Compact name-independent routing with minimum stretch. In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'04)*, pages 20–24. ACM Press, 2004.
7. I. Abraham and D. Malkhi. Compact routing on euclidian metrics. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC'04)*, pages 141–149. ACM Press, 2004.
8. I. Abraham and D. Malkhi. Name independent routing for growth bounded networks. In *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'05)*, pages 49–55. ACM Press, 2005.
9. M. Arias, L. J. Cowen, K. A. Laing, R. Rajaraman, and O. Taka. Compact routing with name independence. *SIAM Journal on Discrete Mathematics*, 20(3):705–726, 2006.
10. B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Compact distributed data structures for adaptive routing. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC'89)*, pages 479–489. ACM Press, 1989.
11. B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *Journal of Algorithms*, 11(3):307–341, 1990.
12. B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31th Annual IEEE Symposium on Foundations of Computer Science (FOCS'90)*, pages 503–513. IEEE Computer Society Press, 1990.
13. A. Brady and L. J. Cowen. Compact routing on power-law graphs with additive stretch. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 119–128. SIAM, 2006.
14. A. Brady and L. J. Cowen. Compact routing with additive stretch using distance labelings. In *Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'06)*, page 233. ACM Press, 2006.
15. A. Brady and L. J. Cowen. Exact distance labelings yield additive-stretch compact routing schemes. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC'06)*, volume 4167 of *Lecture Notes in Computer Science*, pages 339–354. Springer-Verlag, 2006.

16. L. J. Cowen. Compact routing with minimum stretch. In *Proceedings of the 10th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'99)*, pages 255–260. SIAM, 1999.
17. T. Eilam, C. Gavoille, and D. Peleg. Compact routing schemes with low stretch factor. *Journal of Algorithms*, 46(2):97–114, 2003.
18. P. Fraigniaud and C. Gavoille. Routing in trees. In *Proceedings of the 28th International Colloquium on Automata, Languages, and Programming (ICALP'01)*, volume 2076 of *Lecture Notes in Computer Science*, pages 757–772. Springer-Verlag, 2001.
19. P. Fraigniaud and C. Gavoille. A space lower bound for routing in trees. In *Proceedings of the 19th International Symposium on Theoretical Aspects of Computer Science (STACS'02)*, volume 2285 of *Lecture Notes in Computer Science*, pages 65–75. Springer-Verlag, 2002.
20. G. N. Frederickson and R. Janardan. Optimal message routing without complete routing tables. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC'86)*, pages 88–97. ACM Press, 1986.
21. C. Gavoille. A survey on interval routing. *Theoretical Computer Science*, 245(2):217–253, 2000.
22. C. Gavoille. Routing in distributed networks: Overview and open problems. *ACM SIGACT News—Distributed Computing Column*, 32(1):36–52, 2001.
23. C. Gavoille and M. Gengler. Space-efficiency for routing schemes of stretch factor three. *Journal of Parallel and Distributed Computing*, 61(5):679–687, 2001.
24. C. Gavoille and D. Peleg. Compact and localized distributed data structures. *Distributed Computing*, 16(2-3):111–120, 2003.
25. C. Gavoille, D. Peleg, S. Pérennès, and R. Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004.
26. C. Gavoille and S. Pérennès. Memory requirements for routing in distributed networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC'96)*, pages 125–133. ACM Press, 1996.
27. L. Kleinrock and F. Kamoun. Hierarchical routing for large networks; performance evaluation and optimization. *Computer Networks and ISDN Systems*, 1:155–174, 1977.
28. G. Konjevod, A. W. Richa, and D. Xia. Optimal-stretch name-independent compact routing in doubling metrics. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC'06)*, pages 198–207. ACM Press, 2006.
29. H.-I. Lu. Improved compact routing tables for planar networks via orderly spanning trees. In *Proceedings of the 8th Annual International Conference on Computing Combinatorics (COCOON'02)*, volume 2387 of *Lecture Notes in Computer Science*, pages 57–66. Springer-Verlag, 2002.
30. D. Peleg. Proximity-preserving labeling schemes and their applications. In *Proceedings of the 25th International Workshop on Graph-Theoretical Concepts in Computer Science (WG'99)*, volume 1665 of *Lecture Notes in Computer Science*, pages 30–41. Springer-Verlag, 1999.
31. D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM Monographs on Discrete Mathematics and Applications, 2000.
32. D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, 1989.
33. N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 28(1):5–8, 1985.

Originally published in *Algorithms for Sensor and Ad Hoc Networks*, volume 4621 of *LNCS*, pages 187–202. Springer, 2007.

34. M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.
35. M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'01)*, pages 1–10. ACM Press, 2001.
36. M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.
37. 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.
38. J. van Leeuwen and R. B. Tan. Routing with compact routing tables. Technical Report RUU-CS-83-16, Dept. of Computer Science, Utrecht University, 1983.
39. J. van Leeuwen and R. B. Tan. Interval routing. *The Computer Journal*, 30(4):298–307, 1987.
40. J. van Leeuwen and R. B. Tan. Compact routing methods: A survey. In *Proceedings of the 1st International Colloquium on Structural Information and Communication Complexity (SIROCCO'94)*, pages 99–110. Carleton University Press, 1994.

Index

address, 3
address size, 3, 5, 7, 10, 12
algorithm
– distributed, 1

compact routing, *see* routing, compact
compact routing strategy, *see* routing strategy, compact
compactness, 5

designer port model, 4–8
direct routing function, *see* routing function, direct
direct routing scheme, *see* routing scheme, direct
distance, 3
distance labeling, 8

fixed port model, 4–15

graph
– planar, 8, 9
graph minor, 9

hash function, 13, 15
header, 3
– rewritable, 3
header size, 3, 9, 10, 12, 13
hierarchical routing, *see* routing, hierarchical

interval routing, *see* routing, interval

labeled routing, *see* routing, labeled
latency, *see* routing time
local memory, *see* memory, local, 5, 7, 13

memory
– local, 3, 5–10, 12, 13
– total, 3, 5–10

name-independent routing, *see* routing, name-independent
node coloring, 4, 13

port number, 1
port number, 3

routing
– compact, 1–15
– hierarchical, 5, 7
– in planar graphs, 8, 9
– in trees, 5, 7–12
– interval, 4–6, 8–10
– labeled, 4–13
– name-independent, 4–8, 12–15
routing scheme

– shortest path, 8
routing strategy
– compact, 2
– universal, 13
routing time, 13
routing function, 1
– direct, 4
routing scheme, 1, 2
– compact, *see* routing strategy, compact
– direct, 4, 6, 9–12
– interval, *see* routing, interval
– shortest path, 4–6, 8–12
routing strategy, 2
– compact, 6, 10–15
– interval, *see* routing, interval
– labeled, *see* routing, labeled
– name-independent, *see* routing, name-independent
– universal, 4–7, 12–15
routing table, 2
routing time, 3, 10, 12

shortest path, 3
shortest path routing scheme, *see* routing scheme, shortest path
spanning tree, 13
stretch, 2, 3, 5–8, 13, 15

total memory, *see* memory, total

universal routing strategy, *see* routing strategy, universal

vicinity, 13