This item was submitted to [Loughborough's Research Repository](#) by the author.
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

Fully compact routing in low memory self-healing trees

PLEASE CITE THE PUBLISHED VERSION

<https://doi.org/10.1145/3369740.3369786>

PUBLISHER

ACM

VERSION

AM (Accepted Manuscript)

PUBLISHER STATEMENT

© ACM 2020. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in ICDCN 2020: Proceedings of the 21st International Conference on Distributed Computing and Networking, <https://doi.org/10.1145/3369740.3369786>.

LICENCE

All Rights Reserved

REPOSITORY RECORD

Castañeda, Armando, Jonas Lefèvre, and Amitabh Trehan. 2020. "Fully Compact Routing in Low Memory Self-healing Trees". figshare. <https://hdl.handle.net/2134/10260602.v1>.

Fully Compact Routing in Low Memory Self-Healing Trees

Armando Castañeda
Instituto de Matemáticas, UNAM
Mexico City, Mexico
armando.castaneda@im.unam.mx

Jonas Lefèvre
Amitabh Trehan
jalb.pdf@gmail.com
A.Trehan@lboro.ac.uk
Computer Science, Loughborough University
Loughborough, U.K.

ABSTRACT

The paper (*Compact Routing Messages in Self-Healing Trees*, TCS 2017) introduced CompactFTZ, the first self-healing compact routing algorithm that works in a distributed network with each node using only $O(\log n)$ words (i.e. $O(\log^2 n)$ bits) memory and thus $O(\log n)$ sized messages. The routing uses only $O(1)$ and $O(\log n)$ words routing table and packet labels respectively on a self-healing tree that also works using only $O(1)$ words repairing the network in face of a strong adversary deleting nodes. This deterministic algorithm sets up its data structures in a preprocessing phase and then updates the required data structures in only $O(1)$ parallel time per healing round during execution of the algorithm.

However, CompactFTZ has no constraints in its preprocessing phase which could be done in distributed large memory or even centrally. In this paper, we correct that by developing the algorithms for preprocessing of CompactFTZ in a fully distributed manner using only $O(\log n)$ words memory in optimal time. In fact, the preprocessing for the self-healing tree (*ForgivingTree*) component takes only $O(1)$ memory. We develop a local function which each node invokes to instantly compute and then relay its repair instructions (known as its *Will*) in only $O(1)$ time.

We formalise the low memory CONGEST model setting used in previous low memory algorithms (e.g. [24]); nodes' working memory is restricted to be much smaller (in our case, $O(\log n)$) than the numbers of their neighbours to whom they communicate through their I/O ports. We expand the model to allow for non-contiguous ports (e.g. empty ports or neighbours unmarked or lost in dynamic settings) and adversarial order of inputs from neighbours. Besides the Wills, we set up the tree structures and traversals for the routing scheme using only $O(\log n)$ memory and $O(D)$ parallel time, where D is the diameter. Thus, we devise the first self-healing compact routing algorithm that can be fully set up and executed in low memory.

CCS CONCEPTS

• **Mathematics of computing** → **Graph algorithms**; *Trees*; • **Theory of computation** → **Graph algorithms analysis**; **Distributed algorithms**; Data structures design and analysis; • **Networks** → *Network algorithms*.

KEYWORDS

Low Memory Distributed Algorithms, Self-Healing, Compact Tree Routing, Low Memory Preprocessing, Half-full Tree

ACM Reference Format:

Armando Castañeda, Jonas Lefèvre, and Amitabh Trehan. 2020. Fully Compact Routing in Low Memory Self-Healing Trees. In *21st International Conference on Distributed Computing and Networking (ICDCN 2020)*, January 4–7, 2020, Kolkata, India. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3369740.3369786>

1 INTRODUCTION

As the number of nodes in modern distributed networks rapidly increase and large networks of low memory devices evolve, there is a need to design truly scalable solutions. These algorithms should work with small messages while using low memory. For example, networks like the *Internet of Things (IoT)* will deploy systems with large number of weak devices. Similarly, *ad-hoc sensor or vehicular networks* may have dynamically evolving and unpredictable node neighbourhoods. In such scenarios, solutions using local polynomial memory (in n - the number of nodes in the system) may not be desirable. Even with advancements in technology, node memory will remain at a premium with increasing demand for applications and a light memory footprint will be desired. The natural question is *what can be achieved if nodes have limited working memory even if they may have large neighbourhoods?* This, of course, has always been an interesting theoretical question (e.g. [14, 25, 26, 34]). A similarly important question is *what can be achieved despite failures in a distributed network?* There are a multitude of ways in which failure can happen and be handled in distributed networks; in this paper, we concentrate on node crash failures in the self-healing healing paradigm [7, 35–37, 49, 50] developing the first compact self-healing routing scheme fully setup and executed in low (logarithmic in n) memory.

We develop a self-healing compact routing scheme using only $O(\log n)$ words local memory per node with small messages (as in the CONGEST model (see 2 for details). Throughout, unless otherwise mentioned, space is given in the unit of memory words. There has been intense interest in designing efficient routing schemes for distributed networks [1, 8, 13, 19, 33, 53, 56, 58] with compact routing trading stretch (factor increase in routing length) for memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICDCN 2020, January 4–7, 2020, Kolkata, India

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7751-5/20/01...\$15.00

<https://doi.org/10.1145/3369740.3369786>

used. In essence, the challenge is to use small memory per node overcoming the need for large routing tables or/and packet headers. In fact, we define local memory as *compact* if it is $o(n)$ bits and, by extension, an algorithm as compact if it works in compact memory - however, we use much smaller (only $O(\text{polylog } n)$) per node.

A routing scheme (with routing tables at nodes and headers/labels at packets to route them) consists of two parts - a pre-processing algorithm (scheme construction) and a routing protocol [30]. The routing results mentioned previously assume sequential centralized pre-processing. Since routing is inherently a distributed networks problem, it makes sense to have the scheme construction distributed too. Awerbuch et al. [3] were the first to introduce this question which has recently led to a spurt in designing efficient preprocessing algorithms for compact routing schemes in the CONGEST model (discussed in Section 2) [21, 22, 32, 46]. These algorithms did not seek to explicitly optimize internal working memory. We develop a pre-processing scheme (for a tree routing scheme from Thorup and Zwick’s seminal work [58]) assuming that nodes do not even have any excess space and therefore, have, to develop the whole solution in compact memory itself. Elkin and Neiman [24] with a similar motivation develop efficient routing solutions using the least working memory required per node. As they point out, since compact routing is motivated by having limited memory at nodes, it is inconsistent to use much larger memory to setup such schemes in the first place. We develop preprocessing for tree routing in a similar manner to [25] (though they give the scheme for general routing too) but differ in two important aspects: we are interested in deterministic solutions whereas the previous solutions are all randomized, and we are interested in a fault-tolerant solution as discussed in the following paragraph.

Deterministic routing schemes, in the preprocessing phase, rely on discovery and efficient distributed ‘encoding’ of the network’s topology to reduce the memory requirement (a routing scheme on an arbitrary network with no prior topology or direction knowledge would essentially imply large memory requirements). This makes them sensitive to any topology change and, hence, it is challenging to design fault tolerant compact routing schemes. There has been some work in this direction e.g. in the dynamic tree model [38, 40] or with additional capacity and rerouting in anticipation of failures [9, 10, 12, 17, 29, 31]. Self-healing is a responsive fault-tolerance paradigm seeking minimal anticipatory additional capacity and has led to a series of work [35, 37, 48, 50, 55, 57, 59, 60] in the recent past for maintaining topological properties (connectivity, degrees, diameter/stretch, expansion etc.). Algorithms were also proposed to ‘self-heal’ computations e.g. [54].

Combining the above motivations, [7] introduced a fault-tolerant compact routing solution CompactFTZ in the (deletion only) self-healing model (details in Section 2) However, as in previous routing schemes, CompactFTZ’s pre-processing assumed large (not compact) memory. This paper addresses that important problem developing a compact pre-processing deterministic algorithm for CompactFTZ. We also develop a compact pre-processing deterministic algorithm for CompactFT (a compact version of *ForgivingTree* [35]). This leads to a fully compact (i.e. completely distributed and in compact memory) routing scheme, a fully compact self-healing routing scheme and a fully compact self-healing algorithm.

Algorithms for problems	Internal memory	Time (# rounds)	# messages	In Paper
Leader Election: Peleg’s algorithm [51]	$O(1)$	$O(D)$	$O(mD)$	Section 5.1
* BFS Spanning Tree Construction	$O(1)$	$O(D)$	$O(m)$	Section 5.2
* (regular) Convergecast	$O(1)$	$O(D)$	$O(n)$	Section 5.3
Weight labelling with convergecast	$O(1)$	$O(D)$	$O(nD)$	Section 5.3
Compact DFS relabelling of a tree	$O(1)$	$O(D)$	$O(m)$	Section 5.4
* Light Path by BFS construction	$O(\log n)$	$O(D)$	$O(m)$	Section 5.4
Wills (w/ half-full tree labelling) setup	$O(1)$	$O(1)$	$O(n)$	Section 4.2
TZ preprocessing	$O(\log n)$	$O(m)$	$O(m)$	Section 5.4
CompactFT preproc.	$O(1)$	$O(D)$	$O(m)$	Section 4.2
CompactFTZ preproc.	$O(\log n)$	$O(m)$	$O(m)$	Section 4.2

Table 1: Results in this paper

To develop the preprocessing (CompactFTZ and its preprocessing is discussed in detail in Section 5), a number of subproblems such as BFS, DFS traversals, convergecast and *Will* creation need to be solved. Our results are summarised in Table 1. Our memory requirements are the same as that of the final data structures setup in the CompactFTZ algorithm [7] (and the centralised Thorup-Zwick([58]) and ForgivingTree [35]) and, hence, optimal.

2 MODEL

Our communication model is similar to the CONGEST model [52] and the CONGEST RAM model of [23, 24] (in which they develop low memory algorithms). We assume a connected network of arbitrary topology represented by an undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ for n nodes and m bidirectional links. Each node has a *unique* ID and $O(n)$ communication ports with locally unique *Port*-IDs interfacing with the links. In the CONGEST model, nodes communicate with one another in discrete rounds, via short (of size $O(\log n)$ bits) messages. In the CONGEST RAM variant, each message is allowed to contain a vertex ID, a graph distance, or anything of size within the same constant factor size. As in [24], each node also has $o(n)$ working (internal) memory (we call this *compact*) - however, we further restrict node memory to only $O(\log n)$ in this paper.

Again, as in [24], in every (synchronous/asynchronous) round, a node can read from and write to any of its ports while doing internal computation as long as it reads and/or writes to a port only once [20]. However, in addition, in our model, neighbours need not be on contiguous ports i.e. there may be ‘dead’ ports interspersed with live ones. Even starting from contiguous ports, this can happen due to node deletions or due to subnetworks generated (e.g. spanning trees). To accomodate this, we state our model in more formal terms as follows: Every round, v ‘sweeps’ through its ports reading and writing to its ports in some order interleaving the reads and writes with internal processing as long as it reads and writes to a port at most once in that round. That is, a round may be represented as $pr_1pw_1r_2pw_2p \dots$, where p, r and w stand for processing (possibly none), reading and writing, subscripted by port numbers. Notice that it may not be possible for a node to execute any random read order (without repetition) due to memory restrictions, (some orders can only be stored in $\theta(n)$ memory) thus, often, a node may just execute a sequential read order reading from the first port to the last port hosting a neighbour. Note that the above can be viewed as simply a regular message passing round at a finer granularity with interleaved local computations; hence, the time complexity measure (number of distributed rounds) is the same as that of regular message passing.

Note that memory restriction bounds the local computation between reads and writes in the same round and the message size, thus, $O(1)$ local memory generalizes CONGEST with the additional constraint that only polynomial time local computations can be performed between reads and writes. Even though local computations are usually assumed to be negligible compared to the communication, we would like the computation between two consecutive reads and/or writes to be quick: we say that such computations are given by **locally compact functions** where a *locally compact function takes as input the previous read and the node state to produce the next state and message(s)* - the function should be quick (constant/polynomial) taking low memory. The function we devise in Section 4 takes only constant internal computation per execution. The subtlety in designing our algorithms is the awareness of the memory constraints - maybe ‘distribute’ information (e.g. port numbers) to siblings and parents for future use. The overall distributed algorithm coordinates the messages and compact functions.

In effect, this paper only deals with the static setting described above as it only describes the preprocessing phase - the dynamic self-healing setting is handled in [7]. In the following section, we describe the self-healing model for completeness and understanding of the background that follows.

2.1 Bounded Memory Deletion only Self-healing Model

The *deletion only self-healing model* [7, 35, 55, 59] is a model for reconfigurable (e.g. P2P/overlay) networks where an omniscient adversary attacks by removing nodes and the affected nodes distributively respond by adding connections, with the aim of achieving global fault tolerance by local actions.

Before the adversary can delete nodes, a pre-processing phase is permitted where nodes can exchange messages with their neighbors and setup data structures as required on the initial graph G_0 .

The adversarial process can be described as deleting some node v_t from G_{t-1} , forming H_t . All neighbors of v_t are informed of the deletion. In the healing stage nodes of H_t communicate (concurrently, asynchronously) with their immediate neighbors and nodes may decide to add edges to other nodes they know about or drop earlier edges forming G_t . The objective is to minimise the following ‘complexity’ measures (excluding the preprocessing stage):

- **Degree increase:** $\max_{t < n} \max_v (deg(v, G_t) - deg(v, G_0))$
- **Diameter stretch:** $\max_{t < n} D(G_t) / D(G_0)$
- **Communication:** The maximum number of bits/words, sent by a single node in each recovery phase
- **Recovery time:** The maximum total time for a recovery phase, assuming it takes a message no more than 1 time unit to traverse any edge
- **Local Memory:** The amount of memory a single node needs to run the algorithm.

3 BACKGROUND: COMPACT SELF-HEALING ROUTING (COMPACTFTZ)

Algorithm CompactFTZ [5, 7] is a self-healing (i.e. fault-tolerant) compact routing algorithm based on two algorithms: a Thorup-Zwick tree routing scheme, TZ [58] and the self-healing algorithm *ForgivingTree*, CompactFT [35]. In this paper, we give deterministic

setup (preprocessing) of TZ and CompactFT. Since CompactFTZ directly runs in our low memory model (the routing scheme is compact and CompactFTZ works in the bounded memory self-healing model tagging every message with port numbers), we get the first fully distributed, deterministic, fault-tolerant and compact memory routing scheme.

Before going into the details of CompactFT and TZ, let us give the high level picture that makes CompactFTZ work. To begin with, consider that we have a network in the form of a rooted tree and the compact routing scheme TZ (Section 3.2) setup on it: this will successfully route if there were no failures. What happens if there is an adversarial crash failure of a node? How do we seamlessly route packets which are still in the network or new ones which are later generated? CompactFT (Section 3.1) works by replacing the subgraph of the deleted node (i.e. the star with the deleted node at the centre) by a ‘healing’ subgraph of virtual nodes simulated by its children. Importantly, this subgraph structure is routable and has its own (compact) routing scheme, say, R . Also, nodes know if they are connected to a healing structure and through which edges. Now, when a packet is traversing through the undamaged part of the network, it is routed by the nodes following the TZ scheme. However, when the TZ scheme pushes the packet towards a healed substructure, the nodes seamlessly switch to the scheme R . In the paper [7], it is shown that not only does R route the packet to the same intermediate node as it would have done with TZ in an undamaged network (with a small additional stretch) but also that there is no need to update the topological information that is gathered in the preprocessing to implement TZ (CompactFT information is purely local and updated in $O(1)$ time with $O(1)$ messages). Of course, the deleted node may itself have been routing packets and it is possible that the adversary deletes destinations or sources of packets still in transit; CompactFTZ can handle these cases and inform the sender of non-delivery and also continue delivery of dropped packets by retaining copies with neighbouring nodes.

Now, we give a brief overview of CompactFT and TZ, particularly with reference to their combined preprocessing as done in CompactFTZ (Please refer to Algorithm 3.1). Let us begin with CompactFT and lines 2 and 6 of Algorithm 3.1.

Algorithm 3.1 Preprocessing of CompactFTZ: A high level view.

CompactFTZ: *Preprocessing*

- 1: Given a distinguished node v (e.g. by a compact leader election)
 - 2: $T_a \leftarrow$ A BFS spanning tree of graph G_0 (Sec. 5.2) (Or any spanning tree if diameter is not being self-healed)
 - 3: $T_b \leftarrow$ Setup of TZ *heavy arrays* by *compact convergecast* of T_a (Sec. 5.3)
 - 4: $T_c \leftarrow$ DFS traversal and labelling (renaming) of T_b (Sec. 5.4)
 - 5: $T_d \leftarrow$ Setup of TZ *light levels* by BFS *traversal* of T_c (Sec. 5.5)
 - 6: $T_e \leftarrow$ Setup of CompactFT *Wills* on T_d (Sec. 4)
-

3.1 Compact Forgiving Tree

CompactFT is a self-healing algorithm in the bounded memory deletion only self-healing model (Section 2). On a spanning tree T of a network graph G and in face of any series of adversarial

deletions, after attack number t , it guarantees that G_t is connected, for every node $v \in G_t$, $\deg(v, G_t) \leq \deg(v, G_0) + 3$, diameter stretch $D(G_t)/D(G_0) \leq \log(\Delta)$ where Δ is the largest degree of a node in G_0 ; each recovery takes only $O(1)$ parallel rounds with $O(1)$ messages of $O(1)$ size (i.e. in the CONGEST model). Further, the repair at each node uses only $O(1)$ node IDs and, thus, $O(1)$ words for repair assuming $O(1)$ words are enough to store any node's ID.

CompactFT is an example of algorithm design using virtual structures (ref. [59, 60] for a discussion). We will maintain T as a structure with 'virtual' nodes and through a mapping between the virtual forgiving tree T_t and the real graph G_t at any time t , it will be enough for us to maintain bounds on T_t for them to carry over to G_t . To begin with, if T is a BFS spanning tree, we know that $D(T)/D(G) \leq 2$ and we only need to bound $D(T_t)$ to get the diameter bounds wrt $D(G)$. Following is the basic idea of the algorithm: if a non-leaf node is deleted, replace it with a *Reconstruction Tree (RT)* having its children as the leaf nodes and virtual nodes simulated by the children as internal node with the invariant that every leaf simulates no more than one internal node. This is illustrated in Figure 1. Over a series of deletions, we maintain this invariant by a clever mechanism of promotion of designated nodes called *heirs*.

If the RT is a binary tree correctly labeled, we get our degree bound over any series of deletions. If it is also balanced, we also get our diameter stretch. A balanced binary tree is thus sufficient. However, since we want to use CompactFT for routing with a routing scheme R , we make the RT a balanced binary search tree (BBST). Now, our routing scheme R is simply a binary search on the tree which, in fact, uses no routing tables and has zero memory requirement on nodes. In fact, we use a half-full trees (hafts) (introduced in [36, 37]) for its special properties. A haft is simply defined as follows ([37]):

Definition 3.1 (Half -full Tree(haft)). A haft is a rooted binary tree where every non-leaf node v satisfies the following properties:

- (1) v has exactly two children
- (2) The left child of v heads a complete binary subtree containing at least half of v 's descendants.

A complete tree is an example of a haft, so is the tree in figure 2. Definition 4.1 is an alternate definition. Refer to [37] for a detailed exposition.

Healing on a leaf deletion is a bit different: if the deleted node has no virtual duties, it is simply removed otherwise its simulated node is also 'short-circuited' from the graph.

Figure 1: Deleted node replaced by its reconstruction tree with virtual internal nodes simulated by real leaf nodes [37]

How do we implement the above idea in a distributed network? For this we use the idea of *Wills*. A will of a node is nothing but its

RT - i.e. for node v , $will(v)$ is simply $RT(v)$. The idea is that on the demise/deletion of v , its children will execute $will(v)$ i.e. construct $RT(v)$. Thus, each child could be given a copy of $will(v)$ in advance to enable the fix. However, $will(v)$ could be too large ($O(n)$ size) to be transmitted. However, notice a child only needs the subgraph it is involved in - this is what we call a *willportion* and is only of $O(1)$ size. This is illustrated in Fig. 2: the haft on the left corresponding to the Will of a node with children a to e and the boxes on the right giving the children's *willportions*. Once the wills are setup, the major technical challenge is to maintain the wills which is what CompactFTZ does in low memory. It is easy to setup the will in preprocessing with $O(n)$ memory because the node only needs to collect the IDs of its children in its memory to compute $RT(v)$. However, with only $O(1)$ memory, it is a challenging problem. We show how this is done in Section 4

Figure 2: Compact function f to query labeled half-full trees (hafts) (Sec. 4.1): On the left a haft with 5 leaves. $f(5, b)$ returns the second box (with $O(\log n)$ sized subtrees of b and b')

3.2 Thorup-Zwick Tree Routing

TZ is a non fault-tolerant compact routing scheme on trees, inspired on simple interval routing. In interval routing (on a rooted tree), DFS labels are used as routing labels and each node, for each of its children, stores an interval indicating the labels in the subtree rooted at that child. Routing is simple: when a node receives a message to a node with (DFS) label x , it checks which interval x belongs to and forwards the message to the corresponding child; if x does not belong to any interval, the message is forwarded to the parent. This simple solution is not compact as a node in the tree can have $\Omega(n)$ children. TZ solves this problem by storing the interval of only a constant number of children per node at the cost of increasing size of routing labels to $O(\log^2 n)$. To do so, the nodes in a tree are partitioned into two disjoint classes, *heavy* and *light*, depending on the number of nodes in the subtree rooted at the node. A good property of this partition is that each node has only a constant number of heavy children. Each node only stores intervals of heavy nodes. With this information interval routing cannot be performed as a node might need to forward a message to a light child. TZ solves this issue by appending this missing information to DFS labels; this is called *light path* of a node: the sequence of light nodes in the path from the root to the node, which is of length $O(\log^2 n)$. The resulting scheme is still simple: first a process checks intervals of heavy nodes and, if there is no success, then checks the light path in the routing label; if both fail then the message is forwarded to its parent.

Steps 3, 4 and 5 of Algorithm 3.1 perform TZ's preprocessing: first heavy and light nodes are identified, then DFS labels are computed and finally interval for heavy nodes and routing labels are computed.

To conclude CompactFTZ's preprocessing, step 6 computes and distributes Wills and Willportions to all processes in one round. As mentioned, the last step of the preprocessing is the main challenge of the whole process.

4 COMPACTFT PREPROCESSING: COMPUTING WILLPORTIONS IN OPTIMAL TIME

As explained in Section 3.1, the central idea in the fault-tolerant mechanism of CompactFT is that of a node's Will (its RT), which is computed in CompactFT's preprocessing stage (using non-compact memory): each node v gathers all IDs from its children, locally produces its $RT(v)$, and then, to each of its children, sends back part of its Will, called Willportion or *subwill*, of size $O(1)$.

Due to compact local memory, a node v cannot store all its children's IDs, and thus cannot store its whole $RT(v)$. To cope with this problem, in Subsection 4.1 we design a local function using compact memory that works in an *uninterpreted* version of $RT(v)$ in which each node u in $RT(v)$ is replaced with k if the k -th port number of v connects to u . Thus, v can compute the uninterpreted Willportion of its child in its k th port. To compute the interpreted version of the Willportion, v simply needs to send a message asking for IDs through the port numbers involved in the uninterpreted Willportion, and replace port number with IDs accordingly.

As an example, consider again the $RT(v)$ in Figure 1, and suppose that the first port of v connects to a , the second one to b and so on. Thus, in the uninterpreted $RT(v)$, a is replaced with 1, b with 2 and so on. When v wants to compute the Willportion of one of its child, say the one connected to its 6-th port (corresponding to f), v queries our local function which returns that the parent of 6-th port is the 4-th port, and its children are the 5-th and 7-th ports (corresponding to the subgraph involving f' in $RT(v)$), and also the 6-th port as a leaf has parent the 5-th port (corresponding to the leaf e). With this information, v only needs to ask for the IDs of the processors connected to its 4-th, 5-th and 7-th ports, replace the IDs accordingly and send the Willportion to f through its 6-th port in the next round. Using this simple scheme, v can compute and distribute all its Willportions in at most 2Δ rounds. Remarkably, in Section 4.2, we show that, in fact, *all* Willportions of v can be optimally computed and distributed in a single round.

4.1 Computing Half-Full Trees with Compact Memory

Given a power of two, 2^x , consider the full binary tree with 2^x leaves defined recursively as follows. The root of the tree is the string 0, and each node v has left child $v0$ and right child $v1$. It is easy to see that the nodes at height h are the binary representation of $0, \dots, 2^{x-h} - 1$. Let $\tilde{v}$ denote the integer represented by the string v . For any node v , its left and right children represent the number $2\tilde{v}$ and $2\tilde{v} + 1$, respectively. Let $B(2^x)$ denote the previous tree.

We now define a function ℓ used in CompactFT that labels the nodes of $B(2^x)$ in the space $[0, 2^x - 1]$; ℓ represents the ports in the uninterpreted Will of a node. Of course the labelling is not proper but it has nice properties that will allow us to compute it using low memory. Consider a node v of $B(2^x)$. Let h_v denote the height of v in $B(2^x)$. Then, we define ℓ as follows: if $h_v = 0$, $\ell(v) = \tilde{v}$,

Figure 3: The tree $B(2^3)$ with its labelling ℓ . Each circle shows in its interior the binary string identifying the vertex. Near each node appears its label ℓ . Non-leaf nodes correspond to bold line circles.

otherwise $\ell(v) = 2^{h_v-1} - 1 + \tilde{v} 2^{h_v}$. In words, if v is of height 0, its label is simply $\tilde{v}$, otherwise its label is computed using a base number, $2^{h_v-1} - 1$, plus $\tilde{v}$ times an offset, 2^{h_v} . As an illustration, Figure 3 depicts the tree $B(2^3)$ and its labelling ℓ .

LEMMA 4.1. *Let $B(2^x)$ be a non trivial tree, with $x > 0$. For every vertex v , $\ell(v) \in [0, 2^x - 1]$. For the root r , $\ell(r) = 2^{x-1} - 1$. Consider any $y \in [0, 2^x - 1]$. There is a unique leaf v of $B(2^x)$ such that $\ell(v) = y$. If $y \leq 2^x - 2$, there is a unique non-leaf u of $B(2^x)$ such that $\ell(u) = y$, and there is no non-leaf u of $B(2^x)$ such that $\ell(u) = 2^x - 1$.*

PROOF. Let v be a node of $B(2^x)$. As explained, $\tilde{v} \in [0, 2^{x-h_v} - 1]$. It is clear that $\ell(v) \geq 0$. If $h_v = 0$, then $\ell(v) = \tilde{v} \leq 2^x - 1$. Else $\ell(v) = 2^{h_v-1} - 1 + \tilde{v} 2^{h_v} \leq 2^x - 1 - 2^{h_v-1} < 2^x - 1$.

The root r of $B(2^x)$ has height $h_r = x$ and $\tilde{r} = 0$, hence, by definition, $\ell(r) = 2^{x-1} - 1$. Now, consider any $y \in [0, 2^x - 1]$. Since all leaves are at height 0, there is a unique leaf v with $\ell(v) = y$. Suppose that $y \leq 2^x - 2$. There exists a unique integer factorization of $y + 1$ then there exists unique $h \geq 0$ and $p \geq 0$ such that $y + 1 = 2^h(2p + 1)$. This decomposition can be easily obtained from the binary representation of $y + 1$. By construction, we have $h \leq \log(y + 1) < \log(2^x) = x$ then $h + 1 \leq x$ and we have $2p < 2^{x-h}$ then $p \leq 2^{x-h-1} - 1$. Let us consider the unique (non-leaf) node u such that $\tilde{u} = p$ and $h_u = h + 1 \geq 1$. It means that u is the unique node such that $\ell(u) = y$. Finally, there is no non-leaf u of $B(2^x)$ such that $\ell(u) = 2^x - 1$ because we just proved that each element of $[0, 2^x - 2]$ has a unique inverse image under ℓ . Since the number of non-leaf node is exactly $2^x - 1 = |[0, 2^x - 2]|$, there is no non leaf node u such that $\ell(u) = 2^x - 1$. $\square$

By Lemma 4.1, for the labelling ℓ , each $y \in [0, 2^x - 1]$ appears one or two times in $B(2^x)$, on one leaf node and at most on one non-leaf node. Thus, we can use the labelling ℓ to unambiguously refer to the nodes of $B(2^x)$. Namely, we refer to the leaf v of $B(2^x)$ with label $\ell(v) = y$ as *leaf y* , and, similarly, if $y \leq 2^x - 2$, we refer to the non-leaf u of $B(2^x)$ with label $\ell(u) = y$ as *non-leaf y* . By abuse of notation, in what follows $B(2^x)$ denotes the tree itself and its labelling ℓ as defined above. The following lemma directly follows from the definition of ℓ .

LEMMA 4.2. *Let $B(2^x)$ be a non trivial tree ($x \geq 1$). Consider any $y \in [0, 2^x - 1]$. The parent of the leaf y is the non-leaf $2\lfloor \frac{y}{2} \rfloor$. If $y \leq 2^x - 2$, then let $y = 2^i - 1 + z 2^{i+1}$. If $i \leq x - 2$, the parent of the non-leaf y is the non-leaf $2^{i+1} - 1 + \lfloor \frac{z}{2} \rfloor 2^{i+2}$. If $i \geq 1$, the left and right children of the non-leaf y are the non-leaves $2^{i-1} - 1 + 2z 2^i$ and*

Algorithm 4.1 Calculate the parent and children of leaf and non-leaf $y \in [0, 2^x - 1]$ in $B(2^x)$ and in $HT([a, b])$

Function SearchBT($y, 2^x$)

```

if  $x = 1$  then
  return  $\langle 0, \perp, 0, 1 \rangle$ 
else
  if  $y = 2^{x-1} - 1$  then
     $\langle P, P', L', R' \rangle \leftarrow \langle 2^{x-1} - 2, \perp, 2^{x-1} - 1 - 2^{x-2}, 2^{x-1} - 1 + 2^{x-2} \rangle$ 
    { $y$  is the root}
  else if  $y < 2^{x-1} - 1$  then
     $\langle P, P', L', R' \rangle \leftarrow \text{SearchBT}(y, 2^{x-1})$  { $y$  is in the left subtree}
  else
     $\langle P, P', L', R' \rangle \leftarrow \text{SearchBT\_shift}(y, 2^{x-1}, 2^{x-1})$  { $y$  is in the right subtree}
  if  $P' = \perp$  then  $P' \leftarrow 2^{x-1} - 1$  { $y$  is in the root of one of the two subtrees}
return  $\langle P, P', L', R' \rangle$ 

```

Function SearchBT_shift($y, 2^x, a$)

```

 $\langle P, P', L', R' \rangle \leftarrow \text{SearchBT}(y - a, 2^x)$ ; return  $\langle P + a, P' + a, L' + a, R' + a \rangle$ 

```

Function SearchHT(y, a, b)

```

if  $b - a = 2^x$  for some  $x$  then
  return SearchBT_shift( $y, 2^x, a$ ) {The HT is actually a BT}
else
   $\langle P, P', L', R' \rangle \leftarrow \langle \perp, \perp, \perp, \perp \rangle$ 
   $x = \lfloor \log_2(b - a) \rfloor$  {let  $2^x$  be the largest power of two smaller than  $b - a + 1$ }
   $z = \lfloor \log_2(b - a - 2^x) \rfloor$  {let  $2^z$  be the largest power of two smaller than  $b - a - 2^x$ }
  if  $y = a + 2^x - 1$  then
     $\langle P, P', L', R' \rangle \leftarrow \langle 2^x - 2, \perp, 2^x - 1 - 2^{x-1}, 2^x - 1 + 2^z \rangle$  { $y$  is the root}
  else if  $y < a + 2^{x-1} - 1$  then
     $\langle P, P', L', R' \rangle \leftarrow \text{SearchBT\_shift}(y, 2^{x-1}, a)$  { $y$  is in the left subtree}
  else
     $\langle P, P', L', R' \rangle \leftarrow \text{SearchHT}(y, a + 2^x, b)$  { $y$  is in the right subtree}
  if  $P' = \perp$  then
     $P' \leftarrow 2^{x-1} - 1$  { $y$  is in the root of one of the two subtrees}
return  $\langle P, P', L', R' \rangle$ 

```

Function SubWill(y, b, parent)

```

 $\langle P, P', L', R' \rangle \leftarrow \text{SearchHT}(y, 0, b)$ 
 $x = \lfloor \log_2(b) \rfloor$  { $2^x$  is the largest power of two smaller than  $b$ }
if  $y = b - 1$  then  $P' \leftarrow \text{parent}$ ;  $L' \leftarrow 2^x - 1$  { $y$  is in the root of one of the two subtrees}
else if  $y = 2^x - 1$  then  $P' \leftarrow b - 1$  { $y$  is the root}
return  $P, P', L', R'$ 

```

$2^{i-1} - 1 + (2z + 1)2^i$, respectively. If $i = 0$, the left and right children of the non-leaf y are the leaves y and $y + 1$.

The proof of Lemma 4.1 shows how to quickly represent a non-leaf node v with $\tilde{v} \in [0, 2^x - 2]$ in its form $\tilde{v} = 2^i - 1 + \lfloor \frac{\tilde{v}}{2} \rfloor 2^{i+1}$ so that one can easily compute its parent and children, using Lemma 4.2. The function implied by Lemma 4.2, called **SearchBT** in Algorithm 4.1, can be implemented using low memory. This function is what we are looking for, if the node has a number of children

Figure 4: The half-full tree $HT([0, 12])$ with its labeling ℓ' .

that is a power of two, and hence the uninterpreted RT is precisely $B(2^x)$. If not, the Will of a node is a *half-full* tree, defined below.

Let $B(2^x, a)$ denote the tree $B(2^x)$ together with the labelling $\ell'(v) = \ell(v) + a$. Clearly, ℓ' labels the nodes of $B(2^x, a)$ in the space $[a, a + 2^x - 1]$. For ease of representation, we use $B(2^x)$ to represent $B(2^x, 0)$ (i.e. $B(2^x)$ with labelling $\ell(v)$) in the discussion that follows.

REMARK 4.3. The left subtree of the root of $B(2^x)$ is $B(2^{x-1})$ and its right subtree is $B(2^{x-1}, 2^{x-1})$. Thus, the left subtree of the root of $B(2^x, a)$ is $B(2^{x-1}, a)$ and its right subtree is $B(2^{x-1}, 2^{x-1} + a)$.

Definition 4.1 (Half-full Tree [37]). Consider an integer interval $S = [a, b]$. The *half-full tree with leaves in S* , denoted $HT(S)$, is defined recursively as follows. If $|S|$ is a power of two then $HT(S)$ is $B(|S|, a)$. Otherwise, let 2^x be the largest power of two smaller than $|S|$. Then, $HT(S)$ is the tree obtained by replacing the right subtree of the root of $B(2^{x+1}, a)$ with $HT([a + 2^x, b])$. The nodes of $HT(S)$ have the induced labelling ℓ' of every $B(*, *)$ recursively used for defining the half-full tree.

Figure 4 depicts the half-full tree $HT([0, 12])$ and its induced ℓ' labelling. The label ℓ' plays the role of the ports of a node and, as before, our target is to compute the parts of the half-full tree a given label belongs to.

LEMMA 4.4. Consider a half-full tree $HT([a, b])$. For every node v of $HT([a, b])$, $\ell'(v) \in [a, b]$. For the root r of $HT([a, b])$, $\ell'(r) = 2^x - 1 + a$, where 2^x is the largest power of two smaller than $b - a + 1$. For every $y \in [a, b]$, there is a unique leaf v of $HT([a, b])$ such that $\ell'(v) = y$, and if $y \in [a, b - 1]$, there is a unique non-leaf v of $HT([a, b])$ such that $\ell'(v) = y$.

PROOF. First, it directly follows from Definition 4.1 that for every v of $HT([a, b])$, $\ell'(v) \in [a, b]$. By Lemma 4.1, $2^x - 1$ is the root of $B(2^{x+1})$, and thus $2^x - 1 + a$ is the root of $B(2^{x+1}, a)$, and consequently the root of $HT([a, b])$.

By construction and using Remark 4.3, if $y \in [a, a + 2^x - 1]$, there is a unique leaf v in the right subtree of the root such that $y = \ell(v)$ and for all leaf v' outside this the subtree we have $\ell(v') \in [a + 2^x, b]$. By induction on the HT construction, if $y \in [a + 2^x + 1, b]$, there is a unique leaf v in the left subtree of the root such that $y = \ell(v)$ and for all leaf v' outside this the subtree we have $\ell(v') \in [a, a + 2^x]$.

By the inductive construction of the $HT([a, b])$, all leaf nodes belongs to one full binary subtree, and all the full binary subtrees

cover disjoint intervals. By Remark 4.3, there exists a unique leaf node v such that $\ell(v) = y$. $\square$

As with full binary trees, by Lemma 4.4, when considering the labelling ℓ' of $HT([a, b])$, each $y \in [a, b]$ appears one or two times in $HT([a, b])$, on one leaf node and at most on one non-leaf node. Thus, those two nodes in the half-full tree can be unambiguously referred as *leaf node* y and *non-leaf node* y . The very definition of half-full trees, Definition 4.1, Lemmas 4.2 and 4.4 suggest a natural low memory (sublinear on the size of the interval) recursive function that obtains the parent and children of a node in $HT([a, b])$, which is the uninterpreted Willportion of the node. We call this function **SearchHT** in Algorithm 4.1.

LEMMA 4.5. *Function **SearchHT**(y, a, b) computes the parent and children of leaf and non-leaf $y \in [a, b - 1]$ in $HT([a, b])$ using $O(\log b)$ bits.*

4.2 Computing and Distributing Willportions in One Round.

Suppose that we have a rooted spanning tree T of the original graph. Here we present a one-round compact protocol that, for any node v , computes and sends to each child of v in T its corresponding Willportion.

Let δ denote the number of children of v in T . The Will of v is the half-full tree $HT([0, \delta - 1])$, where each label ℓ is replaced with the ID of the ℓ -th child of v in T ; namely, we give an interpretation to the uninterpreted Will $HT([0, \delta - 1])$. Let $RT(v)$ denote this tree with the IDs at its nodes. Thus each child of v with ID y appears two times in y , one as leaf node and one as a non-leaf node, and the subwill of y in $RT(v)$ is made of the parent of the leaf y and the parent and children of the non-leaf y in $RT(v)$. This is the information that v has to compute and send to y . We can efficiently compute the subwill of a child using a slight adaptation of function **SearchHT** previously defined.

The representation of T is compact: v only knows its number of children in T and the port of its first children (the ports of its children do not have to be contiguous). Additionally, the ℓ -th child of v has the port number of v , nxt_port , that is connected to the $(\ell + 1)$ -th child of v . In our solution, shown in Algorithm 4.2, v first indicates to all its children to send its ID and nxt_port so that this data is in the in-buffers of v . Then, with the help of the nxt_port , v can sequentially read and collect the IDs of its children, and in between compute and send will portions. In order to be compact, v has to “forget” the ID of a child as soon as it is not needed anymore for computing the will portion of a child (possibly the same or a different one). For example, if $\delta = 13$, then v uses the half-full tree $HT([0, 12])$ in Figure 4, and the label ℓ in $HT([0, 12])$ denotes to the ℓ -th child of v in T . After reading and storing the IDs of its first four children (corresponding to 0, 1, 2, 3), v can compute and send the subwill of its first and second children (0 and 1). The leaf 0 in $HT([0, 12])$ has parent non-leaf 0 while the non-leaf 0 has parent non-leaf 1 and children leaf 0 and leaf 1. Similarly, the leaf 1 has parent non-leaf 0 while the non-leaf 1 has parent non-leaf 3 and children non-leaf 0 and non-leaf 2. Moreover, at that point v does not need to store anymore the ID of its first child because (leaf or non-leaf) 0 is not part of any other will portion. An invariant of

Algorithm 4.2 Will

Init :

```

if DFS walk is over : then
   $current \leftarrow fst\_p$  ;  $k \leftarrow 0$ 
  if not IsLeader then
     $send \langle MYId, myId, nxt\_p, ichild \rangle$  via  $parent\_port$ 

```

Receive $\langle WILL, p, p_h, c_l, c_r, bool \rangle$ from X :

```

 $nxtparent \leftarrow p$  ;  $nxtpharent \leftarrow p_h$ 
 $nxtchild_l \leftarrow c_l$  ;  $nxtchild_r \leftarrow c_r$ 
 $heir \leftarrow bool$ 

```

Terminate

Receive $\langle MYId, z, nxtPort, _ \rangle$ from $current$:

```

 $Node[k] \leftarrow [0, z, current]$ 
 $Will[k] \leftarrow SubWill(k, n\_child, parent)$ 
for all  $j \in Will[k] \cup \{k\}$  do
  if  $\max(Will[j]) \leq k$  then
     $p, p_h, c_l, c_r = Will[j]$ 
     $send \langle WILL, Node[p][1], Node[p_h][1],$ 
       $Node[c_l][1], Node[c_r][1], [n\_child - 1 = k] \rangle$  via  $Node[j][2]$ 
     $free(Will[j])$ 
    for all  $x \in \{p, p_h, c_l, c_r\}$  do
       $Node[x][0] ++$ 
      if  $Node[x][0] = 4$  then  $free(Node[x])$ 
   $current \leftarrow nxtPort$  ;  $k ++$ 

```

our algorithm is that, at any time, v stores at most four IDs of its children. The rules appear in Algorithm 4.2. The algorithm uses function **SubWill** in Algorithm 4.1, which computes the subwill of a child node.

THEOREM 4.6. *Algorithm 4.2 correctly computes all the subwills and sends all of them in 1 round. Moreover, at any time, the memory contains at most $5 \log \delta$ IDs (node or port) and subwills.*

PROOF. By construction, all the children form a chain starting with fst_p and going on via the pointers nxt_p . If those are correct, all the in-buffers corresponding to children are read during this one round. Thus k goes from 0 to $\delta - 1$ and $current$ from fst_p to the last one through every port. This means that for every k , at some point $Node[k]$ and $Will[k]$ are filled. For each k , $Will[k]$ is freed only after use and $Node[k]$ only after 4 uses. No $Node[k]$ need more than 4 uses. Let $0 \leq k < \delta$. Let $k_0 \leq k_1 \leq k_2 \leq k_3$ the nodes used in $Will[k]$. Eventually $Node[k_3]$ will be filled. At this point $Node[k_i]$ is filled for $i = 0, 1, 2, 3$ and none of them were freed, since there were still at least one missing use. The subwill can be correctly sent to the k^{th} child. In conclusion, each child's port is read and each subwill is correctly computed and sent in one round.

Now, we define the *uncompleted* edges as the edges (u, v) such that u is already registered in $Node$ and v is not. For each of those edges we have to remember informations about u (its id, its port id, its subwill with the ids of corresponding nodes (except v , then at most 3 nodes id)). The size of this information is at most 5 IDs of size $O(\log n)$. If u is linked to the k_0^{th} port and v to the k_1^{th} port, then we have $k_0 \leq k < k_1$. By construction, $k_1 - k_0 = 2^h - \sum_{l=h'}^{h-1} 2^l = 2^{h'}$ for some $0 \leq h' \leq h \leq \log \delta$. For any value of h , there can be only one such edge (this edge being between the port number

$2^{h+1} \lfloor \frac{k-2^{h-1}}{2^{h+1}} \rfloor - 2^{h-1} - 1$ and a port whose number is smaller or equal to $2^{h+1} \lfloor \frac{k-2^{h-1}}{2^{h+1}} \rfloor + 3 \cdot 2^{h-1} - 1$). Thus there are at most $\log n$ such edges. At any moment the total amount of used memory is $O(\log^2 n)$ bits. $\square$

5 THORUP-ZWICK TREE ROUTING PREPROCESSING WITH BASIC TREE TRAVERSALS

We discuss here simple implementations of the following which are used in CompactFTZ preprocessing (Algorithm 3.1 lines 1–5): i) BFS spanning tree construction, ii) Convergecast, iii) DFS traversal, tree construction and renaming. These can, of course, be vital subroutines for a number of distributed algorithms. As mentioned, the subtlety in designing these algorithms is to be aware of memory constraints and maybe ‘distribute’ information (e.g. port numbers) to siblings and parents for future use. Hence, sometimes messages also carry port numbers.

However, before we go further, remember that CompactFTZ assumes a rooted tree. We can either assume a leader to construct one or we can do leader election as shown in the next section:

5.1 Leader Election

We can implement flooding to elect a leader using the flooding based algorithm of Peleg [51]. Flooding a message to every neighbour from a node is fairly straightforward to implement in our low memory model. Consider flooding from a single source(s), say with a message M . The source writes M to all its ports which are received in the corresponding ports of its neighbours in the next round. The following then gets executed:

CLS Flooding at a node v : *Node v sweeps its ports executing all its reads first. If v gets M , it executes writes to all its ports in order copying M to them.*

We can adapt e.g. the simple flooding algorithm of Peleg [51] where D stands for diameter of the graph and each node has a unique ID:

Peleg’s Leader Election Algorithm [51]: *Nodes wake up simultaneously or spontaneously (i.e. not woken up by a message). In each round, nodes flood the highest ID seen so far and an estimate d of the diameter (initially d is 1 at every node). If a node sees a higher ID, it drops out as a candidate otherwise it updates d to the highest d seen. If a surviving candidate does not see a higher d for 3 consecutive rounds, this signals termination and that candidate as the winner.*

All that is needed is a compact function (let’s call it $\text{Max}()$) that stores the maximum value read (for both node IDs and d) as it sweeps through the ports. The nodes only store a constant number of IDs and a counter (of value at most n), therefore they only use $O(1)$ storage. In every round, every edge is used twice, therefore, the following follows:

THEOREM 5.1. *There exists a leader election algorithm using $O(1)$ local memory taking $O(D)$ time and $O(mD)$ messages.*

A simultaneously efficient deterministic LE algorithm ($O(D \log n)$ time, $O(m \log n)$ messages) can be derived from the CONGEST Algorithm 2 of [43]. Though we don’t go into details, using efficient

low memory Breadth First Search (BFS) tree construction and convergecast, which we show in Sections 5.2 and 5.3, this can be derived in a straightforward manner.

5.2 Breadth First Search (BFS) spanning tree construction

A BFS tree can be constructed by a simple flooding process with a node only storing a pointer to its parent but not to its children. Not having children information would make traversals suboptimal, therefore, this information can be ‘distributed’ through a sibling chain where the l -th child of parent p stores, in local variable next_p , the port number in p of p ’s $(l+1)$ -st child. This *compact* representation of the tree will allow us to be round optimal later. Note that p may have much fewer children in the tree than its degree ($O(\Delta)$, where Δ is max degree).

THEOREM 5.2. *Our algorithm constructs a BFS tree in $O(D)$ rounds using $O(m)$ messages.*

5.3 Convergecast (with heaviness computation)

A convergecast on a rooted tree (as previously constructed) can be done by broadcasting top-down and converging parent pointers bottom-up. This can be adapted to TZ *weight* computation: *weight* $\text{wt}(v)$ of a node v is the size of the subtree it heads including itself. For a given constant $b \geq 2$ and parent p , v is *heavy* if $\text{wt}(v) \geq \frac{\text{wt}(p)}{b}$, else v is *light*. Thus, besides aggregating $\text{wt}(v)$, v needs $\text{wt}(p)$. The heavy children inform their parent that stores their ID and port in lists H and P (of size $O(\log n)$). The problem is p ’s subtree could have branches of different heights and thus p could receive children’s weights in different rounds. When $\text{wt}(p)$ is computed, p may have lost information about v ’s port. Thus, p could either broadcast (including to non-children neighbours) or v continuously sends its weights in every round leading to two possible variants.

THEOREM 5.3. *There exist low memory algorithms to compute heaviness (weights) in $O(D)$ rounds with either $O(nD)$ or $O(n\Delta)$ messages.*

5.4 Depth First Walk And Node Relabelling

In TZ, nodes are relabelled to their DFS number on the spanning tree with heavy nodes prioritised, which are computed using a DFS walk. After its DFS exploration, a node sends back its local variable next_p , informing its parent about the next port it should try. A simple DFS exploration on the tree will take $O(n)$ rounds and $O(n)$ messages.

However, we can do much better by executing DFS walks in parallel. Notice that during the previous algorithms, each node knows the size of the subtree it heads and so does the root. If there was enough memory, each node can execute a DFS on its own subtree and store the intervals. These would be local DFS numbers however. The root is the only one with the correct global number at the moment - it can now send to each of its children the correct offset (0 for its first child, size of the first child’s interval to the second child and so on). This can be done in a top-down fashion at each node with each node adding the correct offset to get its DFS number. However, we can do this in low memory too as follows: each node, in every round, keeps sending its subtree

size to its parent. Consider the root - the root sweeps through its input in order (each node has stored in H and P its highest priority traversal children and the rest are in port order) adding the size to the offset and conveying it to the next node in order and so on. Root's children receiving the offsets do the same and so on till all nodes achieve the correct global number. Remember that in the TZ scheme, these are the new labels (ID) of a node. [24] describe a similar algorithm.

THEOREM 5.4. *The relabeling of the nodes using a parallel DFS walk can be performed in $O(D)$ rounds, using $O(n)$ messages overall.*

5.5 Computing Routing Labels.

We now have enough information (a BFS spanning tree, node weights, DFS labels) to produce routing labels in TZ, and hence, to complete the preprocessing. For a node v , its *light path* is the sequence of port numbers for light nodes in the path from the root to v . The routing label of v in TZ is the pair (*NewID*, *LightPath*), where *NewID* is its DFS label and *LightPath* its light path. The second routing label entry for the root is empty.

A simple variant of the BFS tree construction algorithm computes the routing labels if $O(\log n)$ sized messages are permitted. The root begins by sending its path(*empty*) to each port X along with the port number X . When a node receives a message $\langle RL, path, X \rangle$ from its parent, it sets its light path to $path \circ X$, if it is light, otherwise to $path$ only, producing its routing label. Then, for each port Y , it sends its light path together with the port number Y .

THEOREM 5.5. *The routing labels of TZ can be computed in $O(D)$ rounds using $O(m)$ messages of $O(\log n)$ size.*

6 RELATED WORK AND CONCLUSIONS

Our companion paper [6] discusses extensions of the low memory model in detail and provides deterministic CompactFTZ preprocessing in those. Independently, Elkin and Neiman [24] gave randomised low memory preprocessing for general Thorup-Zwick routing schemes with near-optimal complexity with low memory usage at nodes. Towards developing the general scheme, they also devised the preprocessing for Thorup-Zwick tree routing which are similar to ours. Though their overall algorithms are randomised, their tree routing preprocessing can be seen to be deterministic matching ours. However, our algorithm, as mentioned, also handles the preprocessing of the self-healing CompactFTZ and is wholly deterministic. We also state the low memory model more formally in the process.

Earlier works have looked at various memory settings in distributed networks. In the *network finite state machine* model [26], weak computational devices with constant local memory communicate in an asynchronous network. Any node only broadcasts symbols from a constant size alphabet and each time it reads its ports (all of them) can only distinguish up to a constant number of occurrences. They show probabilistic solutions to MST and 3-coloring of trees in this model. In the *beeping* model of communication [14], nodes execute synchronous rounds. Every round, a node either “beeps” and sends or stays silent and listens. A listening node obtains a single bit encoding if one or more of its neighbours beeped. [34] have shown that there are probabilistic solutions to the leader election problem in this model for complete graphs where

each node is a state machine with constant number of states. These solutions imply compact probabilistic solution in low memory models. [18] study the difference between broadcast and unicast, showing that the unicast model is strictly more powerful. [4] study the general case of nodes restricted to sending some number of distinct messages in a round. Finally, dynamic network topology and fault tolerance are core concerns of distributed computing [2, 47] and various models (e.g. [41]) and topology maintenance [42] and self-* algorithms abound especially for self-stabilisation [15, 16, 28, 39] and self-stabilising spanning trees and traversals [11, 44, 45].

To conclude, in this work, we gave the first fully distributed compact self-healing routing algorithm that can be wholly constructed and executed even if the nodes have only $O(\log n)$ memory. There are a number of directions to be explored yet. Fault-tolerant routing has not been well explored; this can be explored with respect to a number of different failure models. Though efficient randomised compact routing for general topologies seem to have been now settled [24], deterministic schemes in low memory still need to be fleshed out. Lastly, algorithms for other important problems (besides compact routing) need to be designed in the low memory settings.

ACKNOWLEDGMENTS

Armando Castañeda was supported by PAPIIT project IA102417. Jonas Lefèvre and Amitabh Trehan were supported on this research by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/P021247/1 (COSHER).

REFERENCES

- [1] Ittai Abraham, Cyril Gavoille, and Dahlia Malkhi. 2004. Routing with Improved Communication-Space Trade-Off. In *Distributed Computing*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–319.
- [2] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons.
- [3] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. 1990. Improved Routing Strategies with Succinct Tables. *J. Algorithms* 11, 3 (1990), 307–341. [http://dx.doi.org/10.1016/0196-6774\(90\)90017-9](http://dx.doi.org/10.1016/0196-6774(90)90017-9)
- [4] Florent Becker, Antonio Fernández Anta, Ivan Rapaport, and Eric Rémila. 2015. Brief Announcement: A Hierarchy of Congested Clique Models, from Broadcast to Unicast. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, Chryssis Georgiou and Paul G. Spirakis (Eds.). ACM, 167–169. <https://doi.org/10.1145/2767386.2767447>
- [5] Armando Castañeda, Danny Dolev, and Amitabh Trehan. 2016. Compact routing messages in self-healing trees. In *Proceedings of the 17th International Conference on Distributed Computing and Networking, Singapore, January 4-7, 2016*. ACM, 23:1–23:10. <https://doi.org/10.1145/2833312.2833328>
- [6] Armando Castañeda, Jonas Lefèvre, and Amitabh Trehan. 2018. Self-healing Routing and Other Problems in Compact Memory. *CoRR* abs/1803.03042 (2018). arXiv:1803.03042 <http://arxiv.org/abs/1803.03042>
- [7] Armando Castañeda, Danny Dolev, and Amitabh Trehan. 2018. Compact routing messages in self-healing trees. *Theoretical Computer Science* 709 (2018), 2 – 19. <https://doi.org/10.1016/j.tcs.2016.11.022> Special Issue of ICDCN 2016 (Distributed Computing Track).
- [8] Shiri Chechik. 2013. Compact routing schemes with improved stretch, See [27], 33–41. <http://doi.acm.org/10.1145/2484239.2484268>
- [9] Shiri Chechik. 2013. Fault-tolerant compact routing schemes for general graphs. *Inf. Comput.* 222 (2013), 36–44. <http://dx.doi.org/10.1016/j.ic.2012.10.009>
- [10] Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. 2012. f-Sensitivity Distance Oracles and Routing Schemes. *Algorithmica* 63, 4 (2012), 861–882. <http://dx.doi.org/10.1007/s00453-011-9543-0>
- [11] Zeev Collin and Shlomi Dolev. 1994. Self-stabilizing depth-first search. *Inform. Process. Lett.* 49, 6 (1994), 297 – 301. <http://www.sciencedirect.com/science/article/pii/0020019094901031>
- [12] Bruno Courcelle and Andrew Twigg. 2007. Compact Forbidden-Set Routing. In *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings (Lecture Notes in Computer*

- Science), Wolfgang Thomas and Pascal Weil (Eds.), Vol. 4393. Springer, 37–48. <https://doi.org/10.1007/978-3-540-70918-3>
- [13] Lenore Cowen. 2001. Compact Routing with Minimum Stretch. *J. Algorithms* 38, 1 (2001), 170–183. <http://dx.doi.org/10.1006/jagm.2000.1134>
 - [14] Julius Degeesys, Ian Rose, Ankit Patel, and Radhika Nagpal. 2007. DESYNC: self-organizing desynchronization and TDMA on wireless sensor networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN 2007, Cambridge, Massachusetts, USA, April 25-27, 2007*. 11–20. <https://doi.org/10.1145/1236360.1236363>
 - [15] Edsger W. Dijkstra. 1974. Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11 (November 1974), 643–644. <http://dx.doi.org/10.1145/361179.361202>
 - [16] Shlomi Dolev. 2000. *Self-stabilization*. MIT Press, Cambridge, MA, USA.
 - [17] Robert D. Doverspike and Brian Wilson. 1994. Comparison of Capacity Efficiency of DCS Network Restoration Routing Techniques. *J. Network Syst. Manage.* 2, 2 (1994).
 - [18] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. 2014. On the power of the congested clique model. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, Magnús M. Halldórsson and Shlomi Dolev (Eds.). ACM, 367–376. <https://doi.org/10.1145/2611462.2611493>
 - [19] Tamar Eilam, Cyril Gavoille, and David Peleg. 2003. Compact Routing Schemes with Low Stretch Factor. *J. Algorithms* 46, 2 (Feb. 2003), 97–114. [https://doi.org/10.1016/S0196-6774\(03\)00002-6](https://doi.org/10.1016/S0196-6774(03)00002-6)
 - [20] Michael Elkin. 2019. Personal communication.
 - [21] Michael Elkin and Ofer Neiman. 2016. On Efficient Distributed Construction of Near Optimal Routing Schemes: Extended Abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, George Giakkoupis (Ed.). ACM, 235–244. <https://doi.org/10.1145/2933057.2933098>
 - [22] Michael Elkin and Ofer Neiman. 2017. Linear-Size Hopsets with Small Hop-bound, and Distributed Routing with Low Memory. *CoRR* abs/1704.08468 (2017). [arXiv:1704.08468](http://arxiv.org/abs/1704.08468) <http://arxiv.org/abs/1704.08468>
 - [23] Michael Elkin and Ofer Neiman. 2017. Linear-Size Hopsets with Small Hop-bound, and Distributed Routing with Low Memory. *CoRR* abs/1704.08468 (2017). [arXiv:1704.08468](http://arxiv.org/abs/1704.08468) <http://arxiv.org/abs/1704.08468>
 - [24] Michael Elkin and Ofer Neiman. 2018. Near-Optimal Distributed Routing with Low Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*. 207–216. <https://doi.org/10.1145/3212734.3212761>
 - [25] Michael Elkin and Ofer Neiman. 2018. On efficient distributed construction of near optimal routing schemes. *Distributed Computing* 31, 2 (2018), 119–137. <https://doi.org/10.1007/s00446-017-0304-4>
 - [26] Yuval Emek and Roger Wattenhofer. 2013. Stone age distributed computing. See [27], 137–146. <https://doi.org/10.1145/2484239.2484244>
 - [27] Panagiotis Fatourou and Gadi Taubenfeld (Eds.). 2013. *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*. ACM. <http://dl.acm.org/citation.cfm?id=2484239>
 - [28] Michael Feldmann and Christian Scheideler. 2017. A Self-stabilizing General De Bruijn Graph. In *Stabilization, Safety, and Security of Distributed Systems*, Paul Spirakis and Philippos Tsigas (Eds.). Springer International Publishing, Cham, 250–264.
 - [29] Xiushan Feng and Chengde Han. 2001. A Fault-Tolerant Routing Scheme in Dynamic Networks. *J. Comput. Sci. Technol.* 16, 4 (2001), 371–380. <http://dx.doi.org/10.1007/BF02948985>
 - [30] Pierre Fraigniaud and Cyril Gavoille. 2002. A Space Lower Bound for Routing in Trees. In *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Antibes - Juan les Pins, France, March 14-16, 2002, Proceedings (Lecture Notes in Computer Science)*, Helmut Alt and Afonso Ferreira (Eds.), Vol. 2285. Springer, 65–75.
 - [31] T. Frisanco. 1997. Optimal spare capacity design for various protection switching methods in ATM networks. In *Communications, 1997 IEEE International Conference on*, Vol. 1. 293–298. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=13277&arnumber=605267&count=107&index=57
 - [32] Cyril Gavoille, Christian Glacet, Nicolas Hanusse, and David Ilcinkas. 2013. On the Communication Complexity of Distributed Name-Independent Routing Schemes. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings (Lecture Notes in Computer Science)*, Yehuda Afek (Ed.), Vol. 8205. Springer, 418–432. https://doi.org/10.1007/978-3-642-41527-2_29
 - [33] Cyril Gavoille and David Peleg. 2003. Compact and localized distributed data structures. *Distributed Computing* 16, 2 (01 Sep 2003), 111–120. <https://doi.org/10.1007/s00446-002-0073-5>
 - [34] Seth Gilbert and Calvin C. Newport. 2015. The Computational Power of Beeps. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015. Proceedings (Lecture Notes in Computer Science)*, Yoram Moses (Ed.), Vol. 9363. Springer, 31–46. https://doi.org/10.1007/978-3-662-48653-5_3
 - [35] Tom Hayes, Navin Rustagi, Jared Saia, and Amitabh Trehan. 2008. The forgiving tree: a self-healing distributed data structure. In *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*. ACM, New York, NY, USA, 203–212.
 - [36] Thomas P. Hayes, Jared Saia, and Amitabh Trehan. 2009. The forgiving graph: a distributed data structure for low stretch under adversarial attack. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, New York, NY, USA, 121–130.
 - [37] Thomas P. Hayes, Jared Saia, and Amitabh Trehan. 2012. The Forgiving Graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing* (2012), 1–18. <http://dx.doi.org/10.1007/s00446-012-0160-1>
 - [38] Amos Korman. 2007. General compact labeling schemes for dynamic trees. *Distributed Computing* 20, 3 (2007), 179–193.
 - [39] Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. 2011. Fast and compact self stabilizing verification, computation, and fault detection of an MST. In *PODC, Cyril Gavoille and Pierre Fraigniaud (Eds.)*. ACM, 311–320.
 - [40] Amos Korman, David Peleg, and Yoav Rodeh. 2004. Labeling Schemes for Dynamic Tree Networks. *Theory Comput. Syst.* 37, 1 (2004), 49–75.
 - [41] Fabian Kuhn, Nancy Lynch, and Rotem Oshman. 2010. Distributed computation in dynamic networks. In *Proceedings of the 42nd ACM symposium on Theory of computing (STOC '10)*. ACM, New York, NY, USA, 513–522. <http://doi.acm.org/10.1145/1806689.1806760>
 - [42] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. 2005. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *4th International Workshop on Peer-To-Peer Systems (IPTPS), Cornell University, Ithaca, New York, USA, Springer LNCS 3640*.
 - [43] Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. 2015. On the Complexity of Universal Leader Election. *J. ACM* 62, 1 (2015), 7:1–7:27. <http://doi.acm.org/10.1145/2699440>
 - [44] Shay Kutten and Avner Porat. 1999. Maintenance of a Spanning Tree in Dynamic Networks. In *Distributed Computing*, Prasad Jayanti (Ed.). Lecture Notes in Computer Science, Vol. 1693. Springer Berlin / Heidelberg, 846–846. http://dx.doi.org/10.1007/3-540-48169-9_24
 - [45] Shay Kutten and Chhaya Trehan. 2014. Fast and Compact Distributed Verification and Self-stabilization of a DFS Tree. In *International Conference on Principles of Distributed Systems*. Springer, 323–338.
 - [46] Christoph Lenzen and Boaz Patt-Shamir. 2013. Fast routing table construction using small messages: extended abstract. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, Dan Boneh, Tim Roughgarden, and Joan Feigenbaum (Eds.). ACM, 381–390. <https://doi.org/10.1145/2488608.2488656>
 - [47] N. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA.
 - [48] Gopal Pandurangan, Peter Robinson, and Amitabh Trehan. 2014. DEX: Self-Healing Expanders. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. IEEE Computer Society, Washington, DC, USA, 702–711. <http://dx.doi.org/10.1109/IPDPS.2014.78>
 - [49] Gopal Pandurangan, Peter Robinson, and Amitabh Trehan. 2016. DEX: self-healing expanders. *Distributed Computing* 29, 3 (2016), 163–185. <https://doi.org/10.1007/s00446-015-0258-3>
 - [50] Gopal Pandurangan and Amitabh Trehan. 2014. Xheal: a localized self-healing algorithm using expanders. *Distributed Computing* 27, 1 (2014), 39–54. <http://dx.doi.org/10.1007/s00446-013-0192-1>
 - [51] David Peleg. 1990. Time-optimal leader election in general networks. *J. Parallel and Distrib. Comput.* 8, 1 (1990), 96 – 99. [https://doi.org/10.1016/0743-7315\(90\)90074-Y](https://doi.org/10.1016/0743-7315(90)90074-Y)
 - [52] David Peleg. 2000. *Distributed Computing: A Locality Sensitive Approach*. SIAM.
 - [53] David Peleg and Eli Upfal. 1988. A Tradeoff between Space and Efficiency for Routing Tables (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, Janos Simon (Ed.). ACM, 43–52.
 - [54] George Saad and Jared Saia. 2017. A theoretical and empirical evaluation of an algorithm for self-healing computation. *Distributed Computing* 30, 6 (2017), 391–412. <https://doi.org/10.1007/s00446-016-0290-y>
 - [55] Jared Saia and Amitabh Trehan. 2008. Picking up the Pieces: Self-Healing in reconfigurable networks. In *IPDPS. 22nd IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–12. <http://arxiv.org/pdf/0801.3710>
 - [56] Nicola Santoro and Ramez Khatib. 1985. Labelling and implicit routing in networks. *The computer journal* 28, 1 (1985), 5–8.
 - [57] Atish Das Sarma and Amitabh Trehan. 2012. Edge-preserving self-healing: keeping network backbones densely connected. In *Workshop on Network Science for Communication Networks (NetSciCom 2012), IEEE InfoComm. IEEE Xplore*.
 - [58] Mikkel Thorup and Uri Zwick. 2001. Compact routing schemes. In *SPAA*. 1–10. <http://doi.acm.org/10.1145/378580.378581>
 - [59] Amitabh Trehan. 2010. *Algorithms for self-healing networks*. Dissertation. University of New Mexico. <http://proquest.umi.com/pqdlink?did=2085415901&Fmt=2&clientId=11910&RQT=309&VName=PQD>
 - [60] Amitabh Trehan. 2012. Self-healing using virtual structures. *CoRR* abs/1202.2466 (2012).