Near-Optimal Distributed Routing with Low Memory

Michael Elkin*[1] and Ofer Neiman[1]

[1]Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel.
Email: {elkinm,neimano}@cs.bgu.ac.il

Abstract

Distributed *routing* is one of the most central and fundamental problems in the area of Distributed Graph Algorithms. It was extensively studied for almost thirty years. Nevertheless, the currently existing solutions for this problem require either prohibitively large construction (aka preprocessing) time, or prohibitively large memory usage either during the construction or during the routing phase, and suffer from suboptimal labels and tables' sizes.

We devise a distributed routing scheme that enjoys the best of all worlds. Specifically, its construction time and memory requirements during the construction phase are near-optimal, and so is also the tradeoff between the sizes of routing tables and labels on the one hand, and the stretch on the other.

On the way to this result, we also improve upon existing solutions for the distributed exact *tree* routing problem. Previous solutions require $\Omega(\sqrt{n})$ memory, and provide tables and labels of size $O(\log n)$ and $O(\log^2 n)$, respectively.

Our solution, on the other hand, requires just $O(\log n)$ memory, and has tables of size $O(1)$, and labels of size $O(\log n)$. These bounds match the bounds of the best-known centralized solution.

*This research was supported by the ISF grant No. (724/15).

1 Introduction

A network is represented as a weighted undirected n-vertex graph $G = (V, E)$. A routing scheme has two main phases: the preprocessing phase, and the routing phase. In the preprocessing phase, each vertex is assigned a routing table and a routing label.[1] In the routing phase, a vertex u gets a message M with a short header $Header(M)$ and with a destination label $Label(v)$ of a vertex v, and based on its routing table $Table(u)$, on $Label(v)$, and on $Header(M)$, the vertex u decides to which neighbor $x \in \Gamma(u)$ to forward the message M, and which header to attach to the message. The *stretch* of a routing scheme is the worst-case ratio between the length of a path on which a message M travels, and the graph distance between the message's origin and destination.

Due to its both theoretical and practical appeal, routing is a central problem in distributed graph algorithms [PU89, ABNLP90, TZ01b, Cow01, EGP03, GP03, AGM04, Che13]. A landmark routing scheme was devised by Thorup and Zwick in [TZ01b]. For an integer $k \geq 1$, the stretch of their scheme is $4k-5$, the tables are of size $\tilde{O}(n^{1/k})$, the labels are of size $O(k \log n)$, and the headers are of size $O(\log n)$. Chechik [Che13] improved this result, and devised a scheme with stretch $3.68k$, and other parameters like in [TZ01b].

An active thread of research [ABNLP90, AP92, LP13, LP15, EN16b] focuses on efficient implementation of the *preprocessing* phase of routing in the distributed CONGEST model (see Section 2 for a definition of the model), i.e., computing compact tables and short labels that allow for future low-stretch routing. This problem was raised in a seminal paper by Awerbuch, Bar-Noy, Linial and Peleg [ABNLP90], who devised a routing scheme with stretch $2^{O(k)}$, overall memory requirement $\tilde{O}(n^{1+1/k})$,[2] individual memory requirement for a vertex v of $\tilde{O}(\deg(v) + n^{1/k})$, and construction time $\tilde{O}(n^{1+1/k})$ (in the CONGEST model). The "individual memory requirement" parameter encapsulates the routing tables and labels, and the memory used while computing the tables and labels.

Lenzen and Patt-Shamir [LP15] devised a distributed routing scheme (based on [TZ01b]) with stretch $4k-3+o(1)$, tables of size $\tilde{O}(n^{1/k})$, labels of size $O(k \log n)$, individual memory requirement of $\tilde{O}(n^{1/k})$, and construction time $\tilde{O}(S + n^{1/k})$, where S is the *shortest-path diameter* of the input graph G, i.e., the maximum number of hops in a shortest path between a pair of vertices in G. Though S is often much smaller than n, it is desirable to evaluate complexity measures of distributed algorithms in terms of n and D, where D is the *hop-diameter* of G, defined as the maximum distance between a pair of vertices u, v in the underlying unweighted graph of G. Typically, we have $D \ll S \ll n$, and it is always the case that $D \leq S \leq n$. (See Peleg's book [Pel00] for a comprehensive discussion.)

Lenzen and Patt-Shamir [LP13] also devised a routing scheme with tables of size $\tilde{O}(n^{1/2+1/k})$, labels of size $O(\log n \cdot \log k)$, stretch at most $O(k \log k)$, and construction time of $\tilde{O}(n^{1/2+1/k} + D) \cdot \log \Lambda$ rounds, where Λ is the ratio between the largest to the smallest distance in the graph. They (based on [SHK+12]) also showed a lower bound of $\tilde{\Omega}(D + \sqrt{n})$ on the time required to construct a routing scheme. In a follow-up paper, [LP15] showed how to improve the stretch of the above scheme to $O(k)$. The main drawback of this result is the prohibitively large size of the routing tables. (The individual memory requirement is consequently prohibitively large as well.) They also exhibited a different tradeoff, that overcame the issue of large routing tables. They devised an algorithm that produced routing tables of size $\tilde{O}(n^{1/k})$, labels of size $O(k \log^2 n)$ and stretch $4k - 3 + o(1)$, albeit with sub-optimal running time $\tilde{O}(\min\{(nD)^{1/2} n^{1/k}, n^{2/3} + D\}) \cdot \log \Lambda$,

[1] In this paper we only consider *labeled* or *name-dependent* routing, in which vertices are assigned labels by the scheme. There is also a large body of literature on name-independent routing schemes; cf. [AGM+08] and the references therein. However, a lower bound [LP13] shows that constructing a name-independent routing scheme with stretch ρ requires $\tilde{\Omega}(n/\rho^2)$ time in the CONGEST model.

[2] $O(\tilde{f}(n))$-notation hides polylogarithmic in $f(n)$ factors.

and no guarantee on the individual memory requirement during the preprocessing phase.[3] In [EN16b], the current authors improved the bounds of [LP13, LP15]. In the current state-of-the-art scheme [EN16b], the stretch is $4k-5+o(1)$, the tables and labels are of the same size as in [LP13, LP15] (i.e., $\tilde{O}(n^{1/k})$ and $O(k\log^2 n)$, respectively), the construction time is $O((n^{1/2+1/k}+D)\cdot\min\{(\log n)^{O(k)}, 2^{\tilde{O}(\sqrt{\log n})}\}\cdot\log\Lambda$. (A similar, though slightly weaker, result was independently achieved by [LPP16].) Still there is no meaningful guarantee on the individual memory requirement in the preprocessing phase. In the last three results the table size is larger by a factor of $\approx \log n$ than Thorup-Zwick's sequential construction (this is hidden by the $\tilde{O}$ notation). See Table 1 for a concise summary of existing bounds, and a comparison with our new results.

To summarize, all currently existing distributed routing algorithms with nearly-optimal running time $\approx D+n^{1/2+1/k}$ suffer from three issues. First, they provide no meaningful guarantee on individual memory requirement on vertices in the preprocessing phase; second, their tables and labels sizes are roughly $O(\log n)$ off from the respective tables and labels' sizes of Thorup-Zwick's sequential construction [TZ01b]; and third, their preprocessing time depends at least linearly on $\log\Lambda$.

We note that the quest to design routing schemes with small tables and labels is typically justified by inherent storage limitations of vertices that run the scheme. This justification is however inconsistent with allowing vertices to use a very large memory (much larger than the eventual size of routing tables and labels) during the preprocessing stage.

We devise an algorithm that addresses all these issues. Most importantly, the individual memory requirement in our algorithm is at most $\tilde{O}(n^{1/k})$, i.e., it is within polylogarithmic factor of the size of routing tables and labels (together), which is an obvious lower bound on the individual memory usage. Previous solutions used at least $\tilde{O}(n^{1/2})$ memory. In particular, for large k we can have polylogarithmic individual memory requirement and construction time $O((n^{1/2}+D)\cdot n^{\epsilon})$, for an arbitrarily small constant $\epsilon>0$. We can also reduce the running time to $(n^{1/2+1/k}+D)\cdot\min\{(\log n)^{O(\max\{k,\log\log n\})}, 2^{\tilde{O}(\sqrt{\log n})}\}$, while the individual memory increases slightly to $\max\{\tilde{O}(n^{1/k}), 2^{\tilde{O}(\sqrt{\log n})}\}$.

The stretch of our scheme is $4k-5+o(1)$, i.e., almost matching the stretch $4k-5$ of [TZ01b]. The sizes of tables and labels match the respective sizes of Thorup-Zwick's construction, i.e., they are $\tilde{O}(n^{1/k})$ and $O(k\log n)$, respectively. Note that as $1\le k=O(\log n)$, our label size is polynomially better than the previous bound $O(k\log^2 n)$. For constant k, the improvement is quadratic. In the opposite end of the spectrum, when $k=\log n$, our label size is $O(\log^2 n)$, instead of the previous $O(\log^3 n)$, but then our table size is polynomially better than the previous bound.

Finally, our construction time $(n^{1/2+1/k}+D)\cdot(\log n)^{O(\max\{k,\log\log n\})}$ is independent of Λ.[4]

Distributed Tree Routing: An important ingredient in the existing distributed routing schemes [LP15, EN16b, LPP16] for general graphs, and in our new routing scheme, is a distributed tree routing scheme. Thorup and Zwick [TZ01b] showed that with routing tables of size $O(1)$ and labels of size $O(\log n)$, one can have an *exact* (i.e., no stretch) tree routing. [LP15, EN16b] showed that in $\tilde{O}(D+\sqrt{n})$ time, one can construct exact tree routing with tables and labels of size $O(\log n)$ and $O(\log^2 n)$, respectively, i.e., there is an overhead of $\log n$ in both parameters with respect to Thorup-Zwick's centralized construction. In addition, both these solutions require $\tilde{O}(\sqrt{n})$ memory per vertex. (In this problem, one is given a graph G of hop-diameter D, and a spanning tree T of G. One then wishes to compute a tree routing scheme for T, using the fact that D is typically much smaller than the hop-diameter of T.) In this paper we significantly improve

[3] The paper [LP15] claimed label size $O(k\log n)$, but in [LP16] it was communicated to us that the actual size is $O(k\log^2 n)$.

[4] We assume that edge weights can be sent in a single message, which we believe is a natural assumption. If one does care about the bit complexity, in our solution the construction time is proportional to $\log_n\log\Lambda$, as opposed to $\Omega(\log\Lambda)$ in all previous solutions. See Section 2 for further discussion.

Reference	Number of Rounds	Table size	Label size	Stretch	Memory per vertex
[ABNLP90]	$O(n^{1+\frac{1}{k}})$	$\tilde{O}(n^{1/k})$	$O(k \log n)$	$2 \cdot 3^k - 1$	$\tilde{O}(\deg(v)+n^{\frac{1}{k}})$
[TZ01b]	NA	$\tilde{O}(n^{1/k})$	$O(k \log n)$	$4k-5$	NA
[Che13]	NA	$\tilde{O}(n^{1/k})$	$O(k \log n)$	$3.68k$	NA
[LP13, LP15]	$\tilde{O}(n^{\frac{1}{2}+\frac{1}{4k}}+D)$	$\tilde{O}(n^{\frac{1}{2}+\frac{1}{4k}})$	$O(\log n)$	$6k-1+o(1)$	$\tilde{O}(n^{\frac{1}{2}+\frac{1}{4k}})$
[LP15]	$\tilde{O}(S+n^{\frac{1}{k}})$	$\tilde{O}(n^{1/k})$	$O(k \log n)$	$4k-3$	$O(n^{\frac{1}{k}} \cdot \log n)$
	$\tilde{O}(\min\{(nD)^{\frac{1}{2}} \cdot n^{\frac{1}{k}}, n^{\frac{2}{3+\frac{2}{3k}}}+D\})$	$\tilde{O}(n^{1/k})$	$O(k \log^2 n)$	$4k-3+o(1)$	$\tilde{O}(n^{\frac{1}{2}})$
[LPP16]	$(n^{\frac{1}{2}+\frac{1}{k}}+D) \cdot 2^{\tilde{O}(\sqrt{\log n})}$	$\tilde{O}(n^{1/k})$	$O(k \log^2 n)$	$4k-3+o(1)$	$\tilde{O}(n^{\frac{1}{2}})$
[EN16b]	$(n^{\frac{1}{2}+\frac{1}{k}}+D) \cdot \beta$	$\tilde{O}(n^{1/k})$	$O(k \log^2 n)$	$4k-5+o(1)$	$\tilde{O}(n^{\frac{1}{2}})$
This paper	$(n^{\frac{1}{2}+\frac{1}{k}}+D) \cdot \gamma$	$\tilde{O}(n^{1/k})$	$O(k \log n)$	$4k-5+o(1)$	$\tilde{O}(n^{\frac{1}{k}})$
	$(n^{\frac{1}{2}+\frac{1}{k}}+D) \cdot 2^{\tilde{O}(\sqrt{\log n})}$	$\tilde{O}(n^{1/k})$	$O(k \log n)$	$4k-5+o(1)$	ϱ

Table 1: Comparison of distributed compact routing schemes for graphs with n vertices, hop-diameter D, and shortest path diameter S. Denote $\beta = \min\{(\log n)^{O(k)}, 2^{\tilde{O}(\sqrt{\log n})}\}$, $\gamma = (\log n)^{O(\max\{k, \log\log n\})}$, and $\varrho = \max\{\tilde{O}(n^{1/k}), 2^{\tilde{O}(\sqrt{\log n})}\}$. By setting $k = \epsilon \log n / \log\log n$, for a small constant $\epsilon > 0$, in the penultimate line one obtains *polylogarithmic* memory requirement, and construction time $(n^{1/2}+D) \cdot n^{O(\epsilon)}$.

Reference	Number of Rounds	Table size	Label size	Memory per vertex
[LP15, EN16b]	$\tilde{O}(D+\sqrt{n})$	$O(\log n)$	$O(\log^2 n)$	$\tilde{O}(\sqrt{n})$
[TZ01b]	NA	$O(1)$	$O(\log n)$	NA
This paper	$\tilde{O}(D+\sqrt{n})$	$O(1)$	$O(\log n)$	$O(\log n)$

Table 2: Comparison of distributed compact exact tree-routing schemes for graphs with n vertices and hop-diameter D.

this result, and devise an $\tilde{O}(D+\sqrt{n})$-time algorithm that constructs tree-routing tables and labels of sizes that match the sequential construction of Thorup and Zwick, i.e., of sizes $O(1)$ (instead of $O(\log n)$) and $O(\log n)$ (instead of $O(\log^2 n)$), respectively. Even more importantly, our algorithm requires only $O(\log n)$ (instead of $\tilde{O}(\sqrt{n})$) individual memory in each vertex. See Table 2 for a concise comparison.

1.1 Technical Overview

The basic approach in previously-existing distributed routing schemes [LP13, LP15, EN16a, LPP16] that have construction time $\approx D+\sqrt{n}$ is to sample a set V' of $\approx \sqrt{n}$ vertices, and to build a "virtual" graph $G' = (V', E', \omega')$ on them. The edge set E' consists of pairs $u', v' \in V'$, such that there exists a u'-v' path in G with $c\sqrt{n} \cdot \log n$ edges, for some fixed sufficiently large constant c. Such a path is said to be $c\sqrt{n} \cdot \log n$-bounded. The weight function $\omega'(u', v')$ is set to be the length of the shortest u'-v' $c\sqrt{n} \cdot \log n$-bounded path in G.

One then constructs a (β, ϵ)-hopset $G'' = (V', E'', \omega'')$ for G', for a positive integer *hop-bound* parameter β, and a positive parameter ϵ, i.e., a sparse graph that satisfies for every $x', y' \in V'$,

$$d_{G'}(x', y') \leq d^{(\beta)}_{G' \cup G''}(x', y') \leq (1+\epsilon) d_{G'}(x', y') \ ,$$

where $d^{(\beta)}_{G'}(x,' y')$ stands for the β-bounded distance between x' and y' in G', i.e., the length of the shortest β-bounded path between them. One then uses $G' \cup G''$ to build an efficient routing scheme for the vertices

of V', and then extends this partial routing scheme to the entire set V. (We remark that this entire approach was originated in Nanongkai's seminal paper [Nan14] on distributed approximate shortest paths.)

The computations of the virtual graph G' and the hopset G'' in [Nan14, LP13, LP15, EN16a, LPP16] are quite costly in terms of local memory. In particular, they require each of the virtual vertices $v' \in V'$ to store up to $|V'|-1 = \Omega(\sqrt{n})$ virtual edges (of E') incident on them. This alone makes the memory requirements of these distributed routing schemes $\Omega(\sqrt{n})$.

To overcome this issue, we use a novel hopset construction from our companion paper [EN17b]. This hopset construction, like the construction of exact hopsets from [Elk17], has the nice property that a hopset G'' for G' can be constructed without ever fully constructing the virtual graph G' itself. Instead, only those edges of G' that are really required for constructing the hopset G'' are computed. Moreover, the hopset G'' itself has a small *arboricity*[5] $\tilde{O}(n^{1/k})$, and thus every vertex $v' \in V'$ needs only to store its $\tilde{O}(n^{1/k})$ parents in the trees of the arboricity decomposition to which it belongs.

Note, however, that even once the hopset G'' of G' is constructed, one still needs the virtual graph G' itself to compute the routing scheme for it. This computation involves Bellman-Ford explorations to hop-depth β (the hop-bound of the hopset G'') in $G' \cup G''$. On the other hand, we cannot store the entire virtual graph G', as it would require some vertices to store $\Omega(\sqrt{n})$ pieces of information.

To resolve this hurdle, we compute only those edges of G' that are required for running these Bellman-Ford explorations *on the fly*, i.e., during these explorations, and as a result we never store (or even compute) the virtual graph G' in its entirety.

Finally, we outline our improved distributed *tree* routing scheme. Previously-existing schemes [LP15, EN16a] partitioned the tree T for which one builds the routing scheme into $|V'|$ subtrees $T(v')$, rooted at v', for every $v' \in V'$. This partition also induces a virtual tree T' on V'. The schemes of [LP15, EN16a] constructed a separate tree-routing scheme for T', and separate tree-routing schemes for each of the subtrees $\{T(v') \mid v' \in V'\}$. Constructing a tree routing scheme for T' involved broadcasting the entire virtual tree, storing it in local memory of all virtual vertices, and computing the scheme locally. This resulted in prohibitively high memory usage. Also, the combination of these routing schemes into a single tree-routing scheme for the original tree T increased the labels' and tables' sizes from $O(1)$ and $O(\log n)$, respectively, to $O(\log n)$ and $O(\log^2 n)$, respectively.

In the current paper we never construct a separate routing scheme for the virtual tree T'. Instead we implement the original Thorup-Zwick tree routing scheme. We use the partition $\{T(v') \mid v' \in V'\}$ of T to conduct local computations in the subtrees. We then incorporate the results of these local computations into a global routing scheme for the original tree T by applying pointer jumping ideas, which involve broadcasts and convergecasts of information associated with vertices of V'. This new approach enables us to achieve labels' and tables' sizes $O(1)$ and $O(\log n)$, respectively, matching the centralized bounds of Thorup and Zwick [TZ01b]. Even more importantly, it enables us to implement tree routing using logarithmic individual memory, as we never store the virtual tree T'. This resolves the last obstacle on our way to a low-memory distributed routing scheme for general graphs. We believe that the technique that we developed for the tree-routing problem will be found useful for other problems in which available memory is limited.

1.2 Structure of the Paper

In Section 3 we describe our tree-routing scheme. Some missing details of it can be found in Appendix A. Due to space limitations, our low-memory routing scheme for general graphs is deferred to Appendix B.

[5] The *arboricity* of a graph $G = (V, E)$ is the minimum number of forests needed to cover the edge set E of G. This collection of forests will be referred to as *arboricity forest decomposition*.

2 Preliminaries

Distributed Models. In the CONGEST model of computation, every vertex of an n-vertex weighted graph $G = (V, E)$ hosts a processor, and the processors communicate with one another in discrete rounds, via short (of size $O(\log n)$ bits) messages. In the CONGEST RAM variant of the CONGEST model, which we introduce here and in the companion paper [EN17a], each message is allowed to contain an identity of a vertex, an edge weight, a distance in the graph, or anything else of no larger (up to a fixed constant factor) size. The local computation is assumed to require zero time, and we are interested in algorithms that run for as few rounds as possible. A parameter of interest is the *hop-diameter* D of the graph, which is the diameter of G viewed as an unweighted graph. The following lemma formalizes the broadcast ability of a distributed network (see, e.g., [Pel00]).

Lemma 1. *Suppose every $v \in V$ holds m_v messages, each of $O(1)$ words, for a total of $M = \sum_{v \in V} m_v$. Then all vertices can receive all the messages within $O(M + D)$ rounds.*

Hopsets. Let $G = (V, E)$ be a weighted undirected graph. Let d_G denote the shortest path metric on G. For an integer t and $u, v \in V$, the *t-bounded distance* $d^{(t)}_G(u, v)$, is the length of the shortest path between u, v that contains at most t edges, aka hops (note that $d^{(t)}_G$ is not a metric). A *(β, ϵ)-hopset* H with weight function ω_H is a set of edges such that for any $u, v \in V$,

$$d_G(u, v) \leq d^{(\beta)}_{G \cup H}(u, v) \leq (1 + \epsilon) \cdot d_G(u, v),$$

i.e., every pair has an approximate shortest path containing at most β hops in $G \cup H$.

Path recovery mechanism. Bearing in mind the current applications of hopsets to approximate shortest paths and routing schemes, our path recovery mechanism enables the vertices on paths implementing hopset edges to compute distances to a certain set of roots (which correspond to the sources of shortest paths we would like to compute, or to the roots of trees in which routing is conducted). We say that a hopset H has a *path-recovery mechanism*, if every edge $e \in H$ corresponds to a path $P(e)$ with $\omega(P(e)) = \omega_H(e)$ in G, and also the following property holds. Suppose we are given a collection of root vertices $R \subseteq V$, so that each hopset edge $e = (x, y)$ is associated with a subset of them $R(e) \subseteq R$, and has approximate distance $\hat{d}(x, z)$ from x to each $z \in R(e)$ (and also the approximate distance from y, since e knows its own weight). Then there exists a *protocol* that enables one to inform each $v \in V$ on $R(v) = \bigcup_{e \,:\, v \in P(e)} R(e)$, and also to compute the approximate distances to all roots in $R(v)$. The approximate distance $\hat{d}(v, z)$ to $z \in R(v)$ is bounded by $d_{P(e)}(v, x) + \hat{d}(x, z)$. In addition, v will know of a *parent*, a neighbor in some path $P(e)$, so that $v \in P(e)$, implementing $\hat{d}(v, z)$. The running time of this protocol is $\tilde{O}(|H| \cdot C + D) \cdot \beta$, where $C = \max_{v \in V} |R(v)|$, and the required memory per vertex is proportional to that required by the hopset construction.

A *virtual graph* is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $d_G(u, v) \leq d_{G'}(u, v)$ for every $u, v \in V'$. (In our setting the edges of E' correspond to $\tilde{O}(n/|V'|)$-bounded distances in G.) The following theorem is shown in [EN17a].

Theorem 1. *Let $G = (V, E)$ be any weighted graph on n vertices with hop-diameter D, given an integer $\kappa > 1$, and parameters $0 < \rho < 1$, $0 < \epsilon < 1/5$, and a virtual graph $G' = (V', E')$ embedded in G on $|V'| = \Theta(\sqrt{n})$ vertices, where E' corresponds to $B = \tilde{O}(\sqrt{n})$-bounded distances in G. Then there is a distributed algorithm in the CONGEST model that runs in $\tilde{O}(n^{(1+\rho)/2} + D) \cdot \beta$ rounds, that computes a*

(β, ϵ)-hopset H with a path-recovery mechanism, of size at most $O(n^{(1+1/\kappa)/2})$, where

$$\beta = O\left(\frac{(\log\kappa + 1/\rho)\cdot\log n}{\epsilon}\right)^{\log\kappa+1/\rho+1},$$

and both the arboricity of H and the internal memory per vertex are bounded by $\tilde{O}(n^{\rho/2})$.

Bellman-Ford algorithm. The classical Bellman-Ford is an algorithm to compute single source shortest paths in a graph. Given a root $u \in V$, every vertex $v \in V$ holds a distance estimate $b_u(v)$. Initially, $b_u(u) = 0$ and all other estimates are ∞. The algorithm is executed in iterations. In every iteration v communicates $b_u(v)$ to its neighbors, and updates $b_u(v)$ according to the messages it received from its neighbors and the edge weights to those neighbors. The following lemma argues that given a bounded arboricity hopset, one can efficiently compute approximate shortest paths with low memory for each vertex. (Its proof appears also in [EN17a]. We provide full details for the sake of completeness.)

Lemma 2. *Let $G'' = (V', E' \cup H)$ be a virtual graph on m vertices embedded in an n-vertex graph $G = (V, E)$ of hop-diameter D, such that edges in E' correspond to B-bounded distances in G, and H has arboricity α. Then one can compute distances from a given root vertex (or set), that are at most those given by β iterations of Bellman-Ford in G'' (and no less than those in G) in the CONGEST model, within $O((m\cdot\alpha + B + D)\cdot\beta\cdot\log n)$ rounds, so that every vertex requires only $O(\alpha + \log n)$ memory.*

Proof. To implement a single iteration of the Bellman-Ford exploration, every vertex $v \in V'$ which holds a current distance estimate $b(v)$ will need to communicate it to its neighbors in G''. First it will initiate an exploration in G for B rounds. In each round, every vertex $u \in V$ will forward the smallest value it received so far. This guarantees that if $\{v, x\} \in E'$, then x will receive a value at most $b(v) + \omega'(v, x)$ (where ω' is the edge weight of $\{v, x\}$ in E').

Next we have to handle the edges of H. Recall that every $v \in V'$ knows of at most α such edges incident on it. Let T be a spanning tree of G with hop-depth D. Every $v \in V'$ will broadcast via T its current distance estimate to the entire graph, and will also send all the existing edges of H incident on it that v knows about. All vertices $w \in V'$ that know of a hopset edge $\{v, w\}$ (or that learn about it from v's message) will update their value accordingly. Since there are $O(m\cdot\alpha)$ messages, by Lemma 1 this can be done in $O(m\cdot\alpha+D)$ rounds. In order to guarantee small internal memory, each v selects at random a number from $\{1, 2, \ldots, m\cdot\alpha\}$ for each message it sends, as a round to start its broadcast (clearly this increases the number of rounds by at most $m\cdot\alpha$); Since each message of v will reach every vertex of T at most once, the probability that some $u \in V$ receives t messages in a single round is at most $\binom{m\cdot\alpha}{t}\cdot 1/(m\cdot\alpha)^t \le (e/t)^t$, thus with high probability no vertex will receive more than $O(\log n)$ messages each round (recall there are n vertices in T). By increasing the number of rounds by a factor of $O(\log n)$, whp there will be no congestion. The total number of rounds required is thus $O(m\cdot\alpha + B + D)\cdot\beta\cdot\log n$, and every vertex stores only the hopset edges, its distance estimate, and the $O(\log n)$ messages it needs to deliver at each round. □

To adapt our algorithm to the standard CONGEST model (i.e., with messages of size $O(\log n)$, rather than messages that can contain $O(1)$ edge weights and Identity numbers), we round all edge weights to the closest power of $(1+\epsilon)$. As a result, each edge weight can now be represented with $O(\log\log\Lambda + \log 1/\epsilon)$ bits, and so the overhead for implementing our algorithm in this model is $O\left(\frac{\log\log\Lambda+\log 1/\epsilon}{\log n}\right)$. (The effect of this rounding on approximation guarantee of the hopset is minor; ϵ is rescaled by a constant factor, and as a result constants hidden by O-notation in other parameters grow.) This is in contrast to previous solutions [LP15, EN16a, LPP16], whose running time is at least linear in $\log\Lambda$.

3 Distributed Tree Routing with Small Memory

In this section we present our exact compact routing scheme for trees that can be efficiently computed in a distributed manner, whose tables' and labels' sizes matches the best known sequential construction, while using small internal memory. In previous constructions of distributed routing schemes for trees [EN16b, LPP16], the internal memory was as high as $\Omega(\sqrt{n})$, and it was also somewhat inefficient: the label size is $O(\log^2 n)$ and the routing tables are of size $O(\log n)$. Compare this to the classical [TZ01b] tree routing, which has label size $O(\log n)$ and routing tables of size $O(1)$.

We select a set $U \subseteq V$, such that each vertex is sampled to U independently with probability $q \leq 1/\sqrt{n}$ (the parameter q will be chosen later). Fix a tree T on vertices $V(T) \subseteq V$ with root z. The vertices $U(T) = (U \cap V(T)) \cup \{z\}$ induce a partition of T into subtrees, by removing the edges from each vertex in $U(T) \setminus \{z\}$ to its parent. Each of the $|U(T)|$ subtrees is rooted in a vertex of $U(T)$. Denote by T_w the subtree rooted at w. We call the trees T_w the *local* trees, and T is the global tree. We also consider T', the virtual tree on the vertices of $U(T)$, which is rooted at z, and contains an edge (x, y) if the T-parent of y lies in T_x. It is not hard to see (e.g., [EN16b]) that whp the depth of each T_w is $\tilde{O}(1/q)$ and that $|U| \leq O(qn)$.

In both [EN16b, LPP16], routing schemes were created for each T_w, and also a routing scheme for the virtual tree T'. This computation required large internal memory, since z had to locally compute the scheme for T'. The inefficiency in the size was due to the fact that when routing in T', traveling over a virtual edge (x, y), one has to route in T_x from x to the parent of y. This requires storing additional routing information for this subtree, increasing both label and table size. We overcome this issue by storing routing information only with respect to the actual tree T, while applying pointer jumping techniques and using broadcasts for the virtual tree T' to quickly compute the full labels.

Before describing our approach, let us briefly recall the Thorup-Zwick construction of tree routing. The idea is to assign to every (non-leaf) vertex $x \in T$ its *heavy child*, which is the child whose subtree has maximal size. Note that the subtree of any non-heavy child of x contains at most 1/2 of the vertices that the tree rooted at x contains. For this reason, any path from the root z to some $y \in T$ contains at most $\log n$ non-heavy edges. They also conduct a DFS search in T that assigns to each y the DFS entry and exit times for its subtree. The label of y is these entry and exit times, and also the names of the non-heavy edges on the z to y path. The routing table y stores consists of its DFS times, the name of the heavy child, and the name of the parent of y in the tree. The routing towards a target v in the tree is done as follows. At any intermediate vertex $y \in T$, if v is not in the subtree rooted at y (can be checked via the DFS times) then y forwards to its parent. If v is in the subtree, y inspects v's label to see if an edge (y, x) is written there. If this is the case, it forwards the message to x. Otherwise it forwards the message to its heavy child.

Now we show how to implement this scheme in a distributed manner, and with $O(\log n)$ internal memory. Our algorithm is composed of several stages, in these stages we compute the subtree sizes (to infer heavy children), the light edges on the path to the root, and the DFS entry and exit times. Each stage has three steps, first we compute the appropriate information for each local tree T_w, and then use pointer jumping to aggregate the local pieces and create a structure for the global tree T, but only for the vertices in $U(T)$. The last step uses each local tree to send the global information to all T, inside each T_w in parallel.

3.1 Stage 1: Computing Subtree Sizes

Computing local subtree sizes. Initially every vertex $y \in T$ only knows that it is in T and its parent $p(y)$. We begin by informing each vertex in which local tree T_w it lies. Every $w \in U(T)$ sends a message about itself to the vertices of T_w, by sending it to its children. Every vertex (in $V(T) \setminus U(T)$) receiving this message infers it is in T_w, and forwards the message to its own children. Note that this message will arrive

to every vertex $x \in U(T)$ who is a child of w in the virtual tree T' (but x will not forward this message to its children), so x will know its (virtual) parent $p'(x)$ in T'.

In order to compute the local subtree sizes, for each $w \in U(T)$, every vertex in T_w sends to its parent the size of the subtree of T_w rooted at it, beginning with the leaves. Every vertex that received messages from all its children, sums up the values, adds 1, and sends this to its own parent. This can be done in parallel for all trees T_w for $w \in U(T)$, and will take $\tilde{O}(1/q)$ (the bound on the height of each T_w) rounds. When this stage concludes, every $x \in U(T)$ knows $|T_x|$.

Computing global subtree sizes. For each $x \in U(T)$, its subtree size is exactly the sum of sizes of subtrees T_w for $w \in U(T)$ that are descendants of x in T'. Note that computing these values from the leaves of T' up will not be efficient, since every message on a virtual edge may require $O(D)$ rounds, and the depth of T' may be as large as qn (which will be approximately $\sqrt{n}$), and thus will result in $O(D\sqrt{n})$ rounds. To alleviate this issue, we use the following "pointer jumping" idea. For a vertex v in the virtual tree T', and a positive integer h, we say that a vertex u is an *h-ancestor* of v, if u lies on the unique $v - z$ path in T' with h hops (virtual edges) from v. Denote by $a_i(x)$ the 2^i-ancestor of x (in particular, $a_0(x)$ is $p'(x)$). We run Algorithm 1 to compute the sizes. (By broadcast we mean using the BFS tree of G to send the messages to every vertex of the graph.)

Algorithm 1 `Global subtree sizes`

for $x \in U(T)$ **do**
  $a_0(x) = p'(x)$; (for the root $a_0(z) = \perp$);
  $s_0(x) = |T_x|$;
end for
for $i = 0, 1, \ldots, \log n - 1$ **do**
  for $x \in U(T)$ **do**
    Broadcast $s_i(x)$ and $a_i(x)$;
    $a_{i+1}(x) = a_i(a_i(x))$;
    $s_{i+1}(x) \leftarrow s_i(x)$;
    for every $w \in U(T)$ such that $x = a_i(w)$ **do**
      Increase $s_{i+1}(x)$ by $s_i(w)$;
    end for
  end for
end for

When the algorithm concludes, denote $s_x = s_{\log n}(x)$. It is not hard to verify that the ancestors $a_i(x)$ are computed correctly.

Claim 3. *For every $x \in U(T)$ we have that s_x is the size of the subtree of T rooted at x.*

Proof. Denote by $T_i(x)$ the maximal subtree (of T) rooted at x that contains at most 2^i vertices of $U(T)$ on any root-leaf path. Clearly $T_{\log n}(x)$ is the subtree of T rooted at x. We prove by induction on i, that $s_i(x) = |T_i(x)|$. The base case $i = 0$ holds by the initial setup, since $T_x = T_0(x)$ is indeed the maximal subtree containing a single vertex of $U(T)$ (the vertex x itself) on any root-leaf path. For $i \geq 0$, at the beginning of the i-th round, each $w \in U(T)$ that has $x = a_i(w)$, holds $s_i(w) = |T_i(w)|$ (by the inductive assumption). Observe that these w are exactly the children of vertices in $T_i(x)$ that lie outside $T_i(x)$, so the

size of $T_{i+1}(x)$ is equal to the sum of these $|T_i(w)|$ together with $|T_i(x)|$. The algorithm ensures $s_{i+1}(x)$ is updated accordingly. □

As there are $O(|U(T)|) \leq \tilde{O}(qn)$ messages sent each iteration for $\log n$ iterations, by Lemma 1 it will take $\tilde{O}(qn + D)$ rounds to implement this algorithm. Each vertex $x \in U(T)$ stores $\{a_i(x)\}_{i=0,1,\ldots,\log n}$ for future use, which requires only $O(\log n)$ memory words.

In order to compute s_y, the size of the subtree of T rooted at y, for all $y \in T$, every $x \in U(T)$ informs its parent in T, $p(x)$, on s_x. Then $p(x)$ updates its size by adding s_x. Once again, for every $w \in U(T)$ in parallel, the leaves of T_w send to their parents their current size. This time, some of these leaves and internal vertices could be parents of vertices in $U(T)$, so these sizes are the actual subtree size in T. In $\tilde{O}(1/q)$ rounds, every vertex $y \in T$ will know s_y. By informing these values to the parents, every vertex can infer who is its heavy child.

3.2 Stage 2: Computing the light edges on the path to the root

Light edges in the local trees. For $y \in T$, its label contains the collection of edges $\{(u, v)\}$ that are on the z to y path in T, such that v is not the heavy child of u. These are the light edges. As noted above, there can be at most $\log n$ such edges on this path. If $y \in T_x$ for some $x \in U(T)$, we start by computing the list of light edges on the path from x to y. We execute Algorithm 2, where $b = \tilde{O}(1/q)$ is the maximal height of a local tree. Note that every vertex $u \in T_x$ will update the lists for all its children, in particular, every

Algorithm 2 `Local light edges`

```
for every u ∈ T do
    L(u) = ⊥;
end for
for every x ∈ U(T) in parallel do
    L(x) = ∅;
    for j = 1, 2, ..., b do
        for every u ∈ T_x of height j, with heavy child v and other children v_1, ..., v_r do
            L(v) = L(u);
            for i = 1, ..., r do
                L(v) = L(u) ∪ {(u, v)};
            end for
        end for
    end for
end for
```

(virtual) child w of x in T' will have a list $L(w)$ that contains all the light edges on the path in T from x to w. Also observe that u does not need to store any information about its non-heavy children, just to send them the message $L(u)$ to which they will append the appropriate edge. This stage clearly runs in $\tilde{O}(1/q)$ rounds, the maximum height of any local tree.

Light edges in the global tree. We again apply pointer jumping to compute all the light edges. The following Algorithm 3 computes this for vertices $x \in U(T)$. The value $L_i(x)$ denote the list of edges x has just before the i-th iteration of the algorithm, and the final list will be $L_{\log n}(x)$.

Algorithm 3 `Global light edges`

for $x \in U(T)$ **do**
  $L_0(x) = L(x)$;
end for
for $i = 0, 1, \ldots, \log n - 1$ **do**
  for $x \in U(T)$ **do**
    Broadcast $L_i(x)$;
    $L_{i+1}(x) \leftarrow L_i(a_i(x)) \cup L_i(x)$;
  end for
end for

Claim 4. *For every $x \in U(T)$, $L_{\log n}(x)$ contains all the light edges in the path from x to z in T.*

Proof. We prove by induction on i that for every $x \in U(T)$, $L_i(x)$ is the set of light edges on the path in T from $a_i(x)$ to x. The base case holds by the initial setup $L_0(x) = L(x)$, which is correct by Algorithm 2. The induction step follows by the value set for L_{i+1}, and by the induction hypothesis on x and $a_i(x)$ (since the set of light edges on the path from x to $a_{i+1}(x)$ is equal to the union of those on the path from x to $a_i(x)$, and those on the path from $a_i(x)$ to $a_i(a_i(x)) = a_{i+1}(x)$). □

Since every list has size $O(\log n)$, it will take $\tilde{O}(qn + D)$ rounds to implement this stage. It remains to update the lists for all $y \in T$. In another $\tilde{O}(1/q)$ rounds, each $x \in U(T)$ sends its updated list $L_{\log n}(x)$ to every vertex $y \in T_x$, and they update their list by appending $L_{\log n}(x)$. This is the correct list, since the light edges on the z to y path are exactly those on the z to x path, together with the light edges $L(y)$ in the local tree T_x.

The computation of DFS entry and exit times is deferred to Appendix A.

Choice of parameter q. If one desires a routing scheme for a single tree, just take $q = 1/\sqrt{n}$, so the running time will be $\tilde{O}(\sqrt{n} + D)$. If we desire to compute a routing scheme in parallel for multiple trees, but have the guarantee that every $v \in V$ belongs to at most s trees, (this is the case when we apply the tree-routing scheme for routing in general graphs), then pick $q = 1/\sqrt{sn}$, and a random start time for each tree, sampled uniformly from $\{1, \ldots, O(\sqrt{sn} \cdot \log n)\}$. Using an argument as in [EN16b], we obtain whp running time $\tilde{O}(\sqrt{sn} + D)$ (rather than the naive $\tilde{O}(s \cdot \sqrt{n} + D)$). We conclude with formally describing our result.

Theorem 2. *For any tree T on n vertices, lying in a network with hop-diameter D, there exists a distributed randomized algorithm in the CONGEST model, that whp runs in $\tilde{O}(\sqrt{n} + D)$ rounds, and computes an exact tree routing scheme with label size $O(\log n)$ and routing tables of size $O(1)$, such that every vertex uses only $O(\log n)$ words of memory throughout the computation.*

In addition, given a network with n vertices and a set of trees so that each vertex is contained in at most s trees, one can compute an exact tree routing scheme as above for all trees in parallel, within $\tilde{O}(\sqrt{sn} + D)$ rounds, while using memory $O(s \cdot \log n)$ at each vertex.

Due to space limitations, our routing scheme for general graphs is deferred to Appendix B.

Acknowledgements

We are grateful to an anonymous reviewer for his helpful comments about the path-recovery mechanism.

References

[ABNLP90] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, 11(3):307–341, September 1990.

[AGM04] Ittai Abraham, Cyril Gavoille, and Dahlia Malkhi. Routing with improved communication-space trade-off. In *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, pages 305–319, 2004.

[AGM+08] Ittai Abraham, Cyril Gavoille, Dahlia Malkhi, Noam Nisan, and Mikkel Thorup. Compact name-independent routing with minimum stretch. *ACM Trans. Algorithms*, 4(3):37:1–37:12, July 2008.

[AP92] B. Awerbuch and D. Peleg. Routing with polynomial communication-space tradeoff. *SIAM J. Discrete Mathematics*, 5:151–162, 1992.

[Che13] Shiri Chechik. Compact routing schemes with improved stretch. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 33–41, 2013.

[Cow01] Lenore Cowen. Compact routing with minimum stretch. *J. Algorithms*, 38(1):170–183, 2001.

[EGP03] Tamar Eilam, Cyril Gavoille, and David Peleg. Compact routing schemes with low stretch factor. *J. Algorithms*, 46(2):97–114, 2003.

[Elk17] Michael Elkin. Distributed exact shortest paths in sublinear time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 757–770, New York, NY, USA, 2017. ACM.

[EN16a] Michael Elkin and Ofer Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 128–137, 2016.

[EN16b] Michael Elkin and Ofer Neiman. On efficient distributed construction of near optimal routing schemes: Extended abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 235–244, New York, NY, USA, 2016. ACM.

[EN17a] Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and constant-hopbound hopsets in RNC, 2017. To be published.

[EN17b] Michael Elkin and Ofer Neiman. Linear-size hopsets with small hopbound, and distributed routing with low memory. *CoRR*, abs/1704.08468, 2017.

[GP03] Cyril Gavoille and David Peleg. Compact and localized distributed data structures. *Distributed Computing*, 16(2-3):111–120, 2003.

[LP13] Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 381–390, 2013.

[LP15] Christoph Lenzen and Boaz Patt-Shamir. Fast partial distance estimation and applications. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 153–162, 2015.

[LP16] Christoph Lenzen and Boaz Patt-Shamir. Personal communication, 2016.

[LPP16] Christoph Lenzen, Boaz Patt-Shamir, and David Peleg. Distributed distance computation and routing with small messages, 2016. manuscript.

[Nan14] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 565–573, 2014.

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[PU89] David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *J. ACM*, 36(3):510–530, 1989.

[SHK+12] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

[TZ01a] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. of the 33rd ACM Symp. on Theory of Computing*, pages 183–192, 2001.

[TZ01b] Mikkel Thorup and Uri Zwick. Compact routing schemes. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 1–10, New York, NY, USA, 2001. ACM.

A DFS entry and exit times for Distributed Tree Routing

Here we complete our distributed tree routing algorithm and present the third stage, which is computing the DFS entry and exit times.

Local DFS. First, for every $x \in U(T)$ we conduct a "parallel" DFS in T_x to assign local DFS range for each $u \in T_x$. Execute Algorithm 4, where $R(x)$ denotes the range given to x.

Initially, x holds the range $[1, s_x]$, and it assigns ranges to its children according to the sizes of their respective subtrees in T (recall that s_x is the size of the subtree of T rooted at x) in parallel. In the next round, each child assigns a range to its own children, and so on. We remark that a vertex y does not need memory proportional to its degree in order to inform its children $y_1, \ldots, y_r$ on their appropriate ranges. We just need to assume there is some order on these children (given by the port numbers, say). Rather, this can implemented in $O(\log n)$ rounds (in parallel for all vertices) as follows. If y has range $[a, b]$, we would like that the child y_j will know of a and $S(y_j) = \sum_{h=1}^{j} s_{y_h}$. This can be achieved by executing Algorithm 5.

We start with an informal sketch of this algorithm. Consider a vertex y that has children $y_1, \ldots, y_r$. Each child y_j has a value s_{y_i} (the size of its subtree), and we want y_j to know the sum $S(y_j) = \sum_{h=1}^{j} s_{y_h}$. To perform this computation using small memory, we use Algorithm 5 in all internal vertices $y \in T$.

Algorithm 4 `Local DFS`

for every $x \in U(T)$ in parallel **do**
  $R(x) = [1, s_x]$;
  for $j = 1, 2, \ldots, b$ **do**
    for every $u \in T_x$ of height j, with $R(u) = [a, b]$ and children $v_1, \ldots, v_r$ **do**
      for $i = 1, \ldots, r$ **do**
        $R(v_i) = [a + 1 + \sum_{h=1}^{i-1} s_{v_h}, a + \sum_{h=1}^{i} s_{v_i}]$;
      end for
    end for
  end for
end for

Algorithm 5 `Range partition`

for every $y \in T$ in parallel **do**
  Let $y_1, \ldots, y_r$ be the children of y;
  for $j = 1, \ldots, r$ **do**
    $S_0(y_j) = s_{y_j}$;
  end for
  for $i = 0, 1, \ldots, \log n - 1$ **do**
    for $t = 1, \ldots, \lceil r/2^{i+1} \rceil$ in parallel **do**
      for $j = (2t-2) \cdot 2^i + 1, \ldots, (2t-1) \cdot 2^i$ in parallel **do**
        $S_{i+1}(y_j) \leftarrow S_i(y_j)$;
      end for
      for $j = (2t-1) \cdot 2^i + 1, \ldots, 2t \cdot 2^i$ in parallel **do**
        $S_{i+1}(y_j) \leftarrow S_i(y_j) + S_i(y_{(2t-1)\cdot 2^i})$;
      end for
    end for
  end for
end for

Assume for simplicity that $r = 2^\lambda$ is an exact power of 2. The algorithm runs for λ phases, and each phase lasts for two rounds. In the first phase, in the first round, for all $t = 0, 1, \ldots, r/2 - 1$, vertices y_{2t+1} send their $s_{y_{2t+1}}$ to y. In the second round, the vertex y forwards $s_{y_{2t+1}}$ to y_{2t+2} (without storing it), and y_{2t+2} locally computes $s_{y_{2t+1}} + s_{y_{2t+2}}$, for every index t as above.

In the second phase, in the first round, every vertex y_{4t+2}, $t = 0, 1, \ldots, r/4 - 1$, sends the message $s_{y_{4t+1}} + s_{y_{4t+2}}$ to y. Then y forwards it to y_{4t+3} and y_{4t+4}. These vertices compute $s_{y_{4t+1}} + s_{y_{4t+2}} + s_{y_{4t+3}}$ and $s_{y_{4t+1}} + s_{y_{4t+2}} + s_{y_{4t+3}} + s_{y_{4t+4}}$, respectively. Continuing in this way for λ phases, we reach the situation that all vertices $y_1, \ldots, y_r$ know their respective values $S(y_1), \ldots, S(y_r)$.

We note that in the algorithm vertex y is responsible for sending information between its children, but we do not need to do any processing on it. Rather, y simply directs in iteration i the message from $y_{(2t-1)\cdot 2^i}$ to the relevant y_j (for all t; see Algorithm 5).

Claim 5. $S_{\log n}(y_j) = \sum_{h=1}^{j} s_{y_h}$.

Proof. The claim is shown via induction on i. We claim that for any positive integer t, every vertex with index $j \in [(t-1) \cdot 2^i + 1, t \cdot 2^i]$ has $S_i(y_j) = \sum_{h=(t-1)\cdot 2^i+1}^{j} s_{y_h}$. The claim will follow by plugging in $t = 1$

and $i = \log n$. The base case $i = 0$ asserts that for every index $j = t$, $S_0(y_j) = s_{y_j}$, which is true by the initial assignment. To show the induction step, fix $j \in [(t-1)\cdot 2^{i+1}+1, t\cdot 2^{i+1}]$. The first case is that j lies in the first half of this range, i.e. $j \in [(t-1)\cdot 2^{i+1}+1, (t-1)\cdot 2^{i+1}+2^i]$. Then by the induction hypothesis with $t' = 2t-1$, since $j \in [(t'-1)\cdot 2^i+1, t'\cdot 2^i]$ we have $S_i(y_j) = \sum_{h=(t'-1)\cdot 2^i+1}^{j} s_{y_h} = \sum_{h=(t-1)\cdot 2^{i+1}+1}^{j} s_{y_h}$, and observe that for j in this range we set $S_{i+1}(y_j) = S_i(y_j)$, which is correct value for the induction step. The second case is that j lies in the second half of the range, i.e. $j \in [(t-1)\cdot 2^{i+1} + 2^i + 1, t\cdot 2^{i+1}]$. Then by the induction hypothesis with $t' = 2t$, we have $j \in [(t'-1)\cdot 2^i + 1, t'\cdot 2^i]$, and thus $S_i(y_j) = \sum_{h=(t'-1)\cdot 2^i+1}^{j} s_{y_h} = \sum_{h=(t-1)\cdot 2^{i+1}+2^i+1}^{j} s_{y_h}$. That is, for $S_{i+1}(y_j)$ we are missing the 2^i elements in positions $(t-1)\cdot 2^{i+1}+1, \ldots, (t-1)\cdot 2^{i+1}+2^i$. Note that the last position is $(2t-1)\cdot 2^i$. By the induction hypothesis on $y_{(2t-1)\cdot 2^i}$, we have that indeed $S_i(y_{(2t-1)\cdot 2^i}) = \sum_{h=(t-1)\cdot 2^{i+1}+1}^{(2t-1)\cdot 2^i} s_{y_h}$ holds these missing values, and the algorithm adds them to $S_{i+1}(y_j)$. □

Note that the children of x in T' will receive these messages as well, i.e., if w is a child of x in T', it will receive a range of size s_w (but will not forward it on). The number of rounds is once again $\tilde{O}(1/q)$.

Global DFS. In this stage we need to shift each DFS range so that they will correspond to a DFS on the entire tree T. Observe that for the root z, each vertex in T_z has the correct range (since the subtree sizes used in the local DFS correspond to the global tree T). Consider some $w \in U(T)$ which is a child of z in T'. Then in the parallel DFS on T_z, w received (from its parent in T) a range of size s_w, say $[q_w+1, q_w+s_w]$, so it needs to "shift" the range of each vertex in T_w by q_w, in order to agree with the DFS search of the root z. We say that z induces a q_w shift for w. In fact, every vertex $x \in U(T)$ is informed about its range, say $[q_x+1, q_x+s_x]$, from its parent. For the root z, $q_z = 0$. In general, observe that such a vertex x, that on the path from x to z has $z = w_0, w_1, \ldots, w_b = x$ vertices of $U(T)$, will need to shift its range $[1, s_x]$ by $\sum_{i=0}^{b} q_{w_i}$. The following Algorithm 6, using pointer jumping, will incur the appropriate shifts efficiently.

Algorithm 6 `Global DFS`

```
for x ∈ U(T) do
    q_0(x) = q_x;
end for
for i = 0, 1, ..., log n − 1 do
    for x ∈ U(T) do
        Broadcast q_x;
        q_{i+1}(x) ← q_i(x) + q_i(a_i(x));
    end for
end for
```

It can be shown by induction on i, that before the i-th round, $q_i(x)$ is the DFS range shift induced by the first 2^i ancestors of x in T'. This algorithm takes $O(qn + D)$ rounds. When the algorithm concludes, each $x \in U(T)$ in parallel informs all the vertices in T_x on the required shift $q_{\log n}(x)$, which takes $\tilde{O}(1/q)$ rounds.

B Distributed Routing in General Graphs

In this section we use the hopsets of [EN17a] with their path-recovery mechanism, combined with our novel distributed tree routing, and design a compact routing scheme that can be efficiently computed in a distributed manner with low memory throughout the computation. Let $G = (V, E)$ be a weighted graph with n vertices and hop-diameter D. Fix a parameter $k > 1$.

We briefly sketch the approach of [EN16b] and the current improvement allowing low memory and improved bounds. First construct the Thorup-Zwick hierarchy $V = A_0 \supseteq A_1 \supseteq \ldots \supseteq A_k = \emptyset$, where each vertex in A_{i-1} is sampled to A_i independently with probability $n^{-1/k}$. Then the cluster $C(v) = \{u \in V : d_G(u, v) < d_G(u, A_{i+1})\}$ for $v \in A_i \setminus A_{i+1}$ can be viewed as tree rooted at v. Computing this cluster is done by a limited Dijkstra exploration from v, i.e., only vertices in $C(v)$ continue the exploration of v. Routing from x to y is done by finding an appropriate cluster $C(v)$ containing both x, y, and routing in that tree. Whenever $i < k/2$ these trees have whp depth $\tilde{O}(\sqrt{n})$, so they can be easily computed in a distributed manner within $\tilde{O}(n^{1/2+1/k})$ rounds. The main issue is computing the clusters for $i \geq k/2$.

The method of [EN16b] was to work with a virtual graph G', whose vertices are $V' = A_{k/2}$, and whose edges correspond to $B = c \cdot \sqrt{n} \log n$-bounded distances in G between the vertices of V'. Then a hopset is computed for this virtual graph, which enables the computation of Bellman-Ford explorations in only $O(\beta) \approx (\log n)^{O(k)}$ rounds. The fact that β-bounded distances can suffer $1 + \epsilon$ stretch creates additional complications; one needs to define *approximate clusters*, and make sure that these approximate clusters correspond to actual trees in G. Finally, since the trees corresponding to $C(v)$ for the high level vertices $v \in A_i$, $i \geq k/2$ can have large depth, one needs to adapt the Thorup-Zwick routing scheme for trees [TZ01b]. In both [EN16b, LPP16] this adaptation significantly increased both the table and label size, and required large memory.

Our improved result has two main ingredients. First, we do not explicitly construct G'; In both [EN16b, LPP16], computing the weights of edges in G' was a rather expensive step, and required large memory and induced factor depending logarithmically on the aspect ratio to the running time. In addition, only approximate values were obtained. We observe that not all the edges of G' are required for the algorithm, and thus we do not compute G' at all. Instead, we conduct the explorations in G' by implementing in each iteration of Bellman-Ford a B-bounded search in G, which not only saves memory and running time, but also simplifies the analysis, since now there is no error in the edge weights of G'. Second, our new tree-routing scheme has both improved label and routing table size, and can be computed with small memory. For more details see Section 3. Our main result is

Theorem 3. *Let $G = (V, E)$ be a weighted graph with n vertices and hop-diameter D, and let $k > 1$ be a parameter. Then there exists a routing scheme with stretch at most $4k - 5 + o(1)$, labels of size $O(k \log n)$ and routing tables of size $\tilde{O}(n^{1/k})$, that can be computed in a distributed manner within $(n^{1/2+1/k} + D) \cdot (\log n)^{O(\max\{k, \log\log n\})}$ rounds, such that every vertex has memory of size $\tilde{O}(n^{1/k})$.*

Alternatively, whenever $k \geq \sqrt{\log n / \log\log n}$, the number of rounds can be made $(n^{1/2+1/k} + D) \cdot 2^{\tilde{O}(\sqrt{\log n})}$ with memory $2^{\tilde{O}(\sqrt{\log n})}$ at each vertex.

In particular, whenever $k = \delta \log n / \log\log n$ for a small constant δ, we get $(\sqrt{n} + D) \cdot n^{O(\delta)}$ rounds with $\mathrm{polylog}(n)$ memory per vertex.

Construction of Routing Scheme. Let $G = (V, E)$ be a weighted graph, fix $k > 1$. We describe a scheme with stretch $4k - 3 + o(1)$, the improvement to $4k - 5 + o(1)$ is done as in [TZ01b, EN16b] (while paying a small polylogarithmic factor in the table size). Sample a collection of sets $V = A_0 \supseteq A_1 \cdots \supseteq A_k = \emptyset$,

where for each $0 < i < k$, each vertex in A_{i-1} is chosen independently to be in A_i with probability $n^{-1/k}$. A point $z \in A_i$ is called an i-pivot of v if $d_G(v,z) = d_G(v,A_i)$. The cluster of a vertex $u \in A_i \setminus A_{i+1}$ is defined as

$$C(u) = \{v \in V \ : \ d_G(u,v) < d_G(v,A_{i+1})\} \ . \tag{1}$$

It was shown in [TZ01a] that

Claim 6. *With high probability, each vertex is contained in at most $4n^{1/k}\log n$ clusters.*

We recall a few definitions from [EN16b]. For each $v \in V$ and $0 \leq i \leq k-1$, a point $\hat{z} \in A_i$ is called an *approximate i-pivot* of v if

$$d_G(v,\hat{z}) \leq (1+\epsilon)d_G(v,A_i) \ . \tag{2}$$

Define

$$C_\epsilon(u) = \{v \in V \ : \ d_G(u,v) < \frac{d_G(v,A_{i+1})}{1+\epsilon}\}. \tag{3}$$

The approximate cluster $\tilde{C}(u)$ will be any set that satisfies the following:

$$C_{6\epsilon}(u) \subseteq \tilde{C}(u) \subseteq C(u) \ . \tag{4}$$

It was shown in [EN16b] that once we obtain approximate clusters as trees of G and provide an exact routing scheme for these trees, it implies a routing scheme for G with stretch $4k-5+o(1)$ whenever $\epsilon \leq 1/(48k^4)$. Concretely, the routing table for x will be the tables for the at most $4n^{1/k}\log n$ trees containing it, and its label will be the collection of labels needed for tree routing in the at most k trees corresponding to its approximate pivots. So the table size is $O(n^{1/k}\log n)$ and the label size is $O(k\log n)$. It remains to show how to efficiently compute approximate clusters as trees of G, and the approximate pivots.

Let $h(u,v)$ denote the number of vertices on the shortest path from u to v in G, and set $B = 4\sqrt{n}\cdot\ln n$. The following were also shown in [EN16b] to hold with high probability.[6]

Claim 7. *For any $u,v \in V$ with $h(u,v) \geq B$, there exists a vertex of $A_{k/2}$ on the shortest path between them.*

Claim 8. *For any $0 \leq i < k-1$, $v \in A_i \setminus A_{i+1}$ and $u \in C(v)$, it holds that $h(u,v) \leq 4n^{(i+1)/k}\ln n$.*

In particular, for $i < k/2$ we can find $C(v)$ (the "exact" cluster) for $v \in A_i \setminus A_{i+1}$ by a simple limited Bellman-Ford exploration from all such v for $4n^{(i+1)/k}\ln n \leq \tilde{O}(\sqrt{n})$ rounds. By Claim 6, the congestion induced at each $u \in V$ by the merit of being a part of many clusters is only $4n^{1/k}\ln n$, so the total number of rounds required is $\tilde{O}(n^{1/2+1/k})$, and each vertex needs to store at most $4n^{1/k}\ln n$ words (the clusters containing it). Finally, note that these clusters indeed correspond to trees, since every vertex $u \in C(v)$ can store as a parent the vertex who last updated the distance estimate u has for v.

From now on we consider the high levels, where $i \geq k/2$. Define $G' = (V',E')$ as a virtual graph where $V' = A_{k/2}$, and E' corresponds to B-bounded distances in G. Observe that Claim 7 implies that $d_{G'}(v,v') = d_G(v,v')$ for any $v,v' \in V'$ (because any shortest path in G has a vertex of V' among any B consecutive vertices on that path). First, we compute a (β,ϵ)-hopset H for the virtual graph G' as in Theorem 1, with parameters $\kappa = \log n$, ϵ and $\rho = 1/\kappa$. We thus get $\beta = (\log n)^{O(\max\{k,\log\log n\})}$ (since $\epsilon \geq \Omega(1/\log^4 n)$). Note that the graph G' is implicit. Since $|A_{k/2}| = \Theta(\sqrt{n})$ whp, the number of rounds required to compute H is at most $(n^{1/2+1/k}+D)\cdot\beta$, and the size of the hopset is $O(n^{1/2})$. The arboricity and memory required per vertex is $\tilde{O}(n^{1/k})$, and it has a path-recovery mechanism. (If one desires the second assertion of the Theorem 3, pick $\rho = \sqrt{\log\log n/\log n}$.)

[6] For the sake of simplicity we will assume k is even, for odd k we can slightly improve the running time by a factor of $n^{1/(2k)}$.

Approximate Pivots To compute the approximate pivots, conduct β iterations of Bellman-Ford in $G'' = G' \cup H$ rooted in A_{i+1}, using low memory as in Lemma 2. Since H is a (β, ϵ)-hopset, each $v \in V'$ has a value $\hat{d}(v, A_{i+1}) \in [d_G(v, A_{i+1}), (1+\epsilon) \cdot d_G(v, A_{i+1})]$. We perform another B-bounded exploration in G, where initially every vertex $v \in V'$ sends its current estimate $\hat{d}(v, A_{i+1})$, and in every step every vertex forwards the smallest value it has heard so far. We claim that every $u \in V$ will learn of an approximate $(i+1)$-pivot $\hat{z} \in A_{i+1}$. To see this, let z be the $(i+1)$-pivot of u. If $h(u, z) \leq B$ then u will hear z's message in the last B-bounded exploration. Otherwise, by Claim 7 there exists a vertex $v \in V'$ on the shortest path from u to z within B hops from u. In the final exploration to range B, v will communicate $\hat{d}(v, A_{i+1})$ on the path towards u, thus u will have a value at most

$$\hat{d}(u, A_{i+1}) \leq d_G^{(B)}(u, v) + \hat{d}(v, A_{i+1}) \leq d_G(u, v) + (1+\epsilon) d_G(v, A_{i+1}) \leq (1+\epsilon) d_G(u, A_{i+1}) \,, \quad (5)$$

where the last inequality used that $d_G(u, v) + d_G(v, A_{i+1}) = d_G(u, A_{i+1})$, which follows since v lies on the shortest path from u to the nearest vertex of A_{i+1}. We conclude that no matter which $\hat{z}$ is the approximate pivot of u, the distance estimate u has for it cannot be larger than $(1+\epsilon) d_G(u, A_{i+1})$. Computing the approximate pivots takes whp $\tilde{O}(n^{(1+\rho)/2} + D) \cdot \beta$ rounds (recall by Theorem 1 we have $\alpha = \tilde{O}(n^{\rho/2})$).

Approximate Clusters Fix some $i \geq k/2$, and for each $v \in A_i \setminus A_{i+1}$ we conduct a *limited* Bellman-Ford exploration in $G'' = G' \cup H$ for β iterations rooted at v, as in Lemma 2. The limit means that any vertex $u \in V'$ receiving a message originated at v, will forward it to its neighbors iff the current distance estimate is strictly less than $\hat{d}(u, A_{i+1})/(1+\epsilon)^2$. We need to avoid congestion at intermediate vertices during the B-bounded exploration in G described in Lemma 2, so these vertices will also need to implement some sort of limitation. Concretely, vertices $u \in V \setminus V'$ will forward v's message iff their current estimate is strictly less than $\hat{d}(u, A_{i+1})/(1+\epsilon)$. The exploration over edges of H is done as before, where Claim 6 guarantees every vertex participates in at most $4n^{1/k} \ln n$ clusters (we will soon show that the approximate clusters are indeed contained in the clusters). This increases the number of rounds required for executing a Bellman-Ford iteration over the edges of E' from $\tilde{O}(n^{1/2})$ to $\tilde{O}(n^{1/2+1/k})$. However, the number of rounds for executing a Bellman-Ford iteration over the edges of H increases only by a constant factor, since in every iteration each $v \in V'$ broadcasts a single distance estimate for each cluster containing it, and $\alpha = \tilde{O}(n^{1/k})$ messages with identities of opposite endpoints of its outgoing hopset edges – but these are the same hopset edges for all clusters. Thus the number of rounds required is at most $\tilde{O}(n^{1/2+1/k} + D) \cdot \beta$. Also the memory each vertex uses for this computation is bounded by $\tilde{O}(n^{1/k})$ (which is essentially the number of clusters containing the vertex).

The exploration rooted at v induces a virtual tree (rooted at v). For every edge $(x, y) \in E'$ on this tree, we add all the vertices in G on the B-bounded path from x to y. This can be done via an acknowledgement message from y back to x on this path, and every vertex updates its parent accordingly. For every hopset edge $e = (x, y) \in H$ of the tree, we use the path-recovery mechanism in order to notify all vertices $u \in P(e)$ (recall $P(e)$ is the path in G implementing the edge e) to join the tree with root v, and also compute their approximate distance $\hat{d}(u, v)$ to v and the corresponding parent. If the computed distance satisfies $\hat{d}(u, v) < b_v(u)$, then u updates $b_v(u) \leftarrow \hat{d}(u, v)$, where $b_v(u)$ is the current distance estimate that u keeps concerning its distance to v. Also, u updates its parent. By Claim 6, any vertex in V is contained in at most $\tilde{O}(n^{1/k})$ clusters, and Claim 9 below will show that the approximate clusters are contained in clusters. So every vertex $u \in V$ may be a part of at most $C = \tilde{O}(n^{1/k})$ different trees. We get that the number of rounds required to implement the path-recovery protocol is $\tilde{O}((|H| \cdot C + D) \cdot \beta) = \tilde{O}(n^{1/2+1/k} + D) \cdot \beta$.

Finally, we perform another limited B iterations of Bellman-Ford in G, where every vertex x in the tree of v has initial value $b_v(x)$, and every vertex $u \in V$ will forward in each iteration the smallest estimate it

heard so far, but iff it is strictly less than $\hat{d}(u, A_{i+1})/(1+\epsilon)$. In that case it will also join the approximate cluster of v, and will update its parent as its neighbor in G whose message caused u to last update its distance estimate to v.

Observe that the same vertex may join a tree more than once, due to several edges in $E' \cup H$ whose path contain it. In such a case the vertex will have as a parent the vertex which minimize the estimated distance to the root. Since every vertex has a single parent, we will have that the approximate cluster of v, $\tilde{C}(v)$, is indeed a tree. It remains to prove (4), which is done in the next two claims. Recall $b_v(u)$ is the distance estimate u has to v in the exploration rooted at v.

Claim 9. *For any $v \in V'$, $\tilde{C}(v) \subseteq C(v)$.*

Proof. Consider any $u \in \tilde{C}(v)$. If it is the case that $u \in V$ joined the approximate cluster by the exploration rooted at v, either by being in V' or on a B-bounded path in G that implements an edge of E', then it must satisfy $b_v(u) < \hat{d}(u, A_{i+1})/(1+\epsilon)$. Now,

$$d_G(u,v) \leq b_v(u) < \hat{d}(u, A_{i+1})/(1+\epsilon) \overset{(5)}{\leq} d_G(u, A_{i+1}) \ ,$$

so indeed $u \in C(v)$. The other case is that u is part of a hopset edge (x, y) that was added to the virtual tree. Since y joins the approximate cluster, it must satisfy $b_v(y) < \hat{d}(y, A_{i+1})/(1+\epsilon)^2$. Recall that the weight of the hopset edge $w_H(x, y)$ is the weight of the path P from x to y in G that u lies on, hence $d_P(x, u) + d_P(u, y) = w_H(x, y)$. It follows that

$$\begin{aligned}
d_G(u, A_{i+1}) &\overset{(5)}{\geq} \frac{\hat{d}(u, A_{i+1})}{1+\epsilon} \geq \frac{d_G(u, A_{i+1})}{1+\epsilon} \geq \frac{d_G(y, A_{i+1}) - d_G(u, y)}{1+\epsilon} \\
&\overset{(5)}{\geq} \frac{\hat{d}(y, A_{i+1})}{(1+\epsilon)^2} - \frac{d_P(u, y)}{1+\epsilon} > b_v(y) - d_P(u, y) \\
&= b_v(x) + w_H(x, y) - d_P(u, y) = b_v(x) + d_P(x, u) \\
&\geq b_v(u) \geq d_G(u, v) \ ,
\end{aligned}$$

where in the penultimate inequality we used the fact that u knows $d_P(x, u)$ so could have updated its distance estimate to v as $b_v(x) + d_P(x, u)$ (note it may have used a smaller estimate). Thus $u \in C(v)$, as required. □

Claim 10. *For any $v \in V'$, $C_{6\epsilon}(v) \subseteq \tilde{C}(v)$.*

Proof. Let $u \in C_{6\epsilon}(v)$, we would like to show that $u \in \tilde{C}(v)$. Consider the shortest path P from u to v in G, then by Claim 7 there is a vertex $u' \in V'$ on P that is within B hops from u. Notice that

$$d_G(v, u') = d_G(v, u) - d_G(u, u') \leq \frac{d_G(u, A_{i+1}) - d_G(u, u')}{1+6\epsilon} \leq \frac{d_G(u', A_{i+1})}{1+6\epsilon} \ . \tag{6}$$

We will show that the limited exploration originated at v will reach u', and in the final depth B exploration it will reach u and include it in $\tilde{C}(v)$. Since H is a (β, ϵ)-hopset, there is a path P' in G'' from v to u' that contains at most β edges, so that

$$d_{P'}(v, u') \leq (1+\epsilon) d_{G'}(v, u') = (1+\epsilon) d_G(v, u') \ . \tag{7}$$

Let $z \in P'$ be any vertex on P' that lies t hops from v, then after t steps of Bellman-Ford exploration from v we have that

$$\begin{aligned} b_z(v) &= d_{P'}(v,z) = d_{P'}(v,u') - d_{P'}(z,u') \\ &\overset{(7)}{\leq} (1+\epsilon)d_G(v,u') - d_G(z,u') \overset{(6)}{\leq} \frac{(1+\epsilon)d_G(u',A_{i+1})}{1+6\epsilon} - d_G(z,u') \\ &\leq \frac{d_G(u',A_{i+1}) - d_G(z,u')}{1+4\epsilon} < \frac{d_G(z,A_{i+1})}{(1+\epsilon)^2} \leq \frac{\hat{d}(z,A_{i+1})}{(1+\epsilon)^2}\ , \end{aligned}$$

where we used that $\epsilon < 1/5$. We conclude that z satisfies the limit condition for the exploration rooted at v, and forwards the message of v onwards. In particular, $b_v(u') \leq d_{P'}(v,u') \leq (1+\epsilon)d_G(v,u')$. In the final phase we make a Bellman-Ford exploration for B rounds in G from each vertex that received v's message. Thus u' will start such an exploration with distance estimate $b_v(u')$. Consider the subpath $Q \subseteq P$ from u' to u, we have to show that every vertex on this path forwards the message of v, that is, that it satisfies the limit condition. Let $y \in Q$ be such a vertex, since this is a shortest path in G we have

$$\begin{aligned} b_v(y) &\leq b_v(u') + d_Q(u',y) \leq (1+\epsilon)d_G(v,u') + d_G(u',y) \\ &\leq (1+\epsilon)d_G(v,y) = (1+\epsilon)(d_G(v,u) - d_G(y,u)) \\ &\overset{(3)}{\leq} \frac{(1+\epsilon)d_G(u,A_{i+1})}{1+6\epsilon} - d_G(y,u) \leq \frac{d_G(u,A_{i+1}) - d_G(y,u)}{1+4\epsilon} \\ &\leq \frac{d_G(y,A_{i+1})}{1+4\epsilon} < \frac{\hat{d}(y,A_{i+1})}{1+\epsilon}\ , \end{aligned}$$

as required.

□