Can't See The Forest for the Trees: Navigating Metric Spaces by Bounded Hop-Diameter Spanners

Omri Kahalon[*] Hung Le[†] Lazar Milenković[‡] Shay Solomon[§]

Abstract

Spanners for metric spaces have been extensively studied, perhaps most notably in low-dimensional Euclidean spaces — due to their numerous applications. *Euclidean spanners* can be viewed as means of *compressing* the $\binom{n}{2}$ pairwise distances of a d-dimensional Euclidean space into $O(n) = O_{\epsilon,d}(n)$ spanner edges, so that the spanner distances preserve the original distances to within a factor of $1+\epsilon$, for any $\epsilon > 0$. Moreover, one can compute such spanners efficiently in the standard centralized and distributed settings. Once the spanner has been computed, it serves as a "proxy" overlay network, on which the computation can proceed, which gives rise to huge savings in space and other important quality measures.

The original metric enables us to "navigate" optimally — a single hop (for any two points) with the exact distance, but the price is high — $\Theta(n^2)$ edges. Is it possible to efficiently navigate, on a sparse spanner, using k hops and approximate distances, for k close to 1 (say $k = 2$)? Surprisingly, this fundamental question has been overlooked in Euclidean spaces, as well as in other classes of metrics, despite the long line of work on spanners in metric spaces.

We answer this question in the affirmative via a surprisingly simple observation on bounded hop-diameter spanners for *tree metrics*, which we apply on top of known, as well as new, *tree cover theorems*. Beyond its simplicity, the strength of our approach is three-fold:

- **Applicable**: We present a variety of applications of our efficient navigation scheme, including a 2-hop routing scheme in Euclidean spaces with stretch $1+\epsilon$ using $O(\log^2 n)$ bits of memory for labels and routing tables — to the best of our knowledge, all known routing schemes prior to this work use $\Omega(\log n)$ hops.
- **Unified**: Our navigation scheme and applications extend beyond Euclidean spaces to any class of metrics that admits an efficient tree cover theorem; currently this includes doubling, planar and general metrics, but our approach is unified.
- **Fault-Tolerant**: In Euclidean and doubling metrics, we strengthen all our results to achieve fault-tolerance. To this end, we first design a new construction of fault-tolerant spanners of bounded hop-diameter, which, in turn, relies on a new tree cover theorem for doubling metrics — hereafter the "Robust Tree Cover" Theorem, which generalizes the classic "Dumbbell Tree" Theorem [Arya et al., STOC'95] in Euclidean spaces.

[*]Tel Aviv University. Email: kahalon@gmail.com.
[†]University of Massachusetts Amherst. Email: hungle@cs.umass.edu.
[‡]Tel Aviv University. Email: milenkovic.lazar@gmail.com.
[§]Tel Aviv University. Email: solo.shay@gmail.com.

Contents

1 Introduction

1.1 Background and motivation

Let $M_X = (X, \delta_X)$ be an n-point metric space, viewed as a complete weighted graph whose weight function satisfies the triangle inequality. For a parameter $t \geq 1$, a subgraph $H = (V, E', w)$ of M_X ($E' \subseteq \binom{V}{2}$) is called a *$t$-spanner* for M_X if for all $u, v \in V$, $\delta_H(u, v) \leq t \cdot \delta_X(u, v)$. (Here $\delta_X(u, v)$ and $\delta_H(u, v)$ denote the distances between u and v in M_X and the spanner H, respectively.) In other words, for all $u, v \in V$, there exists a path in H between u and v whose *weight* (sum of edge weights in it) is at most $t \cdot \delta_X(u, v)$; such a path is called a *t-spanner path* and the parameter t is called the *stretch* of H. Since their introduction in the late 80s [PS89, PU89a], spanners have been extensively studied, and by now they are recognized as a graph structure of fundamental importance, in both theory and practice.

There are a few basic properties of spanners that are important for a wide variety of practical applications; in most applications, a subset of these properties need to be satisfied while preserving small stretch. Although the exact subset of properties varies between applications, perhaps the most basic property (besides small stretch) is to have a small number of edges (or *size*), close to $O(n)$; the spanner *sparsity* is the ratio of its size and the size $n - 1$ of a spanning tree. Second, the spanner *weight* $w(H) := \sum_{e \in E'} w(e)$ should be close to the weight $w(MST(M_X))$ of a minimum spanning tree $MST(M_X)$ of the underlying metric; we refer to the normalized notion of weight, $w(H)/w(MST(M_X))$, as the spanner *lightness*. Third, the *hop-diameter* of a spanner should be close to 1; the hop-diameter of a t-spanner is the smallest integer k such that for all $u, v \in V$, there exists a t-spanner path between u and v with at most k edges (or *hops*). Finally, the *degree* of a spanner, i.e., the maximum number of edges incident on any vertex, should be close to constant.

The original motivation of spanners was in distributed computing. For example, light and sparse spanners have been used in reducing the communication cost in efficient broadcast protocols [ABP90, ABP92], synchronizing networks and computing global functions [Awe85, PU89a, Pel00], gathering and disseminating data [BKR+02, VWF+03, KV02], and routing [WCT02, PU89b, ABLP89, TZ01b]; as another example, spanners with low degree can be used for the design of compact routing schemes [ABLP90, HP00, AM04, Tal04, Sli05, AGGM06, GR08a, CGMZ16]. Since then, graph spanners have found countless applications in distributed computing as well as various other areas, from motion planning and computational biology to machine learning and VLSI-circuit design.

Spanners have had special success in geometric settings, especially in low-dimensional Euclidean spaces. Spanners for Euclidean spaces, namely *Euclidean spanners*, were first studied by Chew [Che86] in 1986 (even before the term "spanner" was coined). Several different constructions of Euclidean spanners enjoy the optimal tradeoff between stretch and size: $(1+\epsilon)$ versus $O(\epsilon^{-d}n)$, for n-point sets in $\mathbb{R}^d$ [LS19]; these include Θ-graphs [Cla87, Kei88, KG92, RS91], Yao graphs [Yao82], path-greedy spanner [ADD+93, CDNS92, NS07], and the gap-greedy spanner [Sal92, AS97]. The reason Euclidean spanners are so important in practice is that one can achieve stretch arbitrarily close to 1 together with a linear number of edges (ignoring dependencies on ϵ and the dimension d). In general metrics, on the other hand, a stretch better than 3 requires $\Omega(n^2)$ edges, and the best result for general metrics is the same as in general graphs: stretch $2k - 1$ with $O(n^{1+1/k})$ edges [PS89, ADD+93]. Moreover, Euclidean spanners with the optimal stretch-size tradeoff can be built in optimal time $O(n \log n)$ in the static centralized setting, and they can be distributed in the obvious way in just one communication round in the Congested Clique model.

Driven by the success of Euclidean spanners, researchers have sought to extend results obtained

in Euclidean metrics to the wider family of *doubling metrics*.[1] The main result in this area is that any n-point metric of doubling dimension d admits a $(1+\epsilon)$-spanner with both sparsity and lightness bounded by $O(\epsilon^{-O(d)})$ [GGN04, CGMZ16, CG09, HPM06, Rod12, GR08a, GR08b, Smi09, ES15, CLNS15, Sol14, Got15, BLW17, FS20].[2] Moreover, here too there are efficient centralized and distributed algorithms, also under some practical restrictions such as those imposed by Unit Ball Graphs [DPP06b, DPP06a, EFN20, EK21].

A fundamental drawback of spanners. Different spanner constructions suit different needs and applications. However, there is one common principle: Once the spanner has been computed, it serves as a "proxy" overlay network, on which the computation can proceed, which gives rise to huge savings in a number of quality measures, including global and local space usage, as well as in various notions of running time, which change from one setting to another; in distributed networks, spanners also lead to additional savings, such as in the message complexity.

Alas, by working on the spanner rather than the original metric, one loses the key property of being able to efficiently "navigate" between points. In the metric, one can go from any point to any other via a direct edge, which is optimal in terms of the weighted distance and the unweighted (or *hop-*) distance. However, it is unclear how to efficiently navigate in the spanner: How can we translate the *existence* of a "good" path into an efficient algorithm finding it?

Moreover, usually by "good" path we mean a t-spanner path, i.e., a path whose weight approximates the original distance between its endpoints — but a priori the number of edges (or *hops*) in the path could be huge. To control the *hop-length* of paths, one can try to upper bound the spanner's hop-diameter, but naturally bounded hop-diameter spanners are more complex than spanners with unbounded hop-diameter, which might render the algorithmic task of efficiently finding good paths more challenging. We stress that most existing spanner constructions have inherently high hop-diameters. In particular, any construction with constant degree must have at least a logarithmic hop-diameter, and in general, if the degree is Δ, then the hop-diameter is $\Omega(\log_\Delta n)$.

In Euclidean spaces, the Θ-graph [Cla87, Kei88, KG92, RS91] and the Yao graph [Yao82] are not only simple spanner constructions, but they also provide simple *navigation algorithms*, where for any two points p and q, one can easily compute a $(1+\epsilon)$-spanner path between p and q. Alas, the resulting path may have a hop-length of $\Omega(n)$, and the *query time* is no smaller than the path length. There is a $(1+\epsilon)$-approximate distance oracle for low-stretch spanners [GLNS08], and while it achieves constant query time, it does not report the respective paths, whose hop-length can be $\Omega(n)$. In doubling metrics, there are $(1+\epsilon)$-approximate distance oracles with constant query time [HM06, BGK+11]. In [BGK+11] the respective paths are not part of a sparse overlay network (such as a spanner); in other words, the union of paths returned by the distance oracle of [BGK+11] may comprise a spanner of $\Theta(n^2)$ edges. Using [HM06], one can return paths that are part of a sparse spanner, but their hop-length is $\Theta(\log \rho)$, where ρ – the metric *aspect ratio*, can be arbitrarily large. This is where bounded hop-diameter spanners may come into play – efficient constructions are known in Euclidean and doubling metrics [CG09, Sol13]. In low-dimensional Euclidean spaces, it is possible to build a $(1+\epsilon)$-spanner with hop-diameter 2 and $O(n \log n)$ edges. In general, for any $k \geq 2$, one can get hop-diameter k with $O(n\alpha_k(n))$ edges, in optimal $O(n \log n)$ time [Sol13]; the function $\alpha_k(n)$ is the inverse of a certain function at the $\lfloor k/2 \rfloor$th level of the primitive

[1] The *doubling dimension* of a metric is the smallest d s.t. every ball of radius r for any r in the metric can be covered by 2^d balls of radius $r/2$. A metric space is called *doubling* if its doubling dimension is constant.

[2] In the sequel, for conciseness, we shall sometimes omit the dependencies on ϵ and the dimension d.

recursive hierarchy, where $\alpha_0(n) = \lceil n/2 \rceil$, $\alpha_1(n) = \lceil \sqrt{n} \rceil$, $\alpha_2(n) = \lceil \log n \rceil$, $\alpha_3(n) = \lceil \log \log n \rceil$, $\alpha_4(n) = \log^* n$, $\alpha_5(n) = \lfloor \frac{1}{2} \log^* n \rfloor$, etc. (For $k \geq 4$, the function α_k is close to log with $\lfloor \frac{k-2}{2} \rfloor$ stars.)

Two points on the tradeoff curve between hop-diameter k and size $O(n\alpha_k(n))$ deserve special attention: (1) $k = 4$ vs. $O(n \log^* n)$ edges; in practice $\log^* n \leq 10$, i.e., one can achieve hop-diameter 4 with effectively $O(n)$ edges. (2) $k = O(\alpha(n))$ vs. $O(n\alpha_k(n)) = O(n)$ edges, where α is a very slowly (more than $\log^*$) growing function; so to achieve a truly linear in n edges, one should take a hop-diameter of $O(\alpha(n))$ (which is effectively a constant). Refer to Section 2.2 for the formal definitions of the functions α_k and α. In some applications where limiting the hop-distances of paths is crucial, such as in some routing schemes, road and railway networks, and telecommunication, we might need to *minimize* the hop-distances; for example, imagine a railway network, where each hop in the route amounts to switching a train – how many of us would be willing to use more than, say, 4 hops? Likewise, what if each hop amounts to traversing a traffic light, wouldn't we prefer routes that minimize the number of traffic lights? In such cases, the designer of the system, or its users, might not be content with super-constant hop-distances, or even with a large constant, and it might be of significant value to achieve as small as possible hop-distances. Motivated by such practical considerations, we are primarily interested in values of hop-diameter k that "approach" 1, mainly $k = 2, 3, 4$, as there is no practical need in considering larger values of k (again, $O(n\alpha_4(n)) = O(n \log^* n)$ edges is effectively $O(n)$ edges).

One can achieve the same result, except for the construction time, also for doubling metrics. However, as mentioned, the drawback of bounded hop-diameter spanner constructions is that they are far more complex than basic spanners; hence, although there *exist* k-hop t-spanner paths between all pairs of points, the crux is to *find such paths efficiently*.

While the original metric enables us to navigate optimally — a single hop (for any two points) with the exact distance, the price is high — $\Theta(n^2)$ edges. The following question naturally arises.

Question 1.1. Can one efficiently navigate, on a *sparse spanner*, using k hops and approximate distances, for k approaching 1? In particular, can we achieve 2, 3 or 4 hops on an $o(n^2)$-sized spanner in Euclidean or doubling metrics?

Surprisingly, despite the long line of work on spanners in Euclidean and doubling metrics, Question 1.1 has been overlooked. By "efficiently navigate" we mean to *quickly* output a path of *small weight*, where ideally: (1) "quickly" means within time linear in the hop-length of the path, and (2) "small weight" means that the weight of the path would be larger than the original metric distance by at most the stretch factor of the underlying spanner.

Clearly, Question 1.1 can be asked in general, for any class of metrics. To the best of our knowledge, this fundamental question was not asked explicitly before. For general graphs, the classic Thorup-Zwick distance oracle [TZ01a] reports $(2\ell - 1)$-approximate distance queries in $O(\ell)$ time, using a data structure of expected size $O(\ell n^{1+1/\ell})$; it is immediate that their distance oracle, when applied to metric spaces, can report 2-hop paths of stretch $2\ell - 1$ in query time $O(\ell)$, which are all part of the same $(2\ell - 1)$-spanner with size $O(\ell n^{1+1/\ell})$. The following question is copied from [MN06]:

"*Since for large values of distortion (i.e., stretch) the query time of the Thorup-Zwick oracle is large, the problem remained whether there exist good approximate distance oracles whose query time is a constant independent of the distortion (i.e., in a sense, true "oracles")*". Mendel and Naor [MN06] gave two distance oracles with $O(1)$ query time and stretch of 128ℓ (the stretch was

improved later to 16ℓ [NT12]), the first has size $O(n^{1+1/\ell})$ and the respective paths can use any edge of the underlying metric and may thus form a network of size $\Omega(n^2)$, whereas the second has size $O(n^{1+1/\ell} \cdot \ell)$ and the respective paths have hop-lengths $\Theta(\log \rho)$. Wulff-Nilsen [Wul13] improved the query time of the Thorup-Zwick distance oracle [TZ01b] to $O(\log \ell)$. Using the Mendel-Naor distance oracle [MN06], Chechik [Che14, Che15] showed how to improve query time of [TZ01b] to $O(1)$, but this approach suffers from the same drawback — the respective paths may have hop-lengths $\Theta(\log \rho)$. Mendel-Naor question can thus be strengthened as:

Question 1.2 (Strengthening Mendel-Naor question [MN06])**.** Is there a good approximate distance oracle for general metric spaces that can report within constant time a constant-hop small-stretch path?

Interestingly, for planar and minor-free graphs, it is immediate that the respective distance oracles ([Tho04, KKS11, AG06]), when applied to the respective metrics, can provide 2-hop paths within constant query time.

Related work (in a nutshell). Thorup [Tho92] introduced the problem of *diameter-reducing shortcuts* for digraphs; the goal is to find a small subset of edges taken from the transitive closure of a digraph so that the resulting digraph has small hop-diameter. Cohen [Coh00] introduced the notion of *hopsets*; informally, an hopset H is an edge set that, when added to a graph G, provides small-stretch small-hop paths between all vertex pairs. (See [BP20, KP22] and references therein for details.) There are also various other related problems, such as *low-congestion shortcuts* [GH16, GH21, KP21]. For all these problems, the focus is on achieving a graph structure in which there *exist* "good" paths, i.e., with small hop-length and possibly additional useful properties, between vertex pairs in the graph; the *existence* of such paths found a plethora of applications in distributed, parallel, dynamic and streaming algorithms, such as to the computation of approximate shortest-paths, DFS trees, and graph diameter [Ber09, Nan14, MPVX15, HKN18, HL18, GP17, LP19]. However, to the best of our knowledge, the computational problem of *efficiently reporting* those paths — which is the focus of our work — has not been the focus of any prior work.

1.2 Our contribution

A key contribution of this work is a conceptual one, in (1) *realizing* that it is possible to efficiently navigate on a much sparser spanner than the entire metric space, and (2) unveiling some of applicability of such a navigation scheme. We start by considering *tree metrics*; a tree metric is a metric for which the distance function is obtained as the shortest-path distance function of some (weighted) tree. For any tree metric, when we relax the navigation requirement to use only $k = 2$ hops (instead of a single hop as in the original metric), we can navigate on a spanner of size $\Theta(n \log n)$, using 2 hops and stretch 1. If we relax the hop-length requirement a bit more, to $k = 3$, we can navigate on a yet sparser spanner, of size $\Theta(n \log \log n)$. In general, our navigation scheme achieves the same tradeoff between hop-diameter and size as the 1-spanner of Solomon [Sol13]. Our result for navigation on trees is stated in the following theorem (proved in Section 3.1); the stretch bound is 1 and one cannot improve the tradeoff between hop-diameter k and size $\Theta(n\alpha_k(n))$, due to lower bounds by [AS87] and [LMS22] that apply to 1-spanners and $(1+\epsilon)$-spanners for line metrics, respectively.

Theorem 1.1. *Let M_T be any tree metric, represented by an n-vertex edge-weighted tree T, let $k \geq 2$ be any integer, and let $G_T = (V(T), E)$ be the 1-spanner for M_T with hop-diameter k and*

$O(n\alpha_k(n))$ edges due to [Sol13]. Then we can construct in time $O(n\alpha_k(n))$ a data structure $\mathcal{D}_T$ such that, for any two query vertices $u, v \in V(T)$, $\mathcal{D}_T$ returns a 1-spanner path in G_T (which is also a shortest path in M_T) between u and v of hop-length $\leq k$ in $O(k)$ time.

The runtime of the 1-spanner construction for tree metrics of [Sol13] is $O(n\alpha_k(n))$, hence the data structure provided by Theorem 1.1 can be built from scratch in time $O(n\alpha_k(n))$. When it comes to 1-spanners for tree metrics, we can restrict attention to unweighted trees; indeed, for any two vertices u and v in tree T, if $P_{u,v}$ denotes the unique path between u and v in T, any 1-spanner path between u and v is a *subpath* of $P_{u,v}$ in the underlying tree metric.

Alon and Schieber [AS87] gave an algorithm for the online tree product that requires $O(n\alpha_k(n))$ time, space and semigroup operations during preprocessing. Their algorithm answers queries following paths of length $2k$, thus achieving $2k$ operations. This result is equivalent to a linear-time 1-spanner for tree metrics with $O(n\alpha_k(n))$ edges and hop-diameter $2k$, and their query algorithm is in fact a navigation algorithm on top of the underlying 1-spanner. They also discuss some applications to MST verification, finding maximum flow values in a multiterminal network, and updating a minimum spanning tree after increasing the cost of one of its edges. [Sol13] presents an improved linear-time construction of 1-spanners for tree metrics, with a hop-diameter of $2k$ rather than k for the same size bound. Since the 1-spanner construction of [Sol13] is more complex than that of [AS87], obtaining a navigation algorithm on top of the 1-spanner of [Sol13] is technically much more intricate than doing so on top of the 1-spanner of [AS87]. A central contribution of our work is in obtaining such a navigation algorithm, and then in realizing that, one can extend it to various families of metrics. Moreover, we demonstrate further applicability of our navigation scheme, and also strengthen our results for Euclidean and doubling metrics to achieve fault-tolerance.

To extend the navigation result of Theorem 1.1 from tree metrics to wider classes of metrics, we apply known results for *tree covers*, and also design a new *robust* tree cover scheme (see Theorem 4.1). Let $M_X = (X, \delta_X)$ be an arbitrary metric space. We say that a weighted tree T is a *dominating* tree for M_X if $X \subseteq V(T)$ and it holds that $\delta_T(x, y) \geq \delta_X(x, y)$, for every $x, y \in X$. For $\gamma \geq 1$ and an integer $\zeta \geq 1$, a (γ, ζ)-*tree cover* of $M_X = (X, \delta_X)$ is a collection of ζ *dominating trees* for M_X, such that for every $x, y \in X$, there exists a tree T with $d_T(u, v) \leq \gamma \cdot \delta_X(u, v)$; we say that the *stretch* between x and y in T is at most γ, and the parameter γ is referred to as the *stretch* of the tree cover. A tree cover is called a *Ramsey tree cover* if for each $x \in X$, there exists a "home" tree T_x, such that the stretch between x and every other vertex $y \in X$ in T_x is at most γ.

The celebrated "Dumbbell Theorem" by Arya et al. [ADM+95] provides a $(1+\epsilon, O(\frac{\log(1/\epsilon)}{\epsilon^d}))$-tree cover in $O(\frac{\log(1/\epsilon)}{\epsilon^d} \cdot n \log n + \frac{1}{\epsilon^{2d}} \cdot n)$ time, for d-dimensional Euclidean spaces. For general metrics, the seminal work of Mendel-Naor [MN06] provides a *Ramsey* (γ, ζ)-*tree cover* with $O(\zeta n^{2+1/\zeta} \log n)$ time, where $\gamma = O(\ell), \zeta = O(\ell \cdot n^{1/\ell})$ for any $\ell \geq 1$. Additional tree cover constructions are given in [BFN19], including a $(1+\epsilon, (1/\epsilon)^{\Theta(d)})$-tree cover for metrics with doubling dimension d. (See Table 1 in Section 2.1.) Plugging Theorem 1.1 on these tree cover theorems, we obtain:

Theorem 1.2. *For any n-point metric $M_X = (X, \delta_X)$ and any integer $k \geq 2$, one can construct a γ-spanner H_X for M_X with hop-diameter k and $O(n\alpha_k(n) \cdot \zeta)$ edges, accompanied with a data structure $\mathcal{D}_X$, such that for any two query points $u, v \in X$, $\mathcal{D}_X$ returns in time τ a γ-spanner path in H_X between u and v of at most k hops, where*

- *$\gamma = (1+\epsilon)$, $\zeta = (1/\epsilon)^{\Theta(d)}$, $\tau = O(k/\epsilon^{\Theta(d)})$, if the doubling dimension of M_X is d.*
- *If M_X is a general metric, there are two possible tradeoffs, for any integer $\ell \geq 1$:*
  - *$\gamma = O(\ell)$, $\zeta = O(\ell \cdot n^{1/\ell})$, $\tau = O(k)$.*

– $\gamma = O(n^{1/\ell} \cdot \log^{1-1/\ell} n)$, $\zeta = \ell$, $\tau = O(k)$.

- $\gamma = (1+\epsilon)$, $\zeta = O(((\log n)/\epsilon)^2)$, $\tau = O(k \cdot ((\log n)/\epsilon)^2)$, *if* M_X *is a fixed-minor-free metric.*

If M_X *is doubling, the running time is* $O(n \log n)$*, for fixed* ϵ *and constant dimension* d*.*

The navigation algorithms provided by Theorem 1.2 work by first determining the right tree for the query points $u, v \in X$, and then applying the tree navigation algorithm of Theorem 1.1 on that tree. This two-step navigation scheme might be advantageous over navigation algorithms that don't employ trees, as navigation on top of a tree could be both faster and simpler to implement in practice. Theorem 1.2 implies that in low-dimensional and doubling metrics, one can navigate along a $(1+\epsilon)$-spanner with hop-diameter k and $O(n\alpha_k(n))$ edges, within query time $O(k)$, ignoring dependencies on ϵ and d. Result of this sort was not known before even in Euclidean spaces, and it affirmatively settles Question 1.1. In metrics induced by fixed-minor-free graphs (e.g., planar metrics), we get a similar result, with the number of edges and query time growing by a factor of $\log^2 n$. For such metrics, as mentioned, there are already efficient navigation algorithms, implicit in [Tho04, KKS11, AG06], so we do not achieve improved bounds here; however, as argued above, our two-step navigation scheme might still be advantageous. Finally, in general metrics, the stretch and size of the spanners on which we navigate nearly match the best possible stretch-size tradeoff of spanners in general metrics, and the number of hops in the returned paths approaches 1. Here too, there are already efficient navigation algorithms, which achieve better bounds on stretch and size, implicit in the works of [TZ01b, MN06, NT12, Che14, Che15]. However, our two-step navigation scheme in general metrics is advantageous over previous ones since it reports an actual path that belongs to the underlying spanner *in constant time*, which also settles Question 1.2; moreover, it uses a Ramsey cover, and is thus of further applicability (e.g., for routing protocols, see below).

A unified approach. Although our original motivation was in Euclidean spaces, our two-step navigation scheme extends far beyond it. Our technique for efficiently navigating 1-spanners for tree metrics, as provided by Theorem 1.1, provides a *unified reduction* from efficient navigation schemes in an arbitrary metric class to *any tree cover* theorem in that class; in other words, any new tree cover theorem will directly translate into a new navigation scheme.

A fault-tolerant spanner and navigation scheme. In Euclidean and doubling metrics, we design a fault-tolerant (FT) navigation scheme, where we can navigate between pairs of non-faulty points in the network even when a predetermined number f of nodes become faulty, while incurring small overheads (factor of at most f) on the size of the navigation data structure and other parameters. We first generalize the Euclidean "Dumbbell Tree" Theorem [ADM+95] for doubling metrics; this generalization is nontrivial and is perhaps the strongest technical contribution of this work. At a high-level, the "Dumbbell Tree" Theorem is quite *robust* against adversarial perturbations of input points; specifically, any internal node in any tree in the cover can be assigned any descendant leaf as its associated point without affecting the stretch bound. This property is not achieved by the tree cover of [BFN19] in doubling metrics. Building on our robust tree cover theorem, we design a new construction of FT sparse spanners of bounded hop-diameter; this construction achieves optimal bounds on all involved parameters for fixed f, and is of independent interest. Our FT navigation scheme is obtained from our new FT spanner just as our basic navigation scheme is obtained from the basic spanner of [Sol13]. See Section 4 for the full details.

Broad applicability. We argue that an efficient navigation scheme is of broad potential applicability, by providing a few applications and implications; we anticipate that more will follow.

Perhaps the main application of our navigation technique is an efficient *routing scheme*, where we achieve small bounds on the local memory at all nodes, even though the maximum degree is huge, which is inevitable for spanners of tiny hop-diameter. Due to space constraints, in this discussion we provide details only on this application. In a nutshell, other applications of our navigation scheme include: (1) *Efficient sparsification of light-weight spanners*, where we start from an arbitrary light-weight but possibly dense spanner and transform it into a spanner that has the original stretch and weight but is also sparse. (2) *Efficient computation on the spanner*, where we are able to compute basic graph structures (such as MST and SPT) efficiently on top of a spanner rather than the underlying metric (which is not as part of our input). (3) *Online tree product queries and applications*, where our basic navigation scheme can be used as a query algorithm for the online tree product problem, which finds applications to MST verification and other problems. More details on these applications are deferred to Section 1.3 (introductory details) and Section 5 (full details).

Our basic result on routing schemes is in providing a routing scheme of stretch 1 on *tree metrics*, for $k = 2$ hops and using labels and local routing tables of $O(\log^2 n)$ bits and headers of $O(\log n)$ bits. The routing scheme works in the labeled, fixed-port model (see Section 5.1 for the definitions). The bound on the number of hops is best possible without routing on the complete graph. We employ this basic routing scheme in conjunction with the aforementioned tree covers and obtain efficient routing schemes for doubling, general and fixed-minor-free metrics. For doubling metrics, we strengthen the result to achieve a fault-tolerant routing scheme, where packets can be routed efficiently even when a predetermined number of nodes in the input metric become faulty.

Theorem 1.3. *For any n-point metric $M_X = (X, \delta_X)$, one can construct a γ-stretch 2-hop routing scheme in the labeled, fixed-port model with headers of $\lceil \log n \rceil$ bits, labels of b_l bits, local routing tables of b_t bits, and local decision time τ, where:*

- $\gamma = (1+\epsilon)$, $b_l = b_t = O(\epsilon^{-O(d)} \log(n) \log(n/\epsilon))$, $\tau = O(\epsilon^{-O(d)})$, *for doubling dimension d.*
- *If M_X is a general metric, there are two possible tradeoffs, for any integer $\ell \geq 1$:*
  - $\gamma = O(\ell)$, $b_l = O(\log^2 n)$, $b_t = O(\ell \cdot n^{1/\ell} \log^2 n)$, $\tau = O(1)$.
  - $\gamma = O(n^{1/\ell} \cdot \log^{1-1/\ell} n)$, $b_l = O(\log^2 n)$, $b_t = O(\ell \log^2 n)$, $\tau = O(1)$.
- $\gamma = (1+\epsilon)$, $b_l = b_t = O((\log n/\epsilon)^3 \log n)$, $\tau = O((\log n/\epsilon)^2)$, *for a fixed-minor-free metric.*

If M_X has doubling dimension d, the running time is $O(n \log n)$, for fixed ϵ and d. In this case, one can achieve an f-fault-tolerant routing scheme, with the bounds on b_l and b_t growing by a factor of f.

This provides the first routing schemes in Euclidean as well as doubling metrics, where the number of hops is as small as 2, and the labels have near-optimal size. To the best of our knowledge, no previous work on routing schemes in Euclidean or doubling metrics achieve a sub-logarithmic bound on the hop-distances, let alone a bound of 2. Some previous works [GR08a, CGMZ16] obtain their routing schemes by routing on constant-degree spanners, which means that the hop-diameters of those spanners are at least $\Omega(\log n)$, hence the hop-lengths of the routing paths are $\Omega(\log n)$ too. The other routing schemes [HP00, AM04, Tal04, Sli05, AGGM06] do not work in this way, but still have a hop-diameter of $\Omega(\log n)$ or even $\Omega(\log \rho)$. We also stress that our routing scheme is fault-tolerant, which is of practical importance, and we are not aware of any previous fault-tolerant routing scheme in Euclidean or doubling metrics.

There are many works on routing in general graphs [ABLP90, AP92, Cow01, TZ01b, EGP03, Che13, RT15, ACE+20, Fil21]. In metrics, it is much easier to get an efficient routing scheme. The Thorup-Zwick routing scheme [TZ01b] can achieve two hops in general metrics with stretch $4\ell - 5$ (improved to 3.68ℓ [Che13]), labels of $O(\ell \log n)$ bits, and table sizes of $\tilde{O}(n^{1/\ell})$. These approaches, when modified to work in metrics, incur a decision time of $O(\ell)$, and it is not clear whether it can be improved. Our result for general metrics from Theorem 1.3, while inferior in terms of the stretch (a constant factor), the label sizes (a $\log n/\ell$ factor) and table size (a $\log n\ell$ factor), achieve constant decision time, which might be an important advantage in real-time routing applications.

1.3 Further discussion on Applications

Efficient sparsification of light spanners. Let $M_X = (X, \delta_X)$ be an arbitrary n-point metric space and let G be any m-edge spanner for M_X of light weight. Our goal is to transform G into a sparse spanner for M_X, without increasing the stretch and weight by much. Let $\mathcal{D}_X$ be the data structure provided by Theorem 1.2. For each edge in G, we can query D_X for the k-hop path between its endpoints and then return the union of the paths over all edges. It is not difficult to verify that the resulting graph is a spanner for M_X, whose stretch and weight are larger than those of G by at most a factor γ, but it includes at most $O(n\alpha_k(n) \cdot \zeta)$ edges — thus it is not only light but also sparse. The runtime of this transformation is $O(m \cdot \tau)$. (As in Theorem 1.2, we denote by γ the stretch of the tree cover, ζ bounds the number of trees in the cover, and τ bounds the query time — which is $O(k \log^2 n)$ for fixed-minor-free graphs, and $O(k)$ for all other metric classes.) For further details, see Section 5.3.

Efficient computation on the spanner. As mentioned already, once a spanner has been constructed, it usually serves as a "proxy" overlay network, on which any subsequent computation can proceed, in order to obtain savings in various measures of space and running time. This means that any algorithm that we may wish to run, should be (ideally) run on top of the spanner itself. Furthermore, in some applications, we may not have direct access to the entire spanner, but may rather have implicit and/or local access, such as via labeling or routing schemes, or by means of a data structure for approximate shortest paths within the spanner, such as the one provided by Theorem 1.2.

Suppose first that we would like to construct a (possibly approximate) shortest path tree (SPT). An SPT for the original metric space is simply a star (in any metric). But the star is (most likely) not a subgraph of the underlying spanner. How can we efficiently transform the star into an approximate SPT in the spanner? If we have direct, explicit access to the spanner, we can simply compute an SPT on top of it using Dijkstra's algorithm, which will provide an approximate SPT for the original metric. Dijkstra's algorithm, however, will require $\Omega(n \log n)$ time (for an n-vertex spanner), even if the spanner size is $o(n \log n)$; there is also another SPT algorithm that would run in time linear in the spanner size, but it is more complex and also assumes that $\log n$-bit integers can be multiplied in constant time [Tho99]. Using our navigation scheme, as provided by Theorem 1.2, we can do both better and simpler, and we don't even need explicit access to the underlying spanner (though we do need, of course, access to the navigation scheme). The data structure provided by Theorem 1.2 allows us to construct, within time $O(n\tau)$, an approximate SPT. In particular, for low-dimensional Euclidean and doubling metrics, we can construct a $(1+\epsilon)$-approximate SPT (for a fixed ϵ) that is a subgraph of the underlying spanner within $O(nk)$ time, where $k = 2, 3, \ldots, O(\alpha(n))$. Refer to Section 5.4 for further details.

Suppose next that we would like to construct an approximate minimum spanning tree (MST). In low-dimensional Euclidean spaces one can compute a $(1+\epsilon)$-approximate MST (for a fixed ϵ) in $O(n)$ time [Cha08], but again this approximate MST may not be a subgraph of the spanner. Running an MST algorithm on top of the spanner would require time that is at least linear in the spanner size; moreover, the state-of-the-art deterministic algorithm runs in super-linear time and is rather complex [Cha00], and the state-of-the-art linear time algorithms either rely on randomization [KKT95] or on some assumptions, such as the one given by *transdichotomous model* [FW94]. Instead, using our navigation scheme, as provided by Theorem 1.2, we can construct an approximate MST easily, within time $O(n\tau)$. In particular, for low-dimensional Euclidean spaces, we can construct in this way a $(1+\epsilon)$-approximate MST (for a fixed ϵ) that is a subgraph of the underlying spanner within $O(nk)$ time, where $k = 2, 3, \ldots, O(\alpha(n))$. Refer to Section 5.5 for further details.

The same principle extends to other metric spaces, but some of the guarantees degrade. In particular, for metric spaces, the approximation factor increases far beyond $1+\epsilon$, at least assuming we would like the size of the underlying spanner to be near-linear. We stress that (approximate) SPTs and MSTs are two representative examples, but the same approach can be used for efficiently constructing other subgraphs of the underlying spanner; we also note that a shallow-light tree (SLT) [ABP90, ABP92, KRY93, Sol14], which is tree structure that combines the useful properties of an SPT and an MST, can be constructed in linear time given any approximate SPT and MST, and the resulting SLT is also a subgraph of these input trees [KRY93]. Thus, after constructing approximate SPT and MST as described above, we obtain, within an additional linear time, an SLT that is a subgraph of the underlying spanner.

Online tree product and MST verification. The paper by [AS87] focuses on the following problem. Let T be a tree with each of its n vertices being associated with an element of a semigroup. One needs to answer online queries of the following form: Given a pair of vertices $u, v \in T$, find the product of the elements associated with the vertices along the path from u to v. They show that one can preprocess the tree using $O(n\alpha_k(n))$ time and space, so that each query can be answered using at most $2k$ semigroup operations. They also showed several applications of their algorithm, such as to finding maximum flow in a multiterminal network, MST verification, and updating the MST after increasing the cost of its edges [AS87].

One can show that the result of [AS87] gives rise to a construction of 1-spanners for tree metrics with $O(n\alpha_k(n))$ edges and hop-diameter $2k$. This is inferior to the spanner of [Sol13], since it achieves a twice larger hop-diameter ($2k$ instead of k) for the same number of edges, $O(n\alpha_k(n))$. In particular, their construction cannot be used to achieve hop-diameters 2 and 3, which is the focus of this paper. We stress that some of the applications that we discussed above cannot be achieved using this weaker result. As a prime example, our routing scheme crucially relies on having hop-diameter 2. Paths of hop-distance 2 have a very basic structure (going through a single intermediate node), which our routing scheme exploits. As a result, the underlying spanner has $O(n \log n)$ edges, which ultimately requires us to use $O(\log^2 n)$ bits of space. Whether or not one can use a spanner of larger (sublogarithmic and preferably constant) hop-diameter for designing compact routing schemes with $o(\log^2 n)$ bits is left here as an intriguing open question. Exactly the same obstacle should render the construction of [AS87] infeasible for constructing efficient routing schemes, since the hop-distances provided by the result of [AS87] are larger than 2.

In Section 5.6.1, we show that our navigation scheme (provided by Theorem 1.1) can be used as a query algorithm that supports all the applications supported by [AS87], but within a factor

2 improvement on the hop-distances (or on other quality measures that are derived from the hop-distances). One such application is to the online MST verification problem, which is the main building block for randomized MST algorithms. For this problem, Pettie [Pet06] shows that it suffices to spend $O(n\alpha_{2k}(n))$ time and space and $O(n \log \alpha_{2k}(n))$ comparisons during preprocessing, so that each subsequent query can be answered using $4k-1$ comparisons.[3] Our algorithm takes $O(n\alpha_{2k}(n))$ time and space and $O(n \log \alpha_{2k}(n))$ comparisons during preprocessing, so that each subsequent query is answered using $2k-1$ comparisons in $O(k)$ time. The result of [Pet06] can also achieve a query time of $O(k)$, by building on [AS87], but using $4k-1$ comparisons rather than $2k-1$ as in our result. Concurrently and independently of us, Yang [Yan21] obtained a similar result.

2 Preliminaries

This section contains definitions and results required for the rest of the paper. In particular, Section 2.1 summarizes known tree cover theorems which we rely on and Section 2.2 introduces variants of Ackermann function which we use.

2.1 Summary of known results on tree covers

The following table summarizes known results on tree cover theorems.

Stretch (γ)	Num. of trees (ζ)	Metric family	Construction time	Authors
$1+\epsilon$	$(1/\epsilon)^{\Theta(d)}$	with doubling dim. d	$O(n \log n)$	[ADM+95, BFN19]
$1+\epsilon$	$O(((\log n)/\epsilon)^2)$	fixed-minor-free (e.g., planar)	$n^{O(1)}$	[BFN19]
$O(\ell)$	$O(\ell \cdot n^{1/\ell})$	general	$O(\ell \cdot n^{2+1/\ell} \log n)$	[MN06]
$O(n^{1/\ell} \cdot \log^{1-1/\ell} n)$	ℓ	general	$n^{O(1)}$	[BFN19]

Table 1: Summary of the tree cover results used throughout the paper. The last two results provide Ramsey tree covers for any integer $\ell \geq 1$.

2.2 Ackermann functions

Following standard notions [Tar75, AS87, Cha87, NS07, Sol13], we will introduce two very rapidly growing functions $A(k,n)$ and $B(k,n)$, which are variants of Ackermann's function. Later, we also introduce several inverses and state their properties that will be used throughout the paper.

[3] In fact, Pettie [Pet06] claimed that the preprocessing time is $O(n \log \alpha_{2k}(n))$, and that each subsequent query can be answered using $2k-1$ comparisons. This is inaccurate, as we elaborate in Section 5.6.2.

Definition 2.1. For all $k \geq 0$, the functions $A(k,n)$ and $B(k,n)$ are defined as follows:

$$\begin{aligned}
A(0,n) &\coloneqq 2n, \text{ for all } n \geq 0,\\
A(k,n) &\coloneqq \begin{cases} 1 & \text{if } k \geq 1 \text{ and } n = 0\\ A(k-1, A(k,n-1)) & \text{if } k \geq 1 \text{ and } n \geq 1 \end{cases}\\
B(0,n) &\coloneqq n^2, \text{ for all } n \geq 0,\\
B(k,n) &\coloneqq \begin{cases} 2 & \text{if } k \geq 1 \text{ and } n = 0\\ B(k-1, B(k,n-1)) & \text{if } k \geq 1 \text{ and } n \geq 1 \end{cases}
\end{aligned}$$

We now define the functional inverses of $A(k,n)$ and $B(k,n)$.

Definition 2.2. For all $k \geq 0$, the function $\alpha_k(n)$ is defined as follows:

$$\begin{aligned}
\alpha_{2k}(n) &\coloneqq \min\{s \geq 0 : A(k,s) \geq n\}, \text{ for all } n \geq 0,\\
\alpha_{2k+1}(n) &\coloneqq \min\{s \geq 0 : B(k,s) \geq n\}, \text{ for all } n \geq 0.
\end{aligned}$$

It is not hard to verify that $\alpha_0(n) = \lceil n/2 \rceil$, $\alpha_1(n) = \lceil \sqrt{n} \rceil$, $\alpha_2(n) = \lceil \log n \rceil$, $\alpha_3(n) = \lceil \log\log n \rceil$, $\alpha_4(n) = \log^* n$, $\alpha_5(n) = \lfloor \frac{1}{2}\log^* n \rfloor$, etc.

The spanner construction of [Sol13] (and thus also our navigation algorithm) uses a slight variant α'_k of the function α_k.

Definition 2.3. We define function $\alpha'_k(n)$ as follows:

$$\begin{aligned}
\alpha'_0(n) &\coloneqq \alpha_0(n), \text{ for all } n \geq 0,\\
\alpha'_1(n) &\coloneqq \alpha_1(n), \text{ for all } n \geq 0,\\
\alpha'_k(n) &\coloneqq \alpha_k(n), \text{ for all } k \geq 2 \text{ and } n \leq k+1,\\
\alpha'_k(n) &\coloneqq 2 + \alpha'_k(\alpha'_{k-2}(n)), \text{ for all } k \geq 2 \text{ and } n \geq k+2.
\end{aligned}$$

By Lemma 2.4 from [Sol13] we know that the functions α'_k and α_k are asymptotically close — for all $k, n \geq 0$, $\alpha_k(n) \leq \alpha'_k(n) \leq 2\alpha_k(n) + 4$.

Finally, for all $n \geq 0$, we introduce the Ackermann function as $A(n) \coloneqq A(n,n)$, and its inverse as $\alpha(n) = \min\{s \geq 0 : A(s) \geq n\}$. In [NS07], it was shown that $\alpha_{2\alpha(n)+2}(n) \leq 4$.

Following [Pet06], we introduce a slight variant of Ackermann's function as follows:[4]

$$\begin{aligned}
P(1,j) &\coloneqq 2^j && \text{for } j \geq 0,\\
P(i,0) &\coloneqq P(i-1,1) && \text{for } i \geq 2,\\
P(i,j) &\coloneqq P(i-1, 2^{2^{P(i,j-1)}}) && \text{for } i \geq 2, j \geq 1.
\end{aligned}$$

Then, its inverse of the ith row is defined as:

$$\lambda_i(n) \coloneqq \min\{j \geq 0 : P(i,j) \geq n\}.$$

Lemma 2.1. *For any $i \geq 1$, if $\lambda_i(n) > 0$, then $\frac{1}{3}\alpha_{2i}(n) \leq \lambda_i(n) \leq \alpha_{2i}(n)$.*

[4] In [Pet06], the function was denoted by letter A.

3 Navigating metric spaces

In this section we present the navigation algorithm for metric spaces. Section 3.1 is devoted to proving Theorem 1.1, which concerns navigation on 1-spanner with bounded hop-diameter for tree metrics by [Sol13]. In Section 3.2, we use tree cover theorems (cf. Table 1) and prove Theorem 1.2, which concerns navigation on metric spaces.

3.1 Navigating the tree spanner

Our navigation algorithm consists of two parts. In Section 3.1.1, we present a preprocessing algorithm, which takes a tree T and an integer parameter $k \geq 2$; it constructs [Sol13] 1-spanner G_T with hop-diameter k for a tree metric $M_T = (V(T), \delta_T)$ induced by T, together with data necessary for efficiently navigating it. Next, in Section 3.1.2, we present a query algorithm which, given any two vertices $u, v \in T$, outputs in $O(k)$ time a 1-spanner k-hop path between u and v in G_T.

The result of [Sol13] considers a generalized problem of constructing 1-spanners for *Steiner tree metrics*. Specifically, suppose that in a given tree T, a subset $R(T) \subseteq V(T)$ of the vertices are set as *required vertices*. The other vertices $S(T) \coloneqq V(T) \setminus R(T)$ are called *Steiner vertices*. We say that a 1-spanner G_T for M_T has hop-diameter k if it contains a 1-spanner path for M_T that consists of at most k edges, for every pair of vertices in $R(T)$.

We next give high-level explanation of the [Sol13] spanner construction algorithm. It relies on the following two procedures, which we shall also use in our preprocessing algorithm.

- $\textsc{Prune}((T, rt(T)), R(T))$: Takes as an input a tree T, its root $rt(T)$, and the set of required vertices $R(T)$. Outputs an edge-weighted tree $(T_{\mathsf{pnd}}, rt(T_{\mathsf{pnd}}))$, which contains $R(T)$ and has at most $|R(T)| - 1$ Steiner vertices. We set the weight $w_{T_{\mathsf{pnd}}}(u, v) = \delta_T(u, v)$ for every edge $(u, v) \in V(T_{\mathsf{pnd}})$. Informally, the procedure keeps the intrinsic properties of T, while reducing the number of Steiner vertices. For more details, see Section 3.2 in [Sol13]. The running time is $O(|V(T)|)$.

- $\textsc{Decompose}((T, rt(T)), R(T), \ell)$: Takes as an input a rooted tree $(T, rt(T))$, the set of required vertices $R(T)$, and an integer parameter $\ell \geq 1$. (The parameter ℓ will be set to $\alpha'_{k-2}(n)$.) Outputs, in $O(|V(T)|)$ time, a set of *cut vertices*, denoted by $CV_\ell \subseteq V(T)$, such that every connected component (tree) of $T \setminus CV_\ell$ contains at most ℓ required vertices. The size of CV_ℓ satisfies $|CV_\ell| \leq \lfloor \frac{|V(T)|}{\ell+1} \rfloor$.

At the beginning of the spanner construction, we find a subset of vertices $CV_\ell \subseteq V(T)$, using $\textsc{Decompose}((T, rt(T)), R(T), \ell)$. We then compute the set of edges E', which interconnects vertices in CV_ℓ. The algorithm distinguishes several cases:

- If $k = 2$, then $|CV_\ell| = 1$ and $E' = \emptyset$.

- If $k = 3$, then connect every pair of vertices in CV_ℓ, i.e., $E' = CV_\ell \times CV_\ell$.

- If $k \geq 4$, then make a copy T' of T, set CV_ℓ as its required vertices and prune it, by invoking $\textsc{Prune}((T', rt(T')), CV_\ell)$; let E' be the set of edges returned by recursive spanner construction on T' with hop-diameter set to $k - 2$.

Denote by $T_1, \ldots, T_p$ the trees in $T \setminus CV_\ell$. The algorithm computes the set of edges E'' that connects the cut vertices of CV_ℓ with the corresponding subtrees. Given a subtree $T_i \in T$, we say that a

vertex $u \in T$ is a *border vertex* of T_i if $u \notin V(T_i)$ is adjacent to a vertex in T_i. Let $\text{border}(T_i)$ denote the set of all border vertices of T_i. With a slight abuse of notation, we let $\text{border}(v) = \text{border}(T_i)$ for all $v \in T_i$; in addition, for $c \in CV_\ell$, let $\text{border}(c) = \{v \in T \mid c \in \text{border}(v)\}$. For every $c \in CV_\ell$, we add an edge from c to all the required vertices in $\text{border}(c)$. Finally, for each i in $[p]$, we let E_i be the set of edges obtained by recursive spanner construction on T_i. The set of spanner edges is $E' \cup E'' \cup \bigcup_{i \in [p]} E_i$. This concludes the high-level description of algorithm for constructing spanner.

The construction guarantees that between any two vertices $u, v \in R(T)$, there is a path of length $\delta_T(u, v)$ in G_T consisting of at most k edges. This path is a shortcut of the path between u and v in T. More formally, denote by $\mathcal{P}_T(u, v)$ the unique path in T between a pair u, v of vertices in T. A path P in G_T between u and v is called T-monotone if it is a subpath of $\mathcal{P}_T(u, v)$, that is, if $\mathcal{P}_T(u, v) = (u = v_0, v_1, \ldots, v_t = v)$, then P can be written as $P = (u = v_{i_0}, v_{i_1}, \ldots, v_{i_q} = v)$, where $0 = i_0 < i_1 < \cdots < i_q = t$. For any two vertices $u, v \in R(T)$, there is a T-monotone path in G_T of at most k edges.

Despite the guarantee of existence of a k-hop path between any two vertices in $R(T)$, it is not a priori clear how one can efficiently find such a path. Consider $k = 2$, as the most basic setting. It is shown in [Sol13] that for any two u and v, there exists an intermediate cut vertex w on the path $\mathcal{P}_T(u, v)$, such that (u, w) and (w, v) are in G_T. (For simplicity, we omit some technical details of handling the corner cases.) But this cut vertex can be anywhere on $\mathcal{P}_T(u, v)$ and (at least naively) finding it could take number of steps linear in the length of the path.

Our key idea is to rely on the recursion tree of the spanner construction algorithm. Since the edges (u, w) and (w, v) are in G_T and w is a cut vertex, there must be a recursive call which had $CV_\ell = \{w\}$. We explicitly build the recursion tree of the spanner construction, and store with each of its vertices the data required for efficient navigation. We call such a tree *augmented recursion tree*, and denote it by Φ. For each vertex v in $R(T)$, we keep track of the vertex in Φ which corresponds to the recursive call when v was chosen to be a cut vertex. To answer a query for k-hop path between u and v, we can find an intermediate cut vertex w as follows. First, we identify two vertices α_u and α_v corresponding to u and v in Φ. Then, we find their lowest common ancestor β in Φ. Vertex β corresponds to a recursive call in which some cut vertex w splits the tree so that u and v are in different subtrees. Clearly, w is on $\mathcal{P}_T(u, v)$. Since u and v are both required vertices and w is a cut vertex, the edges (w, u) and (w, v) are added to the spanner in this recursive call. Hence, we have found a T-monotone 2-hop path between u and v in G_T.

When $k = 3$, the set of cut vertices at each recursion level contains more than one cut vertex. The 1-spanner path between u and v contains two intermediate cut vertices, say u' and v', which are on $\mathcal{P}_T(u, v)$. (Here too, we omit technical details of handling the corner cases.) Let T' be a tree which is passed as an argument to a recursive call in which u' and v' were in CV_ℓ. Since u', v' are on $\mathcal{P}_T(u, v)$, tree $T_u \in T' \setminus CV_\ell$ containing u and $T_v \in T' \setminus CV_\ell$ containing v are different. At that point, u (resp. v) could have many cut vertices in $\text{border}(u)$ (resp., $\text{border}(v)$). To avoid checking every possible pair of cut vertices in $\text{border}(u)$ and $\text{border}(v)$, we construct another tree, called *contracted tree* which facilitate finding the corresponding pair of cut vertices.

Fix a vertex $\beta \in \Phi$, corresponding to a recursive call of spanner construction where a tree $(T', rt(T'))$ is passed as an argument, and let CV_ℓ denote the set of cut vertices chosen for this level. Furthermore, let $T_1, \ldots, T_p = T' \setminus CV_\ell$ be the subtrees obtained by removing vertices in CV_ℓ from T'. The set of vertices of the contracted tree $\mathcal{T}_\beta$, corresponding to vertex β in Φ, consists of p vertices, $t_1, \ldots, t_p$, corresponding to $T_1, \ldots, T_p$, and $|CV_\ell|$ vertices corresponding to cut vertices in CV_ℓ. For each vertex $t_i \in \mathcal{T}_\beta$, we add an edge between t_i and all the vertices in $\mathcal{T}_\beta$ corresponding

to cut vertices in border(T_i). Intuitively, the augmented tree $\mathcal{T}_\beta$ identifies every subtree t_i with a single vertex and keeps the tree structure of given tree T.

We now explain how $\mathcal{T}_\beta$ facilitates finding cut vertices u' and v' corresponding to vertex $\beta \in \Phi$ which are on $\mathcal{P}_T(u, v)$. First, we find the vertex t_u (resp., t_v) in $\mathcal{T}_\beta$ corresponding to subtree T_u (resp., T_v) which contains u (resp., v). (Here too, we consider then most general case, when neither u nor v are in CV_ℓ.) Cut vertex $u' \in CV_\ell$ is the first vertex on the path from t_u to t_v in $\mathcal{T}_\beta$. In other words, it can be either parent of t_u or the first child on the path from t_u to t_v in $\mathcal{T}_\beta$. In both cases, u' can be found using level ancestor data structure. We can similarly find vertex v'. This completes the high-level overview of our navigation algorithm.

3.1.1 Preprocessing

Algorithm description. We proceed to give a detailed description of the preprocessing algorithm. It takes as an input a rooted tree $(T, rt(T))$ which induces a tree metric M_T. Notice that T can be transformed in linear time into a pruned tree $(T_{\mathsf{pnd}}, rt(T_{\mathsf{pnd}}))$ by invoking the procedure Prune$((T, rt(T)), R(T))$. Also, any 1-spanner for pruned tree T_{pnd} provides a 1-spanner for the original tree T with the same diameter. We may henceforth assume that the original tree T is pruned.

Our preprocessing algorithm construct two types of trees — augmented recursion trees and contracted trees. We preprocess every such tree in linear time so that subsequent *lowest common ancestor* (henceforth, LCA) and *level ancestor* (henceforth, LA) queries can be answered in constant time. For more details on these algorithms, refer to [BFC00, BFC04].

We now give details of the procedure PreprocessTree$((T, rt(T)), R(T), k)$. For pseudocode, see Algorithm 1. This procedure takes as parameters a rooted tree $(T, rt(T))$, the set $R(T)$ of required vertices of T, and an integer $k \geq 2$, representing the hop-diameter. It outputs the set of edges of [Sol13] spanner for T, together with the augmented recursion tree $(\Phi, rt(\Phi))$. In addition, it creates a data structure $\mathcal{D}_T$ which supports subsequent queries for k-hop 1-spanner paths in G_T between any two vertices $u, v \in R(T)$.

Let n denote the number of required vertices in T, that is, $n \coloneqq |R(T)|$. When $n \leq k + 1$, the algorithm invokes HandleBaseCase$((T, rt(T)), R(T), k)$, which we proceed to describe. If $n = k + 1$ and $rt(T)$ has exactly two children, u and v, the edge set of spanner, E, consists of the edges of T, denoted by $E(T)$, together with edge (u, v); otherwise, it consists of $E(T)$ only. Every vertex $v \in V(T)$ initializes its special adjacency list, $v.adj$ containing only edges in E. The recursion tree returned by this step, Φ, consists of a single vertex β. For each vertex v in $R(T)$ we create its copy and assign it as an *inner vertex* v' of β. At this stage, we keep a pointer from v to v' and vice-versa. Keeping inner vertices of each vertex in recursion tree will facilitate answering the queries later on. The procedure returns E as the edge set of the spanner, together with a (single-vertex) tree Φ rooted at β.

In what follows, we consider the case $n > k + 1$. The set of cut vertices, CV_ℓ is determined by calling the aforementioned procedure Decompose$((T, rt(T)), R(T), \ell)$, with parameter ℓ set to $\alpha'_{k-2}(n)$. We create a new vertex β and make it a root of the recursion tree Φ. This is done via a call to procedure NewVertex(CV_ℓ), which assigns to β as its inner vertices all the cut vertices in CV_ℓ. The procedure also keeps track of all the relevant pointers.

The algorithm PreprocessTree$((T, rt(T)), R(T), k)$ returns as its output the set of spanner edges, which consists of edges interconnecting the cut vertices, denoted by E', the edges connecting each cut vertex to required vertices in the tree, denoted by E'', and the edges E_i, for each of the

subtrees T_i, $i \in [p]$. If $k = 2$, then there is exactly one vertex v in CV_ℓ and we keep E' empty. For $k = 3$ the set E' consists of an edge between every two cut vertices, i.e., $E' = CV_\ell \times CV_\ell$. For $k \geq 4$, we first create a tree isomorphic to T, which has as its required vertices the inner vertices of β. The edge set E' is then obtained invoking PREPROCESSTREE$((T', rt(T')), CV_\ell, k-2)$.

We next compute the trees $T_1, \ldots, T_p$ in $T \setminus CV_\ell$ and the set of edges E'' consisting of edges between c and every vertex in border(c) for all $c \in CV_\ell$. For each tree T_i, we recursively preprocess it by calling PREPROCESSTREE$((T_i, rt(T_i)), R(T) \cap V(T_i), k)$. Let E_i be the edge set returned by this procedure and $(\Phi_i, rt(\Phi_i))$ be the recursion tree for each of the subtrees. We make $rt(\Phi_i)$ a child of β.

Finally, if $k \geq 3$ we construct the contracted tree $\mathcal{T}_\beta$ which corresponds to β. This is done via a call to procedure CREATECONTRACTED$(\beta, \{T_i\}_{i\in[p]}, \{\Phi_i\}_{i\in[p]})$. This procedure creates representative vertex t_i for each tree T_i, $i \in [p]$. In addition, for every inner vertex c of β it creates a vertex c' corresponding to it. (Recall that β has inner vertices corresponding to vertices CV_ℓ.) At this stage, it creates a pointer from c to c' and vice versa. At this stage, we have constructed a vertex set for $\mathcal{T}_\beta$; it remains to add the edges to it. For every vertex c' corresponding to inner vertex c of β, the algorithm connects it to every representative t_i which represents at least one vertex in border(c). Finally, the root $rt(\mathcal{T}_\beta)$ is the vertex corresponding to the vertex of the lowest level in T. The procedure returns $(\mathcal{T}_\beta, rt(\mathcal{T}_\beta))$.

The algorithm returns the set of edges $E' \cup E'' \cup \bigcup_{i\in[p]} E_i$ together with the recursion tree $(\Phi, rt(\Phi))$. The data structure $\mathcal{D}_T$ consists of all the vertices of T, all the vertices in every recursion tree, and all the vertices in every contracted tree, together with the data assigned to them.

An example of the preprocessing algorithm is given in Figure 1. Tree T is depicted on the left. Its number of (required) vertices is $n = 48$, and it is split into five subtrees $T_1, \ldots, T_5$ using the set of four cut vertices, marked green inside of the dotted area. The size of each subtree is at most $\alpha'_{k-2}(n) = \alpha'_2(48) = 10$. Tree T_1 has size $4 \leq k+1$ and it corresponds to base case of the spanner construction. Trees $T_2, \ldots, T_5$ have size 10 and are recursively split into subtrees of size at most $\alpha'_2(10) = 6$. Finally, one of the subtrees of T_3 (the subtree on the bottom) has size 6 and gets split using a single cut vertex into two subtrees of size 2 and 3. The four cut vertices used in the first level of recursion (inside of the dotted region) are interconnected using the construction for $k = 2$. Before the spanner construction for $k = 2$ is invoked, the algorithm makes the cut vertices required and all the other vertices Steiner vertices and prunes the tree. The pruned tree (as depicted inside of the dotted region) has as its root a Steiner vertex and has four vertices corresponding to the cut vertices; its edges are represented by dashed lines. Since its size is greater than $(k-2)+1 = 3$, it gets split using the cut vertex (which is the Steiner vertex) into two subtrees, each of size 2.

The recursion tree Φ corresponding to the spanner construction is depicted on the right towards the bottom. Each non-leaf vertex of Φ has a contracted tree associated to it. The root of Φ_T, denoted by β has a contracted tree $\mathcal{T}_\beta$ associated to it. Its vertices are the cut vertices of T and the vertices t_1 to t_5 corresponding to subtrees T_1 to T_5. In the image, the root of Φ points (via an arrow with a dotted line) to another recursion tree with two vertices, corresponding to the recursive construction for $k = 2$.

Algorithm guarantees. We shall use the following lemma from [Sol13] which bounds the number of cut vertices returned by DECOMPOSE$((T, rt(T)), R(T), \ell)$.

Lemma 3.1 ([Sol13]). *Let n be the number of required vertices of the tree, and $\ell = \alpha'_{k-2}(n)$. Then, the size of CV_ℓ returned by* DECOMPOSE$((T, rt(T)), R(T), \ell)$ *satisfies: $|CV_\ell| = 1$ if $k = 2$,*

Algorithm 1 Constructs spanner, together with necessary data required for efficient navigation. Each constructed tree is preprocessed for answering LCA and LA queries.

procedure PREPROCESSTREE($(T, rt(T)), R(T), k$) ▷ The main algorithm.
  PRUNE($(T, rt(T)), R(T)$)
  if $n \leq k+1$ **then return** HANDLEBASECASE($(T, rt(T)), R(T), k$) ▷ $n = |R(T)|$
  $CV_\ell \leftarrow$ DECOMPOSE($(T, rt(T)), R(T), \ell$) ▷ $\ell \leftarrow \alpha'_{k-2}(n)$
  create Φ consisting of a single vertex $\beta \leftarrow$ NEWVERTEX(CV_ℓ)
  if $k = 3$ **then** ▷ When $k = 2$, E' is empty.
    $E' \leftarrow CV_\ell \times CV_\ell$
  else if $k \geq 4$ **then**
    make T', a copy of T, with inner vertices of β as required vertices
    $(E', (\Phi', rt(\Phi'))) \leftarrow$ PREPROCESSTREE($(T', rt(T')), CV_\ell, k-2$)
  $\{T_1, T_2, \ldots, T_p\} \leftarrow T \setminus CV_\ell$ ▷ root $rt(T_i)$ of T_i is the vertex of the lowest level in T_i
  $E'' \leftarrow \cup_{u \in CV_\ell} \{u\} \times \text{border}(u)$
  for $i \in [p]$ **do**
    $(E_i, (\Phi_i, rt(\Phi_i)) \leftarrow$ PREPROCESTREE($(T_i, rt(T_i)), R(T) \cap V(T_i), k$)
    make $rt(\Phi_i)$ child of β
  if $k \geq 3$ **then** $(\mathcal{T}_\beta, rt(\mathcal{T}_\beta)) \leftarrow$ CREATECONTRACTED($\beta, \{T_i\}_{i\in[p]}, \{\Phi_i\}_{i\in[p]}$)
  return $(E' \cup E'' \cup \bigcup_{i\in[p]} E_i, (\Phi, \beta))$

procedure HANDLEBASECASE($(T, rt(T)), R(T), k$) ▷ Base case when $n \leq k+1$.
  $E \leftarrow E(T)$
  if $n = k+1$ and $rt(T)$ has exactly two children, u and v, **then** $E \leftarrow E \cup (u, v)$
  create Φ consisting of a single vertex $\beta \leftarrow$ NEWVERTEX($R(T)$)
  for each $v \in T$, create $v.adj$ based on edges in E
  return $(E, (\Phi, \beta))$

procedure NEWVERTEX(U) ▷ Creates a new vertex for Φ.
  create a new vertex β
  for all $v \in U$ **do**
    create a copy v' of v and make it inner vertex of β
    $v.ptr(\Phi) \leftarrow v'$, $v'.ptr(T) \leftarrow v.ptr(T)$, $v'.h \leftarrow \beta$ ▷ If $|U| = 1$, let $\beta.ptr(T) \leftarrow v.ptr(T)$
  return β

procedure CREATECONTRACTED($\beta, \{T_i\}_{i\in[p]}, \{\Phi_i\}_{i\in[p]}$) ▷ Creates a contracted tree for $\beta \in \Phi$.
  $E(\mathcal{T}_\beta) \leftarrow \emptyset$, $V(\mathcal{T}_\beta) \leftarrow \emptyset$
  for all $i \in [p]$ **do**
    create a new vertex t_i corresponding to T_i, add it to $V(\mathcal{T}_\beta)$
    $rt(\Phi_i).ptr(\mathcal{T}) \leftarrow t_i$
  for all inner vertices c of β **do**
    create a new vertex c' corresponding to c, add it to $V(\mathcal{T}_\beta)$
    $c.ptr(\mathcal{T}) \leftarrow c'$, $c'.ptr(\Phi) \leftarrow c$
    add to $E(\mathcal{T}_\beta)$ an edge between c' and every t_i such that $\text{border}(c) \cup T_i \neq \emptyset$
  return $(\mathcal{T}_\beta, rt(\mathcal{T}_\beta))$ ▷ $rt(\mathcal{T}_\beta) \in V(\mathcal{T}_\beta)$ corresponds to vertex of the lowest level in T.

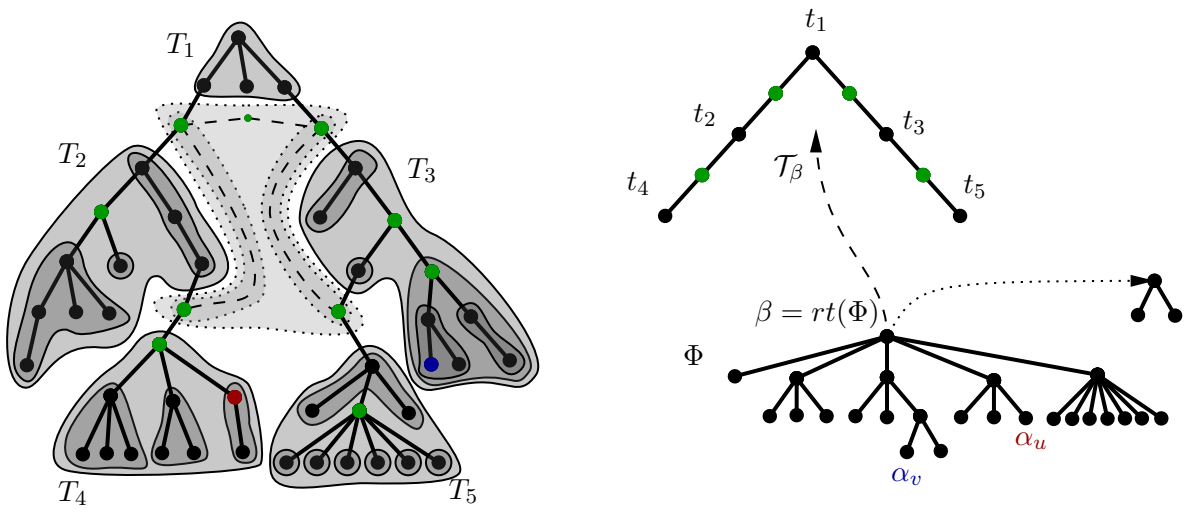

Figure 1: Result of the preprocessing algorithm.

$|CV_\ell| \leq \sqrt{n}$ *if* $k = 3$, *and* $|CV_\ell| \leq \lfloor \frac{|V(T)|}{\ell+1} \rfloor$ *if* $k \geq 4$.

In the following lemma, we bound the preprocessing time of the algorithm in Algorithm 1.

Lemma 3.2. *Let T be a tree with required size $|R(T)| = n$ and $|V(T)| = O(n)$. Algorithm* PreprocessTree$((T, rt(T)), R(T), k)$ *in time $O(n\alpha_k(n))$ outputs a k-hop 1-spanner G_T for $R(T)$ with $O(n\alpha_k(n))$ edges and the corresponding navigation data structure $\mathcal{D}_T$.*

Proof. First, we note that G_T is the same as the 1-spanner in the construction of Solomon [Sol13], which was shown to have $O(n\alpha_k(n))$ edges.

Let $f(n,k)$ be the running time of PreprocessTree$((T, rt(T)), R(T), k)$. Observe that, excluding the recursive calls in lines 14 and 10, the running time is $O(n)$. When $k = 2$, we have that $CV_\ell = 1$ and $|R(T_i)| \leq \alpha'_0(n) = \lceil n/2 \rceil$. Thus, $f(n,2) = \sum_{i\in[p]} f(|R(T_i)|, 2) + O(n)$ which resolves to $f(n,2) = O(n \log n) = O(n\alpha_2(n))$. When $k = 3$, we have that $|R(T_i)| \leq \alpha'_1(n) = \lceil \sqrt{n} \rceil$, and hence, $f(n,3) = \sum_{i\in[p]} f(|R(T_i)|, 3) + O(n)$ which resolves to $O(n \log\log(n)) = O(n\alpha_3(n))$. When $k \geq 4$, recall that $\ell = \alpha'_{k-2}(n)$, $|R(T_i)| \leq \ell$ and $|CV_\ell| \leq \lfloor \frac{n}{\ell+1} \rfloor$ by Lemma 3.1. Thus, we have:

$$f(n,k) = \sum_{i\in[p]} f(|R(T_i)|, k) + f\left(\frac{n}{\alpha'_{k-2}(n)}, k-2\right) + O(n) \tag{1}$$

which resolves to $f(n,k) = O(n\alpha_k(n))$; see Theorem 3.12 in [Sol13] for a detailed inductive proof. □

We next list several properties of the preprocessing algorithm, which follow from the algorithm description. These properties are used in Section 3.1.2, where the query algorithm is presented. Recall that we use $\mathcal{D}_T$ to denote the data structure which consists of all the vertices of T, all the vertices in every recursion tree, and all the vertices in every contracted tree, together with the data assigned to them. See Table 2 for the summary of data stored in $\mathcal{D}_T$.

Property 1. Every tree constructed by the algorithm is preprocessed for answering LCA and LA queries in constant time.

Property 2. For every vertex $x \in \mathcal{D}_T$, pointer $x.ptr(T)$ is either $\emptyset$ or it points to the vertex corresponding to x in T. In particular, it is defined in the following cases: *(i)* $x \in T$, *(ii)* x is a non-leaf vertex in some augmented recursion tree for construction with $k = 2$, *(iii)* x is an inner vertex in some augmented recursion tree.

Property 3. For every vertex $x \in \mathcal{D}_T$, pointer $x.ptr(\Phi)$ is either $\emptyset$ or it points to its corresponding inner vertex in some recursion tree. In particular, it is defined in the following cases: *(i)* $x \in T$, *(ii)* x corresponds to a cut vertex in some contracted tree, *(iii)* x is an inner vertex of a non-leaf in some augmented recursion tree for construction with $k \geq 4$.

Property 4. For every vertex $x \in \mathcal{D}_T$, pointer $x.ptr(\mathcal{T})$ is either $\emptyset$ or it points to its corresponding vertex in some augmented recursion tree. In particular, it is defined in the following cases: *(i)* x is an inner vertex of a non-leaf in some augmented recursion tree for construction with $k \geq 3$, *(ii)* x is a non-root in some recursion tree for construction with $k \geq 3$.

Property 5. For every vertex $x \in \mathcal{D}_T$, pointer $x.h$ is either $\emptyset$ or it points to its corresponding vertex in some augmented recursion tree. In particular, it is defined if x is an inner vertex of a vertex in some recursion tree.

Property 6. Whenever a vertex x is considered in procedure HandleBaseCase() with parameter k, its entry $x.adj$ contains only adjacency list in the subgraph of G_T induced by vertices considered in the same base case. Moreover, this subgraph contains $O(k)$ vertices.

Property 7. Every contracted tree $\mathcal{T}_\beta$, corresponding to a vertex β in some recursion tree Φ satisfies the following:

(*i*) It only contains representative vertices and cut vertices.

(*ii*) There is no edge between two representative vertices.

(*iii*) There is an edge between a cut vertex c and a representative vertex t_i iff $c \in \text{border}(T_i)$.

Observation 3.1. *Given a tree T of required size n and an integer k, the depth of augmented recursion tree corresponding to T is $O(\alpha_k(n))$.*

Proof. The depth of augmented recursion tree satisfies recurrence $D(n,k) = D(\alpha_{k-2}(n),k)+1$, with a base case $D(n,k) = 1$, whenever $n \leq k+1$. This recurrence has solution $D(n,k) = O(\alpha_k(n))$. □

3.1.2 Query algorithm

The algorithm which finds a k-hop path between u and v in 1-spanner G_T of tree T is presented in Algorithm 2. It takes two vertices u and v and a parameter k, representing the hop-diameter of G_T.

Algorithm description. We proceed to give details of the query algorithm. It takes as an input two vertices u and v and an integer parameter $k \geq 2$, representing the hop-diameter. Let Φ denote the recursion tree corresponding to spanner construction which considered u and v with parameter k. The algorithm first checks whether u and v were considered in the same base case corresponding to the call to HandleBaseCase() during the construction of G_T. This check is performed in line 2. We check if u and v point to the same leaf in Φ. The inner vertex corresponding to u (resp., v)

field	meaning	defined for vertices in			
		T	Φ, inner	Φ	$\mathcal{T}$
$ptr(T)$	pointer to vertex in T	yes	yes	if non-leaf & $k = 2$	
$ptr(\Phi)$	pointer to inner vertex in Φ	yes	if non-leaf host & $k \geq 4$		if cut vertex
$ptr(\mathcal{T})$	pointer to vertex in $\mathcal{T}$		if non-leaf host & $k \geq 3$	if non-root & $k \geq 3$	
h	pointer to home vertex in Φ		yes		
adj	adjacency table	yes	yes		
$level$	level in the tree	yes		yes	yes

Table 2: Summary of data stored with every vertex in $x \in \mathcal{D}_T$. Undefined entries take value $\emptyset$. Note that the preprocessing algorithm constructs more than one augmented recursion tree when $k \geq 4$ and more than one contracted tree when $k \geq 3$.

in Φ obtained via $u.ptr(\Phi)$ (resp., $v.ptr(\Phi)$). We use $u.ptr(\Phi).h$ (resp., $v.ptr(\Phi).h$) to obtain the actual vertex in Φ, which contains $u.ptr(\Phi)$ (resp., $v.ptr(\Phi)$) as its inner vertices. If $u.ptr(\Phi).h$ is equal to $v.ptr(\Phi).h$, the algorithm returns the path found by BFS on the subgraph of spanner G_T induced on all the vertices corresponding to the same base case (line 3). This BFS uses adjacency list $u.adj$ stored with vertex u, which contains only the edges of the spanner corresponding to this base case. In other words, the algorithm will only visit the subgraph of G_T induced on the vertices corresponding to the same base case as u and v.

When u and v do not correspond to the same vertex in Φ, the algorithm finds LCA in Φ of $u.ptr(\Phi).h$ and $v.ptr(\Phi).h$, denoted by β (line 4). If $k = 2$, then, by Property 2, β corresponds to a single vertex in T; its corresponding vertex in T is $\beta.ptr(T)$. The algorithm returns path consisting of at most three vertices in T, namely $\{u.ptr(T), \beta.ptr(T), v.ptr(T)\}$. We use braces to denote that consecutive duplicates are removed from it. For example, when $u = \beta$, then $u.ptr(T) = \beta.ptr(T)$ and the algorithm returns two vertices: $\{u.ptr(T), v.ptr(T)\}$.

When $k \geq 3$, the algorithm proceeds to find cut vertices corresponding to u and v. For that purpose, it considers contracted tree $\mathcal{T}_\beta$, corresponding to β. First of all, it locates vertices corresponding to u (resp., v) in $\mathcal{T}_\beta$, via a call to LocateContracted(u, β). If u points to β in Φ, it means that u is a cut vertex at the required level; all we need to do is to find its corresponding vertex in $\mathcal{T}_\beta$, which is obtained via $u.ptr(\Phi).ptr(\mathcal{T}_\beta)$. If u is not a cut vertex at the required level, we use level ancestor data structure to find child of β on the path to vertex corresponding to u. By Property 4, this child corresponds to a unique vertex u' in $\mathcal{T}_\beta$, which is a representative of connected component containing u. Vertex v' corresponding to v is found analogously.

Next, we would like to find the first cut vertex x on the path from u' to v' (resp., the first cut vertex y on the path from v' to u') in the contracted tree $\mathcal{T}_\beta$. First the algorithm finds lowest common ancestor c of u' and v' (cf. line 9). Then, it invokes FindCut(u, u', v', β, c), which we explain next. If u' already corresponds to a cut vertex, we assign u' to x. If that is not the case, when v' is a descendant of u', we let x be the child of u' on the path to v' and otherwise we let it be the parent of u'; in both cases, we can find x using level ancestor data structure on $\mathcal{T}_\beta$. Vertex y is found similarly, using a call to FindCut(v, v', u', β, c). When $k = 3$ the algorithm reports vertices corresponding to u, x, y, v in T (line 13). Otherwise, it proceeds recursively to find a $(k-2)$-hop path between inner vertices of β in Φ corresponding to x and y (line 15).

We refer reader to Figure 1 for an illustration of a query algorithm. Upon a query to navigate

between the red vertex $u \in T$ and the blue vertex $v \in T$, the algorithm uses $u.ptr(\Phi)$ and $v.ptr(\Phi)$ to find the corresponding vertices in Φ_T, denoted by α_u and α_v. Since $\alpha_u \neq \alpha_v$, this means that u and v were not considered together in a base case (corresponding to invocation of HandleBaseCase()). Next, the algorithm finds $LCA(\alpha_u, \alpha_v)$, which is the root $rt(\Phi)$, denoted by β in the picture. Using the level ancestor data structure, we find children α_1, α_2 of β, on the path to α_u and α_v, respectively. Vertex α_1 points to t_4 in the contracted tree $\mathcal{T}_\beta$; similarly, vertex α_2 points to t_3. The LCA of t_3 and t_4 is the root of $\mathcal{T}_\beta$ and the cut vertices corresponding to t_3 and t_4 are their parents. Finally, we recursively find the path (of at most 2 hops) between the chosen cut vertices using the augmented recursion tree for construction with $k-2$ (pointed to by a dotted arrow).

Algorithm 2 Query for a k-hop path in tree 1-spanner G_T between two vertices u and v.

```
1:  procedure FindPath(u, v, k)                                   ▷ The query algorithm.
2:      if u.ptr(Φ).h = v.ptr(Φ).h and u.ptr(Φ).h is a leaf of Φ then
3:          return BFS(u, v)        ▷ BFS on G_T induced on {w ∈ V(T) | w.ptr(Φ).h = u.ptr(Φ).h}.
4:      β ← LCA(u.ptr(Φ).h, v.ptr(Φ).h)
5:      if k = 2 then
6:          return {u.ptr(T), β.ptr(T), v.ptr(T)}
7:      u' ← LocateContracted(u, β)
8:      v' ← LocateContracted(v, β)
9:      c ← LCA(u', v')
10:     x ← FindCut(u, u', v', β, c)
11:     y ← FindCut(v, v', u', β, c)
12:     if k = 3 then
13:         return {u.ptr(T), x.ptr(Φ).ptr(T), y.ptr(Φ).ptr(T), v.ptr(T)}
14:     else
15:         return {u.ptr(T), FindPath(x.ptr(Φ), y.ptr(Φ), k − 2), v.ptr(T)}
16:
17: procedure LocateContracted(u, β)                  ▷ Locates u' corresponding to u in T_β.
18:     if u.ptr(Φ).h = β then
19:         return u.ptr(Φ).ptr(T_β)
20:     else
21:         return LA(u.ptr(Φ).h, β.level + 1).ptr(T_β)
22:
23: procedure FindCut(u, u', v', β, c)     ▷ Finds the first cut vertex x on the path from u' to v'.
24:     if u.ptr(Φ).h = β then
25:         return u'
26:     else if u' = c then
27:         return LA(v', u'.level + 1)
28:     else
29:         return LA(u', u'.level − 1)
```

Algorithm guarantees. We next argue the correctness of the query algorithm.

Lemma 3.3. *Given a tree T preprocessed by* PreprocessTree$((T, rt(T)), R(T), k)$*, and two vertices $u, v \in R(T)$, the algorithm* FindPath(u, v, k) *outputs a 1-spanner path between u and v in G_T consisting of at most k edges.*

Proof. We will prove the lemma by structural induction.

The first base case is when the condition in line 2 is true and the algorithm uses BFS to report the path (line 3). By Properties 3 and 5, we know that $u.ptr(\Phi).h$ and $v.ptr(\Phi).h$ point to vertices corresponding to u and v in Φ. Moreover, since they correspond to a leaf in Φ, this means that u and v were in the same tree T_{base} processed by HandleBaseCase$((T_{base}, rt(T_{base})), R(T_{base}), k)$. By Property 6, we know that vertex u contains adjacency list $u.adj$ restricted to G_T induced on $V(T_{base})$; same holds for v and every other vertex in $V(T_{base})$. This ensures that BFS will find the shortest path from u to v, which is guaranteed in [Sol13] to be the 1-spanner path of at most k edges.

The second base case is when $k = 2$. The algorithm returns path $\{u.ptr(T), \beta.ptr(T), v.ptr(T)\}$, which, by Property 2 correctly map to corresponding vertices in a given tree T. We next argue that this path is a valid 1-spanner path in G_T. Among the common vertices on the paths from $rt(\Phi)$ to $u.ptr(\Phi).h$ and from $rt(\Phi)$ to $u.ptr(\Phi).h$, vertex β has the lowest level. After removing $\beta.ptr(T)$ from T, vertices u and v are not in the same subtree. In other words, $\beta.ptr(T)$ is on the path $\mathcal{P}_T(u, v)$. Suppose that $u \neq \beta.ptr(T)$ and $v \neq \beta.ptr(T)$ and let T_u (resp. T_v) be the tree containing u (resp., v) after $\beta.ptr(T)$ has been removed. By the previous argument, it must be that $T_u \neq T_v$. From the inductive statement, we know that $u.ptr(T)$ (resp., $v.ptr(T)$) is a required vertex in T_u (resp., T_v). This means that G_T contains an edge between $(u.ptr(T), \beta.ptr(T))$ and $(v.ptr(T), \beta.ptr(T))$. Hence, the path $(u, \beta.ptr(T), v.ptr(T))$ exists in G_T and is a valid 1-spanner path in T. Cases when $u = v$, $u = \beta.ptr(T)$, and $v = \beta.ptr(T)$ are handled similarly.

The third base case is when $k = 3$. The algorithm first finds vertices u' and v' corresponding to u and v in the contracted tree $\mathcal{T}_\beta$. The correctness follows by Property 4. Finding relevant cut vertex for u' is done in procedure FindCut(u, u', v', b, c). Correctness follows by Property 7. Hence, x (resp., y) is the closest cut vertex to u (resp. v) on $\mathcal{P}_T(u, v)$. Recall that by inductive statement, u and v are required vertices. Since x is in border(u) and similarly y in border(v), there are edges (x, u) and (y, v) in G_T. In addition, since x and y are cut vertices, they are connected via an edge. This concludes the analysis of base cases.

We take $k \geq 4$ for an inductive step. From the analysis of the base case $k = 3$, we know that cut vertices x and y in $\mathcal{T}_\beta$ are correctly computed, and G_T contains edges (u, x) and (y, v). By Property 3 the pointers $x.ptr(\Phi), y.ptr(\Phi)$ point to inner vertices of β in Φ. Since $x.ptr(\Phi)$ and $y.ptr(\Phi)$ are required vertices for Φ' corresponding to construction with hop-diameter $k - 2$ (cf. lines 9 to 10 in PreprocessTree()), by the inductive hypothesis the recursive call in line 15 returns a 1-spanner path between $x.ptr(\Phi)$ and $y.ptr(\Phi)$ of at most $k - 2$ hops. Recalling that we have at most two more hops, i.e., if $u \notin CV_\ell$, $(u.ptr(T), x.ptr(\Phi).ptr(T))$ and if $v \notin CV_\ell$, $(y.ptr(\Phi).ptr(T), v.ptr(T))$, the statement follows. □

The following lemma states the running time of our query algorithm.

Lemma 3.4. *Algorithm* FindPath(u, v, k) *runs in time $O(k)$.*

Proof. Procedures LocateContracted and FindCut access pointer values and perform LCA and LA queries; the number of such operations is constant, allowing us to conclude that both procedures run in $O(1)$ time. The only nontrivial operation in FindPath is performing BFS in

line 3. By Property 6, this BFS visits only the subgraph of G_T induced on the vertices which correspond to the same base case as u and v. The number of vertices in this subgraph is $O(k)$, hence the running time of BFS is also $O(k)$.

In conclusion, the algorithm either performs $O(k)$ operations and does not continue recursively, or it performs constant number of operations and proceeds recursively with parameter $k-2$. This allows us to conclude that the running time of FindPath is $O(k)$. □

3.2 Navigating tree covers

To prove Theorem 1.2, we rely on tree cover theorems summarized in Table 1. Let ζ denote the number of trees in the cover and γ the stretch of the cover; let $M_X = (X, \delta_X)$ be the metric space we are working on. For each of the ζ trees in the cover, we employ Theorem 1.1 and construct a spanner G_{T_i} and a data structure $\mathcal{D}_{T_i}$. For Ramsey tree covers, in the preprocessing step we store a mapping from every point x in the metric space to its "home" tree. Upon a query for a path between u and v, for the Ramsey tree covers it is sufficient to use this information to find the corresponding tree in constant time. Otherwise, for each of the ζ trees, we query $\mathcal{D}_{T_i}$ for the distance between u and v in T_i in $O(1)$ time. (This step takes $O(\zeta)$ time.) Once the tree with the smallest distance between u and v, T^*, has been found, we query for the k-hop shortest path in T^* between u and v in $O(k)$ time using the result from Theorem 1.1.

4 Fault tolerance in doubling metrics

In this section we strengthen the navigation scheme of Section 3 to achieve fault-tolerance in doubling metrics. We start with required definitions, move on to presenting a new construction of tree covers in doubling metrics, and then build on this tree cover to get a fault-tolerant (FT) spanner of bounded hop-diameter. Equipped with such a spanner, obtaining an FT navigation scheme follows along similar lines to the one presented Section 3 for a non-FT spanner.

Let $X = (X, \delta_X)$ be an n-point metric of doubling dimension d. An *f-fault-tolerant (FT) t-spanner* of X is a t-spanner for X such that, for every set $F \subseteq X$ of size at most f, $f \leq n-2$, called a *faulty set*, it holds that $\delta_{H\setminus F}(x,y) \leq t \cdot \delta_X(x,y)$, for any pair of $x, y \in X \setminus F$. An f-FT spanner H is said to have *hop-diameter* k if the hop-diameter of $H \setminus F$ is at most k for any faulty set F of size at most f, and thus there is a non-faulty t-spanner path in H of at most k hops for any pair of non-faulty points. The main result of this section is a construction of an f-FT spanner with bounded hop-diameter for doubling metrics whose size matches the size bound for non-FT spanners (up to the dependency on f). Our construction relies on the notion of a *robust tree cover* that we introduce below. This new tree cover notion generalizes the Euclidean "Dumbbell Tree" theorem (Theorem 2 in [ADM+95]). In what follows, we shall use $\mathcal{P}_T(u,v)$ to denote the path between leaves x and y in a rooted tree T. We denote by T_v a subtree of T rooted at a vertex $v \in T$.

Definition 4.1 (Robust Tree Cover)**.** A robust (γ, ζ)-tree cover $\mathcal{T}$ for a metric (X, δ_X) is a collection of ζ trees satisfying:

(1) For every tree $T \in \mathcal{T}$, there is a 1-to-1 correspondence between points in X and leaves of T.

(2) For every $x \neq y \in X$, there exists a tree $T \in \mathcal{T}$ such that the path from x to y obtained by replacing each vertex v in $\mathcal{P}_T(x,y)$ with an *arbitrary* leaf point of T_v has weight at most $\gamma \cdot \delta_X(x,y)$. We say that T *covers* x and y.

Property (2) in Definition 4.1, which we call *robustness*, implies that we can obtain an (ordinary) tree cover by replacing any internal vertex of a tree $T \in \mathcal{T}$ with a point associated with an arbitrary leaf in the subtree rooted at that vertex. The robustness is the key in our construction of an f-FT spanner with a bounded hop-diameter. In the following theorem, we show that doubling metrics have robust tree covers with a few trees; the proof is deferred to Section 4.2.

Theorem 4.1. *For any metric (X, δ_X) of doubling dimension d and any parameter $\epsilon > 0$, we can construct a robust $(1+\epsilon, \epsilon^{-O(d)})$-tree cover $\mathcal{T}$ for (X, δ_X) in $O_{d,\epsilon}(n \log n)$ time.*

The $O_{d,\epsilon}$ notation hides the dependency on d and ϵ. The tree cover theorem by [BFN19] for doubling metrics generalizes all but the robustness of the dumbbell tree theorem. By examining the proof closely, we observe that, in the tree cover of [BFN19], each internal vertex of the tree is replaced by a *specific point* chosen from the leaves in the subtree rooted at that vertex; in particular, Claim 8 in [BFN19] fails if the point is chosen arbitrarily from the leaves.

In the following, we show how to construct a FT spanner with a bounded hop-diameter from a robust tree cover.

4.1 Construction of fault-tolerant spanners with bounded hop-diameter

Theorem 4.2. *Given an n-point metric (X, δ_X) of doubling dimension d, a parameter $\epsilon > 0$, and integers $1 \leq f \leq n-2$, and $k \geq 2$, we can construct an f-FT spanner with hop-diameter k and $\epsilon^{-O(d)} n f^2 \alpha_k(n)$ edges in $O_{d,\epsilon}(n(\log(n) + f^2 \alpha_k(n)))$ time.*

Proof. Let $\mathcal{T}$ be a robust $(1+\epsilon, \epsilon^{-O(d)})$-tree cover constructed as in Theorem 4.1. For each tree $T \in \mathcal{T}$, we construct a graph H_T and then form an f-FT spanner H as $H = \cup_{T \in \mathcal{T}} H_T$.

Initially, the vertex set of H_T contains points in X, and the edge set of H_T is empty. We then construct a 1-spanner for T with k hops and $O(n\alpha_k(n))$ edges in $O(n\alpha_k(n))$ time, denoted by K_T, using the algorithm of Solomon [Sol13]. Note that edges in T are unweighted. For every vertex $v \in T$, we choose a set $R(v)$ of (arbitrary) $f+1$ points associated with leaves of T_v; if T_v has strictly less than $f+1$ leaves, $R(v)$ includes all the leaves. For every edge $(u, v) \in K_T$, we add to H_T edges between points in $R(u)$ and $R(v)$ to make a biclique. The weight of each edge is the distance between its endpoints in X. This completes the construction of H_T and hence of H.

Observe by the construction that $|E(H_T)| = O(f^2 |E(K_T)|) = O(f^2 n \alpha_k(n))$. It then follows that $|E(H)| = |\mathcal{T}| O(f^2 n \alpha_k(n)) = \epsilon^{-O(d)} n f^2 \alpha_k(n)$. Observe also by the construction that the running time to construct H_T is $O(f^2 n \alpha_k(n))$. Thus, the running time to construct H is $O_{d,\epsilon}(n \log(n)) + O(f^2 n \alpha_k(n))$, as claimed.

Finally, we bound the stretch and the hop-diameter of H. Let $x \neq y$ be any two non-faulty points in X, and T be a tree in $\mathcal{T}$ that covers x and y. Let Q be any k-hop 1-spanner path between x and y in K_T. Let $x = v_0, \ldots, v_k = y$ be vertices of Q. We claim that for every $i \in [k]$, there exists a non-faulty point in $R(v_i)$. If $|R(v_i)| = f+1$, then clearly it contains a non-faulty point. Otherwise, $R(v_i) \cap \{x, y\} \neq \emptyset$. This is because Q is a 1-spanner path and hence, any vertex in Q is either an ancestor of x or an ancestor of y or both.

We now construct a k-hop path P for Q as follows. For every $i \in [k]$, we replace v_i by a non-faulty point $p_i \in R(v_i)$. Thus, P is a path in H_T (and hence in H) of hop-diameter k. Furthermore, by property (2) in Definition 4.1 and the fact that Q is a 1-spanner path, P has stretch $(1+\epsilon)$, as desired. □

4.2 Construction of Robust Tree Covers

In this section, we prove Theorem 4.1. Our construction follows the construction of tree covers of Bartal et al. [BFN19]. An *r-net* of a metric space (X, δ_X) is a subset of points $N \subseteq X$ such that (a) for every two different points $x \neq y \in N$, $\delta_X(x, y) > r$ and (b) for every point $x \in X$, there exists a point $y \in N$ such that $\delta_X(x, y) \leq r$. We introduce the notion of *pairing* cover for nets (formally defined in Definition 4.2), which is the key to achieving the robustness of our tree cover. We first review the construction of Bartal et al. [BFN19], and then describe how the pairing cover can be used to construct a robust tree cover.

The construction of Bartal et al. [BFN19] can be divided into two steps.

(Step 1) They consider a hierarchy of nets $N_0 \supseteq N_1 \supseteq N_2 \supseteq \ldots$, where N_i is a 2^i-net of (X, δ_X).[5] Each net N_i is then *partitioned* into $\sigma = \epsilon^{-O(d)}$ *well-separated* sets $N_{i1}, \ldots, N_{i\sigma}$ in the sense that for every $x \neq y \in N_{it}$, $\delta_X(x, y) = \Omega(2^i/\epsilon)$ for any $t \in \{1, \ldots, \sigma\}$.

(Step 2) They construct a collection of $O(\sigma \log(1/\epsilon))$ trees $\{T_{j,p}\}$ where $j \in \{1, 2, \ldots, \sigma\}$ and $p \in \{0, 1, \ldots, \log(1/\epsilon) - 1\}$. Each tree $T_{j,p}$ is constructed by considering levels i of the net hierarchy such that $i \equiv p \mod \log(1/\epsilon)$ and marking points as *clustered* along the way. Specifically, for every point $x \in N_{ij}$ that is unclustered, add all unclustered points at distance $O(2^i/\epsilon)$ from x to the tree rooted at x as the children of x; these points are then marked as *clustered*.

It follows from the construction that every internal node of each tree is associated with a unique point $x \in X$. To achieve the robustness, we modify the construction of Bartal et al. [BFN19] in two ways. In Step 1, we construct a *cover* (instead of a partition) of size $\epsilon^{-O(d)}$ for N_i that has a *pairing property*: each point x in a set C_{ij} in the cover has at most one point $y \in C_{ij}$ such that $\delta_X(x, y) \leq 2^i/\epsilon$; y is said to be *paired* with x. (See Definition 4.2 for a formal definition.) In Step 2, for each point $x \in C_{ij}$, we connect the subtree containing x and all other subtrees containing vertices within distances $O(2^i)$ from x, and the subtree containing y where y is paired with x in C_{ij}.

We now give the details of the construction of a robust tree cover. In this section, our focus is primarily on describing the algorithms and proving various properties of the cover. The implementation is discussed in Section 4.3. We say that a collection of subsets $\mathcal{C}$ of a set S is a *cover* for S if $\cup_{C \in \mathcal{C}} C = S$.

Definition 4.2 (Pairing Cover)**.** A cover $\mathcal{C}_i$ of a 2^i-net N_i is a pairing cover if:

(1) For every set $C \in \mathcal{C}_i$ and every $x \in C$, there exists at most one point $y \neq x$ in C such that $\delta_X(x, y) \leq 2^i/\epsilon$.
(2) For every $x \neq y \in N_i$ such that $\delta_X(x, y) \leq 2^i/\epsilon$, there exists a set $C \in \mathcal{C}_i$ such that both x, y are in C. We say that x and y are paired by C.

Next, we construct a pairing cover for N_i with a small number of sets. We use the following well-known packing lemma.

Lemma 4.1 (Packing Lemma)**.** *Let P be a point set in a metric (X, δ_X) of doubling dimension d such that for every $x \neq y \in P$, $r < \delta_X(x, y) \leq R$. Then $|P| \leq \left(\frac{4R}{r}\right)^d$.*

Step 1: Constructing a pairing cover of N_i. The construction has two smaller steps. First, we construct a well-separated partition $\mathcal{P}_i$ of N_i following Bartal et al. [BFN19]. Then in the second step, we construct a pairing cover $\mathcal{C}_i$ from $\mathcal{P}_i$.

[5] We chose indices to start from 0 for the ease of presentation; cf. [BFN19].

- **Step 1a.** Initially $\mathcal{P}_i = \emptyset$. We consider each point $x \in N_i$ in turn, and if there exists a set $P \in \mathcal{P}_i$ such that $\delta_X(x, y) > (3/\epsilon)2^i$ for every $y \in P$, then we add x to P. Otherwise, we add a new set $\{x\}$ to $\mathcal{P}_i$. Let $\sigma_1 = |\mathcal{P}_i|$.
- **Step 1b.** Let $\sigma_2 = \max_{x \in N_i} |\{y \in N_i : \delta_X(x, y) \leq 2^i/\epsilon\}|$. For each set $P \in \mathcal{P}_i$, we construct a collection $\mathcal{C}(P) = \{P_1, \ldots, P_{\sigma_2}\}$ of σ_2 sets as follows. For each $x \in P$, let $\langle y_1, y_2, \ldots, y_{\sigma_2} \rangle$ be a sequence of all points (in arbitrary order) in N_i that have distances at most $2^i/\epsilon$ from x. (Possibly, there could be strictly less than σ_2 such points, and in this case, we duplicate some points to get exactly σ_2 points in the sequence.) We then construct the set $P_j = \cup_{x \in P}\{x, y_j\}$ for each $j \in \{1, 2, \ldots, \sigma_2\}$. That is, P_j contains every point x in P and the j-th point in its sequence. Finally, we set: $\mathcal{C}_i = \cup_{P \in \mathcal{P}_i} \mathcal{C}(P)$.

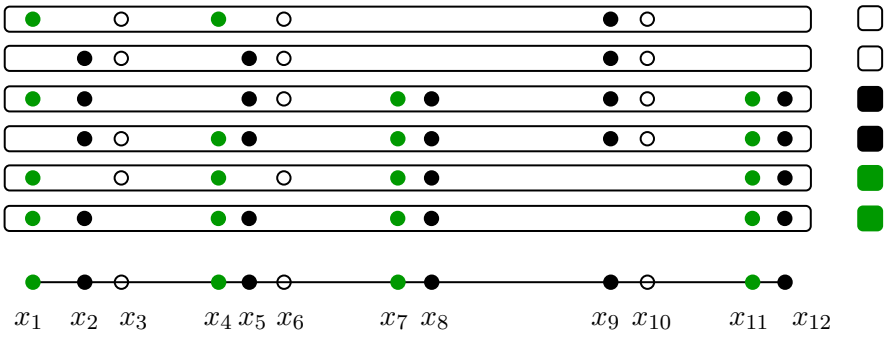

Figure 2: Pairing cover of a net.

An example of a pairing cover is presented in Figure 2. Fix level $i = 0$ for simplicity. We have a net N of 12 points on a line, $x_1, \ldots, x_{12}$. The partition $\mathcal{P}$ consists of $\sigma_1 = 3$ sets: green, black and white points. For the set of green points, we construct two sets. In the first set, we add x_1 and its closest point within distance $1/\epsilon$, x_2, and similarly $\{x_4, x_5\}$, $\{x_7, x_8\}$, and $\{x_{11}, x_{12}\}$. In the second set, for every green point, we add it together with its second-closest point within distance $1/\epsilon$.

In the following lemma, we show that $\mathcal{C}_i$ is a paring cover of N_i.

Lemma 4.2. *$\mathcal{C}_i$ is a pairing cover of N_i of size $\epsilon^{-O(d)}$.*

Proof of Lemma 4.2. First, we bound the size of $\mathcal{C}_i$. Observe by the construction that $|\mathcal{C}_i| = \sigma_1 \cdot \sigma_2$. In what follows, we show that $\sigma_1 = \epsilon^{-O(d)}$ and $\sigma_2 = \epsilon^{-O(d)}$, implying the bound on $|\mathcal{C}_i|$ as claimed in the lemma.

Observe that we add a new set $\{x\}$ to $\mathcal{P}_i$ in Step 1a if every point in N_i considered before x and within a distance of $(3/\epsilon)2^i$ from x belongs to an existing partition of $\mathcal{P}_i$. By the packing lemma (Lemma 4.1), there are at most $\tau = \epsilon^{-O(d)}$ such points. Once we make $\tau + 1$ sets, the algorithm will not add any new set. It follows that $\sigma_1 = \tau + 1 = \epsilon^{-O(d)}$. Also by the packing lemma, since N_i is a 2^i-net, $\sigma_2 = \epsilon^{-O(d)}$, as claimed.

Next, we show the pairing property of $\mathcal{C}_i$. Observe by construction that $\mathcal{C}_i$ is a cover of N_i. Furthermore, for every $C \in \mathcal{C}_i$, $C = P_j$ for some $P_j \in \mathcal{C}(P)$ constructed from a set $P \in \mathcal{P}_i$, for some $j \in [\sigma_2]$. Recall by the construction in Step 1a that the distance between every two points in P is at least $(3/\epsilon)2^i$. It follows that for every $x \in C$, if there exists $y \in C$ such that $y \neq x$ and $\delta_X(x, y) \leq 2^i/\epsilon$, then either (a) $y = x_j$ where x_j is the j-th point in the sequence of x and $x \in P$ or

(b) $x = y_j$ where y_j is the j-th point in the sequence of y and $y \in P$. In either case, there is only one such y.

Finally, consider any pair of points $x \neq y \in N_i$ such that $\delta_X(x,y) \leq 2^i/\epsilon$. Since $\mathcal{P}_i$ is a partition of N_i, there exists a set $P \in \mathcal{P}_i$ such that $x \in P$. Since $\delta_X(x,y) \leq 2^i/\epsilon$, y must be some point x_j in the sequence of x for some $j \in \{1,2,\ldots,\sigma\}$. Thus, $\{x,y\} \in P_j$. That is, there is a set in $\mathcal{C}_i$ that contains both x and y. □

Step 2: Constructing a robust tree cover $\mathcal{T}$. Let $N_0 \supseteq N_1 \supseteq \ldots$ be the hierarchy of nets of (X,δ_X) where N_i is a 2^i-net of X and the last net in the sequence contains a single point. By scaling, we assume that the minimum distance in X is larger than $1/(4\epsilon)$. For each net N_i, we construct a pairing cover $\mathcal{C}_i$, and form a sequence $\langle C_1, C_2, \ldots, C_{\sigma_3}\rangle$ of sets in $\mathcal{C}_i$; here σ_3 is the size of $\mathcal{C}_i$, which is $\epsilon^{-O(d)}$ by Lemma 4.2. For each $j \in \{1,2,\ldots,\sigma_3\}$ and each $p \in \{0,1,\ldots,\lceil\log(1/\epsilon)\rceil - 1\}$, we construct a tree $T_{j,p}$ and form the cover: $\mathcal{T} = \{T_{j,p} : j \in \{1,2,\ldots,\sigma_3\} \wedge p \in \{0,1,\ldots,\lceil\log(1/\epsilon)\rceil - 1\}\}$. Clearly the size of the cover is $O(\sigma_3 \log(1/\epsilon)) = \epsilon^{-O(d)}$.

We now focus on constructing $T_{j,p}$; the construction is in a bottom-up manner as follows. $T_{j,p}$ has n leaves which are in 1-to-1 correspondence with points in X. Let $I = \{i : i \equiv p \mod \lceil\log(1/\epsilon)\rceil\}$ be the set of levels congruent to p modulo $\lceil\log(1/\epsilon)\rceil$. For each level $i \in I$ from lower levels to higher levels, let C_j be the j-th set in the sequence of the pairing cover $\mathcal{C}_i$. Let $i' = i - \lceil\log(1/\epsilon)\rceil$, and $F_{i'}$ be the collection of trees constructed at level i'. (F_0 contains leaves of T.) For each point $x \in C_j$, let $T_x \in F_{i'}$ be the tree containing x, and $F_x \subseteq F_{i'}$ be a collection of subtrees such that each tree $T \in F_x$ contains a point z within distance 2^i from x.

For every two points x, y that are paired by C_j, we add a new node v and make the roots of trees in $\{T_x, T_y\} \cup F_x \cup F_y$ children of v. The resulting forest after this process is denoted by F_i. At the top level $i_{\max}$, if $F_{i_{\max}}$ contains more than one tree, we merge them into a single tree by creating a new node, and making the roots of the trees in $F_{i_{\max}}$ children of the new node. The resulting tree is $T_{j,p}$, and this completes the construction of Step 2.

In the following lemma, we argue that F_i is a forest and bound the diameter of trees in F_i.

Lemma 4.3. *For every level i the following statements are true:*

(i) F_i is a forest.

(ii) Let T be a tree in F_i, and $\mathsf{diam}(T)$ be the diameter of the set of points associated with leaves of T. Then $\mathsf{diam}(T) \leq (1/\epsilon + 20)2^i$ when $\epsilon \leq 1/12$.

Proof of Lemma 4.3. We prove the lemma by induction. The statement is vacuously true for F_0. We assume that the statement is true for $F_{i'}$, where $i' = i - \lceil\log(1/\epsilon)\rceil$, and prove it for F_i.

(i) Let x' be a point in $C_j \setminus \{x,y\}$. Recall that by construction $\delta_X(x,x') > 2^i/\epsilon$. It suffices to argue that: (a) T_x cannot contain x' and (b) no tree in F_x can be in $F_{x'}$. For (a) we use induction hypothesis and observe that for any other $z \in T_x$, we have $\delta_X(x,z) \leq \mathsf{diam}(T_x) \leq (1/\epsilon + 20)2^i\epsilon < 2^i/\epsilon$. For (b) induction hypothesis and observe that for any point z in F_x, we have $d_X(x,z) \leq 2^i + (1/\epsilon + 20)2^i\epsilon$. From triangle inequality, we have $d(x',z) \geq d(x,x') - d_X(x,z) > 2^i/\epsilon + 2^i + (1/\epsilon + 20)2^i\epsilon > 2^i$.

(ii) By construction, either (a) $T \in F_{i'}$, and in this case $\mathsf{diam}(T) \leq (1/\epsilon + 20)2^{i'} \leq (1/\epsilon + 20)2^i$ by induction, or (b) T is formed by merging trees in $\{T_x, T_y\} \cup F_x \cup F_y$ where x and y are

paired by C_j. Recall that $\delta_X(x,y) \le 2^i/\epsilon$, and that for every tree $A \in F_x \cup F_y$, there exists a leaf z such that $\delta_X(x,z) \le 2^i$ or $\delta_X(y,z) \le 2^i$. Thus, by triangle inequality and induction, it follows that:

$$\begin{aligned}\mathsf{diam}(T) &\le 2^i/\epsilon + \mathsf{diam}(T_x) + 2\cdot 2^i + 2\max_{A\in F_x}\mathsf{diam}(A)\\ &\quad + \mathsf{diam}(T_y) + 2\cdot 2^i + 2\max_{B\in F_y}\mathsf{diam}(B)\\ &\le 2^i/\epsilon + 6(1/\epsilon+20)2^{i'} + 4\cdot 2^i\\ &\le 2^i/\epsilon + 6(1/\epsilon+20)2^i\epsilon + 4\cdot 2^i\\ &\le (1/\epsilon + 10 + 120\epsilon)2^i \le (1/\epsilon+20)2^i\end{aligned}$$

when $\epsilon \le 1/12$, as desired.

□

We now show the robustness of the tree cover $\mathcal{T}$ assuming that $\epsilon \le 1/12$. We will show that the stretch is $1+O(\epsilon)$; one can achieve stretch $1+\epsilon$ by scaling ϵ.

Let $x \ne y$ be any two points in (X,δ_X). Let i be the non-negative integer such that:

$$2^{i-2}/\epsilon < \delta_X(x,y) \le 2^{i-1}/\epsilon\ . \tag{2}$$

Recall that we assume that the minimum distance in X is larger than $1/(4\epsilon)$ and hence i exists. Let p and q be two net points of N_i closest to x and y, respectively. By the triangle inequality and Equation (2), it holds that:

$$\begin{aligned}&\delta_X(p,q) \le \delta_X(x,y) + 2\cdot 2^i \le (1/2\epsilon+2)2^i \le 2^i/\epsilon \quad \text{since } \epsilon \le 1/12\\ &\delta_X(p,q) \ge \delta_X(x,y) - 2\cdot 2^i > (1/4\epsilon - 2)2^i > 0 \quad \text{since } \epsilon \le 1/12\end{aligned} \tag{3}$$

It follows from the second inequality in Equation (3) that $p \ne q$. Since $\delta_X(p,q) \le 2^i/\epsilon$, by property (2) of paring cover, there exists a set $C_j \in \mathcal{C}_i$ such that p and q are paired by C_j. Let T_p, T_q be the trees in $F_{i'}$ and $F_p, F_q \subseteq F_{i'}$ associated with p and q as described in the construction. Let T be the tree in F_i resulting from merging trees in $\{T_p,T_q\}\cup F_p \cup F_q$ by the algorithm. Since $\delta_X(x,p) \le 2^i$, x is a leaf of some tree $T_x \in \{T_p\}\cup F_p$. By the same argument, y is a leaf of some tree $T_y \in \{T_q\}\cup F_q$. Thus, both x and y are leaves in T.

Let P be the path from x to y in T. Let r, r_x, r_y be the roots of T, T_x, and T_y, respectively. Then P consists of two paths $T_x[x,r_x]$, $T_y[y,r_y]$ and two edges (r_x,r) and (r_y,r). Let Q be the path obtained from P by replacing each internal vertex v of P with a point chosen from a leaf in T_v. We denote by $S(v)$ the leaf point chosen to replace each vertex $v \in P$. Let Q_x (resp., Q_y) be the subpath of Q from x (resp., y) to $S(r_x)$ (resp., $S(r_y)$). We have:

$$w(Q) \le w(Q_x) + w(Q_y) + \delta_X(S(r_x),S(r)) + \delta_X(S(r),S(r_y)) \tag{4}$$

In the following claim, we bound the weight of each term in Equation (4).

Claim 4.1. $\max\{w(Q_x), w(Q_y)\} \le (2+40\epsilon)2^i$ and $\delta_X(S(r_x),S(r)) + \delta_X(S(r_y),S(r)) \le \delta_X(x,y) + 4(5+60\epsilon)2^i$.

Proof of Claim 4.1. Recall that T_x, T_y are trees in $F_{i'}$, where $i' = i - \lceil \log(1/\epsilon) \rceil$. By Lemma 4.3, we have:

$$\begin{aligned} w(Q_x) &= \sum_{j \le i'} (1/\epsilon + 20) 2^j \le (1/\epsilon + 20) 2^{i'+1} \le (1/\epsilon + 20) 2\epsilon 2^i \le (2 + 40\epsilon) 2^i \\ w(Q_y) &\le (2 + 40\epsilon) 2^i \qquad \text{(by the same argument)} \end{aligned} \tag{5}$$

The summation for computing Q_x in Equation (5) is due to the fact that for $T \in F_j$, we have $\mathsf{diam}(T) \le (1/\epsilon + 20) 2^j$. Let Y and Z be the sets of leaves of $\{T_p\} \cup F_p$ and $\{T_q\} \cup F_q$, respectively. Then we have:

$$\begin{aligned} \mathsf{diam}(Y) &= \mathsf{diam}(T_p) + 2 \cdot 2^i + 2 \max_{A \in F_p} \mathsf{diam}(A) \\ &\le 3(1/\epsilon + 20) 2^{i'} + 2 \cdot 2^i \le 3(1/\epsilon + 20) \epsilon 2^i + 2 \cdot 2^i = (5 + 60\epsilon) 2^i \\ \mathsf{diam}(Z) &\le (5 + 60\epsilon) 2^i \qquad \text{(by the same argument)} \end{aligned} \tag{6}$$

Observe that if $S(r) \in Y$, then $\delta_X(S(r), S(r_x)) \le \mathsf{diam}(Y) \le (5 + 60\epsilon) 2^i$ by Equation (6). Otherwise, $S(r) \in Z$ and hence $\delta_Y(S(r), S(r_x)) \le (5 + 60\epsilon) 2^i$. In either case, by the triangle inequality, we have:

$$\begin{aligned} &\delta_X(S(r_x), S(r)) + \delta_X(S(r_y), S(r)) \\ &\le \delta_X(S(r_x), S(r_y)) + 2 \min\{\delta_X(S(r_x), S(r)), \delta_X(S(r_y), S(r))\} \\ &\le \delta_X(S(r_x), S(r_y)) + 2(5 + 60\epsilon) 2^i \\ &\le \delta_X(x, y) + \mathsf{diam}(Y) + \mathsf{diam}(Z) + 2(5 + 60\epsilon) 2^i \qquad \text{(by the triangle inequality)} \\ &\le \delta_X(x, y) + 4(5 + 60\epsilon) 2^i \qquad \text{(by Equation (6))} \end{aligned} \tag{7}$$

□

By Equation (4) and Claim 4.1, we have that:

$$\begin{aligned} w(Q) &\le 2(2 + 40\epsilon) 2^i + \delta_X(x, y) + 4(5 + 60\epsilon) 2^i \\ &\le \delta_X(x, y) + O(1) 2^i \qquad \text{(since } \epsilon \le 1/12\text{)} \\ &\le \delta_X(x, y) + O(1) 4\epsilon \delta_X(x, y) \qquad \text{(by Equation (2))} \\ &= (1 + O(\epsilon)) \delta_X(x, y) \ , \end{aligned} \tag{8}$$

as claimed.

4.3 Implementing Robust Tree Cover in $O(n \log n)$ time

To make our construction of a robust tree cover efficient, we need two data structures:

(a) An implicit representation of a hierarchy of nets using $O_{\epsilon,d}(n)$ space. Let $\hat{N}_i \subseteq N_i$ be the subset of the net points that are explicitly stored in the hierarchy at level i. We have that $\sum_{i \ge 0} |\hat{N}_i| = O_{\epsilon,d}(n)$.

(b) For each net point $p \in \hat{N}_i$, store all the points in $N_i \cup N_{i'}$ within a distance $O(1/\epsilon)2^i$ from p, assuming that there is at least one such point other than p. (If there are no such points, then p will not be explicitedly store at level i in the hierarchy.) Recall that $i' = i - \lceil \log(1/\epsilon) \rceil$ and hence the pairwise distance of points stored at p in (b) is at least $2^{i-\log\lceil 1/\epsilon \rceil} = \Omega(\epsilon 2^i)$. It follows from the packing lemma (Lemma 4.1) that the number of stored points is $\epsilon^{-O(d)}$.

Both data structures (a) and (b) can be constructed in $O_{\epsilon,d}(n \log n)$ time using the result by Cole and Gottlieb [CG06].

Given these data structures, in Step 1, constructing partition $\mathcal{P}_i$ can be done in $O_{\epsilon,d}(|\hat{N}_i|)$ time following exactly the algorithm where $\hat{N}_i \subseteq N_i$ is the set of points stored explicitly in the hierarchy of nets. Thus, constructing $C(\mathcal{P}_i)$ can also be done in $O_{\epsilon,d}(|\hat{N}_i|)$ time. Note that $\sum_{i \geq 0} \hat{N}_i = O_{\epsilon,d}(n)$ by (a). It follows that the total running time of Step 1 is $O_{\epsilon,d}(n)$. For Step 2, to construct the tree $T_{j,p}$, we need to identify for each point $x \in C_j$ the tree T_x and forest F_x. For T_x, we can store a pointer to T_x at x. For quickly identifying F_x, we (i) relax the definition of F_x to contains subtrees of $F_{i'}$ such that each contains a *net point* of $N_{i'}$ within distance $O(2^i)$ from x, (ii) guarantee that each tree in $F_{i'}$ contains at least one point in $N_{i'}$. Thus, we can identify F_x in $O(\epsilon^{-d}) = O_{\epsilon,d}(1)$ time by looking at all points stored at x in data structure (b). To guarantee (ii) inductively, in Step 2, we not only merge trees from pairs x, y in C_j, but also merge trees in $F_{i'}$ that contain points close to points in N_i. Specifically, for each point $z \in \hat{N}_i$, which might not be in C_j, we merge the tree containing z and all the (unmerged) trees in $F_{i'}$ containing points of $N_{i'}$ within distance 2^i from z. This can be done in $O_{\epsilon,d}(|\hat{N}_i|)$ time. Thus, the total running time of both steps is $O_{\epsilon,d}(n)$, and the final running time is $O_{\epsilon,d}(n \log(n))$.

4.4 Deriving a fault-tolerant navigation (and routing) scheme

In the navigation scheme presented in Section 3, we did not exploit a crucial property of the tree cover theorem in doubling metrics [BFN19]: For every pair u, v of points in M_X, there is a $(1+\epsilon)$-spanner path in one of the trees in the cover — such that the path starts and ends at leaves corresponding to u and v. To achieve FT navigation algorithm, we must rely on this property. For any two points from a doubling metric, the navigation algorithm from Section 3 locates points, which are now the leaves in the corresponding tree of the tree cover. Then, it uses the navigation scheme for that particular tree to navigate between these points. Every vertex in the tree is associated with a single point in the metric space, hence while navigating the tree we can directly obtain the information about the path in the metric space. In the case of FT navigation, every vertex in the tree stores (or is associated with) $f+1$ points (rather than one) that correspond to its descendant leaves. This is the case for all the vertices, except for ones with less than $f+1$ descendant leaves (including the leaves themselves); such vertices store all their descendant leaves. To navigate between any two non-faulty points u and v (corresponding to leaves in the tree), we apply the same navigation scheme as given in Section 3, but for every vertex that we traverse along the path in the tree, we pick a non-faulty point stored in that vertex *arbitrarily*, if it stores $f+1$ points. For every vertex with less than $f+1$ leaves in its subtree, it must store either u or v, since all the nodes along the path in the tree are ancestors of either u or v. Since both u and v are non-faulty, we will have a non-faulty point to choose from (u or v or both). The query time of the navigation scheme remains $O(k)$. The basic (non-FT) routing scheme is deferred to Section 5.1; however, equipped with the FT-navigation scheme that we've just described, it is straightforward to strengthen the basic routing scheme to achieve fault-tolerance (with the size bounds growing by a factor of f).

5 Applications

We argue that an efficient navigation scheme is of broad potential applicability, by providing several applications. The first and perhaps the most important application is an efficient compact routing scheme, which is given in Section 5.1. Next, in Section 5.3 we show that our navigation technique can be used for efficient spanner sparsification without increasing its stretch and lightness by much. In Sections 5.4 and 5.5 we show that one can use the navigation technique to compute the SPT and MST on top of the underlying spanner. Finally, in Section 5.6, we address two related problems: online tree product and MST verification.

5.1 Compact routing schemes

A routing scheme is a distributed algorithm for delivering packets of information from any given source node to any specified destination in a given network. Every node has a *routing table*, which stores local routing-related information. In addition, every node is assigned a unique *label* (sometimes called *address*). Given a destination node v, routing algorithm is initiated at source u and is given the label of v. Based on the local routing table of u and the label of v, it has to decide on the next node w to which the packet should be transmitted. More specifically, it has to output the relevant *port number* leading to its neighbor w. Each packet of information has a *header* attached to it, which contains the label of the destination node v, but may contain additional information that may assist the routing algorithm. Upon receiving the packet, any intermediate node w has at its disposal its local routing table and the information stored in the header. This process continues until the packet arrives at its destination (node v).

We consider routing in metric spaces, where each among n points in the metric corresponds to a network node. Initially, we choose a set of links that induces an overlay network over which the routing must be performed. We would like the overlay network size to be small and yet to be able to route using very few hops. The challenge is to do so while also optimizing the tradeoff between the storage (i.e., size of routing tables, labels, and headers) and the stretch (i.e., the ratio between the distance packet traveled in the network and the distance in the original metric space). In addition, one may try to further optimize the time it takes for every node to determine (or output) the next port number along the path, henceforth *decision time*, and other quality measures, such as the maximum degree in the overlay network.

There are two models, based on the way labels are chosen: *labeled*, where the designer is allowed to choose (typically polylog(n)) labels, and *name-independent*, where an adversary chooses labels. Depending on the way the port numbers are assigned, we distinguish between the *designer-port* model, where the designer can choose the port number, and the *fixed-port* model, where the port numbers are chosen by an adversary. For an additional background on compact routing schemes, we refer the reader to [Pel00, TZ01b, FG01, AGGM06, Che13].

Our basic result is an efficient routing scheme for tree metrics, which we present in Section 5.1.1. Our routing scheme works in the labeled, fixed-port model. Next, in Section 5.1.2, we apply the routing scheme for tree metrics on top of the collection of trees provided by any of the aforementioned tree cover theorems (cf. Table 1) and obtain routing schemes for various metrics. This application is nontrivial for general graphs, as it aims at optimizing the label sizes.

5.1.1 Routing scheme for tree metrics

We show that one can construct an efficient 2-hop routing scheme for tree metrics. The guarantees are summarized in the following theorem.

Theorem 5.1. *Let M_T be a tree metric represented by an edge-weighted tree T with n vertices. We can preprocess T it in $O(n \log n)$ time and construct a routing scheme which works on the overlay network with $O(n \log n)$ edges. The routing scheme works in the labeled, fixed-port model, uses routing tables and labels of $O(\log^2 n)$ bits and headers of $\lceil \log n \rceil$ bits and routes in at most 2 hops and in $O(1)$ decision time.*

The first step of the preprocessing phase consists of constructing spanner G_T of T and the navigation data structure $\mathcal{D}_T$ as described in Theorem 1.1. The number of edges of G_T is $O(n\alpha_k(n)) = O(n \log n)$ for $k = 2$. In addition to G_T, we obtain an augmented recursion tree Φ for navigating G_T. By Observation 3.1, the depth of Φ is $O(\log n)$. (More details on this construction can be found in Section 3.1.) We will show how to construct a routing scheme for spanner G_T using the augmented recursion tree Φ. Recall that, when $k = 2$, every non-leaf vertex in Φ corresponds to exactly one vertex in T.

We shall assume that each vertex in T is assigned a unique identifier between 1 and n, which can be justified via a straightforward linear time procedure. At the beginning, we preprocess the augmented recursion tree Φ using the lowest common ancestor (LCA) labeling scheme by [AHL14].[6] This scheme uses linear preprocessing time to assign $O(\log n)$-bit label to each vertex in the tree so that any subsequent LCA query can be answered in constant time. Recall from Section 3.1 that every vertex u in T uniquely corresponds to a vertex in Φ, denoted by $u.ptr(\Phi)$. For every u in T, let $lca(u)$ be the LCA label of $u.ptr(\Phi)$.

We start by describing the label of each vertex in T. Fix a vertex $u \in T$ and denote by α its corresponding vertex in Φ, i.e., $\alpha \coloneqq u.ptr(\Phi)$. Recall from Section 3.1 that we use $\alpha.level$ to denote the level of α in Φ. Let β_j be the ancestor of α at level j in Φ, so that $\beta_0 = rt(\Phi)$ and $\beta_{\alpha.level} = \alpha$. Let $v_j \coloneqq \beta_j.ptr(T)$ be the vertex in T which corresponds to β_j for all $0 \le j \le \alpha.level - 1$. Denote by h_u a 2-level hash table with key $lca(v_j)$ and value $\text{port}(v_j, u)$ for each $0 \le j \le \alpha.level - 1$. This table occupies space linear in the size of all its key-value pairs. Since there is at most $O(\log n)$ ancestors (the depth of Φ is $O(\log n)$ by Observation 3.1) and each ancestor takes $O(\log n)$ bits of space to store, the total memory required for h_u is $O(\log^2 n)$. Any subsequent query for an element in h_u takes worst-case constant time. Details of implementation can be found in [FKS84, AN96]. Label of u, denoted by $\text{label}(u)$, consists of $\text{label}(u) = (lca(u), h_u)$. Its size is $O(\log^2(n))$ bits.

The routing table of node u, denoted by $\text{table}(u)$ contains information similar to its label. Denote by h'_u a 2-level hash table with key $lca(v_j)$ and value $\text{port}(u, v_j)$ for each $0 \le j \le \alpha.level - 1$. Note that in h_u, we store port numbers leading to u from its ancestors (with respect to Φ), whereas in h'_u we store port number from u to its ancestors. In addition, if u corresponds to a leaf in the augmented recursion tree, we add to $\text{table}(u)$ an array $\text{base}(u)$ containing a constant number of pairs $(v, \text{port}(u, v))$ for every other node v corresponding to the same leaf. Since u might not be directly connected to v, $\text{port}(u, v)$ denotes the port number leading to the first vertex on the shortest path from u to v. Recall that, by Property 6, there can be at most $O(k) = O(1)$ vertices corresponding to the same base case, meaning that $\text{base}(u)$ contains a constant number of $O(\log n)$-bit entries. The routing table of u consists of $\text{table}(u) = (lca(u), h'_u, \text{base}(u))$, which takes $O(\log^2 n)$ bits to store.

[6] This scheme is usually known as *nearest common ancestor (NCA)* scheme, but we use term lowest common ancestor (LCA) in order to be consistent with the rest of the paper.

We note that labels and routing tables can be computed in $O(n \log n)$ time using the augmented recursion tree and techniques similar to those in Section 3.

We now specify the routing protocol. Recall that, upon a query to route from u to v, the algorithm is executed on node u, has access to the local routing table of u (denoted by table(u)), and is passed the label of the destination v (denoted by label(v)). We shall assume that $u \neq v$, since otherwise there is nothing to do. First, we check if u and v correspond to the same base case in Φ. We do so by looking for v in base(u), if base(u) is nonempty. This can be done in constant time since the number of entries in base(u) is constant. If an entry corresponding to v has been found, we extract from base(u) the information about the port corresponding to the first edge on the shortest path from u to v and forward the packed with an empty header. By the guarantees of G_T, there exist a path of at most 2 hops between u and v in the subgraph induced by the vertices corresponding to the same base case as u and v. Hence, the packet is either forwarded directly to v, in which case the algorithm successfully terminates, or it is forwarded to an intermediate node w which corresponds to the same base case and has a direct link to v. In the latter case, the corresponding port can be extracted from base(w), using the local routing table at node w.

If u and v do not correspond to the same base case, we look for the lowest common ancestor, λ, of $u.ptr(\Phi)$ and $v.ptr(\Phi)$. Let $lca(\lambda)$ denote the LCA label of this vertex. At this stage, we distinguish between several cases. If $u.ptr(\Phi) = \lambda$ (which can be checked using their LCA labels), then $u.ptr(\Phi)$ is an ancestor of $v.ptr(\Phi)$ in Φ and the underlying spanner G_T contains an edge between u and v. The corresponding port can be found in h_v, which is in label(v). If $v.ptr(\Phi) = \lambda$, then λ is an ancestor of $u.ptr(\Phi)$ and there is an edge between u and v. The corresponding port can be found in h'_u, which is in table(u). If none of the above is the case (recall that we assumed that $u \neq v$), by the design of G_T, there is edge between u and the vertex w corresponding to λ and from w to v. From label(v), we extract port(w, v), store it in the header and forward the packet to port(u, w), which can be found in table(u). This completes the description of the routing algorithm.

5.1.2 Routing in metric spaces

We proceed to show how to employ the described routing scheme for metric spaces, thus proving Theorem 1.3.

Similarly to what has been done in Section 3, for a given metric $M_X = (X, \delta_X)$ we first construct one of the tree covers from Table 1. Denote the stretch of this cover by γ and the number of the trees by ζ. The underlying graph H is the union of the trees in the cover — it has the same vertex set as M_X, denoted by X, and has an edge set obtained as a union of the edges of ζ trees in the cover. At this stage, port numbers are assigned (by an adversary) for every vertex v in $\{1, \ldots, \deg_H(v)\}$, where $\deg_H(v)$ is the degree of v in H. Then, for each tree in the cover, we construct a 2-hop routing scheme as provided by Theorem 5.1. We distinguish between cases where we use tree covers and Ramsey tree covers.

Routing using tree covers. In addition to the routing schemes, we employ the *distance labeling scheme* with a stretch of $(1 + \epsilon)$, given by [FGNW17]. Their labeling scheme uses $O(\log(1/\epsilon) \log n)$ bits of space per vertex and achieves a constant query time. In our routing scheme, each node stores ζ distance labels, one per tree in the cover, both as a part of its routing table and as a part of its label. In addition, for each of the trees in the cover, we store label and routing table, as described in Section 5.1.1. This means that the memory consumed by the routing table at each node is $O(\zeta \cdot \log^2 n + \zeta \cdot \log(1/\epsilon) \log n) = O(\zeta \cdot \log n \cdot \log(n/\epsilon))$.

The routing algorithm executed at node u first queries the distance labeling scheme for the approximate distance between u and the destination node v in each of the trees. Since each query takes constant time, this step requires time proportional to ζ. The routing proceeds using the routing information of the tree which has the smallest stretch. Suppose that such a tree had index i. We find the next port using the ith entry of the routing table of u and the label of v. If the next step in tree routing correspond to the base case, we only transfer the index i in the header. From the algorithm in Section 5.1.1, we know that the next step is either v, or we route via an intermediate node w which has port number $\text{port}(w, v)$ in its routing table corresponding to the ith tree. If the next step does not correspond to the base case, we either route directly, in which case there is no need to store anything in the header, or route via an intermediate node w. In the latter case, we can extract the information on $\text{port}(u, w)$ and $\text{port}(w, v)$ from the entries in $\text{table}(u)$ and $\text{label}(v)$ (corresponding to the ith tree) and route to $\text{port}(u, w)$, while sending only $\text{port}(w, v)$ in the header. In both cases, the header size is $\lceil \log n \rceil$. Finally, notice that we are using the labeling schemes which return $(1+\epsilon)$-approximate distances, meaning that this step incurs an additional $(1+\epsilon)$ factor to the stretch of our routing path. The stretch of $(1+\epsilon)$ can be achieved by appropriate scaling.

Routing using Ramsey tree covers. When using Ramsey tree covers, we know for each node which of the trees in the cover achieves the desired stretch. The label of a node is now comprised of its label in the routing table for that particular tree, together with the index of that tree. Routing table of every node contains routing tables for each of the ζ trees in the cover. Hence, the label sizes of this scheme are $O(\log^2 n)$, which is the size of routing schemes for tree spanners in Theorem 5.1, and the routing tables have size $O(\zeta \cdot \log^2 n)$. Given a source node u and the label of destination v, the algorithm uses routing table corresponding to the tree which index is in the label of v. In other words, we can in constant time decide which among the ζ routing tables stored at u to use.

In conclusion, we have proved Theorem 1.3, whose guarantees are summarized in the Table 3.

stretch	**storage**		**time**	**metric**
	table	**label**		
1	$O(\log^2 n)$		$O(1)$	tree
$1+\epsilon$	$O(\epsilon^{-O(d)} \log(n) \log(n/\epsilon))$		$O(\epsilon^{-O(d)})$	doubling dim. d
$1+\epsilon$	$O((\log n/\epsilon)^3 \log(n))$		$O((\log n/\epsilon)^2)$	fixed-minor-free
$O(\ell)$	$O(\log^2 n)$	$O(\ell n^{1/\ell} \log^2 n)$	$O(1)$	general
$O(n^{1/\ell} \log^{1-1/\ell} n)$	$O(\log^2 n)$	$O(\ell \cdot \log^2 n)$	$O(1)$	general

Table 3: Summary of our results for 2-hop routing schemes. For each result, header size is $\lceil \log n \rceil$. In the last two results, parameter $\ell \geq 1$ is an arbitrary integer.

5.2 Fault-tolerant routing

Fault-tolerant routing in trees. We describe a fault-tolerant routing scheme for trees of the robust tree cover presented in Section 4. Given a tree T from the robust tree cover, we first compute a spanner for it, as described in Section 4.1. Recall that for every $v \in T$, we choose a set $R(v)$ corresponding to (at most) $f+1$ leaves of the subtree of T rooted at v. We show how to route between any two leaves of this tree, by slightly modifying the tree routing scheme presented in Section 5.1.1.

For every leaf in $u \in T$ (corresponding to some point in the metric), let α be its corresponding vertex in Φ, i.e., $\alpha = u.ptr(\Phi)$. We construct a 2-level hash table h_u for u as follows. For the jth ancestor of α in Φ, let v_j be its corresponding point in T and let $w_1, \ldots, w_\ell$ be the leaves in T from $R(v_j)$, ordered increasingly by their identifiers. We add to the hash table h_u an entry with key $lca(v_j)$ and value $\langle \text{port}(w_1, u), \ldots \text{port}(w_\ell, u)\rangle$. The label of u is $\text{label}(u) = (lca(u), h_u)$; it requires $O(f \log^2 n)$ bits of space.

We next describe the routing table of u, denoted by $\text{table}(u)$. Let h'_u be a 2-level hash table, where for the jth ancestor of α, we store key $lca(v_j)$ and value $\langle \text{port}(u, w_1), \ldots \text{port}(u, w_\ell)\rangle$. If u corresponds to a base case of the spanner construction, we keep a routing table $\text{base}(u)$ for every other node v corresponding to the same leaf in Φ. In particular, let $w_1, \ldots, w_\ell$ be the leaves in T in $R(v)$. We add an entry $(v, \langle \text{port}(u, w_1), \ldots, \text{port}(u, w_\ell)\rangle)$ to $\text{base}(u)$. The routing table of u, denoted by $\text{table}(u)$ consists of $(lca(u), h'_u, \text{base}(u))$; it requires $O(f \log^2 n)$ bits of space.

Given two leaves u and v in T, the routing algorithm is similar to the one from Section 5.1.1. The only difference is that when we route via an intermediate vertex $w \in T$, we scan entries corresponding to $R(w)$ to find a non-faulty vertex. Since the entries in $R(w)$ are ordered increasingly by their identifiers, we can find a port to a non-faulty vertex in $R(w)$ in $O(f)$ steps. This concludes the description of the fault-tolerant routing scheme for trees.

Fault-tolerant routing in metric spaces. Suppose we are given a metric $M_X = (X, \delta_X)$ with doubling dimension d. First, we construct a fault-tolerant spanner for M_X, as described in Section 4.1, and assign port numbers according to spanner edges. The rest of the construction remains the same as for the routing using tree covers described in Section 5.1.2. For each of the $\zeta = O(\epsilon^{-O(d)})$ trees in the cover, we keep labels and routing tables of size $O(f \log^2 n)$. In addition, for each tree, we construct a distance labeling scheme given by [FGNW17]. Labels for this scheme occupy $O(\log(1/\epsilon) \log n)$ bits of space. Hence, the total memory per vertex in the tree is $O(\zeta \cdot \log n \cdot (\log(1/\epsilon) + f \log n)) = O(\epsilon^{-O(d)} \cdot \log n \cdot (\log(1/\epsilon) + f \log n))$. The routing algorithm remains the same. This concludes the description of the fault-tolerant routing; the result is summarized in the following theorem.

Theorem 5.2. *For any n-point metric $M_X = (X, \delta_X)$ with doubling dimension d, one can construct a $(1+\epsilon)$-stretch 2-hop routing scheme in the labeled, fixed-port model with headers of $\lceil \log n \rceil$ bits, labels and local routing tables of $O(\epsilon^{-O(d)} \cdot \log n \cdot (\log(1/\epsilon) + f \log n))$ bits, and local decision time $O(f)$.*

5.3 Spanner sparsification

Let $M_X = (X, \delta_X)$ be an arbitrary n-point metric space and let G_X be any *light* m-edge spanner for M_X. Let $k \geq 2$ be an integer and let H_k be a k-hop spanner for M_X and $\mathcal{D}_x$ data structure for H_k provided by Theorem 1.2. Our goal is to transform G into a sparse spanner for M_X, without increasing the stretch and weight by much.

The transformation is as follows. For each edge (u, v) in $E(G_X)$, we query the data structure $\mathcal{D}_X$ for a k-hop path $P_{u,v}$ between u and v in H_k; the output is $G'_X := (X, \cup_{(u,v) \in E(G_X)} P_{u,v})$. The following theorem summarizes the guarantees of this sparsification procedure. See also Table 4.

Theorem 5.3. *Let G_X be a k-hop spanner for an n-point metric space $M_X = (X, \delta_X)$, with size m, stretch β, and lightness L; let H_k be a spanner for M_X equipped with a data structure $\mathcal{D}_X$ as in*

Theorem 1.2. Then, one can in $O(m \cdot \tau)$ time transform G_X into spanner G'_X with stretch $\gamma \cdot \beta$, lightness $\gamma \cdot L$ and $O(n\alpha_k(n) \cdot \zeta)$ edges.

Proof. For each edge of G_X we perform a query as in Theorem 1.2, and each query takes $O(\tau)$ time, so the total running time of the transformation is $O(|E(G_X)| \cdot \tau)$. The stretch of a spanner is equal to the maximum stretch between any two adjacent points in it; hence, $stretch_{G'}(u,v) \leq \gamma \cdot stretch_G(u,v) \leq \gamma \cdot \beta$, where the first inequality follows since each edge (u,v) in G_X got replaced by a path of total weight $\gamma \cdot w(u,v)$, and the second inequality follows since the stretch of G is β. Similarly, since each edge (u,v) of G_X got replaced by a path $P_{u,v}$ of weight at most $\gamma \cdot w(u,v)$, the lightness of the resulting spanner is at most $\gamma \cdot L$. The resulting spanner is a subgraph of H_k so it has at most $O(n\alpha_k(n) \cdot \zeta)$ edges. □

Stretch	**Size**	**Lightness**	**Time**	**Metric family**
$(1+\epsilon)\beta$	$O(n\alpha_k(n)/\epsilon^{O(d)})$	$(1+\epsilon)L$	$O(mk/\epsilon^{O(d)})$	doubling dim. d
$(1+\epsilon)\beta$	$O(n\alpha_k(n)\epsilon^{-2}\log^2 n)$	$(1+\epsilon)L$	$O(mk\epsilon^{-2}\log^2 n)$	fixed-minor-free
$O(\ell \cdot \beta)$	$O(n\alpha_k(n)\ell n^{1/\ell})$	$O(\ell \cdot L)$	$O(m \cdot k)$	general
$O(\beta n^{1/\ell}\log^{1-1/\ell} n)$	$O(n\alpha_k(n) \cdot \ell)$	$O(Ln^{1/\ell}\log^{1-1/\ell} n)$	$O(m \cdot k)$	general

Table 4: Summary of our result for sparsification of spanner G_X for an n-point metric space. We use β to denote the stretch of G_X, which m edges and lightness L. In the last two entries, $\ell \geq 1$ denotes an arbitrary integer.

Remark 5.1. For fixed ϵ and doubling dimension d, Theorem 5.3 gives rise to a transformation that works in $O(mk)$ time and produces a spanner of size $O(n\alpha_k(n))$ with stretch and lightness increased by a factor of $(1+\epsilon)$. Similarly, for fixed ϵ and fixed-minor-free metrics, we obtain a transformation that works in $O(mk\log^2 n)$ time and produces a spanner of size $O(n\alpha_k(n)\log^2(n))$ with stretch and lightness increased by a factor of $(1+\epsilon)$.

5.4 Approximate shortest path trees

Once a spanner has been constructed, it usually serves as a "proxy" overlay network, on which any subsequent computation can proceed, in order to obtain savings in various measures of space and running time. Thus, we shall focus on devising efficient algorithms that run on the spanner itself. In some applications, we may not have direct access to the entire spanner, but may rather have implicit and/or local access, for example by means of a data structure for approximate shortest paths within the spanner, such as the one provided by Theorem 1.2.

In this section, we explain how our navigation technique can be used for efficiently computing an approximate shortest-path tree (SPT) that is a subgraph of the underlying spanner. In any metric, its SPT is simply a star, which is most likely not a subgraph of the underlying spanner. Assuming that we have direct, explicit access to the spanner, we can simply compute an SPT on top of it using Dijkstra's algorithm, which will provide an approximate SPT for the original metric. For an n-vertex spanner, this approach will require $\Omega(n \log n)$ time, even if the spanner size is $o(n \log n)$. There is also an SPT algorithm that runs in time linear in the spanner size, but it is more complex and also assumes that $\log n$-bit integers can be multiplied in constant time [Tho99]. Using our navigation scheme, as provided by Theorem 1.2, we can do both better and simpler, and we don't even need explicit access to the underlying spanner (though we do need, of course, access to the navigation

scheme). The data structure provided by Theorem 1.2 allows us to construct, within time $O(n\tau)$, an approximate SPT. In particular, for low-dimensional Euclidean and doubling metrics, we can construct a $(1+\epsilon)$-approximate SPT (for a fixed ϵ) that is a subgraph of the underlying spanner within $O(nk)$ time, where $k = 2, 3, \ldots, O(\alpha(n))$.

In what follows, we will assume that we are given a metric (X, δ_X), with $|X| \coloneqq n$ and that we have constructed spanner H_k and the data structure $\mathcal{D}_X$ for (X, δ_X) as in Theorem 1.2. Recall that we use γ to denote the stretch of the path returned by $\mathcal{D}_X$ and τ denotes the time spent per query. The algorithm for computing an approximate SPT rooted at given vertex rt is stated in procedure ApproximateSPT(rt).

Algorithm 3 Computing approximate shortest path trees.

```
1:  procedure ApproximateSPT(rt)
2:      for v ∈ V do let π(v) ← ∅
3:      for v ∈ V \ {rt} do let dist(v) ← ∞
4:      Let dist(rt) ← 0
5:      Let V(T) ← {rt} and E(T) ← ∅
6:      for v ∈ V \ {rt} do
7:          Query for the k-hop γ-approximate shortest path P_{rt,v} from rt to v;
8:          For each edge e = (x, y), ordered from rt to v along P_{rt,v}, invoke Relax(x, y)
9:  procedure Relax(u, v)
10:     if dist(u) = ∞ then
11:         V(T) ← V(T) ∪ {v}
12:         E(T) ← E(T) ∪ {(u, v)}
13:         dist(v) ← dist(u) + w(u, v)
14:         π(v) ← u
15:     else if dist(u) + w(u, v) < dst(v) then
16:         E(T) ← (E(T) \ {(π(v), v)}) ∪ {(u, v)}
17:         dist(v) ← dist(u) + w(u, v)
18:         π(v) ← u
```

We shall prove three claims which will imply the guarantees of the algorithm in Algorithm 3. Intuitively, Claim 5.1 proves that, throughout the execution, the graph T maintained by the algorithm is a tree. Then, Claim 5.2 shows that the value dist for each vertex $v \in T$ will be an upper bound on its distance from the root, denoted by $\delta_T(rt, v)$. Finally, Claim 5.3 implies that upon termination, all the vertices have a γ-stretch path to the root.

Claim 5.1. Throughout the execution of ApproximateSPT(rt), graph T is a tree.

Proof. Initially, the claim holds vacuously, since T consists of vertex rt and no edges. The structure of T changes in lines 14 and 18. It is important to notice that the order of relaxations performed at line 8 allows us to assume that whenever Relax(u, v) is executed, u is in T. We will assume T is a tree before the relaxation and prove that it will also be the case after the relaxation. If v was in T earlier, then T changes a parent of v, thus remaining connected while preserving the number of vertices and edges. If v was not in T, then T grows by one vertex and one edge. In both cases, T is a connected graph with $|V(T)|$ vertices and $|V(T)| - 1$ edges, so it is a tree. □

Claim 5.2. For each vertex $v \in T$ and for each ancestor u of v, the algorithm ApproximateSPT(rt) maintains invariant $\mathsf{dist}(u) + \delta_T(u, v) \leq \mathsf{dist}(v)$.

Proof. At the beginning of the execution, the invariant is trivially satisfied. The values dist change in lines 13 and 17. Similarly, the values δ_T change in lines 14 and 18, due to the change of edges in T. We assume that the invariant was true before relaxation along (u, v) and would like to show that it holds after. In particular, it suffices to show that (i) it holds for v and all its ancestors, (ii) it holds for v and all its descendants, and (iii) it holds for any ancestor x of v and any descendant w of v. As for (i), the invariant trivially holds for v as its own ancestor. Also, after the relaxation, we know that $\mathsf{dist}(v) = \mathsf{dist}(u) + w(u, v) = \mathsf{dist}(u) + \delta_T(u, v)$ so the claim holds for u as a parent of v. The invariant was true before the relaxation, so for any ancestor x of u (and thus ancestor of v), $\mathsf{dist}(x) + \delta_T(x, u) \leq \mathsf{dist}(u)$. Adding $\delta_T(u, v)$ to both sides of inequality completes this case. For (ii), the invariant implies that $\mathsf{dist}'(v) + \delta_T(v, w) \leq \mathsf{dist}(w)$, where $\mathsf{dist}'(v)$ was the value before the relaxation. After the relaxation, $\mathsf{dist}(v)$ decreased, so the invariant remains true. Finally, (iii) holds since $\mathsf{dist}(x) + \delta_T(x, w) = \mathsf{dist}(x) + \delta_T(x, v) + \delta_T(v, w) \leq \mathsf{dist}(v) + \delta_T(v, w) \leq \mathsf{dist}(w)$. □

Claim 5.3. Upon termination of ApproximateSPT(rt), for any $v \in V$, $\delta_T(rt, v) \leq \gamma \cdot \delta_X(rt, v)$.

Proof. We know that after the iteration of line 6 for vertex v, $\mathsf{dist}(v)$ is within γ factor of $\delta_X(rt, v)$. In the subsequent iterations, $\mathsf{dist}(v)$ might only decrease due to relaxations. Finally, by Claim 5.2 we know that $\delta_T(rt, v) \leq \mathsf{dist}(v)$ throughout the execution of the algorithm. □

Claims 5.1 and 5.3 imply that T is a γ-SPT for given metric with root at rt. The running time of the algorithm is dominated by $n - 1$ queries of the path oracle, each of which takes $O(\tau)$ time as in Theorem 1.2. Thus, we have proved the following theorem.

Theorem 5.4. *Given a data structure $\mathcal{D}_X$ for an n-point metric space $M_X = (X, \delta_X)$ as in Theorem 1.2, one can construct a γ-approximate shortest path tree for M_X rooted at any point $rt \in X$ in time $O(n\tau)$.*

Remark 5.2. When M_X is a metric of doubling dimension d, Theorem 5.4 gives rise to a construction of $(1 + \epsilon)$-approximate SPTs in time $O(nk/\epsilon^d)$ time. If M_X is a fixed-minor-free metric, the result is a $(1 + \epsilon)$-SPT in time $O(nk\epsilon^{-2} \log^2 n)$. For general metrics and an integer parameter $\ell \geq 0$, one can construct an $O(\ell)$-SPT in time $O(nk)$.

5.5 Approximate Euclidean minimum spanning trees

Suppose that we would like to construct an approximate minimum spanning tree (MST). Here too, we shall focus on finding an approximate MST that is a subgraph of the underlying spanner. In low-dimensional Euclidean spaces one can compute a $(1 + \epsilon)$-approximate MST (for a fixed ϵ) in $O(n)$ time [Cha08], but again this approximate MST may not be a subgraph of the spanner. Running an MST algorithm on top of the spanner would require time that is at least linear in the spanner size; moreover, the state-of-the-art deterministic algorithm runs in super-linear time and is rather complex [Cha00], and the state-of-the-art linear time algorithms either rely on randomization [KKT95] or on some assumptions, such as the one given by transdichotomous model [FW94]. Instead, using our navigation scheme, as provided by Theorem 1.2, we can construct an approximate MST easily, within time $O(n\tau)$, where τ is the query time as in Theorem 1.2. In particular, for low-dimensional

Euclidean spaces, we can construct in this way a $(1+\epsilon)$-approximate MST (for a fixed ϵ) that is a subgraph of the underlying spanner within $O(nk)$ time, where $k = 2, 3, \ldots, O(\alpha(n))$.

In what follows, we will assume that we are given a Euclidean metric (X, δ_X), with $|X| := n$ and that we have constructed spanner H_k and the data structure $\mathcal{D}_X$ for (X, δ_X) as in Theorem 1.2. The MST construction is as follows. Initially, compute a $(1 + O(\epsilon))$ minimum spanning tree T on (X, δ_X) using [Cha08]. For each edge $(u, v) \in T$, query $\mathcal{D}_X$ for a k-hop path $P_{u,v}$ of weight at most $(1+\epsilon)w(u, v)$. Let $H := (X, \cup_{(u,v)\in E(T)} P_{u,v})$ be the union of the obtained paths. Return a spanning tree of H.

Theorem 5.5. *Given an n-point Euclidean space $M_X = (X, \delta_X)$ and its $(1+\epsilon)$-spanner H_k equipped with a data structure $\mathcal{D}_X$ as in Theorem 1.2, one can compute a $(1 + \epsilon)$-approximate minimum spanning tree for (X, δ_X) in $O(n \cdot k/\epsilon^d)$ time.*

Proof. Computing a $(1+O(\epsilon))$-approximate minimum spanning tree using Chan's algorithm [Cha08] takes $O(n/\epsilon^d)$ time. For each edge $(u, v) \in T$ we query $\mathcal{D}_X$ for path $P_{u,v}$ in H_k in time $O(k/\epsilon^d)$ as in Theorem 1.2; altogether, it takes $O(n \cdot k/\epsilon^d)$ time to compute H. Since the edges graph H consist of union of k-hop paths $P_{u,v}$, the size of $E(H)$ is $O(n \cdot k)$. Every path $P_{u,v}$ is within a factor $(1+\epsilon)$ of $w(u, v)$, so the total weight of $E(H)$ is within a $(1 + \epsilon)$ of the $w(T)$. Computing a spanning tree of H can be done in $O(|E(H)|) = O(n \cdot k)$ time using BFS. □

Remark 5.3. When ϵ is constant, the running time in Theorem 5.5 becomes $O(n \cdot k)$.

5.6 Online tree product and MST verification

The online tree product problem [Tar79, Cha84, AS87, Pet06] is defined as follows.[7] Let T be an n-vertex tree with each of its edges being associated with an element of a semigroup $(S, \circ)$. One needs to answer online queries of the following form: Given a pair of vertices $u, v \in T$, find the product of the elements associated with the edges along the path from u to v. A slight variant of this problem is the online MST verification problem where the edge weights of T are real numbers. One needs to answer online queries of the following form: Given a weighted edge (u, v) not in $E(T)$, report if the weight of (u, v) is larger than each edge weight along the path between u and v in T. In both problems, the goal is to design efficient (in terms of time and space) preprocessing and query algorithms which use as few as possible semigroup operations (or binary comparisons). In Section 5.6.1, we show that our navigation algorithm from Theorem 1.1 can be easily modified to support the online tree product problem. Then, we further optimize it for the online MST verification problem in Section 5.6.2.

5.6.1 Online tree product

Given a tree T and its 1-spanner G_T, the algorithm from Theorem 1.1 builds a data structure $\mathcal{D}_T$ such that, for any two vertices $u, v \in V(T)$, it returns a 1-spanner path of at most k hops in G_T in $O(k)$ time. We proceed to show how to preprocess the spanner G_T and data structure $\mathcal{D}_T$ so that each spanner edge $(u, v) \in E(G_T)$ has assigned to it a value from semigroup $(S, \circ)$. This value corresponds to the product of the edge values along the path from u to v in T. Using this information assigned to the edges of G_T, we can answer online tree product queries between two

[7] This problem is sometimes called the *online tree sum problem*.

vertices u and v in $V(T)$ by querying $\mathcal{D}_T$ for a k-hop path in G_T. This results in online tree product algorithm which uses $k-1$ semigroup operations and $O(k)$ time per query.

The algorithm we proceed to explain uses the notions introduced in Section 3.

We start by describing the preprocessing algorithm for the spanner construction when $k = 2$. In this case, all the spanner edges are either added from from a cut vertex u to every other required vertex in tree T considered in the current level of recursion, or in a base case of the recursive construction (see the description of procedure HandleBaseCase$((T, rt(T)), R(T), k)$ in Section 3.1.1). For the spanner edges added from cut vertex u, we perform DFS traversal on T from u and precompute the semigroup products to every other vertex in T. (Note that $(S, \circ)$ might not be commutative and we have to precompute the values both from u to v and from v to u.) During the DFS traversal, we use stack to maintain the semigroup product (in both directions) along the path from u to the vertex currently visited. This traversal requires time linear in the number of required vertices in T, meaning that it does not asymptotically increase the running time of the preprocessing algorithm (cf. Lemma 3.2). The spanner edges added in a base case always shortcut paths of length 2, so we can precompute and store the edge information in the adjacency lists of its endpoints with a constant overhead in time and space per edge. This concludes the description for the case when $k = 2$.

When $k = 3$, at each each recursion level there is $\Theta(\sqrt{n})$ cut vertices, denoted by CV_ℓ, where n is the number of required vertices in the tree considered at that level, T. The spanner consists of edges from each cut vertex $u \in CV_\ell$ to vertices in border(u), those in $CV_\ell \times CV_\ell$, and the edges added when the recursion ends in a base case. To precompute the information associated with edges from each cut vertex $u \in CV_\ell$ to vertices from border(u), we perform one DFS traversal per cut vertex. For cut vertex u, DFS precomputes the semigroup products from u to all the vertices in border(u), in the same way as when $k = 2$. This information is stored together with the edge information in $\mathcal{D}_T$. The running time of DFS from u is linear in the number of spanner edges added from u to vertices in border(u). From Lemma 3.13 in [Sol13], we know that the union over all vertices in CV_ℓ contains at most $O(n)$ spanner edges of this type, meaning that the total running time spent is linear in n. For the same reason as in the proof of Lemma 3.2, this step does not affect the asymptotic running time of the preprocessing algorithm. We proceed to explain how to precompute the values associated with the spanner edges between vertices in $CV_\ell \times CV_\ell$. First, we set all the vertices in CV_ℓ as required and all the other vertices in T as Steiner and invoke pruning procedure Prune$((T, rt(T)), R(T))$ from Section 3.1.1 (cf. Section 3.2 in [Sol13]). The output of this procedure is tree T_{pnd} which satisfies: (i) it has at most $2|CV_\ell| - 1$ vertices, and (ii) each edge $(x, y) \in E(T_{\mathsf{pnd}})$ has associated to it semigroup products in both directions along the path between x and y in T. This step requires $O(n)$ time. For each cut vertex $u \in CV_\ell$, we perform one DFS on T_{pnd} (starting from u) and precompute the semigroup products from u to every other vertex in T_{pnd}. Each DFS call requires size linear in $|T'|$ so the overall complexity of this step is $\Theta(|T_{\mathsf{pnd}}|^2) = \Theta(n)$. We have described how to preprocess all the semigroup products along the edges in $CV_\ell \times CV_\ell$ within a linear time. Preprocessing for the edges added in a base case is handled in the same way as when $k = 2$ — for every spanner edge, we store the semigroup product associated with it in the adjacency arrays of its endpoints.

Finally, when $k \geq 4$, the spanner edges are either added from a cut vertex u to vertices in border(u), or when the recursion reaches a base case (cf. HandleBaseCase$((T, rt(T)), R(T)), k)$ in Section 3.1.1), or via recursive call using the construction for $k-2$. Precomputing the values associated with the edges in $\{u\} \times$ border(u) is done in the same way as we described for the

case when $k = 3$. The edges in the base case are handled in the same way as when $k = 2$ and $k = 3$. Finally, recall that each composite vertex in Φ keeps a pointer to a data structure containing recursively precomputed information for $k-2$. We can use this data and recursively precompute the values associated with the spanner edges there. In summary, we have proved the following theorem.

Theorem 5.6. *Let T be an n-vertex tree having edges associated with elements of semigroup $(S, \circ)$ and let $k \geq 2$ be any integer. We can preprocess T and build a data structure in $O(n\alpha_k(n))$ time and space, such that upon a query for any two vertices $u, v \in V(T)$, it returns the semigroup product along the path from u to v using $k-1$ semigroup operations and in $O(k)$ time.*

Remark 5.4. For given parameter $k \geq 2$, the online tree product algorithm by [AS87] achieves preprocessing time $O(n\alpha_k(n))$ but each query follows a path with $2k$ (instead of k) hops, thus requiring $2k-1$ semigroup operations.

Finally, we note that [AS87] shows several applications of their algorithm due to [Tar79]: (*i*) finding maximum flow values in a multiterminal network, (*ii*) verifying minimum spanning trees, and (*iii*) updating a minimum spanning tree after increasing the cost of one of its edges. We can naturally support all these applications using smaller complexity per query while maintaining the same preprocessing time and space guarantees.

5.6.2 Online MST verification

We now restrict our attention to the online MST verification problem, i.e., to a variant of the tree sum problem where the edges of the tree are elements of the semigroup $(\mathbb{R}, \max)$. We note that some works considered vertex-weighted versions of this problem, but the two are equivalent up to a linear time transformation which preserves the size of the original tree to within a factor of 2. Indeed, if we are given a vertex-weighted tree T with any weight function $w_T : V(T) \to \mathbb{R}$, then we can build an (edge-weighted) tree T' such that $V(T') \coloneqq V(T)$ and $E(T') \coloneqq E(T)$, and assign the weight of an edge $(u, v) \in E(T')$ to be $\max(w_T(u), w_T(v))$. Conversely, if we are given an edge-weighted tree T' with any weight function $w_{T'} : E(T') \to \mathbb{R}$, then we can build a (vertex-weighted) tree T, where for each edge $(u, v) \in E(T')$, we add to T the two edges (u, w) and (w, v), and set $w_T(u) \coloneqq -\infty$, $w_T(v) \coloneqq -\infty$, $w_T(w) = w_{T'}(u, v)$. The tree T' has $2|V(T)| - 1$ vertices and this transformation works in linear time. Using these transformations, any algorithm for the edge-weighted variant of the problem can be used for solving the vertex-weighted variant within the same up to constant factors preprocessing and query complexities, and vice versa. Thus, we may henceforth consider the two variants of the problem as equivalent.

Komlós [Kom85] showed that for any tree T with values associated with its n vertices and a given set of m simple paths (queries) on T, one can find the maximum value for each of the m queries using only $O(n \log((m+n)/n))$ comparisons; this algorithm was presented without an (efficient) implementation. The first implementation was given by [DRT92], and subsequently simpler ones were proposed [Kin97, BKRW98, Hag09]. These implementations run in time $O(n+m)$ while achieving the same bound $O(n \log((m+n)/n))$ on the number of comparisons during preprocessing. We can use this result to reduce the number of comparisons during preprocessing in Theorem 1.1. Specifically, for any tree T, we first construct a 1-spanner $G_T = (V(T), E)$ and the data structure $\mathcal{D}_T$ as in Theorem 1.1, and then apply an implementation of Komlós' algorithm [DRT92, Kin97, BKRW98, Hag09] with the set of queries being (paths between) the endpoints of edges in E. We

store the precomputed information together with the spanner edges. As a direct corollary, the navigation scheme from Theorem 1.1 can be precomputed in $O(n\alpha_k(n))$ time and space while using only $O(n \log \alpha_k(n))$ comparisons. Given a query edge (u, v), we first query $\mathcal{D}_T$ for a k-hop 1-spanner path between u and v. Using $k-1$ comparisons we find the maximum weight along the path and using another comparison we compare this maximum weight against that of the query edge. Overall, this requires k comparisons. The runtime of the query algorithm remains $O(k)$, as in Theorem 1.1.

When k is even, the number of comparisons per query can be reduced by one using an idea suggested in [Pet06]. We next describe this idea for completeness. As before, for any n-vertex tree T, we start by constructing a 1-spanner $G_T = (V(T), E)$ and the data structure $\mathcal{D}_T$ from Theorem 1.1.

Consider first the case $k = 2$. We store the $n-1$ edges of T in an array, which we denote by S. We sort S using $O(n \log n)$ comparisons and to each edge e in T we assign a unique integer in $\{1, \ldots, n-1\}$, obtained as the position of e in S. We call this position the *order* of e in S. Next, we assign to each edge (u, v) in $E(G_T)$ a number corresponding to the maximum among the edge *orders* (in S) on the path between u and v in T. This step can be done in time linear in the spanner size using an implementation of Komlós' algorithm. By using the edge orders in the sorted array S, rather than their weights, we are not spending any comparisons in this step. Given a query edge (u, v), we first query $\mathcal{D}_T$ for a 2-hop 1-spanner path between u and v. Let the two edges of this path be (u, w) and (w, v). Without using any weight comparison, we can find which of the two edges have larger order associated to them. We then find the edge e in S having this order and using one comparison compare the weight of e to the weight of the query edge, (u, v). Overall the number of comparisons made is $k - 1 = 1$.

Suppose now that $k \geq 2$ is an even integer. Whenever PREPROCESSTREE$((T, rt(T)), R(T), k)$ for constructing $\mathcal{D}_T$ executes a recursive call for some tree T' with parameter $k = 2$, we preprocess the spanner edges added for T' using what we explained in the previous paragraph for the case $k = 2$. Using a recurrence similar to that in Lemma 3.2 (see also the proof of Theorem 3.12 in [Sol13]), it is easy to verify that for $k > 2$, the total number of comparisons over all the trees considered in recursive calls with parameter $k = 2$ is linear in the size of given tree T. For the other spanner edges in $E(G_T)$, we apply an implementation of Komlós' algorithm and store the precomputed information alongside the edges. This step requires $O(n\alpha_k(n))$ time and space and uses $O(n \log \alpha_k(n))$ comparisons. Thus, the total time and space complexity for preprocessing is $O(n\alpha_k(n))$ and the number of comparisons used is $O(n \log \alpha_k(n))$. Given a query edge (u, v), we first query $\mathcal{D}_T$ for a k-hop 1-spanner path between u and v. From procedure FINDPATH(u, v, Φ, k), we know that this path either contains less than k edges or it contains k edges, two edges of which, e_1, e_2, belong to some tree T' that was preprocessed with parameter $k = 2$. (See line 6 in procedure FINDPATH(); the edges e_1 and e_2 correspond to $(u, \phi_{T'}(\beta))$ and $(\phi_{T'}(\beta), v)$.) In the former case, the number of comparisons required is clearly at most $k - 1$. If the latter case, i.e., the number of edges on the spanner path between u and v is equal to k, we save one comparison by comparing the *orders* of edges e_1 and e_2 in S for T', which reduces the number of comparisons from k to $k - 1$.

The following theorem summarizes the guarantees of our online MST verification algorithm.

Theorem 5.7. *Let T be an edge-weighted tree with n vertices and let $k \geq 2$ be any integer. We can preprocess T and build a data structure in $O(n\alpha_k(n))$ time and space and using $O(n \log \alpha_k(n))$ comparisons, such that it answers online MST verification queries on T in $O(k)$ time and using at most $k - 1$ comparisons.*

Remark 5.5. Alon and Schieber [AS87] gave an algorithm for the online MST verification problem that requires $O(n\alpha_k(n))$ time, space and comparisons during preprocessing. Their algorithm answers queries following paths of length $2k$, thus achieving $2k$ comparisons. (The number of comparisons is $2k$, rather than $2k-1$, since the query edge must be compared against the tree weights.) Our algorithm improves this tradeoff both in terms of the number of comparisons required for preprocessing ($O(n \log \alpha_k(n))$ rather than $O(n\alpha_k(n))$) and the number of comparisons per query ($k-1$ rather than $2k-1$). Pettie [Pet06] shows that it suffices to spend $O(n\alpha_{2k}(n))$ time and space and $O(n \log \alpha_{2k}(n))$ comparisons during preprocessing, so that each subsequent query can be answered using $4k-1$ comparisons. In fact, [Pet06] uses a different variant of a row-inverse Ackermann function, λ_k, which satisfies $\lambda_k(n) = \Theta(\alpha_{2k}(n))$ (see Lemma 2.1). This result of [Pet06] builds on [AS87] (and [Cha84]), which requires $4k$ comparisons (rather than $2k$ claimed in [Pet06]) following a preprocessing time of $O(n\lambda_k(n)) = O(n\alpha_{2k}(n))$. To reduce the resources required for preprocessing, Pettie [Pet06] used Komlós' algorithm [Kom85], which reduces the number of *comparisons* to $O(n \log \alpha_{2k}(n))$, but not the running time, since the algorithm is information-theoretic, and all known implementations of Komlós' algorithm take *time* linear in the number of queries (which is $\Theta(n\alpha_{2k}(n))$ in this case) [DRT92, Kin97, BKRW98, Hag09]. In this regime, our algorithm requires $O(n\alpha_{2k}(n))$ time and space and $O(n \log \alpha_{2k}(n))$ comparisons during preprocessing, so that each subsequent query can be answered using $2k-1$ comparisons in $O(k)$ time. The result of [Pet06] can also achieve a query time of $O(k)$ (though not claimed), by building on [AS87], but using $4k-1$ comparisons rather than $2k-1$ as in our result. Concurrently and independently of us, Yang [Yan21] obtained a result similar to ours; the two techniques are inherently different. Interestingly, Yang [Yan21] defines yet another variant of a row-inverse Ackermann function, under the notation λ_k, which is similar to the function used by Pettie [Pet06].

Acknowledgments

The fourth-named author thanks Ofer Neiman for helpful discussions.

References

[ABLP89] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Compact distributed data structures for adaptive routing (extended abstract). In *STOC*, pages 479–489. ACM, 1989.

[ABLP90] Baruch Awerbuch, Amotz Bar-Noy, Nathan Linial, and David Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, 11(3):307–341, 1990.

[ABP90] B. Awerbuch, A. Baratz, and D. Peleg. Cost-sensitive analysis of communication protocols. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, PODC'90, pages 177–187, 1990.

[ABP92] Baruch Awerbuch, Alan E. Baratz, and David Peleg. Efficient broadcast and light-weight spanners. Technical Report CS92-22, The Weizmann Institute of Science, Rehovot, Israel., 1992.

[ACE+20] Ittai Abraham, Shiri Chechik, Michael Elkin, Arnold Filtser, and Ofer Neiman. Ramsey spanning trees and their applications. *ACM Trans. Algorithms*, 16(2):19:1–19:21, 2020.

[ADD+93] Ingo Althöfer, Gautam Das, David Dobkin, Deborah Joseph, and José Soares. On sparse spanners of weighted graphs. *Discrete Comput. Geom.*, 9(1):81–100, December 1993.

[ADM+95] S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: Short, thin, and lanky. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 489–498, 1995.

[AG06] Ittai Abraham and Cyril Gavoille. Object location using path separators. In *PODC*, pages 188–197. ACM, 2006.

[AGGM06] Ittai Abraham, Cyril Gavoille, Andrew V. Goldberg, and Dahlia Malkhi. Routing in networks with low doubling dimension. In *Proc. of 26th ICDCS*, page 75, 2006.

[AHL14] Stephen Alstrup, Esben Bistrup Halvorsen, and Kasper Green Larsen. Near-optimal labeling schemes for nearest common ancestors. In *SODA*, pages 972–982. SIAM, 2014.

[AM04] Ittai Abraham and Dahlia Malkhi. Compact routing on euclidian metrics. In *PODC*, pages 141–149. ACM, 2004.

[AN96] Noga Alon and Moni Naor. Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4/5):434–449, 1996.

[AP92] Baruch Awerbuch and David Peleg. Routing with polynomial communication-space trade-off. *SIAM J. Discret. Math.*, 5(2):151–162, 1992.

[AS87] Noga Alon and Baruch Schieber. *Optimal preprocessing for answering on-line product queries*. Citeseer, 1987.

[AS97] S. Arya and M. H. M. Smid. Efficient construction of a bounded degree spanner with low weight. *Algorithmica*, 17(1):33–54, 1997.

[Awe85] Baruch Awerbuch. Communication-time trade-offs in network synchronization. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing*, PODC'85, pages 272–276, 1985.

[BBD+15] Luis Barba, Prosenjit Bose, Mirela Damian, Rolf Fagerberg, Wah Loon Keng, Joseph O'Rourke, André van Renssen, Perouz Taslakian, Sander Verdonschot, and Ge Xia. New and improved spanning ratios for yao graphs. *J. Comput. Geom.*, 6(2):19–53, 2015.

[BCM+16] Prosenjit Bose, Jean-Lou De Carufel, Pat Morin, André van Renssen, and Sander Verdonschot. Towards tight bounds on theta-graphs: More is not always better. *Theor. Comput. Sci.*, 616:70–93, 2016.

[Ber09] Aaron Bernstein. Fully dynamic (2 + epsilon) approximate all-pairs shortest paths with fast query and close to linear update time. In *FOCS*, pages 693–702. IEEE Computer Society, 2009.

[BFC00] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, LATIN '00, pages 88–94, 2000.

[BFC04] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[BFN19] Y. Bartal, N. Fandina, and O. Neiman. Covering metric spaces by few trees. In *46th International Colloquium on Automata, Languages, and Programming*, ICALP'19, pages 20:1–20:16, 2019.

[BGK+11] Yair Bartal, Lee-Ad Gottlieb, Tsvi Kopelowitz, Moshe Lewenstein, and Liam Roditty. Fast, precise and dynamic distance queries. In *SODA*, pages 840–853. SIAM, 2011.

[BKR+02] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat. Opus: an overlay peer utility service. In *The 5th International Conference on Open Architectures and Network Programming*, OPENARCH'02, 2002.

[BKRW98] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *STOC*, pages 279–288. ACM, 1998.

[BLW17] Glencora Borradaile, Hung Le, and Christian Wulff-Nilsen. Greedy spanners are optimal in doubling metrics. *CoRR*, abs/1712.05007, 2017.

[BP20] Uri Ben-Levy and Merav Parter. New (α, β) spanners and hopsets. In *SODA*, pages 1695–1714. SIAM, 2020.

[CDNS92] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*, 1992.

[CG06] R. Cole and L. Gottlieb. Searching dynamic point sets in spaces with bounded doubling dimension. In *Proceedings of the 38th annual ACM Symposium on Theory of Computing*, STOC '06, 2006.

[CG09] T-H Hubert Chan and Anupam Gupta. Small hop-diameter sparse spanners for doubling metrics. *Discrete & Computational Geometry*, 41(1):28–44, 2009.

[CGMZ16] T.-H. Hubert Chan, Anupam Gupta, Bruce M. Maggs, and Shuheng Zhou. On hierarchical routing in doubling metrics. *ACM Trans. Algorithms*, 12(4):55:1–55:22, 2016. Preliminary version appeared in SODA 2005.

[Cha84] Bernard Chazelle. Computing on a free tree via complexity-preserving mappings. In *FOCS*, pages 358–368. IEEE Computer Society, 1984.

[Cha87] B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2:337–361, 1987.

[Cha00] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[Cha08] Timothy M Chan. Well-separated pair decomposition in linear time. *Information Processing Letters*, 107(5):138–141, 2008.

[Che86] L. P. Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 169–177, 1986.

[Che13] Shiri Chechik. Compact routing schemes with improved stretch. In *PODC*, pages 33–41. ACM, 2013.

[Che14] Shiri Chechik. Approximate distance oracles with constant query time. In *STOC*, pages 654–663. ACM, 2014.

[Che15] Shiri Chechik. Approximate distance oracles with improved bounds. In *STOC*, pages 1–10. ACM, 2015.

[Cla87] K. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 56–65, 1987.

[CLNS15] T-H Hubert Chan, Mingfei Li, Li Ning, and Shay Solomon. New doubling spanners: Better and simpler. *SIAM Journal on Computing*, 44(1):37–53, 2015.

[Coh00] Edith Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. *J. ACM*, 47(1):132–166, 2000.

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

[DPP06a] Mirela Damian, Saurav Pandit, and Sriram V. Pemmaraju. Distributed spanner construction in doubling metric spaces. In *OPODIS*, volume 4305 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2006.

[DPP06b] Mirela Damian, Saurav Pandit, and Sriram V. Pemmaraju. Local approximation schemes for topology control. In *PODC*, pages 208–217. ACM, 2006.

[DRT92] Brandon Dixon, Monika Rauch, and Robert Endre Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992.

[EFN20] Michael Elkin, Arnold Filtser, and Ofer Neiman. Distributed construction of light networks. In *PODC*, pages 483–492. ACM, 2020.

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

[EK21] David Eppstein and Hadi Khodabandeh. Optimal spanners for unit ball graphs in doubling metrics. *CoRR*, abs/2106.15234, 2021.

[ES15] M. Elkin and S. Solomon. Optimal euclidean spanners: Really short, thin, and lanky. *J. ACM*, 62(5):35:1–35:45, 2015.

[FG01] F. Fraigniaud and C. Gavoille. Routing in trees. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming, ICAL '01*, pages 757–772, 2001.

[FGNW17] Ofer Freedman, Paweł Gawrychowski, Patrick K. Nicholson, and Oren Weimann. Optimal distance labeling schemes for trees. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 185–194, New York, NY, USA, 2017. Association for Computing Machinery.

[Fil21] Arnold Filtser. Hop-constrained metric embeddings and their applications. In *62nd FOCS*, 2021.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[FS20] Arnold Filtser and Shay Solomon. The greedy spanner is existentially optimal. *SIAM Journal on Computing*, 49(2):429–447, 2020.

[FW94] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48(3):533–551, 1994. Announced at FOCS'90.

[GGN04] J. Gao, L. J. Guibas, and A. Nguyen. Deformable spanners and applications. In *Proc. of 20th SoCG*, pages 190–199, 2004.

[GH16] Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In *SODA*, pages 202–219. SIAM, 2016.

[GH21] Mohsen Ghaffari and Bernhard Haeupler. Low-congestion shortcuts for graphs excluding dense minors. In *PODC*, pages 213–221. ACM, 2021.

[GLNS08] Joachim Gudmundsson, Christos Levcopoulos, Giri Narasimhan, and Michiel H. M. Smid. Approximate distance oracles for geometric spanners. *ACM Trans. Algorithms*, 4(1):10:1–10:34, 2008.

[Got15] L. Gottlieb. A light metric spanner. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 759–772, 2015.

[GP17] Mohsen Ghaffari and Merav Parter. Near-optimal distributed DFS in planar graphs. In *DISC*, volume 91 of *LIPIcs*, pages 21:1–21:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[GR08a] L. Gottlieb and L. Roditty. Improved algorithms for fully dynamic geometric spanners and geometric routing. In *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'08, pages 591–600, 2008.

[GR08b] L. Gottlieb and L. Roditty. An optimal dynamic spanner for doubling metric spaces. In *Proc. of 16th ESA*, pages 478–489, 2008. Another version of this paper is available via http://cs.nyu.edu/~adi/spanner2.pdf.

[Hag09] Torben Hagerup. An even simpler linear-time algorithm for verifying minimum spanning trees. In *WG*, volume 5911 of *Lecture Notes in Computer Science*, pages 178–189, 2009.

[HKN18] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. Decremental single-source shortest paths on undirected graphs in near-linear total update time. *J. ACM*, 65(6):36:1–36:40, 2018.

[HL18] Bernhard Haeupler and Jason Li. Faster distributed shortest path approximations via shortcuts. In *DISC*, volume 121 of *LIPIcs*, pages 33:1–33:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[HM06] Sariel Har-Peled and Manor Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM J. Comput.*, 35(5):1148–1184, 2006.

[HP00] Yehuda Hassin and David Peleg. Sparse communication networks and efficient routing in the plane (extended abstract). In *PODC*, pages 41–50. ACM, 2000.

[HPM06] S. Har-Peled and M. Mendel. Fast construction of nets in low-dimensional metrics and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006. Announced at SoCG'05.

[Kei88] J. M. Keil. Approximating the complete Euclidean graph. In *Proceedings of the first Scandinavian Workshop on Algorithm Theory*, SWAT '88, pages 208–213, 1988.

[KG92] J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete and Computational Geometry*, 7(1):13–28, 1992.

[Kin97] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[KKS11] Ken-ichi Kawarabayashi, Philip N. Klein, and Christian Sommer. Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In *ICALP (1)*, volume 6755 of *Lecture Notes in Computer Science*, pages 135–146. Springer, 2011.

[KKT95] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, 1995.

[Kom85] János Komlós. Linear verification for spanning trees. *Comb.*, 5(1):57–65, 1985.

[KP21] Shimon Kogan and Merav Parter. Low-congestion shortcuts in constant diameter graphs. In *PODC*, pages 203–211. ACM, 2021.

[KP22] Shimon Kogan and Merav Parter. New diameter-reducing shortcuts and directed hopsets: Breaking the $\sqrt{n}$ barrier. In *SODA'22*, 2022.

[KRY93] S. Khuller, B. Raghavachari, and N. Young. Balancing minimum spanning and shortest path trees. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'93, pages 243–250, 1993.

[KV02] D. Kostic and A. Vahdat. Latency versus cost optimizations in hierarchical overlay networks. *Technical report, Duke University*, (CS-2001-04), 2002.

[LMS22] Hung Le, Lazar Milenkovic, and Shay Solomon. Sparse euclidean spanners with tiny diameter: A tight lower bound. In *SoCG'22, to appear*, 2022.

[LP19] Jason Li and Merav Parter. Planar diameter via metric compression. In *STOC*, pages 152–163. ACM, 2019.

[LS19] Hung Le and Shay Solomon. Truly optimal euclidean spanners. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1078–1100. IEEE, 2019.

[MN06] Manor Mendel and Assaf Naor. Ramsey partitions and proximity data structures. In *FOCS*, pages 109–118. IEEE Computer Society, 2006.

[MPVX15] Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *SPAA*, pages 192–201. ACM, 2015.

[Nan14] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *STOC*, pages 565–573. ACM, 2014.

[NS07] Giri Narasimhan and Michiel Smid. *Geometric Spanner Networks*. Cambridge University Press, USA, 2007.

[NT12] Assaf Naor and Terence Tao. Scale-oblivious metric fragmentation and the nonlinear dvoretzky theorem. *Israel Journal of Mathematics*, 192(1):489–504, 2012.

[Pel00] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, Philadelphia, PA, 2000.

[Pet06] Seth Pettie. An inverse-ackermann type lower bound for online minimum spanning tree verification. *Comb.*, 26(2):207–230, 2006.

[PS89] D. Peleg and A. A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[PU89a] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM Journal on Computing*, 18(4):740–747, 1989.

[PU89b] D. Peleg and E. Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM*, 36(3):510–530, 1989.

[Rod12] Liam Roditty. Fully dynamic geometric spanners. *Algorithmica*, 62(3-4):1073–1087, 2012.

[RS91] J. Ruppert and R. Seidel. Approximating the d-dimensional complete Euclidean graph. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, CCCG '91, page 207–210, 1991.

[RT15] Liam Roditty and Roei Tov. New routing techniques and their applications. In *PODC*, pages 23–32. ACM, 2015.

[Sal92] J. S. Salowe. On Euclidean spanner graphs with small degree. In *Proceedings of the 8th Annual Symposium on Computational Geometry*, SoCG'92, pages 186–191, 1992.

[Sli05] Aleksandrs Slivkins. Distance estimation and object location via rings of neighbors. In *PODC*, pages 41–50. ACM, 2005.

[Smi09] M. Smid. The weak gap property in metric spaces of bounded doubling dimension. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms*, pages 275–289. Springer-Verlag, 2009.

[Sol13] Shay Solomon. Sparse euclidean spanners with tiny diameter. *ACM Trans. Algorithms*, 9(3):28:1–28:33, 2013.

[Sol14] Shay Solomon. Euclidean steiner shallow-light trees. In *SoCG*, page 454. ACM, 2014.

[Tal04] Kunal Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 281–290, 2004.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[Tar79] Robert Endre Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.

[Tho92] Mikkel Thorup. On shortcutting digraphs. In *WG*, volume 657 of *Lecture Notes in Computer Science*, pages 205–211. Springer, 1992.

[Tho99] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[Tho04] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, 2004.

[TZ01a] M. Thorup and U. Zwick. Approximate distance oracles. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, STOC'01, pages 183–192, 2001.

[TZ01b] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA'01, pages 1–10, 2001.

[VWF+03] J. Vogel, J. Widmer, D. Farin, M. Mauve, and W. Effelsberg. Priority-based distribution trees for application-level multicast. In *Proceedings of the 2nd Workshop on Network and System Support for Games*, NETGAMES'03, pages 148–157, 2003.

[WCT02] B. Y. Wu, K. Chao, and C. Y. Tang. Light graphs with small routing cost. *Networks*, 39(3):130–138, 2002.

[Wul13] Christian Wulff-Nilsen. Approximate distance oracles with improved query time. In *SODA*, pages 539–549. SIAM, 2013.

[Yan21] Tianqi Yang. Tree path minimum query oracle via boruvka trees. *CoRR*, abs/2105.01864, 2021.

[Yao82] A. C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM Journal on Computing*, 11(4):721–736, 1982.

A Proof of Lemma 2.1

This appendix is dedicated to proving the following lemma stated in Section 2.

Lemma 2.1. *For any $i \geq 1$, if $\lambda_i(n) > 0$, then $\frac{1}{3}\alpha_{2i}(n) \leq \lambda_i(n) \leq \alpha_{2i}(n)$.*

Let $T(\cdot,\cdot)$ be slightly different Ackermann function as defined by Tarjan [Tar75].

$$\begin{aligned} T(0,j) &= 2j && \text{for } j \geq 0 \\ T(i,0) &= 0 && \text{for } i \geq 1 \\ T(i,1) &= 2 && \text{for } i \geq 1 \\ T(i,j) &= T(i-1,T(i,j-1)) && \text{for } i \geq 1 \text{ and } j \geq 2 \end{aligned}$$

In the following lemma, we show that A and T are almost equal (except for the first column).

Claim A.1. For all $i \geq 0$ and $j \geq 1$, $A(i,j) = T(i,j)$.

Proof. We will show the claim inductively.
Base case $i = 0$. For all $j \geq 0$, it follows by definition that $A(0,j) = T(0,j) = 2j$.
Base case $j = 1$. For any $i > 0$,it follows by definition that $A(i,1) = A(i-1,A(i,0)) = A(i-1,1)$. Since $A(0,1) = 2$ it follows by induction that for every $i \geq 0$, $A(i,1) = 2$, and so $A(i,1) = T(i,1)$.
Inductive step. We proceed to show the inductive step for $i \geq 1, j \geq 2$, assuming that the claim holds for any pair $i' \geq 0, j' \geq 1$ lexicographically smaller than (i,j).

$$\begin{aligned} A(i,j) &= A(i-1,A(i,j-1)) \\ &= A(i-1,T(i,j-1)) && \text{from inductive hypothesis} \\ &= T(i-1,T(i,j-1)) && \text{since } T(i,j-1) \geq 2 \text{ for } j \geq 2 \\ &= T(i,j) \end{aligned}$$

□

We are ready to prove Lemma 2.1.

Proof of Lemma 2.1. Pettie [Pet06] shows that for any $i \geq 1$ his variant of Ackermann function P (cf. Section 2.2) satisfies:

- $T(i,j) \leq P(i,j)$ for $j \geq 0$,
- $P(i,j) \leq T(i,3j)$ for $j \geq 1$.

From the definition of $\lambda_i(\cdot)$:

$$\begin{aligned} \lambda_i(n) &= \min\{j : P(i,j) \geq n\} \\ &\leq \min\{j : T(i,j) \geq n\} \\ &= \min\{j : A(i,j) \geq n\} && \text{for } j \geq 1 \\ &= \alpha_{2i}(n). \end{aligned}$$

On the other hand:

$$\begin{aligned}
\lambda_i(n) &= \min\{j : P(i,j) \geq n\} \\
&\geq \min\{j : T(i,3j) \geq n\} && \text{for } j \geq 1 \\
&= \min\{j : A(i,3j) \geq n\} \\
&\geq \frac{1}{3}\alpha_{2i}(n)
\end{aligned}$$

□