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

Omri Kahalon  
Tel Aviv University  
Tel Aviv, Israel  
kahalon@gmail.com

Hung Le  
University of Massachusetts Amherst  
Amherst, Massachusetts, USA  
hungle@cs.umass.edu

Lazar Milenković  
Tel Aviv University  
Tel Aviv, Israel  
milenkovic.lazar@gmail.com

Shay Solomon  
Tel Aviv University  
Tel Aviv, Israel  
solo.shay@gmail.com

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.

CCS CONCEPTS

- Theory of computation → Sparsification and spanners; Shortest paths; Routing and network design problems.

KEYWORDS

spanners, metric spaces, Euclidean metrics, doubling metrics, hop-diameter, fault-tolerance, tree covers, routing schemes

ACM Reference Format:

Omri Kahalon, Hung Le, Lazar Milenković, and Shay Solomon. 2022. Can’t See The Forest for the Trees: Navigating Metric Spaces by Bounded Hop-Diameter Spanners. In Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (PODC ’22), July 25–29, 2022, Salerno, Italy. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519270.3538414

The full version of the paper is available at [69].

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 \( H = (V, E', \omega) \) 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 \( p \) and \( q \) 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 [89, 90],
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 [14, 15], synchronizing networks and computing global functions [11, 88, 90], gathering and disseminating data [24, 79, 109], and routing [12, 91, 108, 110]; as another example, spanners with low degree can be used for the design of compact routing schemes [3, 4, 13, 26, 60, 67, 91, 101]. Since then, graph spanners have found countless applications in distributed computing as well as various other areas, from motion planning and computational biology to countless applications in distributed computing.

Spanners have special success in geometric settings, especially in low-dimensional Euclidean spaces. Spanners for Euclidean spaces, namely Euclidean spanners, were first studied by Chew [37] in 1986 (even before the term “spanner” was coined). Several different constructions of Euclidean spanners enjoy the optimal tradeoff between stretch and size: $(1+\varepsilon)$-graphs [38, 72, 73, 95], Yao graphs [113], path-greedy spanner [8, 30, 87], and the gap-greedy spanner [10, 96]. 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 [8, 89]. 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. The main result in this area is that any $n$-point metric of doubling dimension $d$ admits a $(1 + \varepsilon)$-spanner with both sparsity and lightness bounded by $O(e^{-O(d)})$ [23, 26–28, 47, 50, 55, 59–61, 65, 93, 98, 100]. Moreover, here too there are efficient centralized and distributed algorithms, also under some practical restrictions such as those imposed by Unit Ball Graphs [42, 43, 46, 48].

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 metrics, 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 $\Delta$, then the hop-diameter is $\Omega(\log \Delta n)$.

In Euclidean spaces, the $\Theta$-graph [38, 72, 73, 95] and the Yao graph [113] 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 $(1 + \varepsilon)$-spanner path between $p$ and $q$. Alas, the resulting path may have a hop-length of $O(\varepsilon n)$, and the query time is no smaller than the path length. There is a $(1 + \varepsilon)$-approximate distance oracle for low-spanner graphs [62], and while it achieves constant query time, it does not report the respective paths, whose hop-length can be $O(\varepsilon n)$. In doubling metrics, there are $(1+\varepsilon)$-approximate distance oracles with constant query time [18, 66]. In [18] 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 [18] may comprise a spanner of $\Theta(n^2)$ edges. Using [66], one can return paths that are part of a sparse spanner, but their hop-length is $O(\log \rho)$, where $\rho$—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 [27, 99]. In low-dimensional Euclidean spaces, it is possible to build a $(1 + \varepsilon)$-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 $O(na_k(n))$ edges, in

---

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 $\varepsilon$ and the dimension $d$. 

---
optimal $O(n \log n)$ time [99]; the function $a_k(n)$ is the inverse of a certain function at the $\lfloor k/2 \rfloor$th level of the primitive recursive hierarchy, where $a_0(n) = \lceil n/2 \rceil$, $a_1(n) = \sqrt{n}$, $a_2(n) = \lfloor \log n \rfloor$, $a_3(n) = \lfloor \log \log n \rfloor$, $a_4(n) = \log^* n$, $a_5(n) = \frac{1}{2} \log^* n$, etc. (For $k \geq 4$, the function $a_k$ is close to log with $\Theta \left(\frac{\log n}{\log \log n}\right)$ stars.)

Two points on the tradeoff curve between hop-diameter $k$ and size $O(n a_k(n))$ 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) $k = O(a(n))$ vs. $O(n a_k(n)) = O(n)$ edges, where $a$ is a very slowly (more than $\log^* \log^* n$) growing function; so to achieve a truly linear in $n$ edges, one should take a hop-diameter of $O(a(n))$ (which is effectively a constant). Refer to the full version of the paper [69] for the formal definitions of the functions $a_k$ and $a$. 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, $O(n a_4(n)) = O(n \log^* n)$ 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 — $O(n^2)$ 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 [107] reports $(2t - 1)$-approximate distance queries in $O(t)$ time, using a data structure of expected size $O(t n^{1+1/t})$; it is immediate that their distance oracle, when applied to metric spaces, can report 2-hop paths of stretch $2t - 1$ in query time $O(t)$, which are all part of the same $(2t - 1)$-spanner with size $O(t n^{1+1/t})$. The following question is copied from [83]:

“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 [83] gave two distance oracles with $O(1)$ query time and stretch of $128t$ (the stretch was improved later to $16t$ [86]), the first has size $O(n^{1+1/t})$ and the respective paths can use any edge of the underlying metric and may thus form a network of size $O(n^2)$, whereas the second has size $O(n^{1+1/t} \cdot t)$ and the respective paths have hop-lengths $O(\log \rho)$. Wulff-Nilsen [111] improved the query time of the Thorup-Zwick distance oracle [108] to $O(\log t)$. Using the Mendel-Naor distance oracle [83], Chechik [35, 36] showed how to improve query time of [108] to $O(1)$, but this approach suffers from the same drawback — the respective paths may have hop-lengths $O(\log \rho)$. Mendel-Naor question can thus be strengthened as:

**Question 1.2 (Strengthening Mendel-Naor question [83]).** 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 $(2, 71, 106)$, when applied to the respective metrics, can provide 2-hop paths within constant query time.

**Related work (in a nutshell).** Thorup [104] 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 [39] introduced the notion of hopsets; informally, a hopset $H$ is an edge set that, when added to a graph $G$, provides small-stretch small-hop paths between all vertex pairs. (See [19, 77] and references therein for details.) There are also various other related problems, such as low-congestion shortcuts [56, 57, 76]. 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 [22, 58, 63, 68, 82, 84, 85]. 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 $O(n \log n)$, 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 [99]. Our result for navigation on trees is stated in the following theorem (proved in Section 2.1); the stretch bound is 1 and one cannot improve the tradeoff between hop-diameter $k$ and size $O(n\ell_1(n))$, due to lower bounds by [6] and [80] 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 $O(n\ell_1(n))$ edges due to [99]. Then we can construct in time $O(n\ell_1(n))$ a data structure $D_T$ such that, for any two query vertices $u, v \in V(T)$, $D_T$ 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 [99] is $O(n\ell_1(n))$, hence the data structure provided by Theorem 1.1 can be built from scratch in time $O(n\ell_1(n))$. 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 [6] gave an algorithm for the online tree product that requires $O(n\ell_1(n))$ 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 $O(n\ell_1(n))$ 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. Solomon [99] 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 [99] is more complex than that of [6], obtaining a navigation algorithm on top of the 1-spanner of [99] is technically much more intricate than doing so on top of the 1-spanner of [6]. 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 3.1). 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 $y \geq 1$ and an integer $\zeta \geq 1$, a $(y, \zeta)$-tree cover of $M_X = (X, \delta_X)$ is a collection of $\zeta$ dominating trees for $M_X$, such that for every $x, y \in X$, there exists a tree $T$ with $\delta_T(x, y) \leq y \cdot \delta_X(x, y)$; we say that the stretch between $x$ and $y$ in $T$ is at most $y$, and the parameter $y$ 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 $y$.

The celebrated "Dumbbell Theorem" by Arya et al. [9] provides a $(1 + \epsilon, O((\log(n)/\epsilon))$-tree cover in $O((\log(n)/\epsilon)^2 \cdot n \log n + n \epsilon^2) = O(n \log n)$ time, for $d$-dimensional Euclidean spaces. For general metrics, the seminal work of Mendel-Naor [83] provides a Ramsey $(y, \zeta)$-tree cover with $O(\zeta n^{1/d} \log n)$ time, where $y = O(\log(n) \epsilon^{-1/d})$, $n \geq l/4$, for any $\epsilon \geq 1$. Additional tree cover constructions are given in [17], including a $(1 + \epsilon, (1/\epsilon)^{O(1)})$-tree cover for metrics with doubling dimension $d$.

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 $(1 + \epsilon)$-spanner $H_X$ for $M_X$ with hop-diameter $k$ and $O(n\ell_1(n) \cdot \zeta)$ edges, accompanied with a data structure $D_X$, such that for any two query points $u, v \in X$, $D_X$ returns in time $\tau$ a $y$-spanner path in $H_X$ between $u$ and $v$ of at most $k$ hops, where

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

If $M_X$ is doubling, the running time is $O(n \log n)$, for fixed $\epsilon$ 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 $O(n\ell_1(n))$ edges, within query time $O(k)$, ignoring dependencies on $\epsilon$ 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 [2, 71, 106], 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 [35, 36, 83, 86, 108]. 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 [9] 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 [17] 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 [99]. See Section 3 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 appear in the full version of the paper [69].

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 [69] 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 \( M_\mathcal{X} = (X, \delta_\mathcal{X}) \), one can construct a \( \gamma \)-stretch 2-hop routing scheme in the labeled, fixed-port model with headers of \( \lceil \log n \rceil \) bits, labels of \( b_1 \) bits, local routing tables of \( b_2 \) bits, and local decision time \( \tau \), where:

- \( \gamma = (1 + \epsilon), \ b_1 = b_2 = O(\epsilon^{-O(d)} \log(n^2) \log(n^3)), \tau = O(\epsilon^{-O(d)}), \) for doubling dimension \( d \).
- If \( M_\mathcal{X} \) is a general metric, there are two possible tradeoffs, for any integer \( \ell \geq 1 \):
  - \( \gamma = O(\ell), \ b_1 = O(\log^2 n), \ b_2 = O(\ell \cdot n^{1/\ell} \log^2 n), \tau = O(1). \)
  - \( \gamma = O(n^{1/\ell} \cdot \log^{-1/\ell}), \ b_1 = O(\log^2 n), \ b_2 = O(\ell \log^2 n), \tau = O(1). \)
- \( \gamma = (1 + \epsilon), \ b_1 = b_2 = O(\log(n^2))^3 \log n), \tau = O((\log(n^2))^2), \) for a fixed-minor-free metric.

If \( M_\mathcal{X} \) is doubling, the running time is \( O(n \log n) \), for fixed \( \epsilon \) and \( d \). In this case, one can achieve an \( f \)-fault-tolerant routing scheme, with the bounds on \( b_1 \) and \( b_2 \) 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 [26, 60] 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 [3, 4, 67, 97, 101] 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 [1, 13, 16, 34, 41, 45, 49, 94, 108], in metrics, it is much easier to get an efficient routing scheme. The Thorup-Zwick routing scheme [108] can achieve two hops in general metrics with stretch \( 4\ell - 5 \) (improved to 3.688 [34]), labels of \( O(\ell \log n) \) bits, and table sizes of \( O(n^{1/\ell}) \). 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 \( \log nf \) factor), achieve constant decision time, which might be an important advantage in real-time routing applications.

## 2 NAVIGATING METRIC SPACES

### 2.1 Navigating the tree spanner

Our navigation algorithm consists of two parts. In Section 2.1.1, we present a preprocessing algorithm, which takes a tree \( T \) and an integer parameter \( k \geq 2 \); it constructs 1-spanner \( G_T \) due to [99] 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 2.1.2, 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 [99] 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 \( S(T) := V(T) \setminus R(T) \) 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 [99] spanner construction algorithm. It relies on the following two procedures, which we shall also use in our preprocessing algorithm. Procedure \( \text{Prune}((T, r(t(T)), R(T)) \) takes as input a tree \( T \), its root \( r(T) \), and the set of required vertices \( R(T) \). Outputs an edge-weighted tree \( (T_{\text{pru}}, r(T_{\text{pru}})) \), which contains \( R(T) \) and has at most \( |R(T)| - 1 \) Steiner vertices. We set the weight \( w_{T_{\text{pru}}} (u, v) = \delta_T (u, v) \) for every edge \( (u, v) \in E(T_{\text{pru}}) \). Informally, the procedure keeps the intrinsic properties of \( T \), while reducing the number of Steiner vertices. For more details, see Section 3.2 in [99]. The running time is \( O(|V(T)|) \).

Procedure \( \text{Decompose}((T, r(t(T)), R(T), \ell) \) takes as input a rooted tree \( (T, r(t(T)), \ell \) takes as the set of required vertices \( R(T) \), and an integer parameter \( \ell \geq 1 \). (The parameter \( \ell \) will be set to \( \frac{k - 1}{2} \) at a later time.) Outputs, in \( O(|V(T)|) \) time, a set of cut vertices, denoted by \( CV_T \subseteq V(T) \), such that every connected component (tree) of \( T \setminus CV_T \) contains at most \( \ell \) required vertices.

At the beginning of the spanner construction, we find a subset of vertices \( CV_T \subseteq V(T) \), using \( \text{Decompose}((T, r(t(T)), R(T, \ell)), \ell) \). We then compute the set of edges \( E' \), which interconnects vertices in \( CV_T \).

The algorithm distinguishes several cases:

- If \( k = 2 \), then \( |CV_T| = 1 \) and \( E' = \emptyset \).
- If \( k = 3 \), then connect every pair of vertices in \( CV_T \), i.e., \( E' = CV_T \times CV_T \).
- If \( k \geq 4 \), then make a copy \( T' \) of \( T \), set \( CV_{T'} \) as its required vertices and prune it, by invoking \( \text{Prune}((T', r(t(T')))), CV_{T'} \); let \( E' \) be the set of edges returned by recursive spanner construction on \( T' \) with hop-diameter \( k \) to \( k - 2 \).

Denote by \( T_1, \ldots, T_p \) the trees in \( T \setminus CV_T \). The algorithm computes the set of edges \( E'' \) that connects the cut vertices of \( CV_T \) 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 \( u \not\in V(T_i) \) is adjacent to a vertex in \( T_i \). Let border\( (T_i) \) denote the set of all border vertices of \( T_i \). With a slight abuse of notation, we let border\( (T_i) = \{ u \in V(T) \mid c \in \text{border}(c) \} \) for every \( c \in CV_T \). We add an edge from \( c \) to all the required vertices in border\( (c) \). Finally, for each \( i \in \{ p \} \), 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 \( P_{T}(u, v) \) 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 \( P_{T}(u, v) \), that is, if \( P_T (u, v) = (u = v_0, v_1, \ldots, v_k = v) \), then \( P \) can be written as \( P = (u = v_0, v_1, \ldots, v_k = v) \), where \( 0 = i_0 < i_1 < \cdots < i_k = t \). For any two vertices \( u, v \in R(T) \), there is a \( T \)-monotone path in \( G_T \) 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 [99] that for any two \( u, v \) there exists an intermediate cut vertex \( w \) on the path \( P_T (u, v) \), 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 \( P_T (u, v) \) 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 \( CV_T = \{ w \} \). 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 \( \Phi \). For each vertex \( v \in R(T) \), we keep track of the vertex in \( \Phi \) which corresponds to the recursive call when \( v \) was chosen as the 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 \( u_0 \) and \( v_0 \) containing \( u \) and \( v \) in \( \Phi \). Then, we find their lowest common ancestor \( \beta \) in \( \Phi \). Vertex \( \beta \) 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 \( P_T (u, v) \). 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 \( P_T (u, v) \). (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 \( CV_T \). Since \( u', v' \) are on \( P_T (u, v) \), tree \( T_0 \subseteq T' \setminus CV_T \) containing \( u \) and \( T_0 \subseteq T' \setminus CV_T \) containing \( v \) are different. At that point, \( u \) (resp. \( v \)) could have many cut vertices in border\( (u) \) (resp. border\( (v) \)). To avoid checking every possible pair of cut vertices in border\( (u) \) and 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 \( (T', r(t(T'))) \) is passed as an argument, and let \( CV_T \) denote the set of cut vertices chosen for this level. Furthermore, let \( T_1, \ldots, T_p = T' \setminus CV_T \) be the subtrees obtained by removing vertices in \( CV_T \) from \( T' \). The set of vertices of the contracted tree \( T_0 \), corresponding to vertex \( \beta \) in \( \Phi \), consists of \( p \) vertices, \( t_1, \ldots, t_p \), corresponding to \( T_1, \ldots, T_p \), and \( |CV_T| \) vertices corresponding to cut vertices in \( CV_T \). For each vertex \( t_i \in T_0 \), we add an edge between \( t_i \) and all the vertices in \( T_0 \) corresponding to cut vertices in border\( (t_i) \). Intuitively, the augmented tree \( T_0 \) identifies every subtree \( t_i \) with a single vertex and keeps the tree structure of given tree \( T \).
We now explain how \( T_B \) facilitates finding cut vertices \( u' \) and \( v' \) corresponding to vertex \( \beta \in \Phi \) which are on \( T_r(u, v) \). First, we find the vertex \( t_u \) (resp., \( t_v \)) in \( T_B \) 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 \( CV_i \).) Cut vertex \( u' \in CV_i \) is the first vertex on the path from \( t_u \) to \( t_v \) in \( T_B \). 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 \( T_B \). 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.

### 2.1.1 Preprocessing

We proceed to give a detailed description of the preprocessing algorithm. It takes as an input a rooted tree \((T, r(T))\) which induces a tree metric \( M_T \). Notice that \( T \) can be transformed in linear time into a pruned tree \((T_{\text{pruned}}, r(T_{\text{pruned}}))\) by invoking the procedure \( \text{PRUNE}(T, r(T)), R(T) \). Also, any \( 1 \)-spanner for pruned tree \( T_{\text{pruned}} \) 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 constructs two types of trees — augmented recursion trees and contracted trees. We preprocess every such tree in linear time so that subsequent \textit{lowest common ancestor} (henceforth, LCA) and \textit{level ancestor} (henceforth, LA) queries can be answered in constant time. For more details, refer to \cite{20, 21}.

We now give details of the procedure \( \text{PREPROCESS}(T, r(T)), R(T), k) \). For pseudocode, see the full version of the paper \cite{99}. This procedure takes as parameters a rooted tree \((T, r(T)), \) 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 \( G_T \) for \( T \), together with the augmented recursion tree \((\Phi, r(\Phi))\). In addition, it creates a data structure \( D_T \) 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, \( n := |R(T)| \). When \( n \leq k + 1 \), the algorithm invokes \( \text{HANDLE-BASE-CASE}(T, r(T)), R(T), k) \), which we proceed to describe. If \( n = k + 1 \) and \( r(T) \) has exactly two children, \( u \) and \( v \), the edge \( E \) between them is the only one in \( T \). The algorithm returns \( E \) and \( \Phi \) as a single vertex \( v \) of the recursion tree. At this stage, we keep a pointer to \( v \). When \( n > k + 1 \), the algorithm invokes \( \text{NEWVERTEX}(CV_i) \), which assigns to \( \beta \) its inner vertices all the cut vertices in \( CV_i \). The procedure also keeps track of all the relevant pointers.

The algorithm \( \text{PREPROCESS}(T, r(T)), R(T), k) \) returns as its output the set of all vertex pairs (\( u, v \)) in \( \Phi \), such that \( u \) and \( v \) are connected by an edge \( e \) in \( T \). For each \( T \), the set \( E' \) consists of the edges between every two cut vertices, i.e., \( E' = CV_i \times CV_i \). For \( k \geq 4 \), \( k \) first create a tree isomorphic to \( T \), which has as its required vertices the inner vertices of \( \beta \). This is done via \( \text{CREATE-CONTRACTED}(\beta, \{T_i\}_{i \in [p]}, \{T_i \}_{i \in [p]} \} \). This procedure creates a data structure \( D_T \) corresponding to \( \Phi \). The algorithm returns the resulting edge set and \( (\Phi, r(\Phi)) \) the recursion tree for each of thesubtrees. We make \( r(\Phi) \) a child of \( \beta \).

Finally, if \( k \geq 3 \) we construct the contracted tree \( T_B \) which corresponds to \( \beta \). This is done via a call to procedure \( \text{CREATE-CONTRACTED}(\beta, \{T_i\}_{i \in [p]}, \{T_i \}_{i \in [p]} \} \). This procedure creates a data structure \( D_T \) corresponding to \( \Phi \). The algorithm returns the resulting edge set and \( (\Phi, r(\Phi)) \) the recursion tree for each of thesubtrees. We make \( r(\Phi) \) a child of \( \beta \).

### 2.1.2 Query algorithm

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 \( \Phi \) denote the recursion tree corresponding to \( \Phi \) constructed during the construction of \( G_T \). We check if \( u \) and \( v \) point to the same leaf in \( \Phi \). The inner vertex corresponding to \( u \) (resp., \( v \)) is contained in \( u \) (resp., \( v \)) of \( \Phi \). We use \( u.ptr(\Phi) \) (resp., \( v.ptr(\Phi) \)) to obtain the actual vertex in \( \Phi \), which contains \( u.ptr(\Phi) \) (resp., \( v.ptr(\Phi) \)) as its inner vertices. If \( u.ptr(\Phi) \) is equal to \( v.ptr(\Phi) \), the algorithm returns the path found by BFS on the subgraph of \( G_T \) induced on all the vertices corresponding to the same base case. 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 \( G_T \) 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 \( \Phi \), the algorithm finds LCA in \( \Phi \) of \( u.ptr(\Phi) \) and \( v.ptr(\Phi) \), denoted by \( \beta \). If \( k = 2 \), then \( \beta \) 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 \( u.ptr(T), \beta.ptr(T), u.ptr(T) \). 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 cuts corresponding to \( u \) and \( v \). For that purpose, it considers contracted tree \( T_β \), corresponding to \( β \). First of all, it locates vertices corresponding to \( u \) (resp., \( v \)) in \( T_β \), via a call to \text{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 \( T_β \), which is obtained via \( u.ptr(Φ).ptr(T_β) \). 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 \( u \). This child corresponds to a unique vertex \( u′ \) in \( T_β \), which is a representative of connected component containing \( u \). Vertex \( u′ \) corresponding to \( u \) 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 \( T_β \). First the algorithm finds lowest common ancestor \( c \) of \( u′ \) and \( v′ \). Then, it invokes \text{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 \( u′ \) 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 \( T_β \). Vertex \( y \) is found similarly, using a call to \text{FindCut}(v, u′, v′, β, c). When \( k = 3 \) the algorithm reports vertices corresponding to \( u, x, y, v \) in \( T \). Otherwise, it proceeds recursively to find a \((k - 2)\)-hop path between inner vertices of \( β \) in \( Φ \) corresponding to \( x \) and \( y \).

The guarantees of the query algorithm are stated in the full version of the paper.

2.2 Navigating tree covers

To prove Theorem 1.2, we rely on tree cover theorems. Let \( ζ \) denote the number of trees in the cover and \( y \) the stretch of the cover; let \( M_ζ = (X, \delta_ζ) \) 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 \( G_ζ \) and a data structure \( D_ζ \). 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 \( D_ζ \) for the distance between \( u \) and \( v \) in \( T_ζ \) in \( O(1) \) time. (This step takes \( O(ζ) \) 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 \( O(κ) \) time using the result from Theorem 1.1.

3 FAULT TOLERANCE IN DOUBLING METRICS

In this section we strengthen the navigation scheme of Section 2 to achieve fault-tolerance in doubling metrics. We start with 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 2 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 \( δ_{H,F}(x, y) \leq t \cdot \delta_X(x, y) \), 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 \( H \), \( k \), 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 [9]). In what follows, we shall use \( P_T(u, v) \) to denote the path between leaves \( x \) and \( y \) in a rooted tree \( T \). We denote by \( T_x \) a subtree of \( T \) rooted at a vertex \( v \in T \).

Definition 3.1 (Robust Tree Cover). A robust \((y, ζ)\)-tree cover \( T \) for a metric \((X, \delta_X)\) is a collection \( ζ \) of \( ζ \) trees satisfying:

(1) For every tree \( T \in T \), 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 \( T \in T \) such that the path from \( x \) to \( y \) in \( T \) has weight at most \( \gamma \cdot \delta_X(x, y) \). We say that \( T \) covers \( x \) and \( y \).

Property (2) in Definition 3.1, which we call robustness, implies that we can obtain an (ordinary) tree cover by replacing any internal vertex of a tree \( T \in T \) 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 3.2.

Theorem 3.1. For any metric \((X, \delta_X)\) of doubling dimension \( d \) and parameter \( ϵ > 0 \), we can construct a robust \((1 + ϵ, e^{-O(d)})\)-tree cover \( T \) for \((X, \delta_X)\) in \( O_d(e)(n \log n) \) time.

The \( O_d(e) \), notation hides the dependency on \( d \) and \( ϵ \). The tree cover theorem by [17] 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 [17], 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 [17] 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.

3.1 Construction of fault-tolerant spanners with small hop-diameter

Theorem 3.2. Given an \( n \)-point metric \((X, \delta_X)\) of doubling dimension \( d \), a parameter \( ϵ > 0 \), and integers \( 1 \leq f \leq n - 2 \), and \( k \geq 2 \), we can construct an \( f \)-FT spanner with hop-diameter \( k \) and \( e^{-O(d)} n^2 ω_k(n) \) edges in \( O_d(e)(n \log(n) + f^2 ω_k(n)) \) time.

Proof. Let \( T \) be a robust \((1 + ϵ, e^{-O(d)})\)-tree cover constructed as in Theorem 3.1. For each tree \( T \in T \), we construct a graph \( H_T \) and then form an \( f \)-FT spanner \( H \) as \( H = ∪_{T \in FT} H_T \).

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 \( O(n\delta_k(n)) \) edges in \( O(n\delta_k(n)) \) time, denoted by \( K_T \), using the
algorithm of Solomon [99]. Note that edges in $T$ are unweighted. For every vertex $x \in T$, we choose a set $R(v)$ of (arbitrary) $f + 1$ points associated with leaves of $T_x$; if $T_x$ has strictly less than $f + 1$ leaves, $R(v)$ includes all the leaves. For every edge $(u, v) \in E(T)$, 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 $|E(H_T)| = O(f^2|E(K_T)|) = O(f^2nq_k(n))$. It then follows that $|E(H)| = |T'|(O(f^2nq_k(n)) = e^{-O(d)}nf^2q_k(n)$. Observe also by the construction that the running time to construct $H_T$ is $O(f^2nq_k(n))$. Thus, the running time to construct $H$ is $O_d(n \log(n)) + O(f^2nq_k(n))$, 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 $T'$ 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 $p_i \in R(v_i)$. Then, $P$ is a path in $H_T$ (and hence in $H$) of hop-diameter $k$. Furthermore, by property (2) in Definition 3.1 and the fact that $Q$ is a 1-spanner path, $P$ has stretch $(1 + e)$, as desired. $\square$

### 3.2 Construction of robust tree covers

In this section, we prove Theorem 3.1. Our construction follows the construction of tree covers of Bartal et al. [17]. An $r$-net of a metric space $(X, \delta_X)$ 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 3.2), which is the key to achieving the robustness of our tree cover. We first review the construction of Bartal et al. [17], and then describe how the pairing cover can be used to construct a robust tree cover.

The construction of Bartal et al. [17] 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 $(X, \delta_X)$. Each net $N_i$ is then partitioned into $\sigma = e^{-O(d)}$ well-separated sets $N_{i1}, \ldots, N_{i\sigma}$ in the sense that for every $x \neq y \in N_{i\alpha}$, $\delta_X(x, y) = \Omega(2^i/e)$ for any $i \in \{1, \ldots, \sigma\}$. (Step 2) They construct a collection of nets $\sigma \log(1/e)$ trees $T_{jp}$, where $j \in \{1, 2, \ldots, \sigma\}$ and $p \in \{0, 1, \ldots, \log(1/e) - 1\}$. Each tree $T_{jp}$ is constructed by considering levels $i$ of the net hierarchy such that $i \equiv p \mod \log(1/e)$ 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 $O(2^i/e)$ 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. [17] in two ways. In Step 1, we construct a cover (instead of a partition) of size $e^{-O(d)}$ for $N_i$ that has a pairing property: each point $x \in C_i$ in the cover has at most one point $y \in C_{ij}$ such that $\delta_X(x, y) \leq 2^i/e$; $x$ is said to be paired with $x$. (See Definition 3.2 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 the full version of the paper [69]. We say that a collection of subsets $C$ of a set $S$ is a cover for $S$ if $\cup_{C \in C} C = S$.

#### Definition 3.2 (Pairing Cover)

A cover $C_i$ of a $2^i$-net $N_i$ is a pairing cover if:

1. For every set $C \in C_i$ and every $x \in C$, there exists at most one point $y \neq x \in C$ such that $\delta_X(x, y) \leq 2^i/e$.
2. For every $x \neq y \in N_i$ such that $\delta_X(x, y) \leq 2^i/e$, there exists a set $C \in C_i$ such that both $x, y \in C$. 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 3.1 (Packing Lemma)

Let $P$ be a point set in a metric $(X, \delta_X)$ of doubling dimension $d$ such that for every $x \notin P$, $r < \delta_X(x, y) \leq R$. Then $|P| \leq (4R/r)^d$.

Step 1: Constructing a pairing cover of $N_i$. The construction has two smaller steps. First, we construct a well-separated partition $P_i$ of $N_i$ following Bartal et al. [17]. Then in the second step, we construct a pairing cover $C_i$ from $P_i$.

- **Step 1a.** Initially $P_i = \emptyset$. We consider each point $x \in N_i$ in turn, and if there exists a set $P \in P_i$ such that $\delta_X(x, y) > (3/e)2^i$ for every $y \in P$, then we add $x \notin P$. Otherwise, we add a new set $\{x\}$ to $P_i$. Let $\sigma_1 = |P_i|$.

- **Step 1b.** Let $\sigma_2 = \max_{x \in P_i} |\{y \in N_i : \delta_X(x, y) \leq 2^i/e\}|$. For each set $P \in P_i$, we construct a collection $C(P) = \{P_1, \ldots, P_{\sigma_2}\}$ of $\sigma_2$ sets as follows. For each $x \in P$, let $\{y_1, y_2, \ldots, y_{\sigma_2}\}$ be a sequence of all points (in arbitrary order) in $N_i$ that have distances at most $2^i/e$ from $x$. (Possibly, there could be strictly less than $\sigma_2$ such points, and in this case, we duplicate some points to get exactly $\sigma_2$ points in the sequence.) We then construct the set $P_j = \cup_{x \in P \in P_i} \{x, y_j\}$ 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 $C_i = \cup_{P \in P_i} C(P)$.

In the following lemma, we show that $C_i$ is a pairing cover of $N_i$. The proof is deferred to the full version of the paper [69].

#### Lemma 3.2

$C_i$ is a pairing cover of $N_i$ of size $e^{-O(d)}$.

Step 2: Constructing a robust tree cover $T$. Let $N_0 \supseteq N_1 \supseteq \ldots$ be the hierarchy of nets of $(X, \delta_X)$ 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 $1/(4e)$. For each net $N_i$, we construct a pairing cover $C_i$, and form a sequence $(C_{i1}, C_{i2}, \ldots, C_{i\sigma_2})$ of sets in $C_i$: here $\sigma_1$ is the size of $C_i$, which is $e^{-O(d)}$ by Lemma 3.2. For each $j \in \{1, 2, \ldots, \sigma_2\}$ and each $P \in C_{ij}$


\{0, 1, \ldots, [\log(1/e)] - 1\}, we construct a tree \(T_{j,p}\) and form the cover: \(\mathcal{T} = \{T_{j,p} : j \in \{1, 2, \ldots, \sigma_j\} \land p \in \{0, 1, \ldots, [\log(1/e)] - 1\}\). Clearly the size of the cover is \(O(\sigma_j \log(1/e)) = e^{-O(d)}\).

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 \(I = \{i : i \equiv p \mod [\log(1/e)]\}\) be the set of levels congruent to \(p\) modulo \([\log(1/e)]\). For each level \(i \in I\) from lower levels to higher levels, let \(C_j\) be the \(j\)-th sets in the sequence of the pairing cover \(C_i\). Let \(i' = i - [\log(1/e)]\), and \(F_{i'}\) be the collection of trees constructed at level \(i'\). \(F_0\) contains leaves of \(\mathcal{T}\) for each point \(x \in C_j\), let \(T_x \in F_x\) be the tree containing \(x\), and \(F_x \subseteq T_x\) be a collection of subtrees such that each tree \(T_x\) 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 \(F\). At the top level \(i_{\text{max}}\), if \(F_{i_{\text{max}}}\) 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 \(F_{i_{\text{max}}}\) 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 bound the diameter of trees in \(F_i\); the proof is deferred to the full version of the paper [69].

**Lemma 3.3.** Let \(T\) be a tree in \(F_i\), and \(\text{diam}(T)\) be the diameter of the set of points associated with leaves of \(T\). Then \(\text{diam}(T) \leq (1/\epsilon + 20)2^i\) when \(\epsilon \leq 1/12\).

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

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

\[
2^{i-2}/\epsilon < \delta_X(x, y) \leq 2^{i-1}/\epsilon.
\]

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 points of \(N_i\) closest to \(x\) and \(y\), respectively. By the triangle inequality and Equation (1), and since \(\epsilon \leq 1/12\), it holds that:

\[
\delta_X(p, q) \leq \delta_X(x, y) + 2 \cdot 2^i \leq (1/2 \epsilon + 2)2^i \leq 2^i/\epsilon
\]

\[
\delta_X(p, q) \geq \delta_X(x, y) - 2 \cdot 2^i > (1/4 \epsilon - 2)2^i > 0
\]

It follows from the second inequality in Equation (2) that \(p \neq q\).

Since \(\delta_X(p, q) \leq 2^i/\epsilon\), property (2) of paring cover, there exists a set \(C_j \subseteq C_i\) such that \(p\) and \(q\) are paired by \(C_j\). Let \(T_p, T_q\) be the trees in \(F_p\) and \(F_q\), \(F_p \subseteq F\) associated with \(p\) and \(q\) as described in the construction. Let \(T\) be the tree in \(F\) resulting from merging trees in \(\{T_p, T_q\} \cup F_p \cup F_q\) by the algorithm. Since \(\delta_X(x, p) \leq 2^i\), \(x\) 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 \(T, T_x, T_y\), 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:

\[
w(Q) \leq w(Q_x) + w(Q_y) + \delta_X(S(r_x), S(r)) + \delta_X(S(r), S(r_y))
\]

In the following claim, we bound the weight of each term in Equation (3); the proof is deferred to the full version of the paper [69].

**Claim 3.1.** \(\max\{w(Q_x), w(Q_y)\} \leq (2 + 40\epsilon)2^i + \delta_X(S(r_x), S(r)) + \delta_X(S(r_y), S(r)) \leq \delta_X(x, y) + 4(5 + 60\epsilon)2^i\).

By Equation (3) and Claim 3.1, we have that:

\[
w(Q) \leq 2(2 + 40\epsilon)2^i + \delta_X(x, y) + 4(5 + 60\epsilon)2^i
\]

\[
\leq \delta_X(x, y) + O(1)2^i
\]

\[
\leq \delta_X(x, y) + O(1)4\epsilon \delta_X(x, y) \quad \text{(by Equation (1))}
\]

\[
= (1 + O(\epsilon))\delta_X(x, y)
\]

### 3.3 Deriving a fault-tolerant navigation (and routing) scheme

In the navigation scheme presented in Section 2, we did not exploit a crucial property of the tree cover theorem in doubling metrics [17]: 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 2 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 2, 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 the full version of the paper [69]; 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 \(f\)).

**ACKNOWLEDGMENTS**

The fourth-named author thanks Ofer Neiman for helpful discussions. The first-, third-, and fourth-named authors are supported by the Israel Science Foundation (ISF) grant No. 1991/1, and by a grant from the United States-Israel Binational Science Foundation (BSF), Israel, and the United States National Science Foundation (NSF). The second-named author is supported by National Science Foundation under Grant No. CCF-2121952.
