Online Metric TSP: Beyond the $\sqrt{n}$ Barrier

Yossi Azar
Department of Computer Science, Tel Aviv University, Israel

Debmalya Panigrahi
Department of Computer Science, Duke University, Durham, NC, USA

Or Vardi
Department of Computer Science, Tel Aviv University, Israel

Abstract

We study an online variant of the Traveling Salesperson Problem (TSP) in which n points arrive sequentially and must be inserted into an evolving tour. In the classical setting where arbitrary insertions are allowed, an $O(\log n)$-competitive algorithm has been known since the 1970s (Rosenkrantz, Stearns and Lewis 1977, Imase and Waxman 1991). Recently, Abrahamsen, Bercea, Beretta, Klausen, and Kozma [ESA 2024] introduced online metric TSP, a stricter model in which each arriving point must be assigned to a distinct cell of an array of size $m \geq n$, with the final tour order induced by the non-empty cells; the parameter m captures the space usage of the algorithm.

When $m = 2^n$, this model recovers arbitrary insertions and therefore admits an $O(\log n)$-competitive algorithm. In contrast, when $m = n$, i.e., when each point's position is fixed on arrival, Bertram [7] recently showed that the competitive ratio is $\Theta(\sqrt{n})$. We investigate the tradeoff between space usage and competitiveness between these extremes. We note that this tradeoff was previously explored by the authors in [6] for the online sorting problem, which is the special case of online metric TSP on a line metric.

Our main result is a deterministic online metric TSP algorithm using $m = (1+\varepsilon)n$ space that achieves a competitive ratio of $O(\log^3 n/\varepsilon)$, for any $\varepsilon \leq 1$. In particular, increasing the space from n to $2n$ improves the competitive ratio from $\Theta(\sqrt{n})$ to $O(\log^3 n)$. We complement this with a lower bound showing that for $m = n^{1+\varepsilon}$, any deterministic algorithm has a competitive ratio $\Omega(1/\varepsilon)$, for all $\varepsilon \geq \Omega(\log\log n/\log n)$. Consequently, even with $m = O(n \cdot \text{polylog}(n))$, deterministic algorithms cannot achieve a constant competitive ratio.

2012 ACM Subject Classification Theory of computation → Online algorithms; Theory of computation → Routing and network design problems; Theory of computation → Approximation algorithms analysis

Keywords and phrases Online algorithms, competitive analysis, metric TSP, space-competitiveness tradeoff, routing problems

Digital Object Identifier 10.4230/LIPIcs.ICALP.2026.18

Category Track A: Algorithms, Complexity and Games

Funding *Debmalya Panigrahi*: D. Panigrahi was supported in part by NSF grants CCF-1955703 and CCF-2329230.

1 Introduction

The (metric) traveling salesman problem (TSP) is a classic problem in combinatorial optimization where the goal is to find the minimum-length tour covering a set of n points in a metric space. In a classic result in approximation algorithms, Christofides [10] (and independently, Serdyukov [25]) gave a 3/2-approximation to this problem, which remained the state-of-the-art for almost 50 years before being eventually improved to $3/2 - \varepsilon$ (for a small $\varepsilon > 0$) in a remarkable recent result of Karlin, Klein, and Oveis Gharan [16]. On the

hardness side, Papadimitrou and Yannakakis [23] showed APX-hardness for this problem as a consequence of the PCP theorem [3]; the current record on the lower bound stands at 123/122 [17]. Resolving this gap remains a central open question in approximation algorithms.

What if the set of points that we need to connect is not known in advance but arrives online? In each online step, the goal is to insert the newly arriving point in the existing order in a way that minimizes the length of the tour following that order. In the classical version of this problem, where the arriving point can be inserted anywhere in the existing order, $O(\log n)$-competitive algorithms are known since the 1970s [24, 14]. Recently, Abrahamsen, Bercea, Beretta, Klausen, and Kozma [2] introduced a stricter model called online metric TSP: here, the online algorithm has to map each arriving point to a distinct cell of an array of size $m \geq n$, and the eventual order is given by the sequence of non-empty cells in the array. (We refer to the parameter m as the space usage of the algorithm.) If $m = 2^n$, this setting recovers arbitrary insertions thereby admitting a competitive ratio of $O(\log n)$. At the other extreme, when $m = n$, the online algorithm commits to an irrevocable position of the new point in the eventual order immediately on arrival. For this setting, Bertram [7] recently gave a deterministic algorithm with a competitive ratio of $O(\sqrt{n})$, matching previously known lower bounds even allowing randomization [1, 2]. The exponential gap in the competitive ratio between the two extremes raises a natural question about what lies in between, namely: *how does the competitive ratio of online metric TSP depend on the space usage m?*

We note that this tradeoff between competitive ratio and space usage was previously explored by the authors for the *online sorting* problem [6] introduced by Aamand, Abrahamsen, Beretta, and Kleist [1], which is the special case of online metric TSP on a line metric. We will describe later that the solution proposed by [6] for online sorting does not generalize naturally to online metric TSP; indeed, the algorithm given in this paper is a different, and arguably more natural, solution for the online sorting problem as well. Moreover, the lower bound in this paper is applicable to the online sorting problem as well. Therefore, in addition to studying online metric TSP with $m > n$, this paper makes contributions to the online sorting problem for the case of $m > n$ as well.

1.1 Our Results

In this paper, we answer this question by showing a poly-logarithmic competitive ratio for online metric TSP even when m only mildly exceeds n. Note that this is an exponential improvement over the case $m = n$. Specifically, we show:

▶ **Theorem 1.** *There is a deterministic online metric TSP algorithm that uses $m = (1 + \varepsilon)n$ space and achieves a competitive ratio of $O(\log^3 n/\varepsilon)$, for any $\varepsilon \in [O(\log n/n), 1]$.*

This implies, for instance, that with $n/\operatorname{polylog}(n) = o(n)$ extra space, the competitive ratio of online metric TSP improves from $\Theta(\sqrt{n})$ to $\operatorname{polylog}(n)$. Prior to our work, the only cases for which a sub-polynomial competitive ratio was known had a space usage of $m = 2^n$. It is interesting to ask whether the extra space can be reduced even further to $n^{1-\delta}$ for any constant $\delta > 0$, or whether there is a lower bound on the extra space required to obtain a $\operatorname{polylog}(n)$ competitive ratio. We leave these are interesting open questions.

Can the competitive ratio be improved further to $O(1)$? To the best of our knowledge, such a result is not known even for unlimited space, but there is no lower bound either, even for $m = O(n)$. We complement our upper bound by showing that it is impossible for a deterministic algorithm to achieve a competitive ratio of $O(1)$, unless m polynomially exceeds n. We show:

▶ **Theorem 2.** *For $m = n^{1+\varepsilon}$, the competitive ratio of any deterministic online metric TSP algorithm is $\Omega(1/\varepsilon)$, for any $\varepsilon > \Omega(\log\log n/\log n)$.*

This implies that using even $m = O(n \cdot \text{polylog}(n))$, the best competitive ratio one can potentially achieve deterministically is $O(\log n / \log \log n)$.

Remark. We note that the lower bound construction that we will give later to prove Theorem 2 is actually for the online sorting problem, which is a special case of onine metric TSP. In [6], the authors showed that using $m = O(n \log^2 n)$, one can achieve a competitive ratio of $O(1)$ for the online sorting problem. However, this result does not contradict the lower bound in this paper since the previous result assumed that the algorithm knows the value of opt from the outset, while the lower bound is for the case where the value of opt is unknown to the algorithm. It remains open whether an $O(1)$-competitive algorithm exists for the online metric TSP problem for the case of known opt.

1.2 Related Work

Online versions of TSP have been studied since the 1970s when Rosenkrantz, Stearns, and Lewis [24] introduced the class of constructive insertion heuristics to iteratively build a TSP tour by inserting new vertices while preserving the relative order of previous vertices. They gave a 2-approximate algorithm when the insertion order is chosen by the algorithm, and an $O(\log n)$-competitive one when it is chosen by an adversary. The latter result also follows from later work by Imase and Waxman on the online Steiner tree problem [14], where the online algorithm additionally commits to the edges being used to connect the new vertex to the previous vertices. This latter commitment also creates a difference between the two problems: while $O(\log n)$-competitiveness is tight for online Steiner tree, it is not known to be tight for creating an online TSP tour. For the latter bound, the only lower bounds known are for specific insertion strategies, such as for the greedy strategy due to Azar [5]. (See also [21] for a recourse version of online MST and TSP where the algorithm commits to edges but these can be partially revoked using a limited budget.)

Later, a different online version of TSP was proposed, where a server has to serve a sequence of requests arriving online on a metric space and the goal is to minimize the makespan, i.e., the total time to serve all requests [4]. This problem differs from prior work (and from online metric TSP) in that there is a real notion of time with release dates for the server requests. We note that several results have been obtained in this line of work, both for general metric spaces [11, 9, 19, 18, 20, 15] and also specifically for the line metric [8].

Online metric TSP was introduced by Abrahamsen, Bercea, Beretta, Klausen, and Kozma [2] as a generalization of the online sorting problem [1] which restricts the points to a line. It is known that online sorting has a tight competitive ratio of $\Theta(\sqrt{n})$ when $m = n$ [1, 2], and that this bound continues to hold for online metric TSP with $m = n$ [7]. For $m > n$, an $O(1)$-competitive algorithm is known for $m = O(n \cdot \text{polylog}(n))$ when opt is known, and an $O(\log n)$-competitive algorithm is known for $m = (1 + \varepsilon)n$ (for $\varepsilon > 0$) when opt is unknown [6]. Note that the lower bound result in the current paper also applies to online sorting; hence, it shows that this distinction between known and unknown opt is necessary. Further results on the online sorting problem appear in [22, 13, 12].

1.3 Our Techniques

We now give an overview of our main techniques. We first describe our upper bound (Theorem 1) and then outline the lower bound construction (Theorem 2).

Upper Bound. Our starting point is the previous result for the online sorting problem with $m > n$ [6]. This paper gave an $O(\log^3 n/\varepsilon)$-competitive algorithm using $m = (1+\varepsilon)n$ space, based on a novel data structure called an *elementary tree*. The core idea is to maintain a collection of binary trees whose leaves, ordered from left to right, correspond to array cells, and which guide the insertion of arriving elements.

We briefly recall how elementary trees are used to insert elements. The algorithm dynamically and sequentially labels internal tree nodes with dyadic intervals, whose lengths correspond to the heights of the nodes. For example, if the element range is $[0, 1]$, the root is labeled $[0, 1]$, but its two children may both be labeled $[0, 1/2]$, or $[1/2, 1]$, or one of each. This continues down the tree with the interval length halving at each level. The label of a node v specifies the range of element values that may occupy the array cells corresponding to the leaves of the subtree rooted at v. Crucially, except at the root, these labels are assigned dynamically and depend on the insertion sequence.

When a new element arrives, the algorithm performs a depth-first search to find a node whose interval label contains the element and which has an unlabeled child. That child is then labeled with the unique dyadic interval at that level containing the element, and the procedure recurses down the tree until the element is placed at a leaf. This approach fundamentally exploits the shared linear order of the line metric and the array, a property that does not extend to general metric spaces.

Extending this framework to general metrics faces several obstacles. First, unlike dyadic intervals, recursive covers (by balls of decreasing radii) of general metric spaces typically involve overlapping regions, which violates laminarity. (Note that laminarity requires that any two of the covering sets should either be disjoint or one should be contained in the other.) As a result, a point may not belong to a unique ball at a given level, introducing ambiguity that the insertion algorithm must resolve. Second, while dyadic intervals split into exactly two subintervals at each level, a ball in a general metric space may require many smaller balls to cover it at the next scale, with the number governed by the doubling dimension. Since elementary trees are binary, this creates a mismatch between the recursive structure of the metric space and the tree arity.[1] Increasing the arity is also generally infeasible, as it would lead to excessive space usage. Finally, there is a more fundamental disconnect: while dyadic intervals inherit a natural left-to-right order from the line metric compatible with the array, there is no canonical linear ordering of balls of the same radius in a general metric space.

Metric Ball-Cover (MBC) Tree. To overcome these difficulties, we introduce a new data structure in this paper called a *metric ball-cover tree* (MBC tree). Our first step is to construct a *dynamic* recursive cover of the metric space using balls of geometrically decreasing radii. Unlike dyadic intervals on the line that are statically defined, the balls in our cover are defined based on the arrival sequence. When a new point arrives, the algorithm identifies the highest level at which it is not already covered by an existing ball and creates balls of progressively smaller radii from that level onward, all centered at the new point.

Given this (dynamically evolving) ball-cover of the metric space, an MBC tree is a binary tree where the nodes are *dynamically labeled* by points in the metric space. Labeling a node v at height h with a point p in the metric space commits the array cells corresponding to the leaves under v in the MBC tree to points in a ball of radius r centered at p, where r is a fixed function of h.

[1] In fact, the doubling dimension can be arbitrarily large for a general metric space.

This brings us to our main challenge: we need to define the rules of labeling an MBC tree culminating in the insertion of a new point in the array. A natural attempt would be to reuse the depth-first insertion procedure of [6], but as we show in Appendix B, this approach fails to preserve the desired guarantees in general metric spaces. Instead, we introduce a new *bottom-up* labeling strategy. To motivate it, consider the risk associated with labeling a node v by a ball of radius r centered at a newly arrived point p. Since labels are irrevocable, this commits the entire subtree under v to points within that ball, even though only a single point has been observed so far. If no other points ever fall into this ball, the remaining array cells in that subtree are wasted.

This phenomenon is significantly more severe in general metric spaces than on the line. Suppose we call a subtree that receives only a single insertion a *wasted* subtree. Note that each ball/interval can only correspond to a single wasted subtree, since any subsequent insertion in the ball/interval would not label a new node at the same level. For dyadic intervals on the line, this is a strong property: it immediately implies that the total wasted space is cumulatively no more than the size of the subarray under a single elementary tree, since each dyadic interval splits into exactly two subintervals at the next level. Indeed, by constructing elementary trees with εn leaves, the algorithm of [6] ensures that the total wasted space is εn.

In contrast, in a general metric space, a ball may split into many smaller balls at the next level. Since the tree remains binary, this can cause the total wasted space to grow well beyond the size of a single tree. Increasing the tree arity to match the doubling dimension does not resolve this issue, as it merely coalesces the wasted space to a larger subarray that now corresponds to the leaves of an elementary tree of fixed height.

To overcome this bottleneck, we adopt a frugal allocation strategy in MBC trees: we label the *deepest* unlabeled node that can legitimately be assigned to the new point. This bottom-up approach minimizes the amount of space committed based on limited evidence. However, abandoning the depth-first insertion procedure in [6] necessitates a fundamentally new analysis of both space usage and competitive ratio of the algorithm. We describe the MBC tree data structure and the online metric TSP algorithm in Section 2. For simplicity, we first assume that the optimal value opt is known to the algorithm. Under this assumption, we obtain an $O(\log^2 n/\varepsilon)$-competitive algorithm using $m = (1+\varepsilon)n$ space. Removing the assumption that opt is known incurs an additional $O(\log n)$ factor in the competitive ratio, following techniques similar to [6]. This extension is presented in Appendix A, completing the proof of Theorem 1.

Lower Bound. Our lower bound construction (Theorem 2) applies to online sorting [1], the special case of online metric TSP on the line. A key feature of the construction is that the optimal value opt is *not* known to the online algorithm. While this assumption is standard in online metric TSP, some prior work on online sorting assumes that all elements lie in $[0, 1]$ and that both endpoints appear in the instance, effectively fixing $\mathsf{opt} = 1$. Our lower bound does not apply in this setting; indeed, when opt is known, $O(1)$-competitive algorithms with $m = O(n \operatorname{polylog} n)$ space are known for online sorting [6].

Recall that in online sorting, elements arrive online and must be placed into an array of size m so as to minimize the sum of absolute differences between consecutive non-empty cells. The lower bound proceeds in multiple phases. Suppose in the first phase, the adversary presents batches of elements that are evenly spaced within the interval $[1, 2]$. The algorithm's behavior can be summarized by the structure of the *holes*, i.e., the empty regions between occupied array cells. If the algorithm keeps these holes small, it is being frugal with space,

but the adversary responds by increasing the batch sizes while keeping the value range fixed. Maintaining small holes under such dense arrivals forces the algorithm to incur a large competitive ratio, as it repeatedly pays cost at the boundaries between batches.

Eventually, to control its competitive ratio, the algorithm must create sufficiently large holes. At this point, the adversary switches strategy and begins presenting elements in a much larger range $[k, 2k]$, for large k. This effectively turns the earlier elements into zeros relative to the new scale. Since the algorithm lacks sufficient unused space, it must reuse the previously created large holes. Inserting large values into these holes immediately incurs cost $\Theta(k)$ at the boundaries, which matches the order of opt at this stage. If this happens in too many holes, the algorithm again suffers a large competitive ratio.

Our construction, given in full in Section 3, iterates this process over multiple rounds and refines the idea presented above to obtain the lower bound stated in Theorem 2.

2 Online Metric TSP using Metric Ball-Cover (MBC) Trees

We introduce a new data structure called a Metric Ball-Cover (MBC) tree. In this section, we describe MBC trees and show how they are used to solve online metric TSP.

An MBC tree of height H is an array data structure of size 2^H that has an associated complete binary tree of height H. We emphasize that the tree is *virtual* in the sense that it is used to define how points are inserted into the array, but the actual data structure is just an array. The leaves of the tree, at height 0, are each associated with a unique array cell, enumerated from left to right. The non-leaf nodes of the tree do not correspond to array cells, and are only used in the algorithm to decide the location where an arriving point in inserted in the array. Thus, all insertions of points are at the leaves of the tree. The root of the tree is at height H, its children are at height $H-1$, their children at height $H-2$, and so on.

Node Labeling. Each node v at height h is associated with a ball $B(c_v, r_h)$.

- **Radius (r_h):** The radius is deterministic and depends only on the height. We set $r_h := R \cdot 2^{h-H}$, where R is a scaling factor denoting the radius of the ball at the root, which will be defined when the tree is created.
- **Center (c_v):** The center is determined dynamically. We say a node in the tree is *marked* if a center c_v has been assigned to it. Otherwise, the node is *unmarked*. Initially, all nodes are unmarked. A marked node v is *partial* if it has one marked child and one unmarked child. For a marked node v, we say that a point x_t is *admissible* at v if $d(x_t, c_v) \leq r_h$ where h is the height of v.

An example of an MBC tree is shown in Figure 1.

The Data Structure. The overall data structure comprises a sequence of identical MBC trees $A_0, A_1, \ldots$, each of fixed height $H := \lfloor \log(\varepsilon n) - \log\log n \rfloor$,[2] and with the roots labeled with balls of radius R. The number of MBC trees is not fixed in advance; the sequence grows dynamically as needed.

▶ Remark 3. We assume $\varepsilon \geq \log n/n$, ensuring that the height H of the MBC trees is non-negative.

[2] All logarithms are with base 2 unless otherwise mentioned.

2.1 The Online Metric TSP Algorithm

We assume knowledge of the value of the optimal solution opt in describing our algorithm in this section. In Appendix A, we will relax this assumption at the cost of an additional $O(\log n)$ factor in the competitive ratio. We use opt to set the radius of the balls at the roots of the MBC trees in our data structure, i.e., $R := \mathsf{opt}$ in our data structure.

We now describe the online algorithm that inserts a point x_t into the array using the sequence of MBC trees. The algorithm searches for a **partial, admissible** node of the **lowest possible height** across all existing MBC trees.

- If such a node v is found, we call $\texttt{insert}(v, x_t)$, which is described below.
- If no such node exists, we create a new MBC tree A_{new}, add it to the sequence, and call $\texttt{insert}(\text{root}(A_{\text{new}}), x_t)$.

Local Insertion ($\texttt{insert}(v, x)$). The procedure starts by identifying a node v' as follows:

- If v is a *partial node*, v' is the *right child* of v.
- Otherwise, v is the root of a new tree; in this case, $v' := v$.

Starting from v', the algorithm traverses down to a leaf by repeatedly choosing the *left* child. For every node w on this path (including v'), we mark w with center $c_w = x$ and radius r_h where h is the height of w. Finally, x is stored at the leaf u at the end of this path.

In Figure 2, we show the insertion of a new point in an MBC tree.

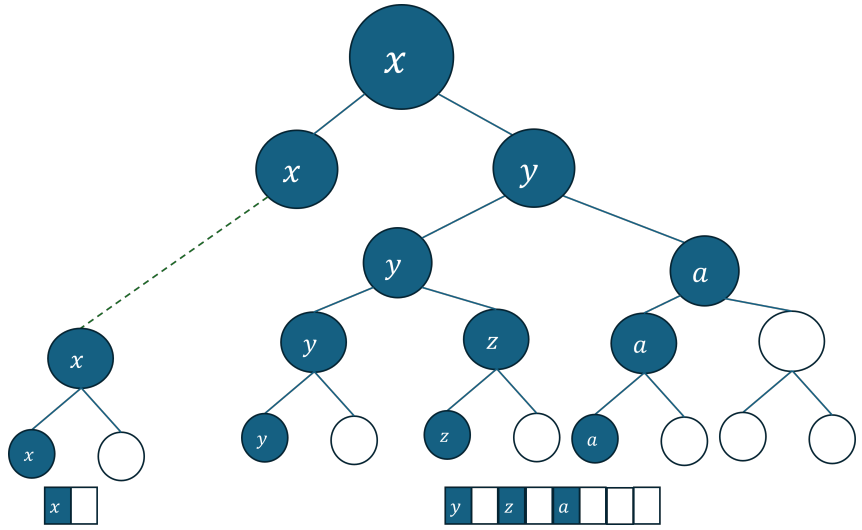
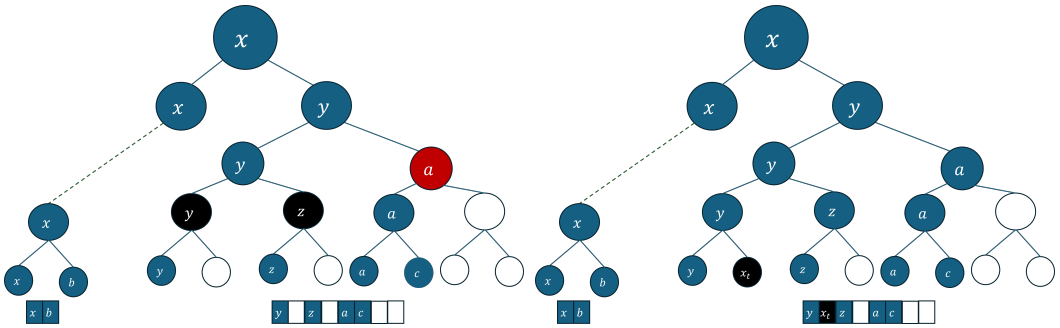


Figure 1 Illustration of an MBC tree. The colored cells in the array represent occupied cells. Similarly, the colored nodes in the tree represent marked nodes.

2.2 Space Usage of the Algorithm

We start by showing that the size of the array used by the online metric TSP algorithm is at most $(1 + \varepsilon)n$. For this, we bound the number of unused cells in the array. Note that each partial node has an unmarked child with an unused subarray under it. Conversely, the unused space in the array can be partitioned into subarrays that are subtended by the unmarked children of partial nodes. Therefore, it suffices to bound the cumulative unused space under unmarked children of partial nodes in the data structure.



(a) Before the insertion of x_t. **(b)** After the insertion of x_t.

Figure 2 Insertion of a point x_t in an MBC tree. The filled parts of the array represent occupied regions. The colored tree nodes represent marked nodes. On the left figure, suppose the nodes colored black and red are the partial nodes whose balls contain the new point x_t. Since the red node is at a higher level than the black nodes, only the black nodes are candidate locations for inserting x_t. On the right figure, we show the MBC tree and the array after inserting x_t.

Our main claim is the following:

▷ Claim 4. For all partial nodes at a fixed height h, the cumulative unused space under their unmarked children is at most 2^{H-1}.

In order to prove this claim, we will use the following important property.

▷ Claim 5. Let u and v be two partial nodes (nodes with exactly one marked child) at the same height h, with centers c_u and c_v. Then $d(c_u, c_v) > r_h$.

Proof. Without loss of generality, assume u was marked before v (created by the insertion of a point x_v). When the algorithm attempted to insert x_v, it checked all existing marked nodes at height h, including u. Since u is partial, it had an available (unmarked) child, which means the insertion into u would have succeeded if x_v were admissible.

The fact that x_v was not inserted in the subtree under u, while the insertion algorithm prefers the partial node at the lowest possible height, implies that x_v is inadmissible at u. Therefore, $d(x_v, c_u) > r_h$. Since $c_v = x_v$, it follows that $d(c_v, c_u) > r_h$. ◁

Using Claim 5, we now prove Claim 4.

Proof of Claim 4. Let N_h be the number of partial nodes at height h. By Claim 5, their centers are N_h points that are pairwise separated by distance $> r_h$. Since the optimal TSP tour (of cost opt) visits all points, it must connect these N_h centers incurring a cost $> (N_h - 1) \cdot r_h$. Substituting $r_h = R \cdot 2^{h-H} = \mathsf{opt} \cdot 2^{h-H}$, we get $N_h \leq 2^{H-h}$.

Since each partial node at height h has an unmarked child node at height $h-1$ that has 2^{h-1} unused array cells under it, the total unused space in the array is at most

$$N_h \cdot 2^{h-1} \leq 2^{H-h} \cdot 2^{h-1} = 2^{H-1}.$$ ◁

We can now derive the space usage of the online algorithm from Claim 4.

▶ **Lemma 6.** *The unused space in the array is at most εn at any time. As a consequence, the array uses at most $(1+\varepsilon)n$ space after inserting all n points.*

Proof. By Claim 4, it follows that the total empty space is at most:

$$H \cdot 2^{H-1} \leq \frac{\varepsilon n}{\log n} \cdot \lfloor \log(\varepsilon n) - \log\log n \rfloor \leq \varepsilon n.$$ ◀

2.3 Competitive Ratio of the Algorithm

We want to prove the following bound:

▶ **Lemma 7.** *The total cost of the solution produced by the algorithm is* $O\left(\frac{\log^2 n}{\varepsilon}\right) \cdot \mathsf{opt}$.

To prove this lemma, we first establish some important properties of the online algorithm.

▷ Claim 8. Let u, v be two marked nodes in the MBC tree such that v is the parent of u. Then, the point c_u at the center of the ball labeling u must be admissible at v.

Proof. There are two possibilities. The first is that $c_v = c_u$. In this case, the claim trivially holds. Otherwise, the labeling at u was done by the procedure `insert`(v, c_u), which can only be run if c_u is admissible at v. ◁

▷ Claim 9. Let x be a point that is stored in the array cell at a leaf node u of an MBC tree, and let v be a node of height h which is an ancestor of u. Then $d(x, c_v) \leq \sum_{k=1}^{h} r_k \leq 2 \cdot r_h$.

Proof. We will prove the claim by induction on h. For the base case of $h = 1$, we have from Claim 8 that x is admissible at v. Therefore, $d(x, c_v) \leq r_1$.

Let $v_0 = u, v_1, \ldots, v_{h-1}, v_h = v$ denote the vertices on the path from a leaf u to its ancestor v at height h. Let z denote $c_{v_{h-1}}$. By the inductive hypothesis, we have $d(x, z) \leq \sum_{k=1}^{h-1} r_k$. Furthermore, using Claim 8, we know that $d(z, c_v) \leq r_h$. Therefore, by the triangle inequality,

$$d(x, c_v) \leq d(x, z) + d(z, c_v) \leq \sum_{k=1}^{h} r_k \leq 2 \cdot r_h.$$ ◁

Using the above claim, we establish a bound on the total cost within an MBC tree.

▶ **Lemma 10.** *Suppose a set of points is inserted into an MBC tree A of height H. Then, the total cost of the points in A is at most* $4H \cdot \mathsf{opt}$.

Proof. Let x_i, x_{i+1} be a pair of points that occupy consecutive non-empty cells in the array (the leaves of A). Let v denote the least common ancestor of x_i and x_{i+1} in A, and let v_ℓ, v_r be the left and right children of v.

By the structure of the tree, x_i is the rightmost non-empty leaf in the subtree rooted at v_ℓ, and x_{i+1} is the leftmost non-empty leaf in the subtree rooted at v_r. By Claim 9, we have $d(x_i, c_v) \leq 2r_h$ and $d(c_v, x_{i+1}) \leq 2r_h$. Therefore, by the triangle inequality, we get

$$d(x_i, x_{i+1}) \leq d(x_i, c_v) + d(c_v, x_{i+1}) \leq 2r_h + 2r_h = 4r_h.$$

We charge this distance $d(x_i, x_{i+1})$ to the node v. Every internal node in the MBC tree is charged at most once. Summing over all heights $h = 1 \ldots H$, we get that the total cost is at most

$$\sum_{h=1}^{H} 2^{H-h} \cdot 4(\mathsf{opt} \cdot 2^{h-H}) = \sum_{h=1}^{H} 4 \cdot \mathsf{opt} = 4H \cdot \mathsf{opt}.$$ ◀

We now return to the proof of Lemma 7.

Proof of Lemma 7. By Lemma 6, the total space used by the algorithm is at most $(1 + \varepsilon)n$. The size of the subarray under a single MBC tree of height H is

$$2^H = 2^{\lfloor \log(\varepsilon n) - \log\log n \rfloor} \geq 2^{\log(\varepsilon n) - \log\log n - 1} = \frac{\varepsilon n}{2 \log n}.$$

Therefore, the number of MBC trees, denoted k, is at most

$$k \leq \frac{(1+\varepsilon)n}{\frac{\varepsilon n}{2\log n}} = 2\left(1+\frac{1}{\varepsilon}\right)\log n \leq \frac{4\log n}{\varepsilon}.$$

We now sum the costs. The total cost consists of the cost within each MBC tree and the cost incurred between MBC trees.

1. Cost within MBC trees. By Lemma 10, the cost of a single MBC tree is at most $4H \cdot \mathsf{opt}$. Since $H \leq \log(\varepsilon n) \leq \log n$, the cumulative internal cost across all k trees is at most:

$$k \cdot (4H \cdot \mathsf{opt}) \leq \frac{4\log n}{\varepsilon} \cdot (4\log n \cdot \mathsf{opt}) = \frac{16\log^2 n}{\varepsilon} \cdot \mathsf{opt}.$$

2. Cost between MBC trees. We are left with the cost incurred *between* MBC trees, i.e., by the rightmost point in A_i and the leftmost one in A_{i+1}. For any pair of points, the distance is at most opt. Thus, the total cost between trees is at most

$$k \cdot \mathsf{opt} \leq \frac{4\log n}{\varepsilon} \cdot \mathsf{opt}.$$

Adding these two types of cost, we get that the total cost is at most

$$\frac{16\log^2 n}{\varepsilon} \cdot \mathsf{opt} + \frac{4\log n}{\varepsilon} \cdot \mathsf{opt} = O\left(\frac{\log^2 n}{\varepsilon}\right) \cdot \mathsf{opt}. \qquad \blacktriangleleft$$

3 Lower Bound for Online Sorting and Online Metric TSP

Recall from Section 1 that our lower bound that establishes Theorem 2 for online metric TSP is actually for the online sorting problem, i.e., on the line metric, Moreover, the lower bound explicitly assumes that the algorithm does not know the value of opt, and therefore, does not apply to prior results in online sorting that assume knowledge of opt.

Before proceeding further, we restate Theorem 2 for the online sorting problem, and also change notation in a way that will make it easier to describe the construction. The reader can easily verify that the theorem below implies Theorem 2.

▶ **Theorem 11.** *Consider the online sorting problem on n elements. Any deterministic online sorting algorithm that is $c/16$ competitive requires at least $n^{1+1/(2c)}/5$ space, for any $c \leq \frac{1}{6} \cdot \frac{\log n}{\log\log n}$.*

We will assume the terminology of an *adversary* and an arbitrary deterministic *algorithm*. In every step of the online instance, the adversary reveals a batch of new elements, which the algorithm inserts in empty cells of the array. This sequence repeats itself until the adversary decides to end the instance.

Epochs and Phases. Overall, the online steps are organized in a sequence of *epochs*. Conceptually, the adversary achieves two properties in each epoch. First, it increases the range of elements from the previous epoch by a multiplicative factor. Second, because of the change in the overall range, all elements in the previous epochs become much smaller than all elements in the current range. In effect, this allows us to treat the elements in previous epochs as (being close enough to) 0, which helps simplify the analysis. More precisely, the elements presented by the adversary in the ith epoch are in the range $[2^{2i-1}, 2^{2i}]$. Note that by induction, all elements in the previous $i-1$ epochs are in the range $[2^1, 2^{2i-2}]$, which are at least 2^{2i-2} far from every element in epoch i.

An epoch is further subdivided into a sequence of *phases*. In each phase, the adversary presents a batch of elements, which are inserted by the algorithm in the array. We now describe the jth phase of the ith epoch. As stated previously, the range of elements in this epoch is $[2^{2i-1}, 2^{2i}]$. Let us call this range R_i. For $j = 0$, the adversary presents 2 elements, 2^{2i-1} and 2^{2i}. Subsequently, for every $j \geq 1$, the adversary presents a batch of $n^{j/c} - n^{(j-1)/c}$ elements *that are all distinct* such that the elements in all phases up to the jth one are *equally spaced in the range* R_i. In particular, the kth element of the ith epoch at the end of the jth phase is given by $2^{2i-1} + k \cdot \frac{2^{2i-1}}{n^{j/c}}$ where k ranges from 0 to $n^{j/c}$. Note that any two consecutive elements in this epoch differ in their values by $\frac{2^{2i-1}}{n^{j/c}}$. In other words, the jth phase supplies the missing elements such that the complete set accumulated by the end of the phase is given by $2^{2i-1} + k \cdot \frac{2^{2i-1}}{n^{j/c}}$, where k ranges from 0 to $n^{j/c}$ (We give a pictorial depiction of the elements arriving in different phases of an epoch in Figure 3.)

Figure 3 Distribution of elements arriving in different phases of epoch i. Initially, the two (blue) elements at the ends of the range for the current epoch arrive. The next two phases (yellow and red respectively) denote subsequent arrivals, illustrating the distribution of the elements over time as the epoch progresses.

Real and Vacuous Holes. This brings us to the crucial decision taken by the adversary after the algorithm inserts the elements presented in a phase. The adversary can perform three possible actions:

- it can continue to the next phase of the current epoch,
- it can terminate the current epoch and go to the first phase of the next epoch, or
- it can terminate the instance altogether.

To describe the conditions for this decision, we need to define the notion of holes in the array:

A *hole* is a set of contiguous cells in the array all of which are vacant.

Initially, the whole array is a single hole. Over time, as the array fills up, this single hole gives rise to multiple holes. For notational convenience, we extend this terminology to say that two consecutive array cells that are both occupied also have a *vacuous* hole between them. In contrast, a hole that comprises an actual non-empty sequence of vacant array cells is called a *real* hole.

It will be useful to define categories of holes, both vacuous and real. First, we categorize holes into three types based on the elements at their two ends:

- *Mixed Holes.* These are holes where one end of the hole is occupied by an element from a previous epoch and the other end is occupied by an element from the current epoch.
- *New Holes.* These are holes where both ends of the hole are occupied by elements from the current epoch.
- *Old Holes.* These are holes where both ends of the hole are occupied by elements from a previous epoch.

Second, orthogonally to the previous categorization, we group holes based on their size:

- *Large Holes.* These are holes that contain at least $n^{1/(2c)}$ (vacant) array cells. (By definition, all large holes are real holes.)
- *Small Holes.* These are holes that contain fewer than $n^{1/(2c)}$ (vacant) array cells.

We define the phase of a hole based on its endpoints: a hole is considered to be *created in phase j* of the current epoch if at least one of the elements defining its ends belongs to phase j.

After the algorithm inserts the elements in the jth phase of the ith epoch, the adversary makes its next move according to the current state of the array. In particular, it uses the following conditions, which are checked in order:

- Condition 1: If the array contains at least c mixed holes (including both real and vacuous holes), then the adversary terminates the instance.
- Condition 2: If the value of j reaches $c/2+1$, then the adversary terminates the instance.
- Condition 3: If the array contains at least $n^{j/c}/2$ large new holes, then the adversary terminates the current epoch and goes to the first phase of the $(i+1)$st epoch.
- If none of the above conditions hold, then the adversary continues with the $(j+1)$st phase of the current epoch.

To establish the lower bound in Theorem 11, we show two facts: first, that the instance ends with at most n elements, and second, that the competitive ratio is at least $c/16$.

Upper Bound on Number of Elements. To show that the instance ends with at most n elements, we first lower bound the number of large holes:

▷ Claim 12. Let n_i denote the total number of elements already inserted at the time that the adversary switches from epoch i to $i+1$. Then, the number of large holes is at least $n_i/4$.

Proof. We prove this claim by induction on i.

Suppose the current epoch i ended with j phases. Let $n_{curr} = n^{j/c}+1$ denote the number of elements inserted in the current epoch. Since the adversary created a new epoch $i+1$, by Condition 3, the number of large new holes at the end of epoch i is at least $n_{curr}/2$. Now, note that by the inductive hypothesis, the number of large holes at the end of epoch $i-1$ was at least $n_{i-1}/4$. All these holes become old holes at the beginning of epoch i. If an element is inserted in such an old hole in epoch i, then that creates at least 1 mixed hole in epoch i. (In fact, 2 mixed holes are created unless the old hole is at one end, in which case only 1 mixed hole is created.) Since Condition 1 was never satisfied, we can conclude that at most c of these old holes have had insertion of elements in epoch i. In other words, at least $n_{i-1}/4 - c$ of the large old holes at the beginning of epoch i have been untouched during epoch i, and therefore, they continue to be large holes at the end of epoch i.

We now sum the number of large holes at the end of epoch i. The count includes the large new holes created during epoch i and the large old holes from the beginning of epoch i that were untouched during epoch i. Thus, the total number of large holes at the end of epoch i is at least $n_{curr}/2 + n_{i-1}/4 - c \geq n_i/4$, since $n_{curr} = n_i - n_{i-1} \geq n^{1/c}+1 \geq 4c$ for $c \leq \frac{1}{6} \cdot \frac{\log n}{\log\log n}$. This completes the inductive proof. ◁

Next, we show that any single epoch cannot contribute more than $O(\sqrt{n})$ elements. This shows that if the instance has n elements overall, then all but $\sqrt{n}$ of them must be from completed epochs.

▷ Claim 13. The maximum number of elements inserted in an epoch is at most $\sqrt{n}+1$.

Proof. Note that an epoch ends if it reaches the beginning of the $(c/2+1)$st phase. Thus, the number of elements in an epoch is at most the number of inserted elements till the end of the $(c/2)$th phase. Since the total number of elements till the jth phase is $n^{j/c}+1$, it follows that the total number of elements in an epoch is at most $n^{(c/2)/c}+1 = \sqrt{n}+1$. ◁

Using the above claims, we now show that the instance cannot have more than n elements.

▶ **Lemma 14.** *The number of elements in the instance constructed by the adversary has at most n elements.*

Proof. Assume for contradiction that $n+1$ elements were inserted. Then, by Claim 13, at least $n - \sqrt{n}$ elements were inserted in completed epochs. Then, by Claim 12, there are at least $(n - \sqrt{n})/4$ large holes at the end of the last completed epoch. Since each large hole has $n^{1/(2c)}$ vacant array cells, it follows that the size of the array is at least

$$\frac{n - \sqrt{n}}{4} \cdot n^{1/(2c)} > \frac{n^{1+1/(2c)}}{5},$$

for $n > 25$. This contradicts the fact that the array is of size $n^{1+1/(2c)}/5$ in the statement of Theorem 11. ◀

Lower Bound on the Competitive Ratio. Finally, we show that the competitive ratio is at least $c/16$. We start with the case when the instance is terminated by the first condition:

▶ **Lemma 15.** *If the array contains at least c mixed holes, then the competitive ratio is at least $c/4$.*

Proof. Recall that $R_i = [2^{2i-1}, 2^{2i}]$ denotes the range for the current epoch i. The elements in all the previous $i-1$ epochs are in the range $\cup_{j=1}^{i-1}[2^{2j-1}, 2^{2j}] \subseteq [2^1, 2^{2i-2}]$. As a consequence, the difference between any element in the current epoch and any element in a previous epoch is at least $2^{2i-1} - 2^{2i-2} = 2^{2i-2}$. Therefore, every mixed hole induces a cost of at least 2^{2i-2}.

Now, note that optimal cost is at most the maximum value in the current range R_i, i.e., $\mathsf{opt} \leq 2^{2i}$. Since there are at least c mixed holes, the total cost is at least $c \cdot 2^{2i-2}$, which implies a competitive ratio of at least $c/4$. ◀

Next, we consider the case where the instance is terminated by the second condition. Note that in this case, the third condition did not hold at the end of the $(c/2)$th phase that immediately preceded the current phase. (Otherwise, the adversary would have moved to the next epoch.) We establish the following claim, which yields a lower bound of the cost of the algorithm based on the index of the current phase.

▷ Claim 16. Suppose $c \leq \frac{1}{6} \cdot \frac{\log n}{\log \log n}$. Then, if the array contains fewer than $n^{j/c}/2$ large new holes after inserting the elements in the jth phase of epoch i, then the cost of the algorithm is at least $j \cdot 2^{2i}/8$.

Proof. In this proof, we obtain a lower bound on the cost by only considering small new holes, and disregarding all other types of holes. Specifically, in each phase, we account only for the cost of small holes newly created during that phase. Once a small new hole is created in a phase, further subdivisions of this hole in subsequent phases are not counted, since these might not cause additional cost.

Suppose we are at the end of the jth phase. If epoch i did not end after phase j, then conditions 1 and 3 did not hold at the end of this phase. Therefore, there can be at most $n^{j/c}/2$ large new holes, in addition to at most c mixed holes. Since $n^{j/c} + 1$ elements were inserted in the current epoch, the number of small new holes is at least:

$$n^{j/c} - n^{j/c}/2 - c = n^{j/c}/2 - c \geq n^{j/c}/3, \tag{3.1}$$

where the inequality holds for any $j \geq 1$, since $n^{1/c} \geq 6c$ for $c \leq \frac{1}{6} \cdot \frac{\log n}{\log \log n}$.

Observe that inserting an element into an empty cell within an existing small new hole does not incur any additional cost if the element's value lies between those of the elements at its two ends. Consequently, to derive a valid lower bound for the cost in phase j, we exclude from our analysis all elements inserted into small holes that were created during the first $j-1$ phases. However, if an element is inserted in an existing large new hole, and this creates a small new hole, this does increase the cost since we explicitly excluded the cost of all large holes from the analysis. Likewise, if the insertion of elements into a mixed hole creates a small new hole, we also include its cost in our accounting.

So, we want to discard small new holes of two types from the above count: (a) those that have elements from a previous phase before j at either end, and (b) those that were created in the current phase j by inserting elements in phase j into a small new hole that existed at the end of phase $j-1$. We count these together by counting the total number of cells that were one of two types at the end of phase $j-1$: (a) either a vacant cell in a small new hole or (b) an occupied cell at one end of a small new hole.

The number of elements inserted in the current epoch up to phase $j-1$ is given by:

$$2 + \sum_{k=1}^{j-1} \left(n^{k/c} - n^{(k-1)/c} \right) = 2 + n^{(j-1)/c} - n^0 = n^{(j-1)/c} + 1.$$

Therefore, the number of small new holes at the end of phase $j-1$ is at most $n^{(j-1)/c}$, since each new hole has an element from the current epoch on each side. Note that each small new hole has at most $n^{1/(2c)}$ vacant cells. Therefore, the total number of cells that are either vacant in a small new hole or at one end of a small new hole at the end of phase $j-1$ is at most:

$$n^{(j-1)/c} \cdot (n^{1/(2c)} - 1 + 2) = n^{(j-1/2)/c} + n^{(j-1)/c} \leq 2n^{(j-1/2)/c}. \tag{3.2}$$

Subtracting (3.2) from (3.1), we get that the number of small new holes that incur additional cost in the jth phase is at least:

$$n^{j/c}/3 - 2n^{(j-1/2)/c} \geq n^{j/c}/4,$$

where the inequality holds since $c \leq \frac{1}{6} \cdot \frac{\log n}{\log \log n}$.

Next, we determine the minimum cost per hole. The elements in range R_i are uniformly spaced such that the minimum difference between any two distinct elements in phase j is

$$\frac{2^{2i} - 2^{2i-1}}{n^{j/c}} = \frac{2^{2i-1}}{n^{j/c}}.$$

Multiplying the number of contributing holes by the minimum cost per hole, the total cost accumulated strictly within phase j is at least

$$\frac{2^{2i-1}}{n^{j/c}} \cdot \frac{n^{j/c}}{4} = \frac{2^{2i-1}}{4} = 2^{2i-3}.$$

Multiplying by the number of phases, the total cost is at least $j \cdot 2^{2i-3}$. ◁

Using the above claim, we immediately get the following:

▶ **Lemma 17.** *If the algorithm terminates with $j = c/2 + 1$, then the competitive ratio of the algorithm is at least $c/16$.*

Proof. This is an immediate corollary of Claim 16. Specifically, if j reaches $c/2+1$, then the cost at the end of the last completed phase is at least $\frac{c/2}{8} \cdot \mathsf{opt} = \frac{c}{16} \cdot \mathsf{opt}$. ◀

4 Closing Remarks

In this paper, we explored the tradeoff between competitive ratio and space usage for the online metric TSP problem. In particular, we gave a deterministic algorithm that uses $(1+\varepsilon)n$ space and improves the competitive ratio from $\Theta(\sqrt{n})$ (for n space) to $O(\log^3 n/\varepsilon)$. We also showed that this cannot be improved further to $O(1)$-competitiveness using a deterministic algorithm, unless the space used increases to $n^{1+\gamma}$ for constant $\gamma > 0$.

Our work raises several interesting questions. First, our lower bound only applies to deterministic algorithms; we believe that the lower bound might also hold for randomized algorithms, but the current construction does not readily extend to the randomized setting. Second, while we rule out $O(1)$-competitive deterministic algorithms with $m = O(n \cdot \text{polylog}(n))$, it is quite possible that setting $m = n^{1+\varepsilon}$ for some constant $\varepsilon > 0$ yields an $O(1)$-competitive algorithm. Indeed, improving on the competitive ratio of $O(\log n)$ [24, 14], even with unlimited space usage, remains open.

References

1 Anders Aamand, Mikkel Abrahamsen, Lorenzo Beretta, and Linda Kleist. Online sorting and translational packing of convex polygons. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 1806–1833. SIAM, 2023. doi:10.1137/1.9781611977554.CH69.

2 Mikkel Abrahamsen, Ioana O. Bercea, Lorenzo Beretta, Jonas Klausen, and László Kozma. Online sorting and online TSP: randomized, stochastic, and high-dimensional. In Timothy M. Chan, Johannes Fischer, John Iacono, and Grzegorz Herman, editors, *32nd Annual European Symposium on Algorithms, ESA 2024, September 2-4, 2024, Royal Holloway, London, United Kingdom*, volume 308 of *LIPIcs*, pages 5:1–5:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ESA.2024.5.

3 Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998. doi:10.1145/278298.278306.

4 Giorgio Ausiello, Esteban Feuerstein, Stefano Leonardi, Leen Stougie, and Maurizio Talamo. Algorithms for the on-line travelling salesman. *Algorithmica*, 29(4):560–581, 2001. doi:10.1007/S004530010071.

5 Yossi Azar. Lower bounds for insertion methods for TSP. *Comb. Probab. Comput.*, 3:285–292, 1994. doi:10.1017/S096354830000119X.

6 Yossi Azar, Debmalya Panigrahi, and Or Vardi. Nearly tight bounds for the online sorting problem. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 6642–6658. SIAM, 2026. doi:10.1137/1.9781611978971.237.

7 Christian Bertram. Online metric TSP. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, LIPIcs, pages 80:1–80:9. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ESA.2025.80.

8 Antje Bjelde, Jan Hackfeld, Yann Disser, Christoph Hansknecht, Maarten Lipmann, Julie Meißner, Miriam Schlöter, Kevin Schewior, and Leen Stougie. Tight bounds for online TSP on the line. *ACM Trans. Algorithms*, 17(1):3:1–3:58, 2021. doi:10.1145/3422362.

9 Michiel Blom, Sven Oliver Krumke, Willem de Paepe, and Leen Stougie. The online TSP against fair adversaries. *INFORMS J. Comput.*, 13(2):138–148, 2001. doi:10.1287/IJOC.13.2.138.10517.

10 Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Operations Research Forum*, 3, 1976. URL: https://api.semanticscholar.org/CorpusID:123194397.

11 Esteban Feuerstein and Leen Stougie. On-line single-server dial-a-ride problems. *Theor. Comput. Sci.*, 268(1):91–105, 2001. doi:10.1016/S0304-3975(00)00261-9.

12 Dimitris Fotakis, Andreas Kalavas, Charalampos Platanos, and Thanos Tolias. A polylogarithmic algorithm for stochastic online sorting, 2025. doi:10.48550/arXiv.2508.12527.

13 Yang Hu. Nearly optimal bounds for stochastic online sorting, 2025. arXiv:2508.07823.

14 Makoto Imase and Bernard M. Waxman. Dynamic steiner tree problem. *SIAM J. Discret. Math.*, 4(3):369–384, 1991. doi:10.1137/0404033.

15 Patrick Jaillet and Michael R. Wagner. Generalized online routing: New competitive ratios, resource augmentation, and asymptotic analyses. *Oper. Res.*, 56(3):745–757, 2008. doi:10.1287/OPRE.1070.0450.

16 Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. A (slightly) improved approximation algorithm for metric TSP. *Oper. Res.*, 72(6):2543–2594, 2024. doi:10.1287/OPRE.2022.2338.

17 Marek Karpinski, Michael Lampis, and Richard Schmied. New inapproximability bounds for TSP. *J. Comput. Syst. Sci.*, 81(8):1665–1677, 2015. doi:10.1016/J.JCSS.2015.06.003.

18 Sven Oliver Krumke, Willem de Paepe, Diana Poensgen, and Leen Stougie. News from the online traveling repairman. *Theor. Comput. Sci.*, 295:279–294, 2003. doi:10.1016/S0304-3975(02)00409-7.

19 Sven Oliver Krumke, Luigi Laura, Maarten Lipmann, Alberto Marchetti-Spaccamela, Willem de Paepe, Diana Poensgen, and Leen Stougie. Non-abusiveness helps: An o(1)-competitive algorithm for minimizing the maximum flow time in the online traveling salesman problem. In Klaus Jansen, Stefano Leonardi, and Vijay V. Vazirani, editors, *Approximation Algorithms for Combinatorial Optimization, 5th International Workshop, APPROX 2002, Rome, Italy, September 17-21, 2002, Proceedings*, volume 2462 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2002. doi:10.1007/3-540-45753-4_18.

20 Maarten Lipmann. *On-line routing*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2003.

21 Nicole Megow, Martin Skutella, José Verschae, and Andreas Wiese. The power of recourse for online MST and TSP. *SIAM J. Comput.*, 45(3):859–880, 2016. doi:10.1137/130917703.

22 Jubayer Nirjhor and Nicole Wein. Improved online sorting, 2025.

23 Christos H. Papadimitriou and Mihalis Yannakakis. The traveling salesman problem with distances one and two. *Math. Oper. Res.*, 18(1):1–11, 1993. doi:10.1287/MOOR.18.1.1.

24 Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(3):563–581, 1977. doi:10.1137/0206041.

25 Anatolii Ivanovich Serdyukov. Some extremal bypasses in graphs. *Upravlyaemye Sistemy*, 17:76–79, 1978. [in Russian].

A The general case for unknown opt

In this section, we extend the result in Section 2 to the case of unknown opt, at the cost of an additional factor of $O(\log n)$ in the competitive ratio.

Following [6], we apply a doubling scheme on the value of the optimal solution. The doubling scheme uses two parameters: the optimal cost opt and the number of inserted points for a given guess on opt. We have an outer loop that (at least) doubles the estimate on opt in each iteration, and an inner loop that doubles the number of inserted points. This nested doubling process is initialized after sequentially inserting the first two points (in the first two array cells); twice their distance provides an initial estimate for opt.

Each iteration of the outer loop is called an epoch. Since MBC trees are not defined for small values of n (see Remark 3), we start any epoch by inserting the first $\frac{\log^2 1/\varepsilon}{\varepsilon}$ points consecutively before starting the doubling scheme. In the i-th epoch, the guess on opt is denoted opt_i. When the arrival of a new point causes the estimated cost (defined as twice the cost of the minimum spanning tree on the points seen so far) to exceed opt_i, the i-th

epoch ends and the $(i+1)$-st epoch starts. The value of opt in the new epoch is set to be at least double that of the previous epoch. Each epoch is assigned new space in the array, which begins immediately after the last cell used in the previous epoch.

Within an epoch, each iteration of the inner loop is called a phase. Let $n_{i,j}$ denote the bound on the number of inserted points in the j-th phase of the i-th epoch. The phase ends when the number of points exceeds $n_{i,j}$, and the $(j+1)$-st phase starts with $n_{j+1} := 2n_{i,j}$. Overall, we define $n_{i,j} := 2^j$.

In the j-th phase, we allocate a new subarray that begins immediately after the subarray used in the $(j-1)$-st phase. The amount of space allocated in the j-th phase is $(1+\varepsilon/3)n_{i,j}$. During the j-th phase, we employ the algorithm from Section 2 for inserting $n_{i,j}$ points with the variable $\varepsilon/3$ (instead of ε) as a black box.

In the proofs below, we use k_i, n_i to respectively denote the number of phases and points in the i-th epoch, and $n_{i,j}$ to denote the number of points in its j-th phase.

▶ **Lemma 18** (Space Bound). *The algorithm inserts n points in $(1+\varepsilon)n$ space.*

Proof. By Lemma 6, in phases $\leq k_i - 1$, the unused space is at most $\varepsilon/3$ times the used space, and in phase k_i, it is at most $\varepsilon/3$ times n_{i,k_i}. But, $n_{i,k_i} \leq 2n_{i,k_i-1}$, i.e., the unused space in phase k_i is at most $2\varepsilon/3$ times the used space in this epoch. Thus, the total unused space is at most an ε-fraction of the used space in this epoch. The lemma follows. ◀

▶ **Lemma 19** (Cost Bound). *The cost incurred by the algorithm is at most $\mathsf{opt} \cdot O(\log^3 n/\varepsilon)$.*

Proof. Fix epoch i. The cost between consecutive phases is at most opt_i, based on the current bound on the optimal solution. This results in a total additive cost of at most $k_i \cdot \mathsf{opt}_i$ between the k_i phases. In addition, since we insert the first $\frac{\log^2(1/\varepsilon)}{\varepsilon}$ points sequentially, we incur an additional cost of at most $\frac{\log^2(1/\varepsilon)}{\varepsilon} \cdot \mathsf{opt}_i$. Finally, there is a cost of $O\left(\frac{\log^2 n_{i,j}}{\varepsilon}\right) \cdot \mathsf{opt}_i = O(j^2/\varepsilon) \cdot \mathsf{opt}_i$ within each phase by Lemma 7. Therefore, the total cost in the i-th epoch is at most

$$\mathsf{opt}_i \cdot O\left(\frac{\log^2(1/\varepsilon)}{\varepsilon} + k_i + \sum_{j=1}^{k_i} \frac{j^2}{\varepsilon}\right) = \mathsf{opt}_i \cdot O\left(\frac{\log^2(1/\varepsilon)}{\varepsilon} + \frac{k_i^3}{\varepsilon}\right).$$

In addition, the cost between epochs $i-1$ and i is at most opt_i. Summing over all ℓ epochs, we get that the total cost is at most

$$\sum_{i=1}^{\ell} \mathsf{opt}_i \cdot O\left(\frac{\log^2(1/\varepsilon)}{\varepsilon} + \frac{k_i^3}{\varepsilon}\right).$$

Note that opt_i at least doubles in every epoch, and at any time it provides a 2-approximation to the true value of opt. As a result, $\sum_{i=1}^{\ell} \mathsf{opt}_i \leq 2\mathsf{opt}_\ell \leq 4\mathsf{opt}$. Moreover, the total number of phases in any epoch is at most $\log n$, i.e., $k_i \leq \log n$ for every i. Applying these facts to the above cost bound, we get that the total cost is at most

$$\mathsf{opt} \cdot O\left(\frac{\log^2(1/\varepsilon)}{\varepsilon} + \frac{\log^3 n}{\varepsilon}\right) = \mathsf{opt} \cdot O(\log^3 n/\varepsilon).$$ ◀

B A Counter-Example to the Analysis from [6] for a General Metric Space

The upper bound analysis in [6] relies on the structural property that all partial nodes at any height h are labeled by disjoint intervals. The adaptation of this key property into balls for a general metric is used in our paper as well in Claim 5. However, the insertion algorithm from [6], adapted to general metrics, does not maintain this key property.

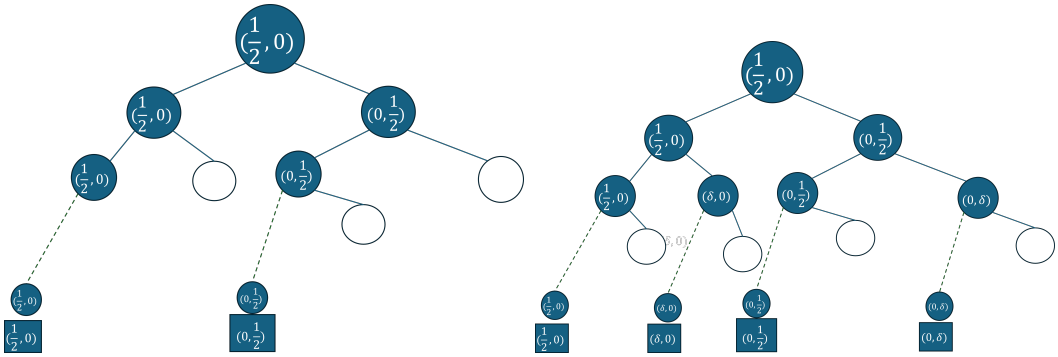
First, we recall the insertion algorithm from [6]. The procedure `insert`(v, x) is defined as follows:

- If v is a leaf node and is unmarked, then label v with the left or right half of its parent's label that contains x, write x in the array cell at the leaf node v, and return success.
- If v is marked and x is inadmissible at v, then return failure.
- If v is marked and x is admissible at v, then call `insert`(v_ℓ, x). If it returns success, return success. Otherwise, if it returns failure, then call `insert`(v_r, x). If this returns success, then return success. Otherwise, if it returns failure as well, then return failure.
- If v is unmarked, then label v with the left or right half of its parent's label that contains x and call `insert`(v_ℓ, x).

In short, `insert`(v, x) attempts to insert x first to the left child of v and then to the right child of v. Note that this operation also applies for balls as labels instead of intervals. As usual, the overall data structure comprises a series of identical MBC trees $A_0, A_1, \ldots$. Let x_t be the point being inserted. The algorithm in [6] iterates over the subarrays corresponding to these trees in order, i.e., it attempts to insert x_t in A_0, then A_1, then A_2, etc., until the point is successfully inserted.

We now demonstrate that the distance property claimed in Claim 5 fails for this insertion algorithm for $\mathbb{R}^2$ with the ℓ_1 metric.

The instance starts by inserting $(1/2, 0), (0, 1/2)$. As shown in Figure 4, these two points mark two sibling nodes (call these u, v) in the MBC tree with balls of radius $1/2$ and centered at $(1/2, 0), (0, 1/2)$ respectively. Next, we insert the points $(\delta, 0), (0, \delta)$ for a small $\delta > 0$. Observe that $\|(\delta, 0) - (1/2, 0)\|_1 = |1/2 - \delta| < 1/2$, whereas $\|(\delta, 0) - (0, 1/2)\|_1 > 1/2$. Thus, $(\delta, 0)$ is inserted under u and creates a partial node (call it u') labeled by a ball centered at $(\delta, 0)$. Similarly, $(0, \delta)$ is inserted under v and creates a partial node (call it v') labeled by a ball centered at $(0, \delta)$. The two partial nodes u', v' at the same height contradict Claim 5 since the points $(\delta, 0), (0, \delta)$ are only at distance 2δ from each other.



(a) After the insertion of $(1/2, 0)$ and $(0, 1/2)$.

(b) After the insertion of $(\delta, 0)$ and $(0, \delta)$.

Figure 4 Example showing that the algorithm in [6] does not satisfy Claim 5 for the ℓ_1-metric on $\mathbb{R}^2$. On the left, after the initial insertion of $(1/2, 0)$ and $(0, 1/2)$, we have sibling nodes u, v with disjoint balls of radius $1/2$. On the right, after the subsequent insertion of $(\delta, 0)$ and $(0, \delta)$, we create partial nodes u', v' at the same height, but their centers are arbitrarily close, within 2δ of each other. This violates Claim 5.