A Tight Lower Bound for Online Service with Deadlines and Lazy Server

Yann Disser
TU Darmstadt, Germany

Linda Thelen
TU Darmstadt, Germany

Abstract

We study the online service with deadlines (or delays) problem, in which a server must serve requests for points in a metric space while balancing travel distance and promptness of service. While the problem has been extensively studied (STOC 2017), (FOCS 2019), (FOCS 2023), the main open question whether a constant competitive ratio can be achieved remains wide open. We prove a logarithmic lower bound for a natural class of algorithms already on uniform line metrics. Our lower bound applies to, and is tight for, the best known algorithms for general metrics and uniform line metrics.

2012 ACM Subject Classification Theory of computation → Online algorithms; Theory of computation → K-server algorithms

Keywords and phrases online algorithms, competitive analysis, lower bound, delay, deadlines

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2025.26

1 Introduction

In the *online service with delay (OSD)* problem, introduced by Azar et al. [1], requests for points of a metric space are released over time in online fashion and need to be served by a single server. The server initially occupies a point of the metric space (its *state*) and must serve each request, once it has been released, by transitioning to the requested point. Crucially, requests do not have to be served immediately, but come with individual delay cost functions that assign a waiting cost to the amount of time between release and serving of the request. The objective is to minimize the sum of the total distance traveled by the server in the metric space (*service cost*) and the waiting costs of all requests (*delay cost*). Note that the service cost does not depend on time and the delay cost does not depend on traveled distance, i.e., in particular, state transitions of the server are instantaneous.

The OSD problem formalizes in a natural way the tradeoff between batching of requests to reduce overall effort and quick response to minimize individual delay, and has sparked a series of strong results that have been very well received. In their seminal paper, Azar et al. [1] initiated the competitive analysis of OSD and presented an $\mathcal{O}(\log^4 n)$-competitive randomized algorithm for metric spaces with n points, based on a randomized embedding of the metric space into a hierarchically separated tree (HST). Later, Azar and Touitou [2] improved this result to a randomized $\mathcal{O}(\log^2 n)$-competitive algorithm, still relying on HSTs but simultaneously addressing several other problems with delay. For uniform line metrics, Bienkowski et al. [6] defined a deterministic algorithm BCKT that further improves the competitive ratio to $\mathcal{O}(\log n)$. Finally, the current state of the art was achieved in a remarkable contribution by Touitou [21], who proved a deterministic bound of $\mathcal{O}(\log n)$ for general metric spaces, by presenting a deterministic algorithm (we call it TOUITOU) that no longer relies on HSTs.

While there is a significant body of work on OSD with meaningful advances in terms of upper bounds, still no super-constant lower bounds on the competitive ratio are known. The question of whether a constant competitive ratio is possible is raised as the main open question

for OSD in [1, 14, 21], and the authors see evidence that a logarithmic competitive ratio might be best-possible in the fact that no better algorithms are known even for prominent special cases. For example, no better competitiveness is known even for the multilevel aggregation problem [3], a further specialization of OSD on a tree.

We consider the important special case *online service with deadlines*, where requests do not incur any waiting costs if they are served until their respective deadlines, and an unbounded cost otherwise. Note that this variant is important in its own right and no better guarantee than $\mathcal{O}(\log n)$ via Bckt [6] and Touitou [21] is known, even for uniform line metrics. We give a tight lower bound for a natural class of algorithms that, in particular, only uses uniform line metrics and applies to Bckt and Touitou.

Our Results. We consider the online service with deadlines problem on a line metric with n equidistant points. We prove a lower bound of $\Omega(\log n/\log\lambda)$ for the competitive ratio of a class of (possibly randomized) online algorithms which we call λ-lazy. These algorithms only perform actions once a request has reached its deadline, and do not serve requests that are more than λ-times farther away than this critical request. This includes every algorithm that allows only λ-times the cost of critical service (i.e., the cost of serving requests whose deadlines expired) for preemptive service to other requests. Since online service with deadlines can be seen as a special case of online service with delay, our lower bound is also valid for the online service with delay problem.

▶ **Theorem 1.** *Every $\lambda(n)$-lazy algorithm has a competitive ratio of at least $\Omega(\log n/\log\lambda(n))$ for online service with deadlines (or delays), even on a line metric with n equidistant points.*

Importantly, the notion of λ-laziness applies to the two best algorithms that are known for online service with deadlines: the Bckt algorithm by Bienkowski et al. [6] for line metrics, and the general algorithm by Touitou [21] (which we call Touitou). Since both algorithms are λ-lazy for constant λ, we can derive a logarithmic lower bound for their competitive ratios that matches the known upper bounds of $\mathcal{O}(\log n)$ for the Bckt algorithm and for the Touitou algorithm, yielding an asymptotically tight bound for the competitive ratio of these algorithms.

▶ **Corollary 2.** Bckt *and* Touitou *have a competitive ratio of $\Theta(\log n)$ for online service with deadlines (or delays) on a line metric with n equidistant points.*

Finally, we address natural generalizations of Bckt and Touitou that increase λ with growing n, by refining our notion of laziness to more accurately capture these types of algorithms. We show that the resulting class of stricly λ-lazy algorithms does not allow for better competitive ratios than $\Omega(\log n)$ either, showing that algorithms that aim to achieve a sublogarithmic competitive ratio must employ more complex strategies for preemptive service.

Related work. A natural extension of the OSD problem is to consider k-OSD with any number $k \in \mathbb{N}$ of servers. The initial result of Azar et al. [1] extends to k-OSD and yields an upper bound of $\mathcal{O}(k\log^5 n)$. For uniform metric spaces, Krnetić et al. [17] improved this to a deterministic competitive ratio of exactly $2k+1$. We can recover the classic k-server problem [18] as a special case of k-OSD by giving all requests immediate deadlines, which means that deterministic algorithms for k-OSD cannot avoid a linear dependence on k (or on $n \geq k$) in the competitive ratio [16, 18]. Note that this also applies to the algorithm of Azar et al. [1], since it only uses randomization when initially embedding the

metric in an HST. Gupta et al. [13] used randomization on weighted star metrics to obtain an $\mathcal{O}(\log k \log n)$-competitive algorithm, and Gupta et al. [14] gave an $\mathcal{O}(\operatorname{poly}\log(\Delta, n))$-competitive randomized algorithm for k-OSD with deadlines, where Δ denotes the aspect ratio between largest and smallest interpoint distance in the metric space. Note that this is compatible with weaker forms of the recently disproved randomized k-server conjecture [7], since $n \geq k$.

Another natural generalization of OSD arises when the (continuous) delay functions are revealed in online fashion, i.e., when the cost of a delay in time is only revealed once this delay has irrevocably been incurred. Azar et al. [1] showed that in this *non-clairvoyant* setting, even for weighted star metrics, the competitive ratio is at least $\Omega(\sqrt{n})$. For uniform line metrics, Bienkowski et al. [6] were able to show that their algorithm is $\mathcal{O}(\log n)$-competitive, despite not requiring clairvoyance. Recently, Touitou [22] proposed an $\mathcal{O}(\sqrt{n}\log n)$-competitive algorithm for non-clairvoyant OSD on general metric spaces.

Problems closely related to OSD arise in network design with deadlines or delay, where instead of requests to visit a point in space, connectivity requests in a graph require transmitting a subgraph. Azar and Touitou [3] introduced a deterministic framework for such problems, avoiding randomized HST embeddings, while Touitou [20] proposed one for the non-clairvoyant setting.

A prominent problem that falls in this framework is the multilevel aggregation (MLA) problem, where leaves of a rooted tree are requested and can be served in bulk by paying for a subtree that includes them and the root. This problem is a specialization of OSD on a tree, where we force the server to return to the root after each service (via urgent requests). The problem was introduced by Charikar et al. [4] with an $\mathcal{O}(D^4 2^D)$-competitive algorithm, where D is the depth of the tree, which was later improved to $\mathcal{O}(\log n)$ [2, 3, 9]. Special cases of MLA include TCP acknowledgment [10, 15] and joint replenishment [5, 8], with the underlying rooted tree being of depth 1 and 2, respectively.

Other related network design problems include node-weighted Steiner forest and directed Steiner tree with deadlines or delay, also with logarithmic upper bounds [3]. In contrast to MLA, super-constant lower bounds are known for these problems [19].

The reordering buffer management (RBM) problem is also closely related to OSD but limits the number of waiting requests instead of charging delay costs. It has been studied in general metrics by Englert et al. [11] and on line metrics by Gamzu and Segev [12], with a best known upper bound of $\mathcal{O}(\log n)$, similar to OSD.

2 Preliminaries

We consider *online service with deadlines*, where requests $r_{j\in\{1,\dots,m\}}$ for points of a metric space $(\mathcal{M}, d)$ are issued over time and need to be served by a single server. Each request r is associated with a tuple $(x_r; a_r, t_r)$ consisting of its location $x_r \in \mathcal{M}$, its arrival time $a_r \geq 0$ and its deadline $t_r \geq a_r$. To make notation more concise, we identify each request with its location.

A solution to this problem is given by a schedule $\pi = ((p_i, t_i) \in \mathcal{M} \times \mathbb{R}_{\geq 0})_{i\in\{0,\dots,N\}}$, where p_0 is the origin of the server, we demand $0 = t_0 \leq t_1 \leq \dots \leq t_N$, and the interpretation is that the server moves (instantaneously) from $p_{i-1} \in \mathcal{M}$ to $p_i \in \mathcal{M}$ at time t_i for $i \in \{1, \dots, N\}$. For each request r, we define $\pi(r) \in \mathbb{R}_{\geq 0}$ to be the earliest time that the server visits its location after time a_r, and consider r served at time $\pi(r)$; in particular, requests that arrive at the server's location are served immediately. Our goal is to minimize

$$\begin{aligned} \min \quad & \sum_{i=1}^{N} d(p_i, p_{i-1}) \\ \text{s.t.} \quad & \pi(r_j) \leq t_{r_j} \forall j \in \{1, \ldots, m\}. \end{aligned}$$

Note that the constraints can be modeled by an additive term $\sum_{j=1}^{m} c_{r_j}(\pi(r_j) - a_{r_j})$ in the objective, where the delay cost functions c_r are given by $c_r(t) = 0$ if $t \leq t_r - a_r$ and $c_r(t) = \infty$ otherwise. In that sense, the online service with deadlines problem is a specialization of the more general online service with delay problem.

For clarity, we assume that deadlines are distinct among all requests, otherwise a total order may be chosen by arbitrary tie-breaking. The events at a time t occur in the following three phases: First, all new requests arrive, then the server moves (instantaneously), serving each request it visits, and, finally, the requests' deadlines expire. We denote the position of the server at the start of time t by s_t^{start}, and the position at the end of time t by s_t^{end}. We use S_t to refer to the set of points that the server visits during time t, i.e., formally $S_t = \{p_i \mid i \in 0, \ldots, N : t_i = t\}$. The set of open (i.e., released but unserved) requests after release of new requests but before the server's movements is denoted by R_t.

Once a request has reached its deadline, we refer to it as *critical*. We say that a request is served *preemptively* if it is served strictly before its deadline.

Online service with deadlines is an online problem in the sense that requests are only revealed once they are released. We evaluate the performance of an online algorithm Alg in terms of competitive analysis, i.e., for each request sequence σ, we compare the cost $\textsc{Alg}(\sigma)$ of its solution to the cost $\textsc{Opt}(\sigma)$ of an optimum offline solution that uses knowledge of the entire input from the beginning. The *competitive ratio* ρ of Alg is then defined as $\rho := \sup_\sigma[\textsc{Alg}(\sigma)/\textsc{Opt}(\sigma)]$.

3 A logarithmic lower bound

We show a lower bound on the competitive ratio for a class of algorithms for online service with deadlines which we call λ-*lazy algorithms* (see Algorithm 1). These algorithms only move the server once some request is critical, which is without loss of generality, since delaying actions while no deadline has expired does not incur any additional cost, but reveals more information about the input sequence. If there is no other request within λ-times the distance between server and critical request, the algorithm serves only the critical request and moves towards it. Note that this definition allows for preemption, e.g. in the case where we allocate a budget of λ-times the cost of critical service for preemptive service. We say that an algorithm for online service with deadlines is $\lambda(n)$-lazy for a given function $\lambda : \mathbb{N} \to \mathbb{N}$ (possibly on a specific class of metric spaces) if it fits this description with parameter $\lambda(n)$ on metric spaces with n points. Note that we allow randomization, e.g., in choosing requests for preemptive service or the next server position. The procedure UponDeadline in Algorithm 1 is triggered at the beginning of the second phase of time t_r for each request r, i.e., after new requests are already released, but before the server has to move and deadlines expire.

▶ **Definition 3.** *We call a request $r \in R_t$* λ-isolated *at time t if every $q \in R_t \setminus \{r\}$ satisfies $d(q, s_t^{\text{start}}) \geq \lambda d(r, s_t^{\text{start}})$.*

Algorithm 1 λ-Lazy Algorithm.

Function UponDeadline(r)**:**
  if r *is* λ*-isolated***:**
    serve only r
    move server to some s_t^{end} with $d(s_t^{\text{end}}, r) = 0$ or $d(s_t^{\text{end}}, r) < d(s_t^{\text{start}}, r)$
  else
    (do something else, e.g. serve requests or move the server)

▶ Remark 4. The requirement to strictly reduce the distance to the critical request can be relaxed to allow any end position satisfying $d(s_t^{\text{end}}, r) \leq d(s_t^{\text{start}}, r)$ by repeating requests in the construction below that lure the algorithm to a specific position. For this, observe that any competitive algorithm must eventually move the server to the position of the repeated requests. To simplify presentation, we will analyze only the case where the distance to isolated requests strictly decreases.

3.1 Adversarial construction

For given $\lambda \in \mathbb{N}$, we define adversarial constructions C_λ^ℓ of a *level* $\ell \in \mathbb{N}_0$ that determines the number of points on the line: C_λ^ℓ uses $(4\lambda+4)^\ell + 1 =: w_\ell + 1$ points. Additionally, we denote by $\tilde{C}_\lambda^\ell$ a variation of C_λ^ℓ that is used to recursively construct adversarial inputs of higher levels. For the start location of the server, we choose position w_ℓ.

We will show later on that every λ-lazy algorithm serves each request at the time of its deadline, while the offline optimum solution can strategically serve some requests at an earlier time, when the server is closer to these requests.

Intuitively, both C_λ^ℓ and $\tilde{C}_\lambda^\ell$ are designed to move the algorithm from the right end of the line to the left, while incurring additional cost logarithmic in the distance between right and left end. The offline optimum solution can serve all requests of the adversarial constructions at cost linear in the number w_ℓ of points of the line segment. The construction $\tilde{C}_\lambda^\ell$ mainly consists of repeating $C_\lambda^{\ell-1}$ and $\tilde{C}_\lambda^{\ell-1}$ to create a sequence of requests that move the algorithm through a larger number of points at roughly the same competitive ratio as $C_\lambda^{\ell-1}$. The request sequence C_λ^ℓ includes additional requests compared to $\tilde{C}_\lambda^\ell$ that cause the algorithm to move the full length of the line segment again after initially reaching the left end. That way, the algorithm suffers an additional penalty linear in the number of points on the line segment, i.e., a constant is added onto the competitive ratio. As the number of points grows exponentially in the level ℓ and the competitive ratio grows linearly, the resulting competitive ratio is logarithmic in the number of points on the line.

Base case ($\ell = 0$)

We define

$$C_\lambda^0 := \tilde{C}_\lambda^0 := \{(0; 0, 0)\} .$$

The deadline and the arrival time of the request coincide, meaning that every algorithm has to serve the request upon arrival. The server's starting position is at $w_0 = 1$.

For the remainder of the construction, we define $t_\ell := |C_\lambda^\ell|$ and $\tilde{t}_\ell := |\tilde{C}_\lambda^\ell|$ as the *duration* of C_λ^ℓ and $\tilde{C}_\lambda^\ell$, respectively. The inputs C_λ^ℓ and $\tilde{C}_\lambda^\ell$ will be constructed in such a way that every request reaches its deadline in the time interval $[0, t_\ell)$ or $[0, \tilde{t}_\ell)$, respectively.

Recursive construction ($\ell > 0$)

We define C_λ^ℓ and $\tilde{C}_\lambda^\ell$ based on the constructions $\tilde{C}_\lambda^{\ell-1}$ and $C_\lambda^{\ell-1}$ of the previous level. Refer to Figures 1 and 2 for an illustration of the recursive construction. We denote by $C_\lambda^{\ell-1}+(x,t)$ (resp. $\tilde{C}_\lambda^{\ell-1}+(x,t)$) the input sequence that results from shifting all requests of $C_\lambda^{\ell-1}$ (resp. $\tilde{C}_\lambda^{\ell-1}$) by x points on the line and delaying the requests' arrival time and deadline by t time units, i.e.

$$C+(x,t) \coloneqq \{(x_r+x; a_r+t, t_r+t) \mid r \in C\} \ .$$

We define

$$\tilde{C}_\lambda^\ell \coloneqq \bigcup_{i=0}^{2\lambda+1} \left(\tilde{C}_\lambda^{\ell-1} + \left(w_\ell - (i+1)\cdot w_{\ell-1}, i\cdot \tilde{t}_{\ell-1}\right)\right) \cup \bigcup_{i=2\lambda+2}^{4\lambda+3} \left(C_\lambda^{\ell-1} + \left(w_\ell - (i+1)\cdot w_{\ell-1}, (i-2\lambda-2)\cdot t_{\ell-1} + (2\lambda+2)\tilde{t}_{\ell-1}\right)\right) \ .$$

The adversarial input C_λ^ℓ has an additional request at the starting position that reaches its deadline towards the end of the construction. We call this request the *background request* $r_{\text{bg}} = (w_\ell; 2\lambda+2, \tilde{t}_\ell)$. This request can be served early by an offline algorithm at small additional cost, while a λ-lazy algorithm will serve it only upon its deadline, incurring large additional cost. Multiple later requests then force the server back to point 0:

$$C_\lambda^\ell \coloneqq \tilde{C}_\lambda^\ell \cup \{r_{\text{bg}}\} \cup \left\{(0; \tilde{t}_\ell + i, \tilde{t}_\ell + i) \mid i = 1, \ldots, 2w_\ell\right\} \ .$$

The definition of λ-lazy algorithms only requires the server to move towards the critical request by at least one unit of distance. Therefore, it is necessary to issue multiple requests at the end of the construction (namely, as many as the maximal possible distance of the server to point 0) to ensure that the server is stationed in point 0 after the construction is finished.

3.2 Analysis

We now analyze the cost of the offline optimum solution, as well as a λ-lazy algorithm on the adversarial inputs, proving Theorem 1. We start by showing that the cost of the offline optimum solution grows linearly with the number of points on the line, while every λ-lazy algorithm has to take a detour to serve additional requests from C_λ^ℓ, thus incurring $\Omega(\ell)$ times the cost of the offline optimum solution.

▶ **Lemma 5.** *For every $\lambda \in \mathbb{N}$ and $\ell \in \mathbb{N}$, the offline optimum solution* Opt *for online service with deadlines satisfies*

$$\textsc{Opt}(C_\lambda^\ell) \leq 2w_\ell \ .$$

Proof. We recursively construct a (suboptimal) offline solution Off that achieves the claimed cost for C_λ^ℓ. We construct solutions for C_λ^ℓ and $\tilde{C}_\lambda^\ell$ for each $\ell \in \mathbb{N}$ that will be used recursively to solve constructions of higher level. Off serves each request upon arrival, and takes a preemptive detour at the start to ensure that all additional requests from C_λ^ℓ can be served at no additional cost. Note that this detour could be omitted when analyzing only $\tilde{C}_\lambda^\ell$, but we will include it to simplify analysis of the recursive constructions.

We show that Off maintains the following invariants for $\ell \in \mathbb{N}$, given that Off starts at position $s_0^{\text{start}} = w_\ell$:

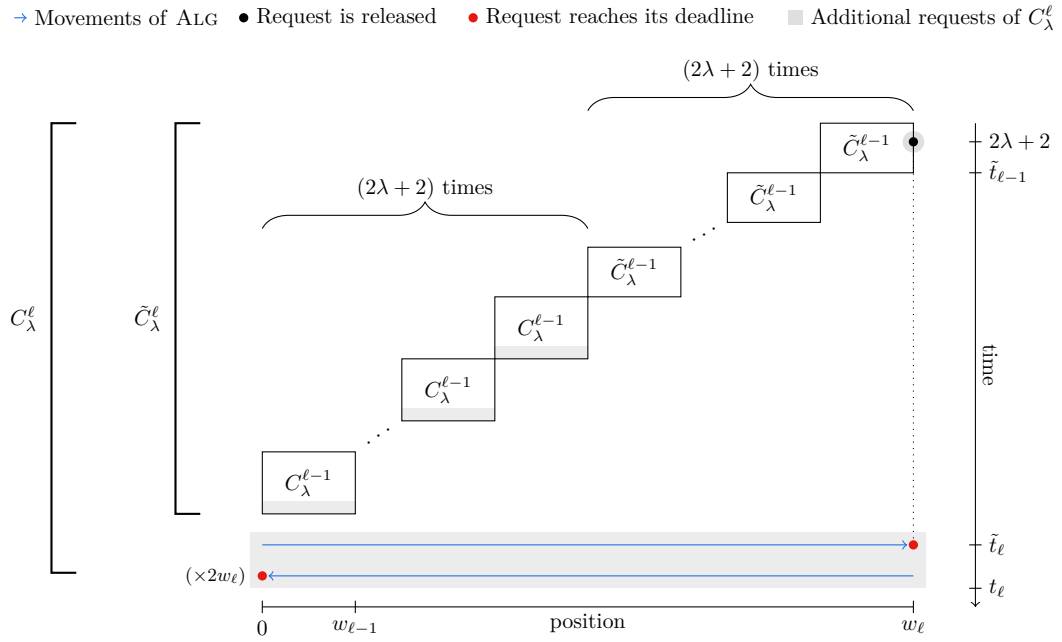

Figure 1 Recursive construction of adversarial inputs C_λ^ℓ and $\tilde{C}_\lambda^\ell$ for a λ-lazy algorithm ALG.

(I1) $\text{OFF}(C_\lambda^\ell) = \text{OFF}(\tilde{C}_\lambda^\ell) = 2w_\ell$.
(I2) OFF's server satisfies $w_\ell \in S_{2\lambda+2}$.
(I3) For each $t \in [\tilde{t}_\ell, t_\ell]$, OFF's server satisfies $s_t^{\text{start}} = s_t^{\text{end}} = 0$.

Base case ($\ell = 1$)

Refer to Figure 2 for an illustration of OFF on C_λ^1. We let OFF serve each request from copies of $\tilde{C}_\lambda^0$ upon arrival, incurring distance cost $2\lambda + 2$. At time $2\lambda + 2$, OFF returns from position $2\lambda + 2$ to $w_1 = 4\lambda + 4$, and back to $2\lambda + 2$ (recall that the server may move instantaneously), proving (I2). Note that it can therefore serve the background request at w_1 from C_λ^1 at time $2\lambda+2$ at no additional cost. OFF then continues to serve the remaining $2\lambda+2$ requests from copies of C_λ^0 upon arrival, each incurring one unit of distance cost. As required by (I1), the total cost of OFF is

$$4 \cdot (2\lambda + 2) = 2 \cdot w_1 \,.$$

OFF ends at position 0 at time $(2\lambda + 1) \cdot t_0 + (2\lambda + 2)\tilde{t}_0 = \tilde{t}_1 - 1$. It can then serve all remaining requests in 0 upon arrival at no additional cost without moving the server, hence (I3) is satisfied for C_λ^1. (I1) is maintained as before.

Inductive step ($\ell > 1$)

OFF handles each construction $C_\lambda^{\ell-1}$, $\tilde{C}_\lambda^{\ell-1}$ as in the previous recursion step, each at cost $2w_{\ell-1}$. This is possible since the required start positions align with the end position of OFF by (I3) and choice of the offsets. (I2) is maintained recursively, and the additional requests in C_λ^ℓ are served at no additional cost, since OFF is at their location at the time of their arrival by (I2) and (I3). OFF may therefore remain at position 0 after arriving there before time $\tilde{t}_\ell$, proving (I3). As required by (I1), the total cost is

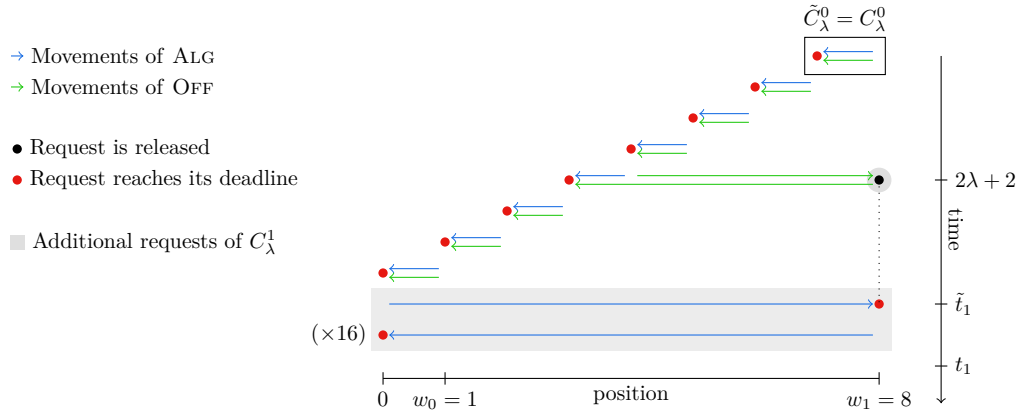

Figure 2 Illustration of C^1_λ for $\lambda = 1$, a λ-lazy algorithm ALG and the offline solution OFF.

$$\begin{aligned}\textsc{Off}(C^\ell_\lambda) = \textsc{Off}(\tilde{C}^\ell_\lambda) &= (2\lambda+2)\textsc{Off}(\tilde{C}^{\ell-1}_\lambda) + (2\lambda+2)\textsc{Off}(C^{\ell-1}_\lambda) \\ &\overset{\text{(I1)}}{=} (4\lambda+4)2w_{\ell-1} = 2w_\ell\,.\end{aligned}$$

◀

▶ **Lemma 6.** *For every $\lambda \in \mathbb{N}$ and $\ell \in \mathbb{N}_0$, every λ-lazy algorithm* ALG *satisfies*

$$\textsc{Alg}(\tilde{C}^\ell_\lambda) \geq \frac{\ell}{2} w_\ell \quad \textit{and} \quad \textsc{Alg}(C^\ell_\lambda) \geq \left(\frac{\ell}{2}+1\right) w_\ell\,.$$

Proof strategy and intuition. We prove the claim by induction on ℓ. In particular, we show that we can analyze copies of $C^{\ell-1}_\lambda$ and $\tilde{C}^{\ell-1}_\lambda$ independently in the recursive construction. To apply this argument, we consider the more general setting where C^ℓ_λ or $\tilde{C}^\ell_\lambda$ are used as part of a larger construction (i.e., there may exist other requests) and start at some later start time t_{start}. Note also that all arguments are translation invariant and can therefore also be applied to shifted copies $C^\ell_\lambda + (x,t)$ and $\tilde{C}^\ell_\lambda + (x,t)$. We will see that ALG serves exactly the requests from C^ℓ_λ (resp. $\tilde{C}^\ell_\lambda$) in the time interval $[t_{\text{start}}, t_{\text{start}} + t_\ell)$ (resp. $[t_{\text{start}}, t_{\text{start}} + \tilde{t}_\ell)$). Since copies of C^ℓ_λ and $\tilde{C}^\ell_\lambda$ are used t_ℓ or $\tilde{t}_\ell$ time units apart, it is then easy to see that ALG handles each copy independently.

To apply the induction hypothesis, we impose a set of assumptions which ensure that requests outside of C^ℓ_λ and $\tilde{C}^\ell_\lambda$ do not interfere with our intended behavior of ALG on C^ℓ_λ and $\tilde{C}^\ell_\lambda$. Intuitively, we assume that no outside request becomes critical during the execution of C^ℓ_λ or $\tilde{C}^\ell_\lambda$ (which would trigger an unexpected service), and that no outside requests are too close by, so that the requests from C^ℓ_λ and $\tilde{C}^\ell_\lambda$ are isolated when becoming critical. Additionally, we require that ALG's server is at the correct position at the start of the execution of C^ℓ_λ or $\tilde{C}^\ell_\lambda$. More formally, we make the following assumptions:

(AStartPosition) ALG's server is at position w_ℓ at the start of time t_{start}.

(ANoDelay) During time $[t_{\text{start}}, t_{\text{start}} + t_\ell)$ (resp. $[t_{\text{start}}, t_{\text{start}} + \tilde{t}_\ell)$), no requests other than those from C^ℓ_λ (resp. $\tilde{C}^\ell_\lambda$) reach their deadlines.

(AIsolationArea) It holds that

$$\begin{aligned} R_t \cap (-\infty, w_\ell + 2\lambda + 2) &\subseteq \tilde{C}^\ell_\lambda \quad && \forall t \in [t_{\text{start}}, t_{\text{start}} + \min\{\tilde{t}_\ell, 2\lambda+2\}) \\ R_t \cap (-\infty, w_\ell) &\subseteq \tilde{C}^\ell_\lambda && \forall t \in [t_{\text{start}} + 2\lambda + 2, t_{\text{start}} + \tilde{t}_\ell)\,, \end{aligned}$$

and, for C^ℓ_λ,

$$R_t \cap (-\infty, (2\lambda+2)w_\ell) \subseteq C^\ell_\lambda \qquad \forall t \in [t_{\text{start}} + \tilde{t}_\ell, t_{\text{start}} + t_\ell)\,.$$

Intuitively, (AIsolationArea) ensures that requests outside those from $\tilde{C}_\lambda^\ell$ or C_λ^ℓ are far enough to not interfere with the behavior of a λ-lazy server on the requests from the constructions. More precisely, we require that during the first $2\lambda+2$ time units of the constructions, no outside requests are less than $2\lambda+2$ points to the right of the starting point, in order not to interfere with the first requests of $\tilde{C}_\lambda^\ell$. After that, when the server has already moved sufficiently far from the starting point, we weaken this requirement and demand only that no outside requests lie to the left of the starting point. For C_λ^ℓ, since the background request forces the server to move back to the starting point later on, we require an isolation area of $(2\lambda+2)$-times the region that requests become critical in during the shifted time interval $[\tilde{t}_\ell, t_\ell)$, to account for the unknown position of the server.

Under these assumptions, we show that the following invariants are maintained:

(ICost) ALG incurs distance cost of at least $\ell/2 \cdot w_\ell$ during time $[t_{\text{start}}, t_{\text{start}} + \tilde{t}_\ell)$ and $(\ell/2+1)w_\ell$ during time $[t_{\text{start}}, t_{\text{start}} + t_\ell)$ (to serve requests from $\tilde{C}_\lambda^\ell$ and C_λ^ℓ, respectively).

(IIsolation) All requests from C_λ^ℓ and $\tilde{C}_\lambda^\ell$ are λ-isolated and critical when ALG serves them.

(IEndPosition) ALG's server is at position 0 at the end of time $t_{\text{start}} + \tilde{t}_\ell - 1$, and at the end of time $t_{\text{start}} + t_\ell - 1$ for C_λ^ℓ.

For the proof, we set $t_{\text{start}} = 0$ for simplicity of notation, but note that other requests may have been released before time t_{start}.

Proof.

Base case ($\ell = 0$)

We first consider the base constructions C_λ^0 and $\tilde{C}_\lambda^0$, which both consist only of a single request $r = (0; 0, 0)$. By (AStartPosition), the algorithm's server starts at position $s_0^{\text{start}} = w_0 = 1$.

The request r is λ-isolated at time $t = 0$, since, by (AIsolationArea), there is no other open request within $[1-\lambda, 1+\lambda]$. As r also reaches its deadline at time 0, r is served, proving (IIsolation). Since $s_t^{\text{start}} = 1$, we have $d(r, s_t^{\text{start}}) = 1$, and λ-lazy algorithms have to reduce the distance to isolated critical requests if possible. Therefore, the only valid next server position is position $0 = s_t^{\text{end}}$, proving (IEndPosition) (recall that $t_0 = \tilde{t}_0 = 1$). To serve r, the server had to incur one unit of distance cost, proving (ICost).

Inductive step ($\ell \geq 1$)

We will prove the induction invariants on C_λ^ℓ and $\tilde{C}_\lambda^\ell$ by applying the induction hypothesis to the copies of $C_\lambda^{\ell-1}$ and $\tilde{C}_\lambda^{\ell-1}$. To this end, we first verify that the assumptions on the copies of $C_\lambda^{\ell-1}$ and $\tilde{C}_\lambda^{\ell-1}$ hold, and then show that, using the induction hypothesis, the invariants remain valid for $\tilde{C}_\lambda^\ell$ and C_λ^ℓ.

Assumptions on $\tilde{C}_\lambda^{\ell-1}$. We constructed C_λ^ℓ and $\tilde{C}_\lambda^\ell$ in such a way that all requests reach their deadlines within t_ℓ (resp. $\tilde{t}_\ell$) time units after the start of the construction. Since all copies of $C_\lambda^{\ell-1}$ (resp. $\tilde{C}_\lambda^{\ell-1}$) used in $\tilde{C}_\lambda^\ell$ are $t_{\ell-1}$ (resp. $\tilde{t}_{\ell-1}$) time units apart, no requests from previous copies remain open at the start of any copy of a construction of level $\ell-1$. Moreover, since the background request r_{bg} from C_λ^ℓ is released only at time $2\lambda+2$, it does not interfere with (AIsolationArea). It follows that (ANoDelay) and (AIsolationArea) carry through from $\tilde{C}_\lambda^\ell$ or C_λ^ℓ to the copies of $\tilde{C}_\lambda^{\ell-1}$. Furthermore, (AStartPosition) on $\tilde{C}_\lambda^\ell$ or C_λ^ℓ implies (AStartPosition) for the first copy of $\tilde{C}_\lambda^{\ell-1}$. It follows that we can apply the induction hypothesis to the first copy of $\tilde{C}_\lambda^{\ell-1}$. Specifically, (IEndPosition) then implies (AStartPosition) for the next copy of $\tilde{C}_\lambda^{\ell-1}$. By repeated application of the invariants, (AStartPosition) is maintained for each copy of $\tilde{C}_\lambda^{\ell-1}$, and we can apply the induction hypothesis.

Assumptions on $C_\lambda^{\ell-1}$. For the copies of $C_\lambda^{\ell-1}$, observe that (ANoDelay) and (AStartPosition) are maintained by the same reasoning. For (AIsolationArea), note that it is sufficient to show that there are no requests $r \le (2\lambda+2)w_{\ell-1}+w_\ell-(2\lambda+3)w_{\ell-1} = w_\ell - w_{\ell-1}$ aside those from copies of $C_\lambda^{\ell-1}$ during time $[(2\lambda+2)\tilde{t}_{\ell-1}, (2\lambda+2)\tilde{t}_{\ell-1}+(2\lambda+1)t_{\ell-1}) \subseteq [0,\tilde{t}_\ell)$. This is guaranteed by (AIsolationArea) on $\tilde{C}_\lambda^\ell$ or C_λ^ℓ and definition of C_λ^ℓ. Hence, the induction hypothesis can again be applied to all copies of $C_\lambda^{\ell-1}$.

Induction invariants for $\tilde{C}_\lambda^\ell$. It is easy to see that (IIsolation) carries through from $\tilde{C}_\lambda^{\ell-1}$ and $C_\lambda^{\ell-1}$ to $\tilde{C}_\lambda^\ell$, as $\tilde{C}_\lambda^\ell$ contains no other requests. For (ICost), we find that by (ICost) on the copies of $\tilde{C}_\lambda^{\ell-1}$ and $C_\lambda^{\ell-1}$ together with $w_\ell = (4\lambda+4)w_{\ell-1}$, the cost incurred by ALG to serve requests from $\tilde{C}_\lambda^\ell$ is at least

$$\begin{aligned}(2\lambda+2)\textsc{Alg}(\tilde{C}_\lambda^{\ell-1}) + (2\lambda+2)\textsc{Alg}(C_\lambda^{\ell-1}) &\overset{\text{(ICost)}}{\ge} (4\lambda+4)\frac{\ell-1}{2}w_{\ell-1} + (2\lambda+2)w_{\ell-1}\\ &= \frac{\ell}{2}w_\ell\,,\end{aligned}$$

as required.

Finally, ALG is at position $w_\ell - (4\lambda+4)w_{\ell-1} = 0$ at the end of time $(2\lambda+2)t_{\ell-1} + (2\lambda+2)\tilde{t}_{\ell-1} - 1 = \tilde{t}_\ell - 1$ by (IEndPosition) on the last copy of $C_\lambda^{\ell-1}$, showing (IEndPosition) for $\tilde{C}_\lambda^\ell$.

Induction invariants for C_λ^ℓ. First note that we can apply the induction hypothesis to $\tilde{C}_\lambda^\ell$, since all assumptions on $\tilde{C}_\lambda^\ell$ hold by definition of the additional requests from C_λ^ℓ and the assumptions on C_λ^ℓ.

By (IEndPosition) on $\tilde{C}_\lambda^\ell$, ALG is at position 0 at the end of time $\tilde{t}_\ell - 1$. As ALG is λ-lazy, the server remains there until another request reaches its deadline. The next such request is the request $r_{\text{bg}} = (w_\ell; 2\lambda+2, \tilde{t}_\ell)$. The request is not served before it reaches its deadline at time $\tilde{t}_\ell$, since by (IIsolation) on $\tilde{C}_\lambda^\ell$, only the triggering request is served in each service prior to r_{bg} becoming critical.

By (AIsolationArea), there is no request at points within $(-\infty, (2\lambda+2)w_\ell)$ at time $\tilde{t}_\ell$. Therefore, r_{bg} is λ-isolated when it becomes critical at time $\tilde{t}_\ell$. ALG then serves r_{bg}, generating at least cost w_ℓ. It follows that ALG incurs distance cost of at least $(\ell/2+1)w_\ell$ for requests from C_λ^ℓ ((ICost) is maintained).

Afterwards, ALG's server is at a position $s_{\tilde{t}_\ell}^{\text{end}} \in (0, 2w_\ell)$, since it must move towards r_{bg}. Since by (AIsolationArea), there are no open requests within $(-\infty, (2\lambda+2)w_\ell)$, each following request $q_i \coloneqq (0; \tilde{t}_\ell + i, \tilde{t}_\ell + i)$ for $i \in \{1, \dots, 2w_\ell\}$ is λ-isolated at the time it reaches its deadline. Hence, the distance of ALG's server to 0 must decrease with each served request (if possible). It follows that ALG is at position 0 after it serves q_{2w_ℓ} at time $\tilde{t}_\ell + 2w_\ell = t_\ell - 1$, proving (IEndPosition). ◀

Proof of Theorem 1. Consider any $\lambda(n)$-lazy algorithm ALG. For a line with n points, we use an adversarial construction $C_{\lambda(n)}^\ell$. Since $C_{\lambda(n)}^\ell$ needs the underlying line metric to have at least $w_\ell + 1 = (4\lambda(n)+4)^\ell + 1$ points, we pick $\ell \coloneqq \left\lfloor \log_{4\lambda(n)+4}(n-1) \right\rfloor$. Lemmas 5 and 6 now imply

$$\frac{\textsc{Alg}(C_{\lambda(n)}^\ell)}{\textsc{Opt}(C_{\lambda(n)}^\ell)} > \frac{\ell}{4} = \Omega(\log n / \log \lambda(n))\,.$$ ◀

4 Applications

In this section, we establish that the algorithms Bckt [6] and Touitou [21] are λ-lazy with constant λ on line metrics. This implies a lower bound of $\Omega(\log n)$ on the competitive ratio of the algorithms by Theorem 1. Since both algorithms have been shown have competitive ratio $\mathcal{O}(\log n)$, the lower bound is tight. The proof details are deferred to appendices A and B.

4.1 The Bckt Algorithm

The algorithm Bckt was presented by Bienkowski et al. [6] (see Algorithm 2). It is $\mathcal{O}(\log n)$-competitive for online service with delay on a line with n equidistant points. Since online service with deadlines can be seen as a special case of online service with delay, Bckt can be applied to the deadlines case by modelling deadlines as the request accumulating very large delay only at the time of its deadline. The algorithm works in phases consisting of a *waiting subphase* and a *serving subphase*. During the waiting subphase, the server waits at its current position until there is a group of requests whose summed delay cost becomes roughly equal to the cost of serving them. At this point, a serving subphase starts, during which the server serves these requests and performs preemptive service to other requests that are similarly close. Figure 3 shows an illustration of a phase of Bckt. Note that we consider the line always centered at the server's position (in terms of the numbering of its points).

To be precise, during the execution, the algorithm partitions the line into $\mathcal{O}(\log n)$ *buckets* of geometrically increasing sizes, starting at the current position of the server. The i-th bucket to the right (resp. left) of the server is the interval $[2^{i-1}, 2^i - 1]$ (resp. $[-2^i + 1, -2^{i-1}]$ on the left), assuming that the points are numbered centered at 0 at the position of the server. We denote this bucket by B_{+i} (resp. B_{-i}) and call i the *label* of $B_{\pm i}$. For a bucket B, $|B|$ denotes the number of points it contains, i.e., $|B_{+i}| = |B_{-i}| = 2^{i-1}$, and we write $w(B)$ for the total delay cost of all open requests in B, i.e., for a given time t,

$$w(B) \coloneqq \sum_{r \in R_t \cap B} \mathrm{delay}_r(t)\,,$$

where delay_r is the function mapping a time t to the accumulated delay cost of request r at time t.

A waiting subphase ends once a bucket becomes *full*, i.e., $w(B) \geq |B|$. At this point, Bckt identifies the largest quarter-full bucket, i.e., the bucket B_i with the largest label $|i|$ such that $w(B_i) \geq |B_i|/4$. We call $|i|$ the *phase label*, and B_i the *critical bucket* of the current phase. In the distinct deadlines setting, a request immediately fills up its bucket when it reaches its deadline and all other buckets are empty, hence the critical bucket is the one containing the request that has just reached its deadline.

In the following serving phase, Bckt defines the *cleaning area* as the region $\bigcup_{j=1}^{|i|+1}(B_{-j} \cup B_{+j}) = [-(2^{|i|+1} - 1), 2^{|i|+1} - 1]$. Bckt serves all requests in the cleaning area and chooses its new position to be the point $\pm 2^{|i|-1}$ within the critical bucket.

▶ **Proposition 9.** Bckt *is* 4*-lazy.*

4.2 The Touitou Algorithm

Similarly to other algorithms for online service with deadlines or delays, Touitou is divided into *services*. Every request maintains a level that corresponds to the reluctance of the algorithm to serve the request. When a request reaches its deadline, a service starts during

Algorithm 2 BCKT (adapted from [6, Algorithm 1]).

Waiting phase:

split the line into buckets $B_{\pm i} = \pm[2^{i-1}, 2^i - 1]$.
wait until there exists a bucket B with $w(B) \geq |B|$.

Serving phase:

$i \leftarrow \arg\max\{|j| \colon w(B_j) \geq |B_j|/4\}$ (B_i is the largest quarter-full bucket)
serve the area $[-2^{|i|+1} + 1, 2^{|i|+1} - 1]$.
if $i > 0$**:** move to $2^{|i|-1}$
else move to $-2^{|i|-1}$

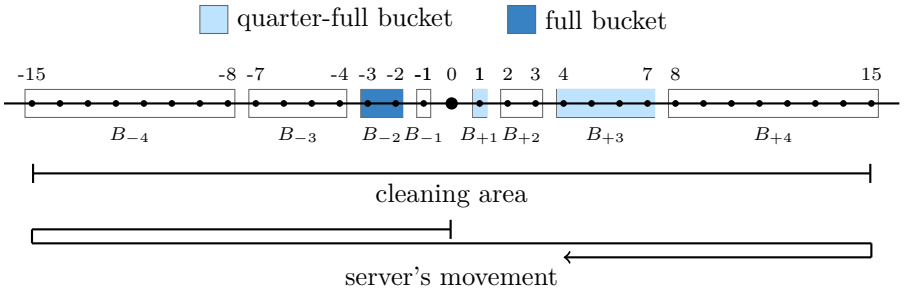

Figure 3 Illustration of the algorithm BCKT for a single phase (adapted from [6, Fig. 1.1]). Individual requests are not depicted.

which the algorithm identifies a set of requests that are eligible for service. A subset of these requests is chosen in order of increasing deadlines. Refer to Algorithm 3 for the pseudocode of TOUITOU.

To be precise, each request r maintains a level l_r that is initially set to $-\infty$. The level of a request increases only when the algorithm considers serving it, but decides not to, which is described below. If s denotes the current position of the server, the *adjusted level* of r is defined as $\bar{l}_r := \max\{l_r, \lceil \log_2 d(r, s) \rceil\}$.

When a request r reaches its deadline, a service S starts with level $l_S := \bar{l}_r + 3$. All requests with adjusted level at most l_S are eligible for service. Eligible requests are added to a list of requests for service in order of increasing deadlines. With each added request, TOUITOU computes a 2-approximation for a Steiner tree that connects the requests in the list to the current server position. Once the solution cost exceeds $4 \cdot 2^{l_S}$, the algorithm stops adding requests to the list. The requests in the list are then served on a DFS tour. Eligible requests that are not served are upgraded to level $l_S + 1$.

A service S is called *primary* if $\bar{l}_r \neq l_r$, i.e., the distance between r and the server dominates the level of r. For primary service, the server moves to r after the service. Otherwise, the server returns to its original position. Intuitively, this is the case if the request's location has been inconvenient compared to other requests in past services.

▶ **Proposition 12.** TOUITOU *is* 16*-lazy for inputs on a line metric.*

Corollary 2 follows immediately from Propositions 9 and 12 and Theorem 1.

▶ Remark 7. For the TOUITOU algorithm, the upper bound on the competitive ratio was also expressed in terms of the number m of requests as $\mathcal{O}(\log m)$. Our construction can be adapted to show that this bound is also tight. Observe that the TOUITOU algorithm

Algorithm 3 TOUITOU (adapted from [21, Algorithm 1]).

Function UponRequest(r)**:**
  $l_r \leftarrow -\infty$
Function UponDeadline(r)**:**
  start a new service S with $l_S := \bar{l}_r + 3$
  $E_S \leftarrow \{q \in R_t \mid \bar{l}_q \leq l_S\}$ (eligible requests)
  $Q_S \leftarrow \emptyset$ (requests chosen for service)
  for $q \in E_S$ *in order of increasing deadlines***:**
    add q to Q_S
    $P \leftarrow$ STEINERTREE(Q_S)
    if $\text{cost}(P) \geq 4 \cdot 2^{l_S}$**:**
      break from the loop
  serve requests in Q_S using a DFS tour on P
  for $q \in E_S \setminus Q_S$**:**
    $l_q \leftarrow l_S + 1$
  if $l_r \neq \bar{l}_r$**:** move the server to r

not only moves closer to critical requests after primary service, but moves exactly to the location of the critical request. We can therefore adapt the adversarial input so that only one request in point 0 is issued at the end of the construction to force the server to its final destination, instead of $2w_\ell$ consecutive requests. In doing so, the total number of requests in the adversarial input is reduced to $m = \mathcal{O}(n)$, implying a tight lower bound of $\Omega(\log m)$ for the competitive ratio of TOUITOU in the number of requests.

4.3 Generalizations

A natural idea to avoid the logarithmic lower bound is to adapt existing algorithms to be λ-lazy with growing values of λ. For both BCKT and TOUITOU, the parameter λ stems from a budget factor of $\Theta(\lambda)$ for preemptive service. To be precise, both algorithms serve all requests within range $c \cdot \lambda$ (for some constant $c \leq 1$) of the critical request, and serve requests within range at most λ of the critical request. We can generalize this to growing functions $\lambda(n)$ by replacing the fixed constant λ with a value depending on the size of the metric space. We can adapt Algorithm 1 accordingly to obtain a new class of *strictly* $\lambda(n)$*-lazy* algorithms (see Algorithm 4). Note that the requirement to serve (all) requests within range $c \cdot \lambda(n)$ ensures $\lambda(n)$-laziness without laziness for asymptotically smaller values of λ – otherwise we cannot hope to improve competitiveness by Theorem 1.

Algorithm 4 Strictly $\lambda(n)$-lazy Algorithm ($c \leq 1$).

Function UponDeadline(r)**:**
  serve all requests q with $d(s_t^{\text{start}}, q) \leq c \cdot \lambda(n) \cdot d(s_t^{\text{start}}, r)$
  (optionally) serve some requests q with $d(s_t^{\text{start}}, q) \leq \lambda(n) \cdot d(s_t^{\text{start}}, r)$
  move server to some s_t^{end} with $d(s_t^{\text{end}}, r) = 0$ or $d(s_t^{\text{end}}, r) < d(s_t^{\text{start}}, r)$

It is now easy to adapt the adversarial construction from Section 3 to obtain a generalized lower bound on the competitive ratio. Note that the following bound becomes $\omega(\log n)$ for $\lambda(n) \in \omega(1)$, i.e., there is nothing to be gained by scaling λ with n.

▶ Remark 8. Every strictly $\lambda(n)$-lazy algorithm has competitive ratio $\Omega(\lambda(n) \cdot \log n / \log \lambda(n))$.

Proof sketch. We can adapt the adversarial construction from Section 3 to exploit the increased budget for preemptive service. To do so, it suffices to add "decoy" requests with a late deadline within range $c \cdot \lambda(n)$ around each critical request. In addition to the factor of $\Omega(\log n / \log \lambda(n))$ proven in Theorem 1, the strictly $\lambda(n)$-lazy algorithm suffers a loss of factor $c \cdot \lambda(n)$ for preemptive service, which was not considered in the proof of Theorem 1. This results in a lower bound of $\Omega(\lambda(n) \cdot \log n / \log \lambda(n))$ on the competitive ratio of every strictly $\lambda(n)$-lazy algorithm. ◀

5 Conclusion

In this paper, we established a logarithmic lower bound on the competitive ratio of $\lambda(n)$-lazy algorithms for online service with deadlines (or delay) on uniform line metrics. This bound applies to the Bckt and Touitou algorithms, showing that their competitive ratios fall in $\Theta(\log n)$. Moreover, we demonstrated that the competitiveness of Bckt and Touitou cannot be improved by scaling the parameter λ with the size n of the metric space.

Our results highlight the inherent limitations of current algorithms and suggest that achieving a constant competitive ratio may be challenging. As evidenced in Section 4.3, avoiding the logarithmic lower bound requires an approach significantly different from the current state of the art for deterministic (or randomized) algorithms.

It remains an open question whether the super-constant lower bound demonstrated in this paper can be extended to a general bound for all online algorithms. While it may be possible to improve the upper bound of $\mathcal{O}(\log n)$, there is evidence that the bound is tight: No better deterministic or randomized algorithms are known for metric spaces other than HSTs of bounded depth [1]; the same holds for related problems, such as RBM [11, 12] and special cases of OSD such as MLA [3]. Notably, for the directed online Steiner tree problem, the directed version of MLA, a nearly-logarithmic lower bound has been established [19].

Also note that, currently, no separation is known between the deadlines and delay settings, or between deterministic and randomized algorithms.

References

1 Yossi Azar, Arun Ganesh, Rong Ge, and Debmalya Panigrahi. Online Service with Delay. *ACM Trans. Algorithms*, 17(3), 2017. (STOC 2017). doi:10.48550/arXiv.1708.05611.

2 Yossi Azar and Noam Touitou. General framework for metric optimization problems with delay or with deadlines. In *Proceedings of the 60th Annual Symposium on Foundations of Computer Science (FOCS)*, 2019. arXiv:1904.07131.

3 Yossi Azar and Noam Touitou. Beyond tree embeddings – a deterministic framework for network design with deadlines or delay. In *Proceedings of the 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1368–1379, 2020. doi:10.1109/FOCS46700.2020.00129.

4 Marcin Bienkowski, Martin Böhm, Jaroslaw Byrka, Marek Chrobak, Christoph Dürr, Lukas Folwarczny, Lukasz Jez, Jiri Sgall, Nguyen Kim Thang, and Pavel Vesely. Online Algorithms for Multi-Level Aggregation. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA)*, volume 57, page 12(17), 2016. doi:10.4230/LIPIcs.ESA.2016.12.

5 Marcin Bienkowski, Jaroslaw Byrka, Marek Chrobak, Łukasz Jeż, Dorian Nogneng, and Jiří Sgall. Better approximation bounds for the joint replenishment problem. In *Proceedings of the 2014 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 42–54, 2014. doi:10.1137/1.9781611973402.4.

6 Marcin Bienkowski, Artur Kraska, and Paweł Schmidt. Online service with delay on a line. In *25th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 237–248, June 2018. doi:10.1007/978-3-030-01325-7_22.

7 Sébastien Bubeck, Christian Coester, and Yuval Rabani. The randomized k-server conjecture is false! In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*, pages 581–594, 2023. doi:10.1145/3564246.3585132.

8 N. Buchbinder, T. Kimbrelt, R. Levi, K. Makarychev, and M. Sviridenko. Online make-to-order joint replenishment model: primal dual competitive algorithms. *Operations Research*, 61(4), 2013. (SODA 2008). doi:10.1287/OPRE.2013.1188.

9 Niv Buchbinder, Moran Feldman, Joseph (Seffi) Naor, and Ohad Talmon. O(depth)-competitive algorithm for online multi-level aggregation. In *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1235–1244, 2017. doi:10.1137/1.9781611974782.80.

10 Daniel R. Dooly, Sally A. Goldman, and Stephen D. Scott. TCP dynamic acknowledgment delay (extended abstract): theory and practice. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC)*, pages 389–398, 1998. doi:10.1145/276698.276792.

11 Matthias Englert, Harald Räcke, and Matthias Westermann. Reordering buffers for general metric spaces. *Theory of Computing*, 6(1):27–46, 2010. (STOC 2007). doi:10.4086/TOC.2010.V006A002.

12 Iftah Gamzu and Danny Segev. Improved online algorithms for the sorting buffer problem on line metrics. *ACM Transactions on Algorithms*, 6(1):15:1–15:14, 2009. doi:10.1145/1644015.1644030.

13 Anupam Gupta, Amit Kumar, and Debmalya Panigrahi. Caching with time windows and delays. *SIAM Journal on Computing*, 51(4):975–1017, 2022. doi:10.1137/20M1346286.

14 Anupam Gupta, Amit Kumar, and Debmalya Panigrahi. A hitting set relaxation for k-server and an extension to time-windows. In *Proceedings of the 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 504–515, 2022. doi:10.1109/FOCS52979.2021.00057.

15 Anna R. Karlin, Claire Kenyon, and Dana Randall. Dynamic TCP acknowledgement and other stories about e/(e-1). *Algorithmica*, 36(3):209–224, 2003. (STOC 2001). doi:10.1145/380752.380845.

16 Elias Koutsoupias and Christos H. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42(5):971–983, September 1995. doi:10.1145/210118.210128.

17 Predrag Krnetić, Darya Melnyk, Yuyi Wang, and Roger Wattenhofer. The k-Server Problem with Delays on the Uniform Metric Space. In *Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC)*, volume 181, pages 61:1–61:13, 2020. doi:10.4230/LIPIcs.ISAAC.2020.61.

18 Mark S Manasse, Lyle A McGeoch, and Daniel D Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11(2):208–230, 1990. doi:10.1016/0196-6774(90)90003-W.

19 Noam Touitou. Nearly-Tight Lower Bounds for Set Cover and Network Design with Deadlines/Delay. In *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC)*, volume 212, page 53(16), 2021. doi:10.4230/LIPIcs.ISAAC.2021.53.

20 Noam Touitou. Frameworks for Nonclairvoyant Network Design with Deadlines or Delay. In *Proceedings of the 50th International Colloquium on Automata, Languages, and Programming (ICALP)*, page 105(20), 2023. doi:10.4230/LIPIcs.ICALP.2023.105.

21 Noam Touitou. Improved and deterministic online service with deadlines or delay. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC)*, pages 761–774, 2023. doi:10.1145/3564246.3585107.

22 Noam Touitou. Nearly-Optimal Algorithm for Non-Clairvoyant Service with Delay. In *Proceedings of the 42nd International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 327, pages 74:1–74:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.STACS.2025.74.

A Laziness of Bckt

In this section, we restate and prove the laziness property of the Bckt algorithm.

▶ **Proposition 9.** Bckt *is* 4*-lazy.*

Proof. To prove that Bckt fits the description of a 4-lazy algorithm as defined in Algorithm 1, we need to show the following in the deadlines setting:

- Bckt does not perform any actions while no request has reached its deadline.
- When a single 4-isolated request r reaches its deadline, Bckt serves only r and, if possible, moves closer to r.

Since Bckt waits for a bucket to become full before serving any requests or moving the server, it does not perform any actions while no request has reached its deadline.

In the case where a request at the server's current position becomes critical (i.e., reaches its deadline), Bckt does not initiate a serving phase, but just serves the request without moving. This aligns with the requirements for a 4-lazy algorithm.

Now consider the case where a single 4-isolated request r reaches its deadline at a time t with $s_t^{\text{start}} \neq x_r$ (i.e., the server is not at r when r becomes critical). Let i be the label of the bucket that contains r. Since, in the deadlines setting, no requests other than r have positive delay cost, the critical bucket must be B_i. Bckt serves all requests within the cleaning area $\bigcup_{1\leq j\leq |i|+1}(B_{-j}\cup B_{+j}) \subseteq (s_t^{\text{start}}-4d(r,s_t^{\text{start}}), s_t^{\text{start}}+4d(r,s_t^{\text{start}}))$. Since r is 4-isolated, Bckt serves only r, as required for a 4-lazy algorithm. The next server position s_t^{end} is the closest point of the critical bucket B_i (the point $\pm 2^{|i|-1}$), which satisfies $d(s_t^{\text{end}}, r) < d(s_t^{\text{start}}, r)$. Therefore, the server moves closer to the critical request if possible. ◀

B Laziness of Touitou

In this section, we restate and prove the laziness property of the Touitou algorithm. In particular, we observe that on line metrics, the level of every request remains at the initial value $-\infty$, and that the resulting simplified algorithm is 16-lazy.

▶ **Lemma 10.** *During a service S of* Touitou*, triggered by a request r with $l_r = -\infty$, only requests $q \in R_t$ with $d(q, s_t^{\text{start}}) < 16d(r, s_t^{\text{start}})$ are eligible for service.*

Proof. Every request q that is eligible for service must satisfy $l_S \geq \bar{l}_q \geq \lceil \log_2 d(q, s_t^{\text{start}}) \rceil$. Given that $l_r = -\infty$, we know that $l_S = \bar{l}_r + 3 = \lceil \log_2 d(r, s_t^{\text{start}}) \rceil + 3$. It follows that

$$d(q, s_t^{\text{start}}) \leq 2^{l_S} = 2^{\lceil \log_2 d(r, s_t^{\text{start}}) \rceil + 3} < 16d(r, s_t^{\text{start}}) \,.$$ ◀

▶ **Lemma 11.** *On a line metric,* Touitou *never increases levels of requests.*

Proof. During a service S, only requests q with $d(s_t^{\text{start}}, q) \leq 2^{l_S}$ are eligible for service. On a line metric, the cost of an optimal Steiner tree connecting any such requests to s_t^{start} is at most $2 \cdot 2^{l_S}$. It follows that the cost of a 2-approximation for the Steiner tree problem on this instance is bounded by $4 \cdot 2^{l_S}$. The loop over eligible requests therefore does not break early, meaning that all eligible requests are selected for service. Since the level of a request is only increased if it is eligible for service but not served, the level of a request is never increased. ◀

▶ **Proposition 12.** Touitou *is* 16*-lazy for inputs on a line metric.*

Proof. As before, we need to show that for inputs on a line metric, the following properties hold:

- TOUITOU does not perform any actions while no request has reached its deadline.
- When a single 16-isolated request r reaches its deadline, TOUITOU serves only r and, if possible, moves closer to r.

TOUITOU only performs actions when a request reaches its deadline during the call to UponDeadline, as required.

Now consider the case where a single 16-isolated request r reaches its deadline. By Lemma 11, we have $l_r = -\infty$. We can therefore apply Lemma 10 to find that in the following service S, only r is eligible for service. Hence, only r will be served.

Consider the case where the server is at the location of r at time t_r. By definition of TOUITOU, the server never moves farther away from the critical request, proving that $d(s^{\text{end}}_{t_r}, r) = 0$. In any other case, since all levels are $-\infty$ (Lemma 11), the adjusted level of each request is dictated by its distance to the server. It follows that the service is primary, and the server moves to r. ◀