On Online Algorithms with Advice for the k-Server Problem

Marc P. Renault and Adi Rosén [*]

[1] LIAFA, Université Paris Diderot - Paris 7 and UPMC,
`mrenault@liafa.univ-paris-diderot.fr`
[2] CNRS and Université Paris Diderot - Paris 7,
`adiro@liafa.univ-paris-diderot.fr`

Abstract. We consider the model of online computation with advice [6]. In particular, we study the k-server problem under this model. We prove three upper bounds for this problem. First, we show a $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$-competitive online algorithm for general metric spaces with b bits of advice per request, where $3 \leq b \leq \log k$. This improves upon the result of [1]. Moreover, we believe that our algorithm and our analysis are more intuitive and simpler than those of [1]. Second, we give a 1-competitive online algorithm for finite trees which uses $2+2\lceil \log(p+1) \rceil$ bits of advice per request, where p is the caterpillar dimension of the tree. Lastly, we present a variant of the algorithm for the tree that is optimal for the line with 1-bit of advice.

Keywords: online computation with advice, k-server problem, online algorithms, competitive analysis

1 Introduction

Online algorithms have been the subject of intense research activity over the past decades. The traditional setting is that of an online algorithm that does not have any knowledge about the future and that of a worst-case analysis using competitive analysis (cf. [3]). In the present paper, we consider a model introduced by Emek et al. [6], dubbed *online computation with advice*, which is aimed at relaxing the "absolutely no knowledge about the future" setting and at giving a general framework to quantify the interplay between the amount of knowledge about the future and the possible improvement in the competitive ratio. Roughly speaking, this model augments the power of the online algorithm by a series of queries. Each query is issued by the online algorithm when it receives a new request. These queries map the whole request sequence, including future requests, to some domain of *advice*. Thus, they provide the online algorithm with some information about the future. One is typically interested in the interplay between the size of the domain of advice, i.e., how many bits of advice are received with

[*] Research supported in part by ANR projects QRAC and NeTOC.

each request, and the attainable competitive ratio. For a formal definition of this model, see Section 2.

A number of results for various online problems have been obtained in the above model and in a variant thereof introduced by Böckenhauer et al. [2]. In the present paper, we consider the k-server problem under the model of online computation with advice. Emek et al. [6] gave an upper bound of $k^{O(\frac{1}{b})}$ on the competitive ratio of deterministic algorithms on general metric spaces, where b is the number of bits of advice per request. This upper bound was improved to $2\left\lceil \frac{\lceil \log k \rceil}{b-1} \right\rceil$ by Böckenhauer et al. [1]. Better bounds for specific metric space where also given (see "Related Work" below).

In this paper, we improve the upper bound for deterministic k-server algorithms with advice on general metric spaces by giving a deterministic online algorithm with b bits of advice per request, for $3 \leq b \leq \log k$, whose competitive ratio is $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$. While the improvement over the previous result is only about a factor of 2, we believe that our algorithm and analysis are more intuitive and simpler than previous ones, and may lead to further improvements in the upper bound. Also, we consider the class of metric spaces of finite trees, and give a 1-competitive deterministic online algorithm. The number of bits of advice per request used by this algorithm is $2 + 2\lceil \log(p+1) \rceil$, where p is the caterpillar dimension of the tree (cf. [9]). If $\log k < 2 + 2\lceil \log(p+1) \rceil$, then the trivial algorithm with advice, i.e., encoding the server id used by an optimum algorithm per request could be used instead of the algorithm presented in this paper.

The caterpillar dimension of the tree is preferable over other measures, such as height, because it remains constant for degenerate trees, such as the line, the spider and the caterpillar. Moreover, the caterpillar dimension is at most the height of the tree, and it is at most $\log N$, where N is the number of nodes in the tree [9]. For the line, the caterpillar dimension is 1 implying that 4 bits of advice are needed using the algorithm for the tree. In the last section, we present an algorithm that is strictly 1-competitive for the line using 1 bit of advice.

Related Work. The model of online computation with advice considered in the present paper was introduced by Emek et al. [6]. In that paper, the authors gave tight bounds of $\Theta(\log n/b)$ for deterministic and randomized online algorithms with advice for Metrical Task Systems [4], where n is the number of states of the system and b is the number of bits of advice per request. They also gave a deterministic online algorithm with advice for the k-server problem which is $k^{O(\frac{1}{b})}$-competitive, where $\Theta(1) \leq b < \log k$. This was improved by Böckenhauer et al. [1] who gave a deterministic online algorithm with advice for general metric spaces with a competitive ratio of $2\left\lceil \frac{\lceil \log k \rceil}{b-1} \right\rceil$. Böckenhauer et al., also, gave a deterministic algorithm for the Euclidean plane with a competitive ratio of $\frac{1}{1-2\sin(\frac{\pi}{2^b})}$, where $b \geq 3$ is the number of bits of advice per request. For the uniform metric space (the problem of paging), a 1-competitive deterministic online algorithm with 1 bit of advice per request is implicit in [5].

Böckenhauer et al. [2] introduced a somewhat similar model for online algorithms with advice, where the advice is a single tape of bits instead of being given separately for each request. This allows an algorithm to read a different number of bits of advice per request, but it requires that the online algorithm knows how many bits of advice to read with each request. Thus, the two models are, in general, incomparable. We note that upper bounds in the model of [6], as those given in the present paper, carry over to the model of [2]. An algorithm defined in the model of [6] can be transformed into an algorithm defined in the model of [2] by concatenating all the bit strings received per request in the model of [6], using it as the advice tape in the model of [2] and reading the same number of bits from the advice tape per request in [2] as received in [6].

Several results were given in this related model [5, 2, 7, 8, 1]. For example, in [5, 2], the authors explore the number of bits of advice required for deterministic and randomized paging algorithms, algorithms for the DiffServe problem, algorithms for a special case of the Job Shop Scheduling problem, and algorithms for the Disjoint Path Allocation problem to be 1-competitive.

2 Preliminaries

Online algorithms receive their input piece by piece. Each piece, or request, is an element of some set R, and the algorithm receives a *request sequence* denoted $\sigma = r_1, \ldots, r_n$, where $n = |\sigma|$ and r_i is the ith request. An online algorithm must perform all of the actions pertaining to a request before receiving the subsequent requests. The online algorithm incurs some cost performing these actions. In this paper, we consider only minimization problems.

We use the definition of *deterministic online algorithms with advice* as presented in [6]. An online algorithm with advice is defined as a request-answer game that consists of a request set, R; a sequence of finite nonempty answer sets, $A_1, A_2, \ldots$; and a sequence of cost functions, $\mathrm{cost}_n : R^n \times A_1 \times A_2 \times \cdots \times A_n \to \mathbb{R}^+ \cup \{\infty\}$ for $n = 1, 2, \ldots$. In addition, online algorithms with advice have access via a query to an advice space, U, which is a finite set. The advice space has a size of 2^b, where $b \geq 0$ is the number of bits of advice provided to the algorithm with each request. With each request, the online algorithm receives some advice that is defined by a function, $u_i : R^* \to U$, where i is the request index, that is applied to the whole request sequence, including future requests. A deterministic online algorithm with advice can, thus, be represented as a sequence of pairs (g_i, u_i) and $g_i : R^i \times U^i \to A_i$ for $i = 1, 2, \ldots$, where g_i is the action of the online algorithm. The action that the online algorithm takes upon receiving request r_i is a function of the first i requests, $r_1, \ldots, r_i$, and the advice received so far, $u_1(\sigma), \ldots, u_i(\sigma)$, i.e., the action on request r_i is $g_i(r_1, \ldots, r_i, u_1, \ldots, u_i)$.

The cost of the online algorithm is defined as $\mathrm{ALG}(\sigma) = \mathrm{cost}_n(\sigma, \mathrm{ALG}[\sigma])$, where $\mathrm{ALG}[\sigma] =< a_1, \ldots, a_n >\in A_1 \times \cdots \times A_n$ and, for $j = 1, \ldots, n$, $a_j = g_j(r_1, \ldots, r_j, u_i, \ldots, u_j)$. At the risk of a slight abuse of notation, we will denote the cost of a subsequence of σ as $\mathrm{ALG}(r_i, \ldots, r_j)$, where the prefix is understood implicitly.

For a minimization problem, we say that an algorithm is *c-competitive*, or has a *competitive ratio* of c, if, for every finite request sequence σ, $\text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma) + \zeta$, where ζ is not dependent on the request sequence and $\text{OPT}(\sigma)$ is the optimum cost over σ. If $\zeta = 0$, we say that an online algorithm is *strictly c-competitive*.

The *k-server problem* consists of a metric space, $\mathcal{M}$, k mobile servers, an initial configuration, and a finite request sequence, σ. Let $\mathcal{M} = (M, d)$, where M is a set of nodes, $d : M \times M \rightarrow \mathbb{R}^+$ is a distance function on M and $|M| = N > k$. A *configuration* is a multiset $X \subseteq M$ such that $|X| = k$. Each $x \in X$ indicates the position one of the k servers in the metric space M. Each request of σ will be to a node of $\mathcal{M}$, and a server must be moved to the requested node before the algorithm will receive the subsequent request. The goal is to minimize the distance travelled by the k servers over σ. A *lazy* k-server algorithm is an algorithm that, upon each request, only moves a single server to the request if it is uncovered.

For a metric space which is a tree, we say that a server, s, is *adjacent* to a request, r_i, if, along the path between the positions of s and r_i, there are no other servers.

The *caterpillar dimension* of a rooted tree, T, with root r, is denoted $\text{cdim}(T)$. We will define it as it is defined in [9] which is as follows. For a tree, T, composed of a single node, $\text{cdim}(T) = 0$. For a tree, T, with two or more nodes, $\text{cdim}(T) \leq k + 1$ if there exist edge disjoint paths, $P_1, \ldots, P_q$, beginning at the root r such that each component T_j of $T \setminus E(P_1) \setminus \ldots \setminus E(P_q)$ has $\text{cdim}(T_j) \leq k$, where $E(P_i)$ are the edges of P_i. The components T_j are rooted at their unique vertex lying on some P_i. The decomposition of T into these edge disjoint paths is called the *caterpillar decomposition* of the tree. All the nodes of P_i, $1 \leq i \leq q$, except the root, are assigned path level $k + 1$. The root is assigned path level $k + 2$. Note that the root of the tree has a path level one more than the caterpillar dimension of the tree.

Given an unrooted tree, G, we define the caterpillar dimension of G as the minimum over all nodes, $v \in G$, of the caterpillar dimension of G when rooted at v. In what follows, we refer to the caterpillar dimension of unrooted trees as defined here.

3 An Upper Bound for General Metric Spaces

In this section, we present a $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$-competitive deterministic online algorithm with advice, called CHASE, for the k-server problem on general metric spaces with b bits of advice per request, where $b \geq 3$. For convenience of notation, we use $\alpha = \left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$.

In order to clearly present the algorithm and the proof, we will first design and analyze the algorithm such that it gets a variable number of bits of advice with each request. The algorithm will receive at least 2 bits of advice with each request, and the total number of advice bits will not exceed bn for any prefix of

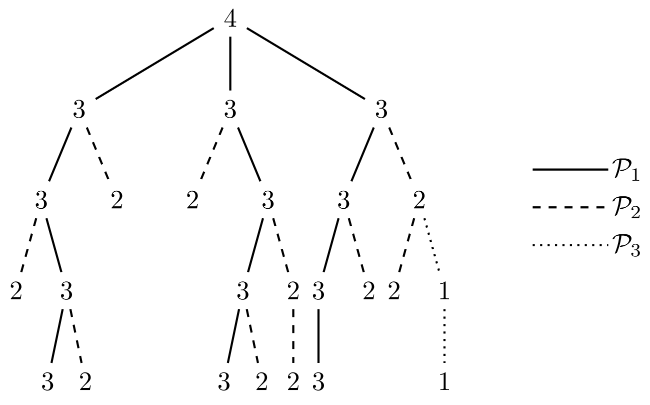

Fig. 1. An illustration of a caterpillar decomposition of a tree where the nodes are marked by their assigned path level. $\mathcal{P}_1$ is the set of edge disjoint paths rooted at the root, $\mathcal{P}_2$ is the set of edge disjoint paths rooted at a node of some path $\in \mathcal{P}_1$ and $\mathcal{P}_3$ is the set of edge disjoint paths rooted at a node of some path $\in \mathcal{P}_2$.

n requests. Afterwards, we will show how to adapt the algorithm so that it gets at most b bits of advice with each request using a transformation of [1].

Roughly speaking, our algorithm works as follows: given a request sequence, σ, we consider an optimal algorithm for this sequence. Based on this optimal algorithm, we partition σ into k subsequences, σ^s for $1 \leq s \leq k$, such that all the requests of σ^s are served according to the optimal algorithm by server s. With $\log k$ bits of advice per request, we can indicate, with each request of σ^s, the identity of the server s, and, thus, our online algorithm with advice would precisely follow the optimum algorithm. If, however, we have only $b < \log k$ bits of advice per request, we can do that only roughly every $\log k/b$ requests of σ^s. We call these requests "anchors". The rest of the requests of σ^s are served in a greedy manner, i.e., they are served by the closest server to the request which then returns to its previous position. By serving requests in this way, server s always stays at its last anchor. Thus, the cost of serving the $(\log k/b) - 1$ non-anchor requests of σ^s between any two anchors is bounded from above by $2 \log k/b$ times the distance from the last anchor to the furthest non-anchor request. This gives us a competitive ratio of $O(\log k/b)$. Some fine tuning of the above ideas gives us our result. In what follows, we formally define the algorithm and prove its competitive ratio.

Algorithm CHASE: At the beginning, all servers are unmarked.
Given a request, r_j, and the advice, do:

– If the advice is 00, serve r_j with the closest server to r_j and return it to its previous position.
– If the advice is 10, serve r_j with the closest unmarked server and mark this server. Do not return the server to its previous position.

– If the advice is $11t$, where t is a server number encoded in $\lceil \log k \rceil$ bits, serve the request with server number t.

In order to define the advice, we will fix a optimum algorithm, OPT, that we assume to be a lazy algorithm. We will then partition the request sequence into k subsequences, $\sigma^1, \ldots, \sigma^k$, where σ^s is the trace of the server s in OPT. In other words, σ^s consists of the requests served by server s in the lazy optimum. It should be noted that the requests of σ^s are not necessarily consecutive requests in σ. Let r_j^s be the jth request served by server s over σ^s. Recall that $\alpha = \left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$. Independently, for each server, we choose an index $1 \leq q^s \leq \alpha$. The way to choose this index will be defined later . The request sequence σ^s is divided into α-length cycles starting at $r^s_{q^s+1}$. We will denote the ith cycle of σ^s by c_i^s. The first cycle, c_1^s, which starts at request r_1^s and ends at request $r^s_{q^s}$, may have a length less than α. Let C^s be the total number of cycles in σ^s.

The advice will be defined as follows for request r_j^s:

– 10, if $j = q^s$, i.e., the last request of the first cycle.
– $11t$, if $j = q^s + i\alpha$ for some $i \geq 1$, i.e., the last request of all cycles except the first one. Here, t is the server number that serves request $r^s_{q^s}$ in CHASE encoded in $\lceil \log k \rceil$ bits.
– 00, if $j \neq q^s + i\alpha$, i.e., everywhere else.

The first two bits of the advice per request will be referred to as the control bits.

First, we state a technical lemma that we will use in our proof.

Lemma 1. *Given a sequence of α non-negative values, $a_1, \ldots, a_\alpha$, there is an integral value, q, where $1 \leq q \leq \alpha$, such that*

$$\sum_{i=1}^{q}(2(q-i)+1)a_i + \sum_{i=q+1}^{\alpha}(2(\alpha+q-i)+1)a_i \leq \alpha \sum_{i=1}^{\alpha} a_i \ .$$

Proof. Summing the expression over all possible values of q, we get

$$\sum_{q=1}^{\alpha}\left[\sum_{i=1}^{q}(2(q-i)+1)a_i+\sum_{i=q+1}^{\alpha}(2(\alpha+q-i)+1)a_i\right]$$
$$=\sum_{q=1}^{\alpha}\left[\sum_{i=1}^{\alpha}(2(\alpha+q-i)+1)a_i-\sum_{i=1}^{q}2\alpha a_i\right]$$
$$=\left[\sum_{q=1}^{\alpha}\big(2(\alpha-q)+1\big)\right]\cdot\sum_{i=1}^{\alpha}a_i+\left[\sum_{q=1}^{\alpha}4q\right]\cdot\sum_{i=1}^{\alpha}a_i-\sum_{q=1}^{\alpha}\sum_{i=1}^{\alpha}2ia_i-\sum_{q=1}^{\alpha}\sum_{i=1}^{q}2\alpha a_i$$
$$=\left[\sum_{q=1}^{\alpha}\big(2(\alpha-q)+1\big)\right]\cdot\sum_{i=1}^{\alpha}a_i+(2\alpha^2+2\alpha)\sum_{i=1}^{\alpha}a_i-\left[2\alpha\sum_{i=1}^{\alpha}ia_i+2\alpha\sum_{i=1}^{\alpha}(\alpha-i+1)a_i\right]$$
$$=\left[\sum_{q=1}^{\alpha}\big(2(\alpha-q)+1\big)\right]\cdot\sum_{i=1}^{\alpha}a_i+(2\alpha^2+2\alpha)\sum_{i=1}^{\alpha}a_i-(2\alpha^2+2\alpha)\sum_{i=1}^{\alpha}a_i$$
$$=\left[\sum_{q=1}^{\alpha}\big(2(\alpha-q)+1\big)\right]\cdot\sum_{i=1}^{\alpha}a_i$$
$$=\alpha^2\sum_{i=1}^{\alpha}a_i\ .$$

It follows that one of the α possible values of q gives at most the average value, i.e., $\alpha\sum_{i=1}^{\alpha}a_i$. The lemma follows. □

Now, we prove the main theorem of this section.

Theorem 1. *For every $b \geq 3$, algorithm CHASE is a $\left\lceil\frac{\lceil\log k\rceil}{b-2}\right\rceil$-competitive k-server algorithm for general metric spaces with b bits of advice per request.*

Proof. For the proof, we will compare the cost of CHASE and OPT separately for every subsequence σ^s, and cycle by cycle within each σ^s. Recall that $\alpha = \left\lceil\frac{\lceil\log k\rceil}{b-2}\right\rceil$. Note that the first cycle and the last cycle may be of length less than α.

If the last cycle is less than α for some σ^s, we can repeat the last request of σ^s until the last cycle is of length α. The repeated requests for each server should be appended to σ. Let σ' be the request sequence with the additional requests. Clearly, $\text{CHASE}(\sigma) \leq \text{CHASE}(\sigma')$. As the additional requests of σ' will be to nodes containing the servers of OPT, $\text{OPT}(\sigma') = \text{OPT}(\sigma)$. Hence, $\text{CHASE}(\sigma) \leq \alpha\text{OPT}(\sigma)$ if $\text{CHASE}(\sigma') \leq \alpha\text{OPT}(\sigma')$. Therefore, for this proof, we can assume without the loss of generality that the last cycle for each server is of length α.

Consider the ith cycle of server s in OPT for $i > 1$ (we will deal with the first cycle later). Let t be the server in CHASE that serves request $r^s_{q^s}$. We will denote the position of $r^s_{(i-2)\alpha+q^s}$, the last request of the previous cycle, by INIT^s_i. We claim that, just before the cycle starts, both OPT and CHASE will have a server at INIT_i^s. This is true because the advice for request $r^s_{(i-2)\alpha+q^s}$ indicates to CHASE to bring server t to INIT_i^s and, by the definition of the algorithm, t will always return to INIT_i^s between $r^s_{(i-2)\alpha+q^s}$ and $r^s_{(i-2)\alpha+q^s+1}$. By definition of the subsequence σ^s, OPT serves $r^s_{(i-2)\alpha+q^s}$ with s and does not move s between request $r^s_{(i-2)\alpha+q^s}$ and request $r^s_{(i-2)\alpha+q^s+1}$.

Also, observe that just before each of the requests between $r^s_{(i-2)\alpha+q^s+1}$ and $r^s_{(i-1)\alpha+q^s}$ inclusive, i.e., the requests of the ith cycle, server t of CHASE is at INIT_i^s. Recall that CHASE serves these requests except the last one by using the closest server to the request and, then, returns that server to its prior position. Therefore, the cost to CHASE for any request $r^s_{(i-2)\alpha+q^s+j}$, where $1 \leq j \leq \alpha-1$, i.e., the requests of cycle i except the last one, is

$$\text{CHASE}(r^s_{(i-2)\alpha+q^s+j}) \leq 2d(\text{INIT}_i^s, r^s_{(i-2)\alpha+q^s+j}) \ . \tag{1}$$

By the triangle inequality and Inequality (1),

$$\text{CHASE}(r^s_{(i-2)\alpha+q^s+j}) \leq 2\sum_{l=1}^{j} d(r^s_{(i-2)\alpha+q^s+l-1}, r^s_{(i-2)\alpha+q^s+l}) \ . \tag{2}$$

For request $r^s_{(i-1)\alpha+q^s}$, i.e., the last request of cycle i, CHASE serves the request using server t that is at $r^s_{(i-2)\alpha+q^s}$. We have, by the triangle inequality,

$$\begin{aligned}\text{CHASE}(r^s_{(i-1)\alpha+q^s}) &= d(\text{INIT}_i^s, r^s_{(i-1)\alpha+q^s}) \\ &\leq \sum_{l=1}^{\alpha} d(r^s_{(i-2)\alpha+q^s+l-1}, r^s_{(i-2)\alpha+q^s+l}) \ . \end{aligned}\tag{3}$$

Observe that the cost of OPT to serve $r^s_{(i-2)\alpha+q^s+j}$ for $1 \leq j \leq \alpha$, i.e., the requests of cycle i, is $d(r^s_{(i-2)\alpha+q^s+j-1}, r^s_{(i-2)\alpha+q^s+j})$. Using this fact and Inequalities (2) and (3), we can bound the cost of CHASE over a cycle by the cost of OPT as follows:

$$\begin{aligned}\sum_{j=1}^{\alpha}\text{CHASE}(r^s_{(i-2)\alpha+q^s+j}) &\leq \sum_{j=1}^{\alpha-1}\left(2\sum_{l=1}^{j}\text{OPT}(r^s_{(i-2)\alpha+q^s+l})\right) \\ &\qquad + \sum_{l=1}^{\alpha}\text{OPT}(r^s_{(i-2)\alpha+q^s+l}) \\ &= \sum_{j=1}^{\alpha}[2(\alpha-j)+1]\text{OPT}(r^s_{(i-2)\alpha+q^s+j}) \ . \end{aligned}\tag{4}$$

The analysis of the first cycle is, essentially, the same as the analysis of the ith cycle, $i > 1$, with the exception that an additive constant is introduced per request of the first cycle. The additive constant results from the fact that, during the first cycle of σ^s, CHASE does not necessarily maintain a server at the initial position of s. Recall that in CHASE, s may have been used to follow the trace of another server of OPT that is not s. In such a case, s would be marked and may be at a position in the metric space that is not the initial position of s. Nevertheless, by the definition of CHASE, there will always be an unmarked server in one of the locations of the initial configuration. Let $\varDelta$ be the diameter of the initial configuration. Therefore, for any request of the first cycle, r^s_l, of σ^s, analogously to Inequality (2), we have

$$\mathrm{CHASE}(r^s_l) \leq 2\Big(\varDelta + \sum_{m=1}^{l} d(r^s_{m-1}, r^s_m)\Big) \ , \tag{5}$$

where r^s_0 is the initial position of s. Analogous to Inequality (4), summing Inequality (5) over all requests of the first cycle of s, gives

$$\sum_{l=1}^{q^s} \mathrm{CHASE}(r^s_l) \leq \sum_{l=1}^{q^s} [2(q^s - l) + 1]\mathrm{OPT}(r^s_l) + 2\alpha\varDelta \ . \tag{6}$$

Note that the first cycle is of length $q^s \leq \alpha$. If we define the cost for requests with indexes less than 0 to be 0 for both OPT and CHASE, we can rewrite Inequality (6) to be more congruent with Inequality (4) as follows:

$$\sum_{j=1}^{\alpha} \mathrm{CHASE}(r^s_{-\alpha+q^s+j}) \leq \sum_{j=1}^{\alpha} [2(\alpha - j) + 1]\mathrm{OPT}(r^s_{-\alpha+q^s+j}) + 2\alpha\varDelta \ . \tag{7}$$

Using Inequalities (4) and (7), and summing over all cycles, gives

$$\mathrm{CHASE}(\sigma^s) \leq \sum_{i=1}^{C^s} \sum_{j=1}^{\alpha} [2(\alpha - j) + 1]\mathrm{OPT}(r^s_{(i-2)\alpha+q^s+j}) + 2\alpha\varDelta \ . \tag{8}$$

Now, we deal with assigning the values of q^s. Define $a_1, \ldots, a_\alpha$ to be $a_j = \sum_{i=1}^{C^s} \mathrm{OPT}(r^s_{(i-1)\alpha+j})$, i.e., the cost of OPT for the requests in σ^s in jumps of α requests. We can rewrite Inequality (8) as

$$\mathrm{CHASE}(\sigma^s) \leq \sum_{i=1}^{q} (2(q - i) + 1)a_i + \sum_{i=q+1}^{\alpha} (2(\alpha + q - i) + 1)a_i + 2\alpha\varDelta \ . \tag{9}$$

By Lemma 1, there is a value $1 \leq q^s \leq \alpha$ such that

$$\mathrm{CHASE}(\sigma^s) \leq \alpha \sum_{i=1}^{\alpha} a_i + 2\alpha\varDelta = \alpha\mathrm{OPT}(\sigma^s) + 2\alpha\varDelta \ .$$

We chose this q^s separately for each server s in order to define the cycles. Summing over all k subsequences σ^s concludes the proof of the competitive ratio.

Finally, we show that the algorithm uses at most bn bits of advice over any prefix of n requests. There are two control bits with each request. Let t be the server in CHASE that serves $r^s_{q_s}$, i.e., the last request of the first cycle of σ^s. There are at least α requests of σ^s between any two requests, where the id of t is given in the advice. Since $\alpha = \left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$, the claim follows. $\square$

In order to adapt the algorithm so that it receives b bits of advice per request, we use a transformation of [1]. Two control bits will be given with each request, and the remaining $b-2$ bits will contain portions of server ids. The control bits will be as defined previously. We then define a string as the concatenation of all server ids given for the whole sequence. This string will be broken into $(b-2)$-bit chunks and a single chunk will be given with each request. The algorithm can store these $(b-2)$-bit chunks in a FIFO queue and will have $\lceil \log k \rceil$ bits available to be read from the queue when dictated by the control bits.

4 k-Server with Advice on Trees

In this section, we describe a deterministic online algorithm with advice for the k-server problem on finite trees, called PATH-COVER, that is 1-competitive and uses $2 + 2\lceil \log(p+1) \rceil$ bits of advice per request, where p denotes the minimal caterpillar dimension of the tree.

The algorithm and advice are such that the actions of the algorithm will mimic the actions of a non-lazy optimum algorithm, denoted by OPT_{nl} which has specific properties with respect to the given request sequence. First, we will describe the construction of the non-lazy algorithm and show that it has optimum cost. Then, we will analyze the online algorithm with advice based on OPT_{nl}.

In this section, the definition of the algorithm, the advice and OPT_{nl} is based on the caterpillar decomposition of the tree that minimizes the caterpillar dimension. If there is not a unique caterpillar decomposition that minimizes the caterpillar dimension, one of the minimizing caterpillar decomposition can chosen arbitrarily to define the algorithm, the advice and OPT_{nl}. Further, in this section, an ancestor of a node is with respect to the root used for the chosen caterpillar decomposition and we assume that a node is an ancestor of itself.

The intuition for our algorithm can be seen most easily using the height of a rooted tree, h, instead of the caterpillar dimension. Consider any lazy optimum algorithm on the tree. A server move on the tree of such an algorithm can be broken into two parts. The first part is to the lowest common ancestor of the server and the request and the second part is to the request. Such an optimum can be altered into a non-lazy optimum algorithm that before the initial request and immediately after serving each request, it moves the server to the lowest common ancestor between the server's current position and the next request it would serve in the lazy optimum. $2 \log h$ bits of advice are sufficient to communicate the height of the node containing the server used for the request in the non-lazy optimum and the height of the lowest common ancestor between this request and the next request handled by the active server in the non-lazy optimum. Therefore, modulo an initialization phase, an algorithm with $2 \log h$ bits of advice can follow the non-lazy algorithm for each request by using a server at the position indicated by the advice and, after serving the request, moving the server to the second position indicated by the advice. For the algorithm described below, we choose the caterpillar dimension over the height of the tree since it gives a 1-competitive algorithm with a constant number of bits of advice per request for degenerate trees such as the line or a caterpillar. Furthermore, the caterpillar dimension is at most the height of the tree, and is at most $\log N$, where N is the number of nodes in the tree [9].

4.1 Non-Lazy Optimum

Let OPT_l be any lazy optimum for the given request sequence. For a given caterpillar decomposition of the tree, OPT_{nl} will be constructed from OPT_l such that OPT_{nl} has three properties. Let s be the server used by OPT_{nl} to serve request r_i. Then, given that OPT_{nl} can choose the starting configuration, the three properties are:

1. Immediately after serving r_i and before serving r_{i+1}, s and only s can be moved, and, if s is moved, it can only be to the nearest node of a higher path level in the tree.
2. The position of s, immediately before r_i is given, is at the same path level or higher than r_i.
3. Immediately before r_i is given, s is adjacent to r_i.

Let T be the tree representing the metric space $\mathcal{M}$. Given the caterpillar decomposition of T that minimizes $\mathrm{cdim}(T)$ as described above, we first construct OPT'_{nl} which has the first two properties. For any $u, v \in T$, let $\mathrm{maxPath}(u, v)$ be the ancestor of u with the maximum path level on the path between u and v (see Figure 2). The initial position of each server, $1 \leq s \leq k$, in OPT'_{nl} is $\mathrm{maxPath}(u_s, v_s)$, where u_s is the initial position of server s in OPT_l and v_s is the position of the first request served by s in OPT_l.

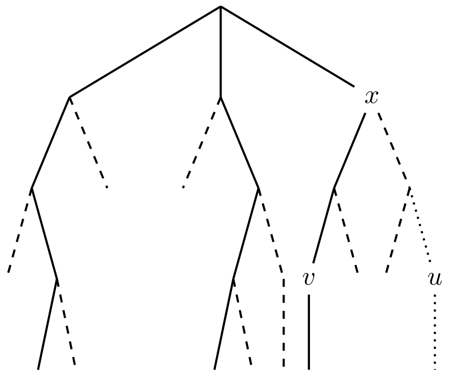

Fig. 2. An illustration of $x = \mathrm{maxPath}(u, v)$, using the caterpillar decomposition from Figure 1.

For each request, r_i in σ, let s_i be the server used by OPT_l to serve r_i. For r_i, OPT'_{nl} will:

1. Serve r_i with s_i.
2. Immediately after serving r_i, move s_i to $\mathrm{maxPath}(r_i, r_j)$ where r_j is the next request served by s_i in OPT_l.

Fact 1 *OPT'_{nl} has the first two properties.*

Proof. The definition of the algorithm guarantees Property 1.

As OPT'_{nl} uses the same server as OPT_l for each request, the initial configuration of OPT'_{nl} guarantees that each server s will be at the same level or higher than the first request it serves; and the second step for r_i guarantees Property 2 for the rest of the request sequence. □

Fact 2 *For any σ, $OPT'_{nl}(\sigma) = OPT_l(\sigma)$.*

Proof. The claim follows from the fact that the trajectories followed by each of the servers according to OPT'_{nl} and OPT_l are the same. The only difference being that some of the moves are done earlier in OPT'_{nl} than OPT_l. □

Next, we construct OPT_{nl} from OPT'_{nl} so that Property 3 will be satisfied, properties 1 and 2 will be maintained and the cost will not increase.

Intuitively, we will iterate over all the server moves of OPT'_{nl} and, in each instance where a non-adjacent server moves to a request, we will swap the non-adjacent server and the adjacent server. I.e., the adjacent server will serve the request and the non-adjacent server will be used for the next move of the adjacent server.

OPT_{nl} will be defined by induction on the server moves of OPT'_{nl}. Let $T^* = (t^*_1, q_1), \ldots, (t^*_m, q_m)$ be the sequence, in order, of server moves performed by OPT'_{nl} such that the ordered pair (x, y) represents a move of a server from position x to position y in the metric space. Note that $m \geq n$. All requests are represented in the sequence even if there is a server at the position of a request. In that case, the two positions of the ordered pair are the same. We will build T^i, which is a sequence of location pairs representing the server moves of an optimum algorithm, inductively for $i \geq 0$ where $T^0 = T^*$. For each T^i, $i \geq 0$, Property 1 and Property 2 hold for the entire sequence and Property 3 holds for the subsequence $(t^i_1, q_1), \ldots, (t^i_i, q_i)$.

Assume that T^{i-1} is defined and has the above properties. This is trivially true for T^0. In order to construct T^i, we need to first consider q_i. If q_i is not a request, then $T^i = T^{i-1}$. If q_i is a request and, in the configuration obtained in T^{i-1} after the $(i-1)$th pair, the server at t^{i-1}_i is adjacent to q_i, then $T^i = T^{i-1}$. Otherwise, there is a server at some position u_i between t^{i-1} and q_i. In this case, T^i will be defined as T^{i-1} from q_1 to q_{i-1}. That is, for all $1 \leq j < i$, $(t^i_j, q_j) = (t^{i-1}_j, q_j)$. The server at u_i will be used in T^i for request q_i, i.e., $(t^i_i, q_i) = (u_i, q_i)$. The remainder of the server moves will be the same in T^i as T^{i-1} except for the *next* time after i that a server at the position u_i is moved in T^{i-1}. In that case, a server at position t^{i-1}_i will be used. More formally, let l be the *first* index of the ordered pairs of T^{i-1} where $t^{i-1}_l = u_i$ such that $i < l \leq m$. In T^i, $(t^i_l, q_l) = (t^{i-1}_i, q_l)$.

Lemma 2. *For a given caterpillar decomposition of the tree, the cost of OPT_{nl} is no more than the cost of OPT_l, and OPT_{nl} has three properties. Let s be the server used by OPT_{nl} to serve request r_i. The three properties are:*

1. *Immediately after serving r_i and before serving r_{i+1}, s and only s can be moved, and, if s is moved, it can only be to the nearest node of a higher path level in the tree.*
2. *The position of s, immediately before r_i is given, is at the same path level or higher than r_i.*
3. *Immediately before r_i is given, s is adjacent to r_i.*

Proof. We will prove by induction on i that, for T^i, the cost does not increase compared to OPT_l, Property 1 and Property 2 hold for the enter sequence T^i, and Property 3 holds for $(t^i_1, q_1), \ldots, (t^i_i, q_i)$.

Cost: We will show by induction on i that the cost of T^i is no more than OPT_l. For $i = 0$, as T^0 is OPT'_{nl}, the claim follows from Fact 2. For the inductive step from T^{i-1} to T^i, note that the two sequences differ by at most two moves in the construction above. If $T^{i-1} = T^i$, then the cost remains the same between T^{i-1} and T^i. Using the notation in the construction above, if the two sequences differ at a single move then it is the move at index i. T^i uses a server at position u_i which is on the path between t^{i-1}_i and q_i and, therefore, the cost of T^i is less than that of T^{i-1}. If two moves differ between T^{i-1} and T^i, then they are the moves at indexes i and l. The cost to T^i for i and l is

$$\begin{aligned}
&d(u_i, q_i) + d(t^{i-1}_i, q_l)\\
&\qquad \le d(u_i, q_i) + d(t^{i-1}_i, u_i) + d(u_i, q_l) \text{ , by the triangle inequality,}\\
&\qquad = d(t^{i-1}_i, q_i) + d(u_i, q_l) \text{ , as } u_i \text{ is on the path between } t^{i-1}_i \text{ and } q_i,\\
&\qquad = d(t^{i-1}_i, q_i) + d(t^{i-1}_l, q_l) \text{ , as } u_i = t^{i-1}_l,
\end{aligned}$$

which is the cost to T^{i-1} for i and l.

Property 1: Property 1 restricts the moves that an algorithm can make between serving requests. Specifically, the algorithm can only move a server from the current request. The intuition of the proof is that, in the construction above, the only moves that differ between T^{i-1} and T^i are the moves to serve requests while any moves between requests are consistent between T^{i-1} and T^i.

We will show that Property 1 holds for the entirety of T^i by induction on i. The claim is trivial for T^0 as Property 1 holds for OPT'_{nl}. For the inductive step from T^{i-1} to T^i, if $T^i = T^{i-1}$, the claim must hold as it was true for T^{i-1}. If only the move at index i differs between T^{i-1} and T^i, then, according to the construction, this move must be that of a server to a request. Specifically, the difference between T^{i-1} and T^i is the position of the server that serves r_i, i.e., the first item of the ith ordered pair is the only difference between T^{i-1} and T^i. So, the claim holds as it was true for T^{i-1}. If the moves at index i and l, as defined in the construction above, differ between T^{i-1} and T^i, the move at index i is that of a server to a request as in the previous case. The second change is the first move of the server at u_i, in T^{i-1}, after (t_i^{i-l}, q_i) which must be to a request in T^{i-1} as Property 1 holds for T^{i-1}. Therefore, the second move changed between T^{i-1} and T^i is of a server to a request as well, which implies that Property 1 holds for the whole sequence T^i.

Property 2: By induction on i, we will show that Property 2 holds for the entirety of T^i. The claim is trivial for T^0 as Property 2 holds for OPT'_{nl}. For the inductive step from T^{i-1} to T^i, if $T^i = T^{i-1}$, the claim holds as it was true for T^{i-1}. If only the move at index i differs between T^{i-1} and T^i, then the claim holds as the different move, using the notation above, uses a server at u_i instead of t_i^{i-1}, and by the construction above, u_i is on the path from t_i^{i-1} to q_i. Therefore, u_i must be at the same path level or higher than that of q_i due to the monotonic nature of the caterpillar decomposition and that t_i^{i-1} is at the same path level or higher than q_i which follows from the induction hypothesis. If both the moves at indexes i and l, as defined in the construction above, differ between T^{i-1} and T^i, then the claim holds for the move at index i by the same argument of the previous case. For the difference at index l, the move must be that of a server to a request (as shown in the proof for Property 1). As shown in the previous case, t_i^{i-1} is at the same path level or higher than u_i. Therefore, t_i^{i-1} will be at the same path level or higher than any subsequent request served by the server at u_i in T^{i-1}, since Property 2 holds for the whole of T^{i-1}. Hence, Property 2 holds for the whole sequence T^i.

Property 3: The fact that Property 3 holds for $(t_1^i, q_1), \ldots, (t_i^i, q_i)$ is immediate from the inductive construction. □

4.2 The Algorithm PATH-COVER

There will be two stages to the algorithm. The initial stage will be for the first k requests and will be used to match the configuration of PATH-COVER to that of OPT_{nl} as defined in the previous section. Over the remaining requests, PATH-COVER will be designed to act exactly as OPT_{nl}. PATH-COVER will receive $2(l+1)$ bits of advice per request, where $l = \lceil \log(p+1) \rceil$ and p is the minimal caterpillar dimension of the tree. The advice will be of the form $wxyz$, where w and x will be 1 bit in length, and y and z will be l bits in length.

Note that, in the definition of the algorithm and the advice that follows, we assume that the servers are arbitrarily numbered from 1 to k.

Algorithm and Advice for $\boldsymbol{r_1, \ldots, r_k}$ From r_1 to r_k, PATH-COVER will serve each request with the nearest server regardless of the advice. As for the advice, for request r_i, where $1 \le i \le k$,

- if $w = 1$, the algorithm stores the ancestor node nearest r_i which has the path level y.

– if $x = 1$, the algorithm stores the ancestor node nearest the initial position of server i which has the path level z.

Note that both w and x can be 1 for request r_i. Immediately after serving r_k, PATH-COVER will use the first k stored nodes as a server configuration and will move to this configuration at minimal cost (minimum matching).

Immediately before serving r_{k+1}, the k servers of OPT_{nl} will be at their initial position or a higher path level; or at one of the first k requests or a higher path level. The advice over the first k requests is defined so as to encode the configuration of OPT_{nl} immediately before OPT_{nl} serves r_{k+1}. For $1 \leq i \leq k$, the advice for r_i will be defined as follows:

w**:** 1, if the server used for r_i in OPT_{nl} does not serve another request up to r_k.
0, otherwise

x**:** 1, if server i does not serve any of the first k requests in OPT_{nl}.
0, otherwise

y**:** A number in binary indicating the path level of the node to which the server used for r_i is moved to in OPT_{nl} after serving r_i.

z**:** A number in binary indicating the path level of the node to which server i is moved to before r_1 in OPT_{nl}.

Note that, over the first k requests, w and x will be 1 a total of k times (once for each of the k servers of OPT_{nl}). For r_i, when w is 1, this implies that, immediately before r_{k+1}, OPT_{nl} will have a server at the the nearest ancestor of r_i with path level y. When x is 1, this implies that, immediately before r_{k+1}, the server i of OPT_{nl} will be at the nearest ancestor of the initial position of server i with path level z. Observe that this indeed encodes the configuration of OPT_{nl} just before r_{k+1} in the advice bits of the first k requests.

Algorithm and Advice for $r_{k+1}, \ldots, r_n$ From r_{k+1} to r_n, given a request, r_i, where $k + 1 \leq i \leq n$, and the advice, let P be the path formed by all the adjacent nodes of path level y that includes the nearest ancestor of r_i with path level y. That is, P is the unique path from the chosen caterpillar decomposition of T with nodes of path level y that intersects the path between r_i and the root. Now, define a path, Q, on the tree as follows:

– if $x = 1$, Q runs from r_i to the end of P nearest the root.
– if $x = 0$, Q runs from r_i to the end of P furthest from the root.

PATH-COVER will serve r_i with the closest server along Q. After serving r_i, PATH-COVER will move this server to the nearest ancestor of r_i with path level z.

The advice is defined so that PATH-COVER and OPT_{nl} will use a server from the same position for each request. For $k+1 \leq i \leq n$, the advice for r_i will be defined as follows:

w**:** 0 (not used)
x**:** Let s be the position of the server used by OPT_{nl} to serve r_i, and let c be the ancestor of r_i with the same path level as s (see Figure 3).
1, if s is between c and the root of the tree.
0, otherwise.
y**:** A number in binary indicating the path level of s.
z**:** A number in binary indicating the path level of the node to which the server used for r_i is moved to in OPT_{nl} immediately after serving r_i.

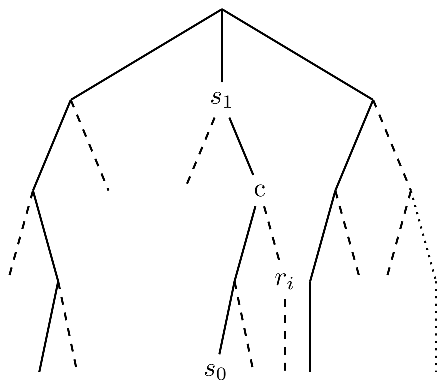

Fig. 3. An illustration of the definition of the advice bit x for r_i, where $k+1 \leq i \leq n$, using the caterpillar decomposition from Figure 1. s_1 represents a position of s such that $x = 1$ and s_0 represents a position of s such that $x = 0$.

Fact 3 *The algorithm uses* $2 + 2\lceil \log(p+1) \rceil$ *bits of advice per request, where* p *is the minimal caterpillar dimension of the tree.*

Proof. For each request, the bits of advice are composed of two control bits and two numbers encoded in binary, representing a path level. The two numbers encoded in binary range from 1 to $p+1$ where p is the minimal caterpillar dimension of the tree. □

Analysis

Theorem 2. *PATH-COVER is* 1*-competitive on finite trees.*

Proof. From r_1 to r_k, all the requests are served by the nearest server. This cost can be bounded by $k\Delta$, where Δ is the diameter of the tree. Immediately after serving r_k, PATH-COVER matches the configuration of OPT_{nl}. The cost to match a configuration can also be bounded by $k\Delta$.

We will show, by induction on $i \geq k+1$, that:

1. Just before r_i, the configuration of PATH-COVER and the configuration of OPT_{nl} match.
2. To serve r_i, they use a server from the same position.

By the definition of the algorithm and the advice, the configurations of PATH-COVER and OPT_{nl} match just before r_{k+1}. This establishes point 1 for the base case of r_{k+1}. Property 2 of OPT_{nl} guarantees that the position of the server, s_{k+1}, used by OPT_{nl} to serve r_{k+1} is at the same path level or higher that r_{k+1}. Property 3 of OPT_{nl} ensures that, just before serving r_{k+1}, there are no servers between s_{k+1} and r_{k+1}. Given that the configurations of PATH-COVER and OPT_{nl} match just before r_{k+1}, the first server, in PATH-COVER, along the path defined by the advice is at s_{k+1}. This establishes point 2 for the base case of r_{k+1}.

Assume that the induction hypothesis is true for j, $k+1 \leq j \leq i-1$. From the induction hypothesis, we know that the configurations of PATH-COVER and OPT_{nl} prior to serving r_{i-1} match, and that both PATH-COVER and OPT_{nl} will move a server from the same position in the tree to r_{i-1}. Therefore, the configurations of PATH-COVER and OPT_{nl} still match upon serving r_{i-1}. Property 1 of OPT_{nl} guarantees that only the server used for r_{i-1} can be moved if there is a server moved between the time that r_{i-1} is served and just before r_i is served. In that case, by the definition of the advice and the algorithm, PATH-COVER will move the server located at r_{i-1} to the same position that it is moved in OPT_{nl}. Property 1 also guarantees that this is the only server moved by OPT_{nl}. Therefore, the configurations of PATH-COVER and OPT_{nl} will match immediately before r_i, proving point 1. As shown in the base case, properties 2 and 3 of OPT_{nl}, and the fact that the configurations of OPT_{nl} and PATH-COVER match just before r_i, ensures that the first server along the path defined by the advice is at the same position as the server used by OPT_{nl}. This proves point 2.

It follows that PATH-COVER mimics the moves of OPT_{nl} from r_{k+1} to r_n, and, therefore,

$$\text{PATH-COVER}(\sigma) \leq \mathrm{OPT}(\sigma) + 2k\Delta \ .$$

□

5 The Line

In this section, we consider a special case of the tree: the line. The caterpillar dimension for the line is 1 which implies that our algorithm PATH-COVER requires 4 bits of advice. However, as the servers essentially do not change path levels on the line, a single bit of advice indicating the direction of the server to be used is all that is needed for a strictly 1-competitive, i.e., optimal, algorithm. In this section, we present the details of such an optimal algorithm, COMPLIANT that is a variant of PATH-COVER, the algorithm with advice for the tree presented in the previous section. Note that for this section, we will assume without the loss of generality that the optimum algorithm is lazy.

The definition of the advice will be based on a minimum weighted matching between the servers of COMPLIANT and those of OPT after OPT serves a request but before COMPLIANT serves it, where the server used by OPT for the request is matched to a server of COMPLIANT immediately to the right or the left. For completeness, we show in the appendix that on the line such a matching always exists.

5.1 The Algorithm COMPLIANT

For any request on the line, there are at most two adjacent servers. For request r_i, the advice is defined based on a minimum weighted matching between the servers of OPT immediately after OPT has served r_i and the servers of COMPLIANT immediately before serving r_i, where the server that OPT used for r_i, denoted by s, is matched to the first server of COMPLIANT to the left or the right. The bit of advice will indicate left or right according to this minimum matching.

Algorithm COMPLIANT: Serve the request with the first server in the direction indicated by the advice.

The one bit of advice given to the algorithm with each request is formally defined as follows:

1, if s is matched to the first server of COMPLIANT on the right of r_i.
0, otherwise.

5.2 Analysis

For the analysis of the algorithm, we will break up the process of serving a request, r_i, by OPT and COMPLIANT into two operations. They are:

1. OPT serves r_i
2. COMPLIANT serves r_i

Further, the analysis will make use of a potential function Φ which is the weight of the minimum weighted matching between the servers of OPT and COMPLIANT. Φ_i will denote Φ after Operation 2 for request r_i, and Φ'_i will denote Φ after Operation 1 for request r_i.

Theorem 3. *COMPLIANT is optimal.*

Proof. We first prove that for all requests of σ

1. $\text{OPT}(r_i) \geq \Phi'_i - \Phi_{i-1}$
2. $-\text{COMPLIANT}(r_i) \geq \Phi_i - \Phi'_i$

for Φ_i and Φ'_i as defined above.

Point 1: $OPT(r_i) \geq \Phi'_i - \Phi_{i-1}$. Let M be the matching used to calculate Φ_{i-1} and let $\Phi_i^{'*}$ be the weight of M after Operation 1 of r_i. By definition $\Phi'_i \leq \Phi_i^{'*}$, hence $\Phi'_i - \Phi_{i-1} \leq \Phi_i^{'*} - \Phi_{i-1}$. The claim follows since $\Phi_i^{'*} - \Phi_{i-1} \leq \mathrm{OPT}(r_i)$.

Point 2: $-COMPLIANT(r_i) \geq \Phi_i - \Phi'_i$. Let t be the server used by COMPLIANT to serve r_i as directed by the advice and let y be the distance that t is moved. By the definition of the advice and the algorithm, in the minimum matching M', used to calculate Φ'_i, t is matched to s, where s is the server used by OPT to serve r_i. Let Φ_i^* be the weight, after Operation 2, of M'. As t moves to the position of s, i.e., r_i, we have $\Phi_i^* - \Phi'_i = -y = -\mathrm{COMPLIANT}(r_i)$. Since $\Phi_i \leq \Phi_i^*$, point 2 is proved.

Adding the two inequalities gives

$$\mathrm{COMPLIANT}(r_i) \leq \mathrm{OPT}(r_i) + \Phi_{i-1} - \Phi_i \ . \tag{10}$$

Summing Inequality 10 over all requests and using the fact that $\Phi_n \geq 0$, we get

$$\mathrm{COMPLIANT}(\sigma) \leq \mathrm{OPT}(\sigma) + \Phi_0 \ .$$

COMPLIANT is strictly 1-competitive, i.e., optimal, as $\Phi_0 = 0$ since COMPLIANT and OPT start from the same configuration. □

6 Conclusions

We give an improved upper bound for the k-server problem with advice on general metric spaces. Moreover, we believe that our algorithm and our analysis are more intuitive and simpler than previous ones, and may, thus, lead to further improvements in the upper bound. We, also, give a 1-competitive k-server algorithm with advice for finite trees, using a number of bits of advice which is a function of the caterpillar dimension of the tree. For the line, we give an optimum algorithm with 1-bit of advice. The obvious open problem that remains is to give tight bounds for the k-server problem with advice on general metric spaces or for specific metric spaces.

Acknowledgements We thank Yuval Emek, Pierre Fraigniaud, Amos Korman, and Manor Mendel for useful discussions.

References

1. Böckenhauer, H.J., Komm, D., Královic, R., Královic, R.: On the advice complexity of the k-server problem. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP (1). Lecture Notes in Computer Science, vol. 6755, pp. 207–218. Springer (2011), also as technical report at ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/7xx/703.pdf

2. Böckenhauer, H.J., Komm, D., Královic, R., Královic, R., Mömke, T.: On the advice complexity of online problems. In: Dong, Y., Du, D.Z., Ibarra, O.H. (eds.) ISAAC. Lecture Notes in Computer Science, vol. 5878, pp. 331–340. Springer (2009)
3. Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press, New York, NY, USA (1998)
4. Borodin, A., Linial, N., Saks, M.E.: An optimal online algorithm for metrical task systems. Journal of the ACM 39, 373–382 (1992)
5. Dobrev, S., Královič, R., Pardubská, D.: How much information about the future is needed? In: SOFSEM'08: Proceedings of the 34th conference on Current trends in theory and practice of computer science. pp. 247–258. Springer-Verlag, Berlin, Heidelberg (2008)
6. Emek, Y., Fraigniaud, P., Korman, A., Rosén, A.: Online computation with advice. Theor. Comput. Sci. 412(24), 2642–2656 (2011)
7. Hromkovic, J., Královic, R., Královic, R.: Information complexity of online problems. In: Hlinený, P., Kucera, A. (eds.) MFCS. Lecture Notes in Computer Science, vol. 6281, pp. 24–36. Springer (2010)
8. Komm, D., Královic, R.: Advice complexity and barely random algorithms. In: Cerná, I., Gyimóthy, T., Hromkovic, J., Jeffery, K.G., Královic, R., Vukolic, M., Wolf, S. (eds.) SOFSEM. Lecture Notes in Computer Science, vol. 6543, pp. 332–343. Springer (2011)
9. Matoušek, J.: On embedding trees into uniformly convex banach spaces. Israel Journal of Mathematics 114, 221–237 (1999), http://dx.doi.org/10.1007/BF02785579

A Minimum Matching

Lemma 3. *Let A and B be two sets of points of equal size on the line. For any $a \in A$ there is a minimum weighted matching between the points of A and the points of B such that a is matched to a point $b \in B$, where b is the first element of B to the right or the left of a.*

Proof. Given a minimum weighted matching between the sets A and B on the line, let $a \in A$ be matched to a server $e \in B$ such that e is not the first element of B to the right or the left of a and let $b \in B$ be a point that is the first element of B to the left or the right of a and between a and e on the line. Let $c \in A$ be the point to which b is matched. Let $d(x, y)$ be the distance between the points x and y on the line. Without loss of generality, assume that b and e are to the right of a.

If c is between b and e, then $d(a, b) + d(c, e) \leq d(a, e) + d(b, c)$, so a can be matched to b and c can be matched to e without increasing the cost of the matching.

If c is to the right of e, then $d(a, b) + d(e, c) \leq d(a, e) + d(b, c)$ since $d(a, b) + 2d(b, e) + d(e, c) = d(a, e) + d(b, c)$. So, a can be matched to b and c can be matched to e without increasing the cost of the matching.

If c is to the left of a, then $d(a, b) + d(c, e) = d(c, b) + d(a, e)$ since $d(a, b) + d(c, e) = d(a, e) + 2d(a, b) + d(c, b) = d(a, b) + d(c, e)$. So, a can be matched to b and c can be matched to e without increasing the cost of the matching.

The final case is that c is between a and b. Note that this is the previous case with a and c swapped. As in the previous case, a can be matched to b and c can be matched to e without increasing the cost of the matching. $\square$