Reordering Buffers for General Metric Spaces[*]

Matthias Englert
RWTH Aachen
52056 Aachen, Germany
englert@cs.rwth-aachen.de

Harald Räcke
Toyota Technological Institute
Chicago, IL 60637, USA
harry@tti-c.org

Matthias Westermann
RWTH Aachen
52056 Aachen, Germany
marsu@cs.rwth-aachen.de

ABSTRACT

In the reordering buffer problem, we are given an input sequence of requests for service each of which corresponds to a point in a metric space. The cost of serving the requests heavily depends on the processing order. Serving a request induces cost corresponding to the distance between itself and the previously served request, measured in the underlying metric space. A reordering buffer with storage capacity k can be used to reorder the input sequence in a restricted fashion so as to construct an output sequence with lower service cost. This simple and universal framework is useful for many applications in computer science and economics, e. g., disk scheduling, rendering in computer graphics, or painting shops in car plants.

In this paper, we design online algorithms for the reordering buffer problem. Our main result is a strategy with a polylogarithmic competitive ratio for general metric spaces. Previous work on the reordering buffer problem only considered very restricted metric spaces. We obtain our result by first developing a deterministic algorithm for arbitrary weighted trees with a competitive ratio of $O(D \cdot \log k)$, where D denotes the unweighted diameter of the tree, i. e., the maximum number of edges on a path connecting two nodes. Then we show how to improve this competitive ratio to $O(\log^2 k)$ for metric spaces that are derived from HSTs. Combining this result with the results on probabilistically approximating arbitrary metrics by tree metrics, we obtain a randomized strategy for general metric spaces that achieves a competitive ratio of $O(\log^2 k \cdot \log n)$ in expectation against an oblivious adversary. Here n denotes the number of distinct points in the metric space. Note that the length of the input sequence can be much larger than n.

[*]Supported in part by DFG grant WE 2842/1 and by the EU within the 6th Framework Programme under contract 001907 (DELIS).

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

General Terms

Theory, Algorithms

Keywords

Online Algorithms, Reordering Buffer, Sorting Buffer, General Metric Spaces

1. INTRODUCTION

In the reordering buffer problem, we are given an input sequence of requests for service each of which corresponds to a point in a metric space (V, d), where V is a set of points and d is a distance function. The cost of serving the requests heavily depends on the processing order. Serving a request $p \in V$ following the service to a request $q \in V$ induces cost $d(p, q)$, i. e., the distance between these two requests.

A reordering buffer can be used to reorder the input sequence in a restricted fashion so as to construct an output sequence with lower service cost. At each point in time, the reordering buffer contains the first k requests of the input sequence that have not been processed so far. A scheduling strategy has to decide which request to serve next. Upon its decision, the corresponding request is removed from the buffer and appended to the output sequence, and thereafter the next request in the input sequence takes its place.

This simple and universal framework is useful for many applications in computer science and economics. In the following we give three examples (for further examples see [2, 6, 10, 11, 12, 14]).

- In hard disks, the latency of an access is mainly induced by the movement of the head to the respective cylinder. The latencies are the dominating factor for the performance of a hard disk. A reordering buffer can be used to rearrange the incoming sequence of accesses in such a way that latencies are reduced. This problem is known as disk scheduling (see, e. g., [15]).
- In computer graphics, a rendering system displays a 3D scene which is composed of primitives. A significant factor for the performance of a rendering system are the state changes performed by the graphics hardware. A state change occurs when two consecutively rendered

primitives differ in their attribute values, e. g., in their texture or shader program. The exact time required to perform a state change depends on the attribute values of the primitives causing the change. A reordering buffer can be included between application and graphics hardware to rearrange the incoming sequence of primitives in such a way that the cost of the state changes are reduced (see [13]).

- In the painting shop of a car manufacturing plant, car bodies traverse the final layer painting where each car body is painted with its own top coat. If two consecutive cars have to be painted in different colors a color change is required which causes non-negligible set-up and cleaning cost. This cost can be reduced by preceding the final layer painting with a reordering buffer (see, e. g., [9]).

In this paper, we design online scheduling algorithms for the reordering buffer problem. An online algorithm does not have knowledge about the whole input sequence in advance, but at any point in time only knows the k requests stored in its buffer. The cost of the online algorithm is compared to the cost of an optimal offline strategy that knows all requests in the input sequence in advance but, of course, may only select requests stored in its buffer for immediate service. Note that if the buffer size k is equal to the length of the input sequence, the reordering buffer problem reduces to the minimum weight Hamiltonian path problem and is therefore NP-hard.

Our main result is a scheduling strategy with a polylogarithmic competitive ratio for general metric spaces. Previous work on the reordering buffer problem only considered very restricted metric spaces like line metrics [8, 10, 11] and star metrics [5, 6, 14]. We obtain our result by first developing a deterministic algorithm for arbitrary weighted trees with a competitive ratio of $O(D \cdot \log k)$, where D denotes the unweighted diameter of the tree, i. e, the maximum number of edges on a path connecting two nodes. Then we show how to improve this competitive ratio to $O(\log^2 k)$ for metric spaces that are derived from HSTs. Combining this result with the results on probabilistically approximating arbitrary metrics by tree metrics [3, 4, 7], we obtain a randomized scheduling strategy for general metric spaces that achieves a competitive ratio of $O(\log^2 k \cdot \log n)$ in expectation against an oblivious adversary. Here n denotes the number of distinct points in the metric space. Note that the length of the input sequence can be much larger than n.

1.1 Related Work

The reordering buffer problem with uniform metric spaces in which two points are either at distance 0 or at distance 1 was introduced by Räcke, Sohler, and Westermann [14]. This setting models the paint shop scenario: Two requests are at distance 1, if the corresponding cars are to be painted in different colors, and at distance 0, otherwise. With this definition the total distance traveled by the server is equal to the total number of color changes. The authors present a deterministic online algorithm with a competitive ratio of $O(\log^2 k)$. This has subsequently been improved by Englert and Westermann [6] to a competitive ratio of $O(\log k)$, which also holds for a slightly more general class of metric spaces, the class of so-called star metrics, which can be represented as the shortest path metric space induced by weighted trees of height one. Note that our tree-algorithm is $O(\log k)$-competitive for trees of constant height, i. e., it obtains the best known bound for this special case.

Khandekar and Pandit [11] analyze the reordering buffer problem for n uniformly-spaced points on a line with the motivation that this scenario models the disc scheduling problem well. They present a randomized algorithm with a competitive ratio of $O(\log^2 n)$ in expectation against an oblivious adversary. Gamzu and Segev [8] improve this by presenting a deterministic $\Theta(\log n)$-competitive strategy that can also be used to derive an algorithm for the continuous line. However, the performance then depends polylogarithmically on the length of the input sequence. In addition, they give, for the line metric space, a lower bound of ≈ 2.154 on the competitive ratio of any deterministic algorithm. This is the only non-trivial lower bound known so far.

In terms of approximating the offline scenario most of the work has been done in the maximization version of the problem where the goal is to maximize the total cost-savings that result from reordering the sequence. Note that in terms of an optimal solution the minimization and maximization scenario are identical. However, in terms of approximation they behave quite differently. For uniform metric spaces, Kohrt and Pruhs [12] present an approximation algorithm with approximation ratio 20. Bar-Yehuda and Laserson [2] improve on this result with an approximation guarantee of 9.

Khandekar and Pandit [10] investigate the offline version of the minimization problem. They obtain a constant factor approximation guarantee with an algorithm that runs in quasi-polynomial time. To the best of our knowledge, the best polynomial time approximation algorithms for the minimization problem in the different scenarios discussed above are actually the corresponding online algorithms.

Alborzi et al. [1] analyze the k-client problem in which we are given k clients, each of which generates an input sequence of requests for service in a metric space. At each point in time a scheduling strategy has to decide which client's request to serve next. The authors present a deterministic strategy that achieves a competitive ratio of $2k - 1$. Further, they give a lower bound of $\Omega(\log k)$ on the competitive ratio of any deterministic strategy. The k-client problem is closely related to our problem, in the sense that in each time step a scheduling strategy has to choose between k requests in a metric space. At least for the online algorithm both problems look more or less identical as in each time step it chooses a request to be appended to the output sequence and a new request appears. A crucial difference however may be that in the k-client problem an optimal offline algorithm can take into account that processing different requests results in different requests to be released next. The offline algorithm can leverage this to its advantage, and therefore the bounds on the competitive ratio for the k-client problem are much larger.

1.2 Our Results

In Section 2, we start by introducing an online algorithm for the reordering buffer problem on tree metric spaces. The algorithm is inspired by the MAP strategy for star metrics introduced in [6]. However, our algorithm is not a generalization of this strategy as the behavior of both algorithms on a star metric can be different. In fact, analyzing our algorithm for the case of a star metric would lead to a simpler proof of a logarithmic competitive ratio for this special case.

We analyze our algorithm for tree metric spaces in Section 3 and obtain the following result.

THEOREM 1. *Let T denote an arbitrary weighted tree, and let D denote the unweighted diameter of T. For the shortest path metric space induced by T, our deterministic online algorithm achieves a competitive ratio of $O(D \cdot \log k)$, where k denotes the size of the reordering buffer.*

In Section 4, we then show how to improve the analysis for the special case that the underlying metric space is the shortest path metric induced by an HST (hierarchically well-separated tree). For $c > 1$, a c-HST is a rooted tree for which the edge lengths fulfill the following properties: For every vertex u on some level i (where the level of a node is its unweighted distance to the root), all incident edges connecting u to a node on level $i+1$ have the same length, and this length is at most ℓ/c, where ℓ denotes the length of the edge connecting u to its parent in the tree. We show the following result.

THEOREM 2. *For metric spaces that can be represented as the shortest path metric induced by an HST our online algorithm achieves a competitive ratio of $O(\log^2 k)$, where k denotes the size of the reordering buffer.*

Fakcharoenphol, Rao, and Talwar [7] present a randomized approximation of arbitrary n-point metric spaces by tree metric spaces with an approximation ratio of $O(\log n)$. The tree metric spaces used in this result are in fact the shortest path metrics induced by the leaf nodes of HSTs. Combining this result with our strategy for tree metric spaces, gives a randomized strategy for general metric spaces. This yields the following result.

COROLLARY 3. *For an n-point metric space, our randomized strategy achieves a competitive ratio of $O(\log^2 k \cdot \log n)$ in expectation against an oblivious adversary, where k denotes the size of the reordering buffer.*

2. THE ALGORITHM

The online algorithm for processing a sequence of requests in a tree works in phases where each phase consists of a *selection step* and a *processing step*. The different steps work as follows.

- **Selection Step**

  In this step, the online algorithm selects a set of requests to be removed from its buffer and to be appended to the output sequence. This selection is done as follows. We assign a variable $\mathsf{pay}(e)$ to each edge e of the tree, which at any given point in time has a value between 0 and the length $\ell(e)$ of the edge. We call an edge e a *paid edge* if $\mathsf{pay}(e) = \ell(e)$, and otherwise we call e an *unpaid edge*.

  During the selection process, the requests currently stored in the buffer are buying edges towards v_{onl}, where v_{onl} denotes the current position of the online server in the tree. This is done in the following continuous process. In a time interval[1] $[t, t+dt)$ each request at each node u increases the payment $\mathsf{pay}(e)$ by dt, where e denotes the first unpaid edge on the path from u to v_{onl}. This process continues, until there exist a connected component induced by paid edges that contains v_{onl}.

- **Processing Step**

  In this step, the online algorithm outputs all requests within the connected component. The order in which these items are visited is not important. The online algorithm only ensures that each edge of the component is traversed at most twice and that the final position $\hat{v}_{\text{onl}}$, i. e., the new position of the online server for the next phase, is the node in the component that is farthest away from v_{onl}.[2] Note that requests appearing during the processing step are ignored and will not be served in this processing step.

  After serving the requests the payment counter $\mathsf{pay}(e)$ on edges of the component is reset to 0. Note however that the payment counter of edges not in the component is not reset and that this payment will influence the selection step in future phases. This ends the phase.

The steps above are repeated as long as there exist at least k unprocessed requests. If the number of unprocessed requests drops below k, the online algorithm starts a *clean-up phase*, during which it simply processes all remaining requests in an optimal fashion. Note that despite of the continuous nature of our algorithm description all steps can be easily discretized and implemented efficiently.

3. GENERAL TREE ANALYSIS

For the analysis of the algorithm, we fix an optimal offline algorithm OPT, and we compare the performance of OPT to the performance of our algorithm, which is denoted as ONLINE. We view OPT and ONLINE as working in a synchronized manner. After a phase of ONLINE during which f requests were processed, i. e., appended to the output sequence, we simulate OPT until OPT processed f requests as well. Then we start the next phase of ONLINE.

Throughout the analysis, we use v_{opt} to denote the current position of the optimal server, i. e., the position of the last request that was appended to OPT's output sequence, and we use v_{onl} to denote the current position of the online server.

The following observation forms the basis for our analysis.

OBSERVATION 4. *The online cost induced by all processing steps is at most twice the total payment generated during the run-time of ONLINE.*

PROOF. Whenever ONLINE visits the requests in a connected component, it visits each edge of the component at most twice. At the same time it removes a payment of $\ell(e)$ from each edge e of the component. Hence, the removed payment in such a step is at least half the online cost of the step. □

The final goal of our analysis is to relate the total payment that is generated by ONLINE to the cost of OPT. The idea is to fix an edge e and to analyze the payment that is

[1]Note that the notion of time we use is only important for this selection process.

[2]At first glance this requirement may seem like a subtlety, but it is actually critical for achieving a sublinear competitive ratio.

generated on e between two consecutive traversals of e by OPT. If we can show that this payment is comparable to the length $\ell(e)$ of the edge we have our desired result, since the payment reflects the online cost on the edge. However, this approach fails as one can easily construct scenarios in which OPT can avoid using some edge for a long time at the cost of using other edges much more frequently. This means we cannot compare the optimal and online cost on an edge-by-edge basis.

In order to account for this we introduce the notion of *discount*. We say that a request that is in ONLINE's or OPT's buffer at time t generates a *discount* of $dt/4D$ during the time interval $[t, t + dt)$ on all edges between its position and v_{onl}. Note that this discount generation is only used for the analysis. Hence, we can assume that OPT is known.

OBSERVATION 5. *The total generated discount is at most half the total payment.*

PROOF. The total number of requests that generate discount is $2k$. Each of these requests generates discount on at most D edges. This means that in a time interval of length dt a total discount of at most $k \cdot dt/2$ is generated. On the other hand the k requests stored in the online buffer generate a payment of $k \cdot dt$ in each time interval of length dt. $\square$

The idea now is to analyze the *real payment* created on an edge e, i. e., the payment minus the discount, between two consecutive traversals of e by OPT. If we can show that this is less than $f \cdot \ell(e)$, for some factor f, then the above observations give a competitive ratio of $O(f)$.

Unfortunately, this approach still fails. To overcome the remaining difficulties we allow payment to be moved between edges. To describe this approach formally, we introduce the counters $\mathsf{buffer}(e)$, $\mathsf{accepted}(e)$, and $\mathsf{discount}(e)$, in addition to the counter $\mathsf{pay}(e)$, for each edge e. A rough intuition behind these counters is as follows.

- The counter $\mathsf{pay}(e)$ describes the payment on e as generated by the algorithm. In particular this means that, for time t, $0 \leq \mathsf{pay}_t(e) \leq \ell(e)$, where $\mathsf{pay}_t(e)$ denotes the value of the counter at time t.
- The intuition of the counter $\mathsf{accepted}(e)$ is that it stores the whole payment that is generated on e during the run-time of ONLINE (However, this is not exactly true. See below.). Finally, we want to show that $\mathsf{accepted}(e) \leq f \cdot \text{OPT}(e)$, for some factor f, where $\text{OPT}(e)$ denotes the cost of OPT on e.

  From time to time the algorithm resets the counter $\mathsf{pay}(e)$ to 0. Consequently, we would have to increase $\mathsf{accepted}(e)$ accordingly if the above intuition behind the counter $\mathsf{accepted}(e)$ was correct. Instead we sometimes increase the counter $\mathsf{accepted}(e')$ of some other edge e'. Basically, this means that we have moved the payment from edge e to edge e'.
- Sometimes we neither increase $\mathsf{accepted}(e)$ nor the counter $\mathsf{accepted}(e')$ of some other edge e' accordingly. In these cases the counter $\mathsf{buffer}(e)$ comes into play.

  Instead of increasing an $\mathsf{accepted}(e)$-counter we increase the counter $\mathsf{buffer}(e)$. This payment will later be accepted on e by adding the value of $\mathsf{buffer}(e)$ to $\mathsf{accepted}(e)$ and resetting $\mathsf{buffer}(e)$ to 0. The reason for differentiating between $\mathsf{accepted}(e)$ and $\mathsf{buffer}(e)$ is purely technical. Our goal is to show that whenever $\mathsf{accepted}(e)$ is increased, "something terrible" is happening for OPT. This "something" cannot happen too often without OPT traversing e. Because of this approach we have to take special care when to increase $\mathsf{accepted}(e)$. Hence, we sometimes use $\mathsf{buffer}(e)$ to delay an increase of $\mathsf{accepted}(e)$.
- The counter $\mathsf{discount}(e)$ can be viewed as describing the discount generated on an edge e. This means that whenever a request generates discount on an edge e the counter $\mathsf{discount}(e)$ is increased by the respective value.

The counter $\mathsf{pay}(e)$ is only changed by the online algorithm. Further, the increments of the $\mathsf{discount}(e)$-counter are also solely determined by the algorithms. The other counter changes, i. e., the changes to $\mathsf{buffer}(e)$ and $\mathsf{accepted}(e)$ and the decrements of $\mathsf{discount}(e)$, are determined by a counter manipulation scheme that we describe in the following section.

This scheme has the following properties.

PROPERTY 1 (PAYMENT IS CONSERVED). *In a manipulation step, the total payment on edges i. e.,*

$$\sum_e (\mathsf{pay}(e) + \mathsf{buffer}(e) + \mathsf{accepted}(e) - \mathsf{discount}(e)) \ ,$$

does not decrease.

PROPERTY 2 (NOT ACCEPTED PAYMENT IS MARGINAL). *For each point in time t and each edge e,*

$$\mathsf{pay}_t(e) + \mathsf{buffer}_t(e) \leq 2 \cdot \ell(e)$$

and, furthermore,

$$\mathsf{pay}_t(e) + \mathsf{buffer}_t(e) \leq 2 \cdot \text{OPT}(e) \ ,$$

where $\text{OPT}(e)$ denotes the cost of OPT on e.

PROPERTY 3 (ACCEPTED PAYMENT IS BOUNDED). *For each edge e,*

$$\mathsf{accepted}(e) \leq O(D \cdot \log k) \cdot \text{OPT}(e) \ .$$

Basically, the first two properties guarantee that in the end (nearly) all generated payment is stored in the $\mathsf{accepted}(e)$-counters. Property 3 combined with the previous observations then give a bound of $\sum_e O(D \cdot \log k) \cdot \text{OPT}(e)$ on the online cost induced by all processing steps during the run-time of ONLINE. This means that ONLINE is $O(D \cdot \log k)$-competitive as the cost induced by the clean-up phase is trivially bounded by the optimum cost.

3.1 Manipulation Scheme

We describe our scheme for manipulating the counters such that the scheme fulfills the properties presented in the previous section. Suppose ONLINE traverses a connected component starting at some node v_{onl} and ending at node $\hat{v}_{\text{onl}}$, which is farthest from v_{onl} among all nodes of the component. This induces the following changes in the counters.

1. For each edge e in the component, $\mathsf{pay}(e)$ is reset to 0 as ONLINE deletes the payment on all edges of the component.

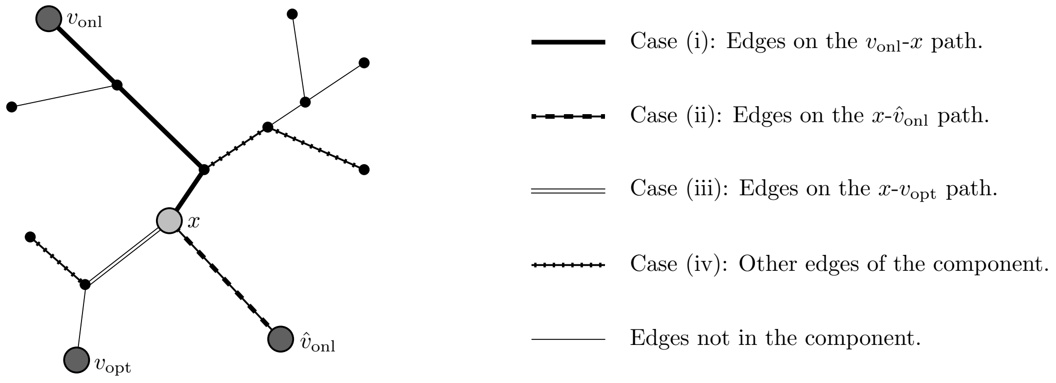
v_{onl}
x
$\hat{v}_{\text{onl}}$
v_{opt}
Case (i): Edges on the v_{onl}-x path.
Case (ii): Edges on the x-$\hat{v}_{\text{onl}}$ path.
Case (iii): Edges on the x-v_{opt} path.
Case (iv): Other edges of the component.
Edges not in the component.

Figure 1: The different types of edges in the component traversed by ONLINE.

In order to guarantee Property 1 (payment is conserved), we have to increase other counters. This is done as follows.

2. Let x denote the node at which the paths from v_{onl} to $\hat{v}_{\text{onl}}$ and from v_{onl} to v_{opt} split, where v_{opt} denotes the current position of the optimal server in the tree. We partition the edges of the connected component into four different sets.

   The first set contains the edges on the v_{onl}–x path, the second set contains the edges on the x–$\hat{v}_{\text{onl}}$ path, the third set contains the edges on the x–v_{opt} path[3], and the fourth set contains all other edges of the connected component. Figure 1 gives an overview over the different types of edges.

   The counters are changed as follows.

   (i) Each edge e on the path from v_{onl} to x increases its $\mathsf{buffer}(e)$-counter by the length $\ell(e)$ of the edge. This exactly offsets the decrease caused by resetting $\mathsf{pay}(e)$ to 0. Additionally the $\mathsf{discount}(e)$-counter is reset to 0. Altogether, the payment on these edges does not decrease.

   (ii) Each edge e on the path from x to $\hat{v}_{\text{onl}}$ increases the counter $\mathsf{accepted}(e)$ by $2\ell(e) + \mathsf{buffer}(e) - \mathsf{discount}(e)$ and resets the counters $\mathsf{buffer}(e)$ and $\mathsf{discount}(e)$ to 0. This means that the increase in $\mathsf{accepted}(e)$ exceeds the decrease of $\mathsf{pay}(e) + \mathsf{buffer}(e) - \mathsf{discount}(e)$ by the length $\ell(e)$ of the edge. Altogether, the payment on these edges increases by the length of the x–$\hat{v}_{\text{onl}}$ path.

   (iii) Each edge e on the path from x to v_{opt} does not increase any of its counters, it only resets the counter $\mathsf{discount}(e)$ to 0. This means that the payment on these edges actually decreases due to the $\mathsf{pay}(e)$-counter being reset to 0. The total decrease is at most the length of the intersection of the x–v_{opt} path with the component.

   In order to account for this we observe that, since $\hat{v}_{\text{onl}}$ is the farthest node from v_{onl} in the connected component, the length of the x–$\hat{v}_{\text{onl}}$ path is larger than the intersection of the x–v_{opt} path with the component. Hence, we can amortize the decrease against the excess on the edges on the x–$\hat{v}_{\text{onl}}$ path (see Case ii).

   (iv) All remaining edges of the component increase their counter $\mathsf{accepted}(e)$ by $\ell(e) + \mathsf{buffer}(e) - \mathsf{discount}(e)$ and reset their counters $\mathsf{buffer}(e)$ and $\mathsf{discount}(e)$ to 0. This means that their increase in $\mathsf{accepted}(e)$ exactly offsets the decrease of $\mathsf{pay}(e) + \mathsf{buffer}(e) - \mathsf{discount}(e)$.

   In addition to the counter changes above, OPT triggers other counter changes when it moves its server between two online phases.

   (v) When OPT moves its server over an edge e, the counter $\mathsf{accepted}(e)$ is increased by $\mathsf{buffer}(e)$ and the counter $\mathsf{buffer}(e)$ is reset to 0. This counter manipulation does also not decrease the payment.

[3]Note that in general not all edges on the x–v_{opt} path are contained in the component.

3.2 Properties of the Manipulation Scheme

The above counter manipulation scheme fulfills Property 1, i. e., the payment conservation, by design. Further, it has the property that at the end of each phase the counters $\mathsf{pay}(e)$ and $\mathsf{discount}(e)$ for an edge in the connected component are 0. The proof of the following lemma shows that the $\mathsf{buffer}(e)$-counter is either 0 or $\ell(e)$.

LEMMA 6. *For each edge e, the value of* $\mathsf{buffer}(e)$ *is at most the length $\ell(e)$ of the edge.*

PROOF. Fix an edge $e = (u, v)$, and let T_u and T_v denote the two trees that are obtained when deleting e from T. Suppose that $\mathsf{buffer}(e)$ has been increased to $\ell(e)$ in the last phase. Note that this only happens in Case i. This means that e has been on the v_{onl}–x path in the component just processed by ONLINE. Now, the online and offline server are both on the same side of e, i. e., they are either both in T_u or both in T_v. Assume without loss of generality that they are located in T_u.

We show that the counter $\mathsf{buffer}(e)$ is reset to 0 before the next increase occurs. First, assume that the next change to $\mathsf{buffer}(e)$ happens because OPT traverses the edge. However, OPT traversing the edge means that $\mathsf{buffer}(e)$ is reset to 0 (Case v).

Now, assume that the next change to $\mathsf{buffer}(e)$ occurs because e is part of the connected component traversed by ONLINE. Then v_{opt}, v_{onl}, and x are all located in T_u because OPT did not traverse the edge. This means that e is either an edge on the x–$\hat{v}_{\text{onl}}$ path (Case ii) or one of the other edges of the component (Case iv). In both cases $\mathsf{buffer}(e)$ is reset to 0. □

Lemma 7. *The counter manipulation scheme fulfills Property 2.*

Proof. The $\mathsf{pay}(e)$-counter and the $\mathsf{buffer}(e)$-counter are both bounded by the length $\ell(e)$ of the edge, which gives the first statement of Property 2.

All counters of an edge have value 0 if ONLINE never generates payment on this edge. The edges on a minimum Steiner tree connecting the initial starting position to all requests from the input sequence are the only edges on which ONLINE generates payment. OPT has to visit all these edges at least once. This completes the second statement of Property 2. □

It remains to show that the counter manipulation scheme fulfills Property 3. We need to compare the accepted payment on an edge e to the cost of OPT on e. Note that whenever $\mathsf{accepted}(e)$ is changed (Case ii and Case iv), it increases by at most $2\ell(e) + \mathsf{buffer}(e) \leq 3\ell(e)$. However, it may also decrease depending on the value of the $\mathsf{discount}(e)$-counter.

In order for the inequality $\mathsf{accepted}(e) \leq O(D \cdot \log k) \cdot \text{OPT}(e)$ to be violated there need to be long sequences of changes to the counter $\mathsf{accepted}(e)$ — and many of these changes have to *increase* the counter — without OPT visiting e, as this would increase $\text{OPT}(e)$ by $\ell(e)$. The following lemma forms the crucial part of our analysis and shows that this is not possible.

Lemma 8. *Let $[i_{start}, \dots, i_{end}]$ denote a sequence of consecutive phases during which OPT does not traverse edge e. Then the number of phases $i \in [i_{start}, \dots, i_{end}]$ in which the counter $\mathsf{accepted}(e)$ increases is at most $O(D \cdot \log k)$.*

Proof. Let T_u and T_v denote the trees obtained when deleting e from T, and assume without loss of generality that at the beginning of the phase i_{start} the optimal server is located in T_u. We call a request *opt-exclusive* (in phase i) if at the beginning of the phase the request is in OPT's buffer but not in ONLINE's buffer. Similarly, we call a request *online-exclusive* if it is held by ONLINE but not by OPT.

Let $\mathsf{onl\text{-}excl}_i(T_v)$ and $\mathsf{opt\text{-}excl}_i(T_v)$ denote the number of online-exclusive and opt-exclusive requests, respectively, that are in sub-tree T_v at the beginning of phase i. Note that during phases in $[i_{\text{start}}, \dots, i_{\text{end}}]$ the number of online-exclusive requests in T_v cannot increase and the number of opt-exclusive requests in T_v cannot decrease, as this would require OPT to visit the sub-tree.

Let $i_{\text{first}} \geq i_{\text{start}}$ denote the first phase in which the $\mathsf{accepted}(e)$-counter changes. If such a phase does not exist, then the lemma obviously holds. The following proposition shows that an increase in the counter $\mathsf{accepted}(e)$ occurring after i_{first} is always accompanied by either a large decrease in $\mathsf{onl\text{-}excl}_i(T_v)$ or a large increase in $\mathsf{opt\text{-}excl}_i(T_v)$. This allows us to derive a bound on the total number of increases to the $\mathsf{accepted}(e)$-counter during phases in $[i_{\text{start}}, \dots, i_{\text{end}}]$.

Proposition 9. *Let $i \in [i_{first}+1, \dots, i_{end}]$ denote a phase in which the counter $\mathsf{accepted}(e)$ increases. Then either*

$$\mathsf{opt\text{-}excl}_{i+1}(T_v) > \left(1 + \frac{1}{12D}\right) \cdot \mathsf{opt\text{-}excl}_i(T_v)$$

or

$$\mathsf{onl\text{-}excl}_{i+1}(T_v) < \left(1 - \frac{1}{12D}\right) \cdot \mathsf{onl\text{-}excl}_i(T_v) \ .$$

Proof. First observe that in the beginning of the phase i the online server is located in T_u, as otherwise e lies either on the v_{onl}–x path or on the x–v_{opt} path, and hence no payment would be accepted on e (Case i and Case iii).

Let n_{rem} denote the number of requests that generate payment on e in phase i. Note that since the online server is located in T_u all these payment generating requests are in T_v. Further, observe that all these requests are removed from the online buffer at the end of phase i. Let $n_{\text{rem}}^{\text{opt}} \leq n_{\text{rem}}$ denote the number of payment generating requests that are held by OPT and by ONLINE, and let $n_{\text{rem}}^{\text{onl-excl}}$ denote the number of *online-exclusive* requests that generate payment on e. Note that $n_{\text{rem}} = n_{\text{rem}}^{\text{opt}} + n_{\text{rem}}^{\text{onl-excl}}$.

Observe that all requests contributing to $n_{\text{rem}}^{\text{opt}}$ are held by OPT and are removed from ONLINE's buffer at the end of phase i. Hence, these requests become opt-exclusive for phase $i+1$. Similarly, requests contributing to $n_{\text{rem}}^{\text{onl-excl}}$ are removed from ONLINE's buffer and decrease $\mathsf{onl\text{-}excl}(T_v)$ accordingly. Hence,

$$\begin{aligned} \mathsf{opt\text{-}excl}_{i+1}(T_v) - \mathsf{opt\text{-}excl}_i(T_v) &= n_{\text{rem}}^{\text{opt}} \quad \text{and} \\ \mathsf{onl\text{-}excl}_i(T_v) - \mathsf{onl\text{-}excl}_{i+1}(T_v) &= n_{\text{rem}}^{\text{onl-excl}} \ . \end{aligned}$$

Now assume for contradiction that

$$\begin{aligned} \mathsf{opt\text{-}excl}_{i+1}(T_v) &\leq \left(1 + \frac{1}{12D}\right) \cdot \mathsf{opt\text{-}excl}_i(T_v) \quad \text{and} \\ \mathsf{onl\text{-}excl}_{i+1}(T_v) &\geq \left(1 - \frac{1}{12D}\right) \cdot \mathsf{onl\text{-}excl}_i(T_v) \ , \end{aligned}$$

which gives

$$\frac{\mathsf{opt\text{-}excl}_i(T_v)}{12D} + \frac{\mathsf{onl\text{-}excl}_i(T_v)}{12D} \geq n_{\text{rem}}^{\text{opt}} + n_{\text{rem}}^{\text{onl-excl}} = n_{\text{rem}} \ .$$

Let $i_{\text{first}} \leq j < i$ be the most recent phase before phase i during which ONLINE visited T_v. The requests contributing to n_{rem} are the only requests that generate payment on e during the phases $j+1, \dots, i$. All the requests contributing to $\mathsf{opt\text{-}excl}_i(T_v)$ and $\mathsf{onl\text{-}excl}_i(T_v)$ generate discount on e during these phases. Note that the number of opt-exclusive and online-exclusive requests in T_v does not change during the phases $j+1, \dots, i-1$, since the online and the optimal server are both located in T_u during these phases.[4] Therefore the total discount generated on e during these phases is at least

$$\begin{aligned} \mathsf{discount}(e) &\geq \frac{\mathsf{pay}(e)}{n_{\text{rem}}} \cdot \frac{\mathsf{opt\text{-}excl}_i(T_v) + \mathsf{onl\text{-}excl}_i(T_v)}{4D} \\ &\geq \frac{\mathsf{pay}(e)}{n_{\text{rem}}} \cdot \frac{12D \cdot n_{\text{rem}}}{4D} \\ &= 3\mathsf{pay}(e) \\ &\geq 3\ell(e) \\ &\geq 2\ell(e) + \mathsf{buffer}(e) \ , \end{aligned}$$

where the first step follows since $\mathsf{pay}(e) = 0$ at the beginning of phase $j+1$ and $\mathsf{pay}(e) = \ell(e)$ right before the processing step of phase i.

However, the counter $\mathsf{accepted}(e)$ increases by at most $2\ell(e) + \mathsf{buffer}(e) - \mathsf{discount}(e)$ (Case ii), and hence $\mathsf{accepted}(e)$ does not increase in phase i. This is a contradiction which completes the proof of the proposition. □

[4]This means that the following contradiction also holds if we only consider the discounts on e created by requests in T_v that already existed at the start of phase $j+1$. These are *old* requests according to the definition in Section 4.

Now, we can deduce the lemma from the proposition above. Since the number of opt-exclusive and online-exclusive requests in T_v are both bounded by k, the counter $\mathsf{accepted}(e)$ can only increase $O(D \cdot \log k)$ times. □

The lemma directly implies that our counter manipulation scheme fulfills Property 3. As previously outlined, the existence of a scheme that fulfills all three properties guarantees that ONLINE achieves a competitive ratio of $O(D \cdot \log k)$.

4. HST ANALYSIS

In this section, we give an improved analysis for the competitive ratio of our online algorithm on metric spaces that can be represented as the shortest path metric induced by the leaf nodes of so-called 2-HSTs.

Definition 10. *A* 2*-HST is a rooted tree such that*

- *all leaf nodes are at the same distance from the root,*
- *all edges on the same level, i. e., distance from the root, have the same length, and*
- *the length of each edge connecting a level i node to a level $i+1$ node is half the length of an edge connecting a level $i-1$ node to a level i node.*

Remark 11. *The literature contains several slightly different definitions of c-HSTs. We note that, for constant c, all these c-HSTs can be suitably approximated by a* 2*-HST according to the above definition. Also, the restriction to leaf nodes can be easily removed using this technique. This means that Theorem 2 follows from the arguments about* 2*-HSTs in this section.*

The key idea for improving the analysis of the previous section for the special case of HSTs is to generate and distribute the discount in a more sophisticated manner. The goal is to increase the amount of discount an edge receives in a time interval $[t, t+dt)$ by a single request from $dt/4D$ to dt/z, for some parameter $z \ll 4D$. If we can do this while maintaining the properties of the discount distribution, Theorem 1 will improve to a competitive ratio of $O(z \cdot \log k)$.

More precisely, we change the discount generation and counter manipulation in such a way that Observation 5, Property 1, Property 2, and Lemma 6 hold. Then the competitive ratio of the algorithm only hinges upon Lemma 8, i. e., how often the counter $\mathsf{accepted}(e)$ of an edge e can increase without OPT traversing the edge.

Observation 12. *Property 1, Property 2, and Lemma 6 hold independently of the discount generation process.*

This means that changing the amount of discount created by a request can only change Observation 5 (bound on the total amount of discount) and Lemma 8 (bound on the number of increases to $\mathsf{accepted}(e)$).[5]

Fix an edge $e = (u, v)$, where u is the parent of v, and let T_u and T_v denote the trees obtained when deleting e from T. Suppose that the online server is located in T_u and let v_{onl} denote its current position. We call a request in T_v *old* if it already existed right after ONLINE traversed e (from v to u) the last time. The first change to our discount generation scheme is that only *old requests* generate discount on an edge.

Observation 13. *If in a time interval $[t, t+dt)$ a request generates a discount of dt/z only on edges for which it is an old request, then Lemma 8 holds with a bound of $O(z \cdot \log k)$.*

Proof. As already mentioned in Footnote 4 in the proof of Lemma 8, the proof also works if only old requests generate discount. Further, the rate of discount generation directly influences the bound in the lemma. □

Further, we change the discount generation by defining that a request does not generate discount on edges that are ancestor edges of v_{onl} in the tree. This means that a request at position u generates discount on an edge e only if

- e lies on the u–v_{onl} path, and
- e is an ancestor edge of u.

Unfortunately, changing the discount generation in this way creates a problem in our analysis when we accept payment on an edge that is an ancestor edge of v_{onl} because Lemma 8 may not hold anymore. (Informally stated: If we still accept payment when traversing such edges, $\mathsf{accepted}(e)$ may increase very often because there is no discount generated on these edges anymore.) An edge e that at some point is an ancestor edge of v_{onl}, and hence gets a reduced discount, must lie on the v_{onl}–$\hat{v}_{\text{onl}}$ path the next time it is contained in the connected component because $\hat{v}_{\text{onl}}$ is chosen as the farthest node. In the case that e lies on the v_{onl}–x portion of this path the discount on e is not used and simply set to 0 (Case i). This means that the reduced discount that e receives does no harm.

The problematic edges lie on the x–$\hat{v}_{\text{onl}}$ path (Case ii) by the next time they are contained in the connected component. To deal with these edges, we change our counter manipulation scheme in the following way. Let r denote the root of the connected component, i. e., the node on the lowest level in the component. We change the scheme by splitting Case ii, i. e., edges on the x–$\hat{v}_{\text{onl}}$ path, into two sub-cases, namely edges in the intersection of the r–$\hat{v}_{\text{onl}}$ path and the x–$\hat{v}_{\text{onl}}$ path (downward edges) and edges in the intersection of the v_{onl}–r path and the x–$\hat{v}_{\text{onl}}$ path (upward edges) (see Figure 2).

(ii.a) If the edge e is an ancestor edge of v_{onl}, i. e., e lies in the intersection of the v_{onl}–r path and the x–$\hat{v}_{\text{onl}}$ path, reset the counters $\mathsf{buffer}(e)$ and $\mathsf{discount}(e)$ to 0 *without* increasing the counter $\mathsf{accepted}(e)$.

(ii.b) If the edge e is not an ancestor edge of v_{onl}, i. e., e lies in the intersection of the r–$\hat{v}_{\text{onl}}$ path and the x–$\hat{v}_{\text{onl}}$ path, increase the counter $\mathsf{accepted}(e)$ by $3\ell(e) + \mathsf{buffer}(e) - \mathsf{discount}(e)$, and then reset the counters $\mathsf{buffer}(e)$ and $\mathsf{discount}(e)$ to 0. This means that the increase of the counter $\mathsf{accepted}(e)$ exceeds the decrease of $\mathsf{pay}(e) + \mathsf{buffer}(e) - \mathsf{discount}(e)$ by $2\ell(e)$.

This excess is used to counteract the decrease in $\mathsf{pay}(e)$ on each edge e of the component on the x–v_{opt} path (Case iii) and the decrease in $\mathsf{pay}(e) + \mathsf{buffer}(e)$ on each edge e in the intersection of the v_{onl}–r path and the x–$\hat{v}_{\text{onl}}$ path (Case ii.a). We argue that the excess is

[5]Recall that the sole purpose of the discount generation scheme is to make a lot of these counter changes to $\mathsf{accepted}(e)$ negative or zero, so that the number of increases can be bounded (Lemma 8).

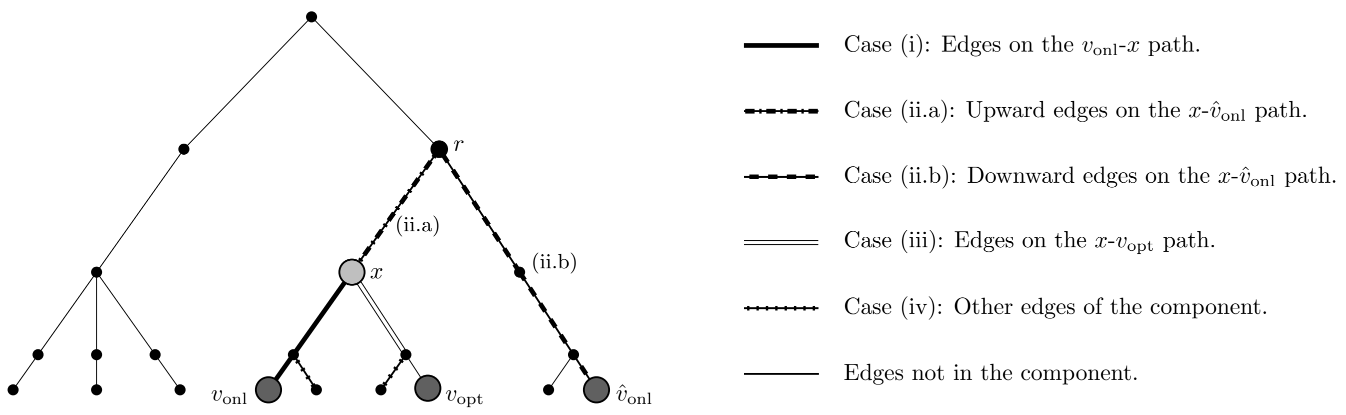
r
(ii.a)
(ii.b)
x
v_{onl}
v_{opt}
$\hat{v}_{\mathrm{onl}}$
Case (i): Edges on the v_{onl}-x path.
Case (ii.a): Upward edges on the x-$\hat{v}_{\mathrm{onl}}$ path.
Case (ii.b): Downward edges on the x-$\hat{v}_{\mathrm{onl}}$ path.
Case (iii): Edges on the x-v_{opt} path.
Case (iv): Other edges of the component.
Edges not in the component.

Figure 2: The different types of edges considered in the HST analysis.

at least as large as this decrease. First observe that the counter decrease on any of these edges is at most $2\ell(e)$, i. e., we only need to show that the total length of edges generating excess (Case ii.b edges) is larger than the length of edges that experience a decrease.

First, assume that the root r of the component lies on the x–$\hat{v}_{\mathrm{onl}}$ path. Then edges experiencing a decrease are edges on the x–v_{opt} path and edges on the x–r path, while all edges on the r–$\hat{v}_{\mathrm{onl}}$ path generate excess. The argument follows, since v_{opt} and $\hat{v}_{\mathrm{onl}}$ are both leaf nodes, and hence the r–v_{opt} path and the r–$\hat{v}_{\mathrm{onl}}$ path have the same length.

Now, assume that the root r does not lie on the x–$\hat{v}_{\mathrm{onl}}$ path, i. e., there are no edges corresponding to Case ii.a. Then edges experiencing a decrease are located on the x–v_{opt} path, while all edges on the x–$\hat{v}_{\mathrm{onl}}$ path generate excess. The argument follows, since v_{opt} and $\hat{v}_{\mathrm{onl}}$ are both leaf nodes, and hence the x–v_{opt} path and the x–$\hat{v}_{\mathrm{onl}}$ path have the same length.

This shows that the new counter manipulation scheme still fulfills Property 1.

The new counter manipulation scheme leads to the following result.

Observation 14. *If a request r at position u generates a discount of dt/z on each edge e for which*

- *r is an old request,*
- *e is on the path from u to v_{onl}, and*
- *e is an ancestor edge of u,*

then Lemma 8 holds with a bound of $O(z \cdot \log k)$.

The above restriction in the discount generation reduces the number of edges on which a request generates discount to the height h of the tree. If we generate discount at a rate $dt/4h$ per time interval of length dt, i. e., a uniform rate among all edges on which discount is generated, Observation 5 (bound on the total discount) still holds, and together with the above observation this would give a competitive ratio of $O(h \cdot \log k)$, which however is no asymptotic improvement over $O(D \cdot \log k)$.

To further improve the discount generation process, we observe that a discount of more than $4\ell(e)$ on an edge is useless and will never be used by the scheme. The reason is that Lemma 8 (the central part of our argument) counts how often $\mathsf{accepted}(e)$ is increased. Because of our bound on $\mathsf{buffer}(e)$ from Lemma 6 this is never the case if the discount on an edge exceeds $4\ell(e)$ (see Case ii.b and Case iv).

The idea now is to change the discount generation in such a way that for some very small edges much less discount is generated but that the generation is still sufficient to guarantee that by the next counter change the accumulated discount on such an edge e will be at least $4\ell(e)$.

Fix a request o, and let r denote the node with lowest level on the o–v_{onl} path. We only generate discount on edges on the path from r to o, and only on edges for which o is an old request. We call the first $\log k + 6$ edges on the r–o path *long edges* and the remaining edges on this path *short edges*. Now we define that in a time interval of length dt, the request o generates a discount of $dt/(8(\log k + 6))$ on every long edge and a discount of $k \cdot dt \cdot 4\ell(e)/\ell(e_{\max})$ on every short edge e, where $e_{\max}$ denotes the longest edge on the r–o path. Altogether, the total discount generated by this request is at most

$$\begin{aligned}\sum_{\text{long edge } e} \frac{dt}{8(\log k+6)} &+ \sum_{\text{short edge } e} \frac{k \cdot dt \cdot 4\ell(e)}{\ell(e_{\max})}\\ &\le \frac{dt}{8} + \frac{4k \cdot dt}{\ell(e_{\max})} \cdot \sum_{\text{short edge } e} \ell(e)\\ &\le \frac{dt}{4}\ ,\end{aligned}$$

where the last step follows since in a 2-HST the edge lengths are decreasing by a factor of 2, and hence $\sum_{\text{short edge } e} \ell(e) \le \ell(e_{\max})/(32k)$. This implies that Observation 5 holds, i. e., the total generated discount is at most half the total payment.

Further, every long edge receives discount at a rate of $\Theta(dt/\log k)$, which is fairly large. The following lemma shows that the reduced discount on short edges does not cause any additional increases to the $\mathsf{accepted}(e)$-counters.

Lemma 15. *If an edge $e = (u, v)$, where u is the parent of v, receives discount as a short edge for some old request o in T_v, the accumulated discount on e is at least $4\ell(e)$ by the time the next change to $\mathsf{accepted}(e)$ triggered by ONLINE occurs.*

This means that if an edge receives discount for being short, the next counter change to $\mathsf{accepted}(e)$ triggered by

ONLINE will not be an increase, i. e., it does not hurt at all that short edges receive less discount. Hence, the total number of increases during a sequence of phases in which OPT does not traverse e is $O(\log^2 k)$ due to Observation 14, because we generate the same number of increases as a scheme with uniform rate $dt/\Theta(\log k)$.

PROOF OF LEMMA 15. Assume that by the time of the next counter change on e the smallest contribution rate by request o to the discount of e has been $k \cdot dt \cdot 4\ell(e)/\ell(e_{\max})$, where $e_{\max} = (u', v')$ is some ancestor edge of e and u' is the parent node of v'. Now, consider the most recent traversal of $e_{\max}$ by ONLINE in direction from v' to u'.

At this point the request o already exists in T_v and generates a discount of $k \cdot dt \cdot 4\ell(e)/\ell(e_{\max})$ on e in every time interval of length dt. Right after ONLINE moved over $e_{\max}$ there is no payment on this edge, and in order for ONLINE to return into the sub-tree $T_{v'}$ the edge $e_{\max}$ has to be paid for. However, in a time interval of length dt only a total payment of $k \cdot dt$ is generated by the k requests stored in ONLINE's buffer. Hence, by the time ONLINE returns into $T_{v'}$ a discount of at least $4\ell(e)$ has been generated by the request o on the edge e. □

5. CONCLUSIONS

Our online algorithm does not need to know the tree metric space in advance. However, one limitation of our results is that to derive a strategy for general metric spaces we need to know the metric in advance in order to compute a FRT-tree of the metric [7], and then to run the online algorithm on this tree. Interestingly, using the decomposition due to Bartal [3] can help here. We can obtain a competitive ratio of $O(\log^3 \delta \cdot \log k)$, where δ denotes the aspect ratio of the metric space, i. e., the maximal ratio of any two distances in the metric space, without knowing the metric in advance. This is achieved by constructing a Bartal-tree of the decomposition in an online manner.

It is an interesting open problem whether it is also possible to obtain a polylogarithmic dependency on k and n (as opposed to δ) without knowing the metric space in advance. Further, it seems interesting to investigate whether the FRT-scheme can be adapted, as well, to allow for an online construction of the tree. This would lead to an improved competitive ratio for the scenario in which the metric space is initially unknown.

We strongly believe that in a refined analysis of our algorithm it might be possible to remove the dependency on D for trees, and to show a competitive ratio that only has a polylogarithmic dependency on k. An even more challenging task would be to remove the dependency on n for arbitrary metric spaces. However, this problem seems to require entirely new techniques as one needs to find a way around using Bartal/FRT as a black box tool.

6. REFERENCES

[1] H. Alborzi, E. Torng, P. Uthaisombut, and S. Wagner. The k-client problem. *Journal of Algorithms*, 41(2):115–173, 2001.

[2] R. Bar-Yehuda and J. Laserson. Exploiting locality: Approximating sorting buffers. In *Proceedings of the 3rd Workshop on Approximation and Online Algorithms (WAOA)*, pages 69–81, 2005.

[3] Y. Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–193, 1996.

[4] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 30th ACM Symposium on Theory of Computing (STOC)*, pages 161–168, 1998.

[5] M. Englert, H. Röglin, and M. Westermann. Evaluation of online strategies for reordering buffers. In *Proceedings of the 5th International Workshop on Efficient and Experimental Algorithms (WEA)*, pages 183–194, 2006.

[6] M. Englert and M. Westermann. Reordering buffer management for non-uniform cost models. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 627–638, 2005.

[7] J. Fakcharoenphol, S. B. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *Journal of Computer and System Sciences*, 69(3):485–497, 2004.

[8] I. Gamzu and D. Segev. Improved online algorithms for the sorting buffer problem. In *Proceedings of the 24th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 658–669, 2007.

[9] K. Gutenschwager, S. Spiekermann, and S. Voß. A sequential ordering problem in automotive paint shops. *International Journal of Production Research*, 42(9):1865–1878, 2004.

[10] R. Khandekar and V. Pandit. Offline sorting buffers on line. In *Proceedings of the 17th International Symposium on Algorithms and Computation (ISAAC)*, pages 81–89, 2006.

[11] R. Khandekar and V. Pandit. Online sorting buffers on line. In *Proceedings of the 23rd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 584–595, 2006.

[12] J. S. Kohrt and K. Pruhs. A constant factor approximation algorithm for sorting buffers. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 193–202, 2004.

[13] J. Krokowski, H. Räcke, C. Sohler, and M. Westermann. Reducing state changes with a pipeline buffer. In *Proceedings of the 9th International Fall Workshop Vision, Modeling, and Visualization (VMV)*, pages 217–224, 2004.

[14] H. Räcke, C. Sohler, and M. Westermann. Online scheduling for sorting buffers. In *Proceedings of the 10th European Symposium on Algorithms (ESA)*, pages 820–832, 2002.

[15] T. J. Teorey and T. B. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177–184, 1972.