THEORY OF COMPUTING, Volume 6 (2010), pp. 27–46
www.theoryofcomputing.org

Reordering Buffers for General Metric Spaces*

Matthias Englert† Harald Räcke‡ Matthias Westermann§

Received: December 18, 2008; published: February 15, 2010.

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. When serving a request the cost is equal to the distance, in the metric space, between this request and the previously served request. 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 where the goal is to minimize the total cost. 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

*An earlier version of this work appeared in Proceedings of the 39th Annual ACM Symposium on Theory of Computing (STOC), 2007.

†Supported in part by DFG grant WE 2842/1, EPSRC grant EP/F043333/1, and the Centre for Discrete Mathematics and its Applications (DIMAP).

‡Supported in part by the EU within the 6th Framework Programme under contract 001907 (DELIS) and the Centre for Discrete Mathematics and its Applications (DIMAP).

§Supported in part by DFG grant WE 2842/1.

ACM Classification: F.2.2

AMS Classification: 68W27

Key words and phrases: online algorithms, reordering buffer, sorting buffer, general metric spaces

DOI: 10.4086/toc.2010.v006a002

deterministic algorithm for weighted trees whose competitive ratio depends on k and the hop-diameter of the tree. Then we show how to improve this competitive ratio to $O(\log^2 k)$ for metric spaces that correspond to hierarchically well-separated trees. Combining this result with the results on the probabilistic approximation of arbitrary metrics by tree metrics due to Fakcharoenphol, Rao, and Talwar, 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.

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.

Another formulation of the problem is the following: An undirected weighted graph G and an input sequence of requests which correspond to vertices in G is given. At each point in time the first k unprocessed requests from the input sequence are located at their respective vertices in the graph. The scheduling algorithm controls one server that moves through the graph. The initial position of the server can be chosen arbitrarily from the vertices that contain at least one of the first k requests from the input sequence without cost. Thereafter, to serve a request, the server has to be moved to the vertex containing that request. The cost of this movement is given by the length of the chosen path. If a request is served, it is removed from the graph and the next request from the input sequence is placed at its corresponding vertex. The objective is to process all requests from the input sequence while minimizing the total distance the server moves.

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 [4, 9, 13, 14, 16]).

- In hard disks, the latency of a disk 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., [17]).
- 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 inserted 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 [15]).

- 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., [12]).

In this paper, we design online scheduling algorithms for the reordering buffer problem where the goal is to minimize the total cost. 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 and the requests already served. The cost of the online algorithm is compared to the cost of an optimal offline strategy which knows all requests in the input sequence in advance. However, the optimal offline strategy also has to use the limited capacity buffer to reorder the input sequence. Note that if the buffer size k is equal to the length of the input sequence, we allow arbitrary reorderings. Therefore there exists a straightforward reduction from the traveling salesperson problem and hence, in general, the offline problem of finding an optimal reordering is NP-hard. On the other hand, using dynamic programming the problem can be solved in time $O(N^{k+1})$ where N is the number of requests in the input sequence (see [13]).

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 [11, 13] and star metrics [1, 8, 9, 16]. 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 hop-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 correspond to hierarchically well-separated tree (HSTs). Combining this result with the results on probabilistically approximating arbitrary metrics by tree metrics [5, 6, 10], 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

Räcke, Sohler, and Westermann [16] introduced the reordering buffer problem for the uniform metric, in which two points x and y are either at distance 0 (if $x = y$) or at distance 1 (if $x \neq y$). 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 [9] to a competitive ratio of $O(\log k)$ and later to $O(\log k/(\log\log k))$ by Avigdor-Elgrabli and Rabani [3]. Both improvements also holds for a slightly more general class of metrics, the class of so-called star metrics, which can be represented as the shortest path metric induced by weighted trees of height one.

Khandekar and Pandit [13] analyze the reordering buffer problem for n uniformly-spaced points on a line with the motivation that this scenario models the disc scheduling problem. They present a randomized algorithm with a competitive ratio of $O(\log^2 n)$ in expectation against an oblivious adversary. Gamzu and Segev [11] 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, 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.

Most of the work on approximating the offline scenario 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. In terms of an optimal solution, the minimization and maximization scenario are identical. However, in terms of approximation they behave quite differently in the sense that a c-approximate solution for the maximization problem usually has very different cost from a c-approximate solution for the minimization problem. For the uniform metric, Kohrt and Pruhs [14] present an approximation algorithm with approximation ratio 20 for the maximization variant of the problem. Bar-Yehuda and Laserson [4] improve on this result with an approximation guarantee of 9.

For n uniformly spaced points on a line, Khandekar and Pandit [13] 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.

Englert, Özmen, and Westermann [7] show that the concept of a reordering buffer is also useful for other scheduling objectives. They present an extensive study of the power and limits of online reordering for minimum makespan scheduling. Their main result concerns the minimum makespan scheduling problem with reordering buffers on identical machines. They obtain tight bounds on the competitive ratio that are much improved over the bounds for the problem variant without reordering.

Another problem similar to the one studied here is the k-client problem introduced and analyzed by Alborzi et al. [2]. In the k-client problem we are given k clients, each of which generates an input sequence of requests for service in a metric space. In each step, every client presents its first outstanding request to the algorithm. The scheduling algorithm has to decide which of these k requests to serve. Again, the cost of serving a request is equal to this requests distance to the previously served one (from any client). The objective is to minimize the total cost. 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 PAY for the reordering buffer problem on tree metrics. The algorithm is inspired by the MAP strategy for star metrics introduced in [9]. However, our algorithm is not a generalization of this strategy as the behavior of MAP and PAY 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 metrics in Section 3 and obtain the following result.

Theorem 1.1. *Let T denote an arbitrary weighted tree, and let D denote the hop-diameter of T. For the shortest path metric induced by T, our deterministic online algorithm PAY 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 a hierarchically well-separated tree (HST). 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 1.2. *For metric spaces that can be represented as the shortest path metric induced by an HST, PAY achieves a competitive ratio of $O(\log^2 k)$, where k denotes the size of the reordering buffer.*

Fakcharoenphol, Rao, and Talwar [10] present a randomized approximation of arbitrary n-point metrics by tree metrics with an approximation ratio of $O(\log n)$. The tree metrics 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 1.3. *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

In the following, we present the PAY algorithm for the reordering buffer problem in tree metrics. Initially, the first k requests from the input sequence are stored in the reordering buffer. The server is placed at an arbitrary point corresponding to one of the k requests. PAY works in phases where each phase consists of a *selection step* and a *processing step*. To simplify the presentation of the algorithm and the analysis, the selection step is described as a continuous process. For this, we describe the behavior of the algorithm for infinitesimal short time intervals $[t, t+dt)$. Note that our notion of time is only used to describe the selection step and despite the continuous nature of our description the selection step can be easily discretized and implemented efficiently.

The two steps work as follows:

Selection Step In this step, PAY 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 $\mathsf{pay}(e) = \ell(e)$. Otherwise, we call e an *unpaid edge*.

During the selection process, the requests currently stored in the buffer are buying edges towards v_{pay}, where v_{pay} denotes the current position of PAY's server in the tree. This is done in the following continuous process: In a time interval $[t, t+dt)$ each request at each node u increases the payment $\mathsf{pay}(e)$ by dt, where e is the first unpaid edge on the path from u to v_{pay}. This process continues until there exists a connected component induced by paid edges that contains v_{pay}.

Processing Step In this step, PAY outputs all requests within the connected component. The order in which these requests are visited is not important. The online algorithm only has to ensure that each edge of the component is traversed at most twice and that the final position $\hat{v}_{\text{pay}}$, i. e., the new position of the PAY-server for the next phase, is a node in the component that is farthest away from v_{pay}.[1] 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.

These steps above are repeated as long as there exist at least k unprocessed requests. If the number of unprocessed requests drops below k, PAY starts a *clean-up phase*, during which it simply processes all remaining requests in an optimal fashion. For trees, this can be easily done in polynomial time (just as, e. g., TSP can be solved optimally for tree metrics).

3 Analysis for general trees

Fix an input sequence and a tree T. 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 PAY. We view OPT and PAY as working in a synchronized manner. After a phase of PAY 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 PAY.

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

If the PAY-server traverses an edge, it traverses the edge either towards the OPT-server or away from it. Due to the following observation it is sufficient to derive a bound on the cost induced by traversals away from OPT.

[1] At first glance, the requirement that the new position of the server is a node that is farthest away from the previous position may seem like a subtlety, but it will be used in the proof of the following theorem and is, in fact, critical for achieving a sublinear competitive ratio.

Observation 3.1. *Let* $\mathrm{PAY}_{\mathrm{away}}(e)$ *and* $\mathrm{PAY}_{\mathrm{towards}}(e)$ *denote the total cost induced by traversal of edge e by the* PAY*-server away from* OPT *and towards* OPT*, respectively. Let* $\mathrm{OPT}(e)$ *denote the cost induced due to traversals of edge e by the* OPT*-server. For each edge e,*

$$\mathrm{PAY}_{\mathrm{towards}}(e) \leq \mathrm{PAY}_{\mathrm{away}}(e) + 2 \cdot \mathrm{OPT}(e)\,.$$

Proof. Fix an edge e. First suppose that the PAY-server and the OPT-server both start on the same side of e. In this case,

$$\mathrm{PAY}_{\mathrm{towards}}(e) \leq \mathrm{PAY}_{\mathrm{away}}(e) + \mathrm{OPT}(e)\,.$$

To see this, we study how the left and the right side of the inequality change over time. In the beginning, $\mathrm{PAY}_{\mathrm{away}}(e) = \mathrm{OPT}(e) = \mathrm{PAY}_{\mathrm{towards}}(e) = 0$. The right side of the inequality is increased first (by $\ell(e)$) since both servers start on the same side of e.

Every but the last increase of the left side of the inequality is followed by an increase of the right side (both increases are by the amount $\ell(e)$). To see this, suppose PAY traverses e towards the OPT-server. After the traversal both servers are on the same side of e. Thus, the next traversal of e is either a traversal by the OPT-server or a traversal by the PAY-server away from the OPT-server.

Now assume that the servers start on different sides of e. Observe that the OPT-server has to traverse e at least once since there is at least one request on the side of e on which the PAY-server starts (recall that the initial position of the PAY-server is one of the first k requests). Thus, $\mathrm{OPT}(e) \geq \ell(e)$.

After e is traversed once—either by the OPT-server or the PAY-server—both servers reside on the same side of e. Ignoring the cost of the first traversal of e and using the arguments above, we get $\mathrm{PAY}_{\mathrm{towards}}(e) \leq \mathrm{PAY}_{\mathrm{away}}(e) + \mathrm{OPT}(e)$. Taking the first traversal into account gives

$$\mathrm{PAY}_{\mathrm{towards}}(e) \leq \mathrm{PAY}_{\mathrm{away}}(e) + \mathrm{OPT}(e) + \ell(e) \leq \mathrm{PAY}_{\mathrm{away}}(e) + 2 \cdot \mathrm{OPT}(e)\,. \qquad \square$$

In order to obtain our result we have to relate $\sum_e \mathrm{PAY}_{\mathrm{away}}(e)$ to the cost of an optimum solution. Ideally we would like to make an argument of the following type: For every fixed edge e, analyze the number of traversal of e by the PAY-server between two consecutive traversals of e by the OPT-server. If this number is bounded by at most $O(D \cdot \log k)$, we have our desired result. Unfortunately, doing an edge-by-edge analysis like this 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.

Therefore we use two concepts to amortize cost. First we introduce a counter $\mathsf{collected}(e)$ for each edge e with the intuition that at any point in time this counter describes the total cost that was generated on e in previous phases. Consequently, during a processing step in which e is traversed, the counter $\mathsf{collected}(e)$ would be increased by $\ell(e)$ while the counter $\mathsf{pay}(e)$ is set to 0. Note that the counter $\mathsf{pay}(e)$ is part of the algorithm whereas the $\mathsf{collected}(e)$ counter is only introduced as part of our analysis.

Now, to move cost between edges we slightly modify the handling of the $\mathsf{collected}(e)$-counters. At the end of a processing step, when the algorithms sets all counters $\mathsf{pay}(e)$ for edges of the connected component to 0, we do not increase some of the $\mathsf{collected}(e)$ counters and instead increase others by more than $\ell(e)$ in such a way that

$$\sum_{\substack{e \text{ traversed} \\ \text{away from } OPT}} \ell(e) \leq \sum_{e \,\in\, \text{component}} \Delta\,\mathsf{collected}(e)\,,$$

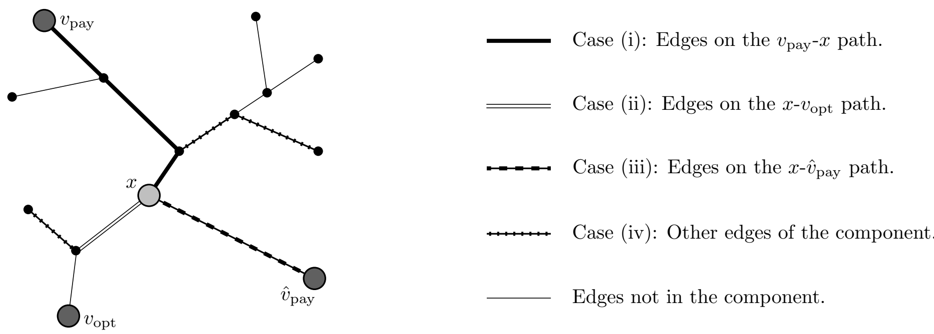

Figure 1: The different types of edges in the component traversed by PAY.

where $\Delta\mathsf{collected}(e)$ denotes the increase of counter $\mathsf{collected}(e)$. If we show that we can relate $\sum_e \mathsf{collected}(e)$ to the cost of the optimum solution, we get our result as $\sum_e \mathsf{collected}(e)$ still bounds the cost of the algorithm PAY.

As a second concept we introduce the notion of *discount*. In the selection step of the algorithm, requests generate payment in a continuous process. Similarly, we now let requests stored in PAY's or OPT's buffer generate discount. Again, this is solely done for the analysis.

Fix a selection step and a request p at position v_p. Recall that v_{pay} denotes the current position of the PAY-server. Let E_p denote the set of edges on the v_p–v_{pay} path that have been traversed by the PAY-server after the request p arrived.[2] We say that a request p that is in PAY's or OPT's buffer at time t generates a *discount* of $dt/(8D)$ during the time interval $[t, t+dt)$ on all edges in E_p. Similar to the counter $\mathsf{pay}(e)$ which was introduced in the algorithm, we use a counter $\mathsf{discount}(e)$ to keep track of the current discount on an edge e. That is, the above request p increases $\mathsf{discount}(e)$ by $dt/(8D)$ during each time interval of length dt for every edge e in E_p. This discount generation is only done in the analysis. Hence, we can assume that OPT and especially OPT's buffer content is known.

We now describe precisely how the $\mathsf{collected}(e)$ and $\mathsf{discount}(e)$ counters are changed during a processing step.

For a given processing step, let x denote the node at which the paths from v_{pay} to $\hat{v}_{\text{pay}}$ and from v_{pay} to v_{opt} split, where v_{opt} denotes the current position of the optimal server in the tree and $\hat{v}_{\text{pay}}$ denotes the position of the PAY-server at the end of the processing step. The PAY-server traverses the connected component in such a way that the edges on the v_{pay}–x path are traversed once towards the OPT-server, the edges on the x–$\hat{v}_{\text{pay}}$ path are traversed once away from the OPT-server and the remaining edges in the connected component are traversed twice (once towards the OPT-server and once away from it).

At the end of a processing step we change the counters as follows (see Figure 1):

(i) For edges on the v_{pay}–x path, we do not change $\mathsf{collected}(e)$ and reset $\mathsf{discount}(e)$ to 0.

This is reasonable as these edges are not traversed in direction away from OPT, and, hence, do not contribute to the increase in $\sum_e \text{PAY}_{\text{away}}(e)$.

[2]This technicality is only needed for the results in Section 4. For the results in Section 3 it would be sufficient to generate discount on every edge on the v_p–v_{pay} path.

(ii) For edges in the intersection of the component with the x–v_{opt} path, we do not change $\mathsf{collected}(e)$ and reset $\mathsf{discount}(e)$ to 0.

Note that these edges are traversed by the PAY-server in direction away from OPT. Hence, these edges contribute to the increase in $\sum_e \mathrm{PAY}_{\text{away}}(e)$, and we still need to account for them.

(iii) For edges that are on the x–$\hat{v}_{\text{pay}}$ path, we increase $\mathsf{collected}(e)$ by $2\ell(e) - \mathsf{discount}(e)$ and reset $\mathsf{discount}(e)$ to 0.

Since we choose $\hat{v}_{\text{pay}}$ as a farthest request from v_{pay} in the component, it holds that the total length of edges of type (iii) is at least as large as the length of edges of type (ii). Therefore, the stronger increase in the $\mathsf{collected}(e)$ counter for these edges offsets the missing increase for type (ii) edges.

(iv) For the remaining edges in the connected component, we increase $\mathsf{collected}(e)$ by $\ell(e) - \mathsf{discount}(e)$ and reset $\mathsf{discount}(e)$ to 0.

With these counter changes, the total collected payment after the whole input has been processed is $\sum_e \mathsf{collected}(e) \geq \sum_e \mathrm{PAY}_{\text{away}}(e) - \mathsf{discount}$, where $\mathsf{discount}$ denotes the total amount of discount generated over the entire duration of the execution of the algorithm. In order to derive a meaningful bound from this, we need an upper bound on $\mathsf{discount}$.

Observation 3.2. *The total generated discount is at most $C_{\text{PAY}}/4$, where C_{PAY} denotes the total cost of PAY on the input sequence.*

Proof. At any point in time, the total number of requests that generate discount is at most $2k$. 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 $k \cdot dt/4$ is generated. On the other hand, the k requests stored in PAY's buffer generate a payment of $k \cdot dt$ in each time interval of length dt. Hence, the total generated discount is at most a fourth of the total generated payment, where total generated payment is the total increase of $\mathsf{pay}(e)$ counters over the entire duration of the execution of the algorithm.

The total generated payment is at most the total cost of PAY since payment is not removed from an edge e unless PAY moves over e and, after the whole input has been processed, all $\mathsf{pay}(e)$ counters are 0 (otherwise there has to be an unprocessed request that is responsible for the remaining payment). □

We will show that, for every edge e, $\mathsf{collected}(e) \leq O(D \log k) \cdot \mathrm{OPT}(e)$. This gives

$$
\begin{aligned}
C_{\text{PAY}} &= 4 \cdot (C_{\text{PAY}}/2 - C_{\text{OPT}} - C_{\text{PAY}}/4) + 4 \cdot C_{\text{OPT}} \\
&\leq 4 \cdot \Big(\sum_e \mathrm{PAY}_{\text{away}}(e) - C_{\text{PAY}}/4\Big) + 4 \cdot C_{\text{OPT}} \\
&\leq 4 \cdot \Big(\sum_e \mathrm{PAY}_{\text{away}}(e) - \mathsf{discount}\Big) + 4 \cdot C_{\text{OPT}} \\
&\leq 4 \cdot \sum_e \mathsf{collected}(e) + 4 \cdot C_{\text{OPT}} \\
&\leq O(D \log k) \cdot \sum_e \mathrm{OPT}(e) + 4 \cdot C_{\text{OPT}} \\
&= O(D \log k) \cdot C_{\text{OPT}},
\end{aligned}
$$

where C_{OPT} denotes the total cost of an optimal offline algorithm. The second step holds due to Observation 3.1 and the fact that $C_{\mathrm{PAY}} = \sum_e (\mathrm{PAY}_{\mathrm{away}}(e) + \mathrm{PAY}_{\mathrm{towards}}(e))$. The third step holds due to Observation 3.2.

In order to show the missing statement $\mathsf{collected}(e) \le O(D \log k) \cdot \mathrm{OPT}(e)$ we need to compare the collected payment on an edge $e = \{u, v\}$ to the cost of OPT on e. Note that whenever $\mathsf{collected}(e)$ is changed, it increases by at most $2\ell(e)$. However, it may also decrease depending on the value of the $\mathsf{discount}(e)$-counter.

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

Lemma 3.3. *Let $[i_{start}, \ldots, i_{end}]$ denote a sequence of consecutive phases during which OPT does not traverse edge e. Then the number of phases in $[i_{start}, \ldots, i_{end}]$ in which the counter $\mathsf{collected}(e)$ increases is $O(D \log k)$.*

Proof. Let T_u and T_v denote the trees obtained when deleting $e = \{u, v\}$ from T, and assume without loss of generality that at the beginning of the phase i_{start} OPT's 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 PAY's buffer. Similarly, we call a request *pay-exclusive* if it is held by PAY but not by OPT.

Let $\mathsf{pay\text{-}excl}_i(T_v)$ and $\mathsf{opt\text{-}excl}_i(T_v)$ denote the number of pay-exclusive and opt-exclusive requests, respectively, that are in sub-tree T_v at the beginning of phase i. Note that during phases in $[i_{\mathrm{start}}, \ldots, i_{\mathrm{end}}]$ the number of pay-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 $i_{\mathrm{first}} \ge i_{\mathrm{start}}$ denote the first phase in which the $\mathsf{collected}(e)$-counter changes. If such a phase does not exist, then the lemma obviously holds. The following proposition shows that an increase in the counter $\mathsf{collected}(e)$ occurring after i_{first} is always accompanied by either a large decrease in $\mathsf{pay\text{-}excl}_i(T_v)$ 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 of the $\mathsf{collected}(e)$-counter during phases in $[i_{\mathrm{start}}, \ldots, i_{\mathrm{end}}]$.

Proposition 3.4. *Let $i \in [i_{first}+1, \ldots, i_{end}]$ denote a phase in which the counter $\mathsf{collected}(e)$ increases. Then either*

$$\mathsf{opt\text{-}excl}_{i+1}(T_v) > \left(1 + \frac{1}{16D}\right) \cdot \mathsf{opt\text{-}excl}_i(T_v)$$

or

$$\mathsf{pay\text{-}excl}_{i+1}(T_v) < \left(1 - \frac{1}{16D}\right) \cdot \mathsf{pay\text{-}excl}_i(T_v)\,.$$

Proof. First observe that in the beginning of the phase i the PAY-server is located in T_u, as otherwise e lies either on the v_{pay}–x path or on the x–v_{opt} path, and hence the $\mathsf{collected}(e)$-counter would not be increased. Furthermore, the edge e has to be part of the connected component traversed by the PAY server at the end of phase i, since otherwise the value of $\mathsf{collected}(e)$ also does not change.

Let n_{rem} denote the number of requests that generate payment on e in phase i, i. e., the number of requests that, at some point during the selection step of the phase, increase the $\mathsf{pay}(e)$-counter used by the algorithm. Note that since PAY's 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. This is due to the fact that e is part of the connected component at the end of phase i and, consequently, so is every request that increases the $\mathsf{pay}(e)$-counter during the phase.

Let $n_{\text{rem}}^{\text{opt}} \leq n_{\text{rem}}$ denote the number of payment generating requests that are held by OPT and by PAY, and let $n_{\text{rem}}^{\text{pay-excl}}$ denote the number of *pay-exclusive* requests that generate payment on e. Note that $n_{\text{rem}} = n_{\text{rem}}^{\text{opt}} + n_{\text{rem}}^{\text{pay-excl}}$.

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

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

Now assume for contradiction that

$$\begin{aligned} \mathsf{opt\text{-}excl}_{i+1}(T_v) &\leq \left(1+\frac{1}{16D}\right)\cdot \mathsf{opt\text{-}excl}_i(T_v) \quad \text{and} \\ \mathsf{pay\text{-}excl}_{i+1}(T_v) &\geq \left(1-\frac{1}{16D}\right)\cdot \mathsf{pay\text{-}excl}_i(T_v) , \end{aligned}$$

which gives

$$\frac{\mathsf{opt\text{-}excl}_i(T_v)}{16D} + \frac{\mathsf{pay\text{-}excl}_i(T_v)}{16D} \geq n_{\text{rem}}^{\text{opt}} + n_{\text{rem}}^{\text{pay-excl}} = n_{\text{rem}} .$$

Let $j \in \{i_{\text{first}}, \ldots, i-1\}$ be the most recent phase before phase i during which PAY visited T_v. The requests contributing to n_{rem} are the only requests that generate payment on e during phases $j+1, \ldots, i$. Note that the set of opt-exclusive and pay-exclusive requests in T_v does not change during phases $j+1, \ldots, i-1$, since PAY's and OPT's server are both located in T_u. Thus, all opt-exclusive and pay-exclusive requests arrived before PAY's server last traversed e in phase j. Hence, all the requests contributing to $\mathsf{opt\text{-}excl}_i(T_v)$ and $\mathsf{pay\text{-}excl}_i(T_v)$ generate discount on e. Therefore, the total discount generated on e during phases $j+1, \ldots, i$ is at least

$$\begin{aligned} \mathsf{discount}(e) &\geq \frac{\ell(e)}{n_{\text{rem}}} \cdot \frac{\mathsf{opt\text{-}excl}_i(T_v) + \mathsf{pay\text{-}excl}_i(T_v)}{8D} \\ &\geq \frac{\ell(e)}{n_{\text{rem}}} \cdot \frac{16D \cdot n_{\text{rem}}}{8D} \\ &= 2 \cdot \ell(e) , \end{aligned}$$

where the first inequality follows since $\mathsf{pay}(e) = 0$ at the beginning of phase $j+1$, $\mathsf{pay}(e) = \ell(e)$ right before the processing step of phase i, and only n_{rem} requests generate payment on e. Thus, $\ell(e)/n_{\text{rem}}$

is a lower bound on the time needed to pay for e and hence also for the time during which discount is generated.

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

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

The lemma directly implies $\mathsf{collected}(e) \le O(D \log k) \cdot \mathrm{OPT}(e)$. As previously outlined, this shows that PAY achieves a competitive ratio of $O(D \log k)$.

4 Analysis for HSTs

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 4.1. A 2-HST is a rooted tree such that

- all leaf nodes are on the same level, i. e., have the same hop-distance from the root,
- all edges on the same level have the same length, and
- the length of an 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 4.2. The literature contains several slightly different definitions of c-HSTs. It is well-known 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 1.2 follows from the arguments about 2-HSTs in this section. For completeness, Appendix A.1 contains more details on these approximations.

Our analysis for general trees in the last section consists of the following main arguments:

1. For every edge e,
$$\mathrm{PAY}_{\mathrm{towards}}(e) \le \mathrm{PAY}_{\mathrm{away}}(e) + 2 \cdot \mathrm{OPT}(e)\,,$$
as shown by Observation 3.1.

2. Then we introduced the notion of discount and, for every edge e, a counter $\mathsf{collected}(e)$ and a counter $\mathsf{discount}(e)$. We described how these counter-values are changed at the end of each processing step.

   The counter-values are changed in such a way that, after the whole input has been processed, we have
$$\sum_e \mathsf{collected}(e) \ge \sum_e \mathrm{PAY}_{\mathrm{away}}(e) - \mathsf{discount}\,,$$
where $\mathsf{discount}$ denotes the total amount of discount generated over the entire duration of the execution of the algorithm.

3. Observation 3.2 shows that the way in which we generate discount ensures that $\mathsf{discount} \le C_{\mathrm{PAY}}/4$.

4. And finally, Lemma 3.3 shows that, for every edge e, $\mathsf{collected}(e) \le O(D\log k)\cdot \mathrm{OPT}(e)$.

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 $dt/(8D)$ to $dt/\Theta(\log k)$. If we can do this while otherwise maintaining the properties of the discount distribution, Theorem 1.1 will improve to a competitive ratio of $O(\log^2 k)$.

Recall that, in our previous analysis, each element p either stored in PAY's buffer or OPT's buffer (or both) generates discount in the following way: E_p denotes the set of edges on the v_{pay}–v_p path that have been traversed by the PAY-server after the request p arrived. Then p generates a discount of $dt/(8D)$ in each time interval of length dt on each edge in E_p.

We now change this discount generation as follows. Let r denote the node with lowest level on the v_{pay}–v_p path, i. e., the node on the path that has the smallest distance to the root of the tree. We divide the edges in E_p into three different types:

1. Edges that are ancestor edges of v_{pay} in the rooted tree, i. e., edges on the v_{pay}–r path.

2. The first $\log k+7$ edges on the r–v_p path. We call these edges *long edges*.

3. The remaining edges (if any) on the r–v_p path. We call these edges *short edges*.

Let $e_{\max}$ denote the longest edge on the r–v_p path (which, due to the definition of an HST, is the first edge). We now define that, in a time interval of length dt, request p generates no discount on edges on the v_{pay}–r path, generates a discount of $dt/(16(\log k+7))$ on every long edge contained in E_p, and generates a discount of $k\cdot dt\cdot 4\ell(e)/\ell(e_{\max})$ on every short edge contained in E_p.

With this modified discount generation we still can show Observation 3.2, as follows. In a time interval of length dt, each request generates a discount of at most

$$\sum_{\text{long edge } e}\frac{dt}{16(\log k+7)}+\sum_{\text{short edge } e}\frac{k\cdot dt\cdot 4\ell(e)}{\ell(e_{\max})}\le\frac{dt}{16}+\frac{4k\cdot dt}{\ell(e_{\max})}\cdot\sum_{\text{short edge } e}\ell(e)\,.$$

The edge lengths on the r–v_p path are geometrically decreasing. Hence, $\sum_{\text{short edge } e}\ell(e)\le \ell(e_{\max})/(64k)$ since the first $\log k+7$ edges on the path are long edges. Thus, the discount generated by a single request is bounded by

$$\frac{dt}{16}+\frac{4k\cdot dt}{\ell(e_{\max})}\cdot\sum_{\text{short edge } e}\ell(e)\le\frac{dt}{8}\,.$$

Following the arguments in the proof of Observation 3.2, this shows that $\mathsf{discount}\le C_{\mathrm{PAY}}/4$.

We are now almost ready to establish an improved version of Lemma 3.3 for HSTs. Unfortunately, since edges that are ancestor edges of v_{pay} do not receive any discount anymore, some of the $\mathsf{collected}(e)$-counters could be increased much more frequently than before. Lemma 3.3 may not hold anymore. We have to make one last technical adjustment to the proof in the previous section.

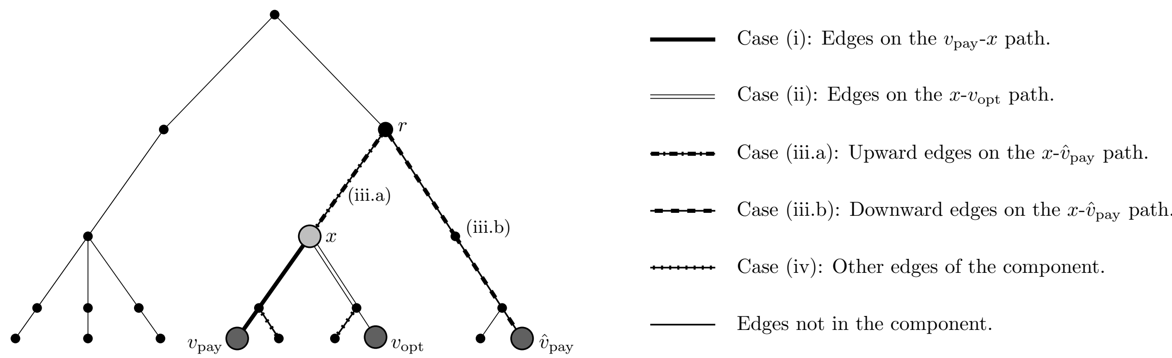

Figure 2: The different types of edges considered in the HST analysis.

In the last section, we defined how the counters $\mathsf{collected}(e)$ and $\mathsf{discount}(e)$ are changed at the end of each processing step. This was done in such a way that, after the whole input has been processed, we have $\sum_e \mathsf{collected}(e) \geq \sum_e \text{PAY}_{\text{away}}(e) - \mathsf{discount}$. There were four different types of edges in the connected component. We are now changing the third type. These were the edges on the x–$\hat{v}_{\text{pay}}$ path, and in the previous section we increased their $\mathsf{collected}(e)$-counters by $2\ell(e) - \mathsf{discount}(e)$.

Let r denote the root of the connected component, i. e., the node on the lowest level in the component. Note that r lies on the v_{pay}–$\hat{v}_{\text{pay}}$ path as we always choose $\hat{v}_{\text{pay}}$ as the node in the connected component that is furthest from v_{pay}. We split edges of the third type (Case (iii)), i. e., edges on the x–$\hat{v}_{\text{pay}}$ path, into two sub-cases, namely those edges on the x–$\hat{v}_{\text{pay}}$ path that also lie on the v_{pay}–r path (upward edges), and those edges on the x–$\hat{v}_{\text{pay}}$ path that also lie on the r–$\hat{v}_{\text{pay}}$ path (downward edges) (see Figure 2). The counters for these edges are changed as follows:

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

(iii.b) If the edge e is not an ancestor edge of v_{pay}, i. e., e lies in the intersection of the r–$\hat{v}_{\text{pay}}$ path and the x–$\hat{v}_{\text{pay}}$ path, increase the counter $\mathsf{collected}(e)$ by $4\ell(e) - \mathsf{discount}(e)$, and then reset the counters $\mathsf{discount}(e)$ to 0. This means that the increase of the counter $\mathsf{collected}(e)$ exceeds the previously needed increase of $2\ell(e) - \mathsf{discount}(e)$ by $2\ell(e)$.

The excess in Case (iii.b) is used to counteract the now omitted increase of $\mathsf{collected}(e')$-counters on each edge e' in the intersection of the v_{pay}–r and the x–$\hat{v}_{\text{pay}}$ path (Case (iii.a)). First observe that previously each $\mathsf{collected}(e')$-counter was at most increased by $2\ell(e')$, i. e., we only need to show that the total length of edges generating excess (Case (iii.b) edges) is at least as large as the length of edges for which the $\mathsf{collected}(e')$ counter is not increased anymore (Case (iii.a) edges).

First, assume that the root r does not lie on the x–$\hat{v}_{\text{pay}}$ path, i. e., there are no edges corresponding to Case (iii.a). Thus, trivially, the total length of edges corresponding to Case (iii.a) is at most the total length of edges corresponding to Case (iii.b).

Now, assume that the root r of the component lies on the x–$\hat{v}_{\text{pay}}$ path. In this case, edges corresponding to Case (iii.a) are the edges on the x–r path and edges corresponding to Case (iii.b) are edges on the

r–$\hat{v}_{\text{pay}}$ path. The latter is at least as long as the former since the x–r path is completely contained in the v_{pay}–r path and the v_{pay}-r path has the same length as the r–$\hat{v}_{\text{pay}}$ path. This shows that the new counter changes still fulfill

$$\sum_e \mathsf{collected}(e) \geq \sum_e \mathrm{PAY}_{\text{away}}(e) - \mathsf{discount}\,.$$

Observe that, up to Lemma 3.3, all the main arguments from the previous section about general trees go through, even with these changes to the discount generation process and the changes to the $\mathsf{collected}(e)$-counters.

The central claim in the proof of Lemma 3.3 is stated in Proposition 3.4. For HSTs, we are now ready to give an improved version of Proposition 3.4. This directly improves the bound on the number of increases to the $\mathsf{collected}(e)$-counter given by Lemma 3.3, which, in turn, gives Theorem 1.2.

Let $e = \{u, v\}$ denote an edge in the connected component of phase i, and let u be the parent of v. Let T_u and T_v denote the two trees obtained by deleting e from the HST.

Proposition 4.3. *Let $i \in [i_{first}+1, \ldots, i_{end}]$ denote a phase in which the counter $\mathsf{collected}(e)$ increases. Then either*

$$\mathsf{opt\text{-}excl}_{i+1}(T_v) > \left(1 + \frac{1}{64(\log k + 7)}\right) \cdot \mathsf{opt\text{-}excl}_i(T_v)$$

or

$$\mathsf{pay\text{-}excl}_{i+1}(T_v) < \left(1 - \frac{1}{64(\log k + 7)}\right) \cdot \mathsf{pay\text{-}excl}_i(T_v)\,.$$

Proof. If, in the beginning of phase i, PAY's server is located in T_v, $\mathsf{collected}(e)$ is not changed in the phase. To see this observe that e lies on the v_{pay}–$\hat{v}_{\text{pay}}$ path because the root of the connected component lies both in T_u and on the v_{pay}–$\hat{v}_{\text{pay}}$ path. This excludes Case (ii) and Case (iv). The fact that e is an ancestor edge of v_{pay} excludes Case (iii.b). Hence, e belongs either to Case (i) or Case (iii.a) and $\mathsf{collected}(e)$ is not changed.

If PAY's server is located in T_u but OPT's server is located in T_v, $\mathsf{collected}(e)$ is not changed either, since in this case e lies on the v_{pay}–v_{opt} path and hence, belongs either to Case (i) or Case (ii). Thus, we may assume that both servers are located in T_u.

Following the same arguments and notation as in Proposition 3.4, the assumption that the proposition does not hold implies

$$\frac{\mathsf{opt\text{-}excl}_i(T_v)}{64(\log k + 7)} + \frac{\mathsf{pay\text{-}excl}_i(T_v)}{64(\log k + 7)} \geq n_{\text{rem}}^{\text{opt}} + n_{\text{rem}}^{\text{pay-excl}} = n_{\text{rem}}\,.$$

The following claim ensures that any request that generated discount on e did so at rate $1/(16/(\log k + 7))$, i. e., the request did not consider e to be a short edge.

Claim 4.4. *There is no request that generates discount on e as a short edge.*

Proof. We show that in the case that a request p in T_v generates discount on edge e as a short edge, the accumulated discount on e in phase i when e is contained in the connected component is at least $4\ell(e)$.

This means that the $\mathsf{collected}(e)$ counter would not increase as $\Delta\mathsf{collected}(e) \le 4\ell(e) - \mathsf{discount}(e)$ is negative. This would be a contradiction to the choice of the phase i.

Fix a request $p \in T_v$ that generates discount on edge e as a short edge. Assume that by the end of phase i the smallest rate at which request p generated discount on 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 PAY in direction from v' to u', i. e., away from p (this traversal happens at some point between phase j and phase i).

Since request p generates discount on edge e, e has to be contained in the edge-set E_p. Therefore, request p already exists in T_v at this point and generates a discount of at least $k \cdot dt \cdot 4\ell(e)/\ell(e_{\max})$ on e in every time interval of length dt. Right after PAY moved over $e_{\max}$ there is no payment on this edge, and in order for PAY 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 PAY's buffer. Hence, by the time PAY returns into $T_{v'}$ a discount of at least $4\ell(e)$ has been generated by the request p on the edge e. □

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

$$\begin{aligned}
\mathsf{discount}(e) &\ge \frac{\ell(e)}{n_{\text{rem}}} \cdot \frac{\mathsf{opt\text{-}excl}_i(T_v) + \mathsf{pay\text{-}excl}_i(T_v)}{16(\log k + 7)} \\
&\ge \frac{\ell(e)}{n_{\text{rem}}} \cdot \frac{64(\log k + 7) \cdot n_{\text{rem}}}{16(\log k + 7)} \\
&= 4 \cdot \ell(e)\,,
\end{aligned}$$

where the first inequality follows since $\mathsf{pay}(e) = 0$ at the beginning of phase $j+1$, $\mathsf{pay}(e) = \ell(e)$ right before the processing step of phase i, and only n_{rem} requests generate payment on e. Thus, $\ell(e)/n_{\text{rem}}$ is a lower bound on the time needed to pay for e and hence also for the time during which discount is generated. However, the counter $\mathsf{collected}(e)$ increases by at most $4\ell(e) - \mathsf{discount}(e)$, and hence $\mathsf{collected}(e)$ does not increase in phase i. This contradiction completes the proof of the proposition. □

A Appendix

A.1 Approximating different types of HSTs

In the following we sketch how to approximate a c-HST according to the original definition by Bartal [5] by a 2-HST according to Definition 4.1.

Definition A.1 (Bartal [5])**.** Let $c > 1$. A c-hierarchically well separated tree (c-HST) is defined as a weighted, rooted tree with the following properties.

1. The edge weight from any node to each of its children is the same.

2. The edge weights along any path from the root to a leaf are decreasing by a factor of at least c.

Given a c-HST according to the above definition we transform it into a 2-HST according to Definition 4.1 while changing distances by at most a constant factor. First, we round all edge-length up to the nearest power of two. This only increases distances by a factor of 2.

Now, let $T_1, T_2, \ldots$ denote the sub-trees that result from this new tree when only considering edges of a certain length ℓ. The rules for an HST (both definitions) require that the T_i's are trees of height 1. However, we now may have trees of height up to $1/\log_2 c = O(1)$ as edges of different length may have been rounded to the same value. We remedy this by attaching all nodes in a tree T_i to the root node of that tree. This process decreases distances by at most a factor $1/\log_2 c = O(1)$. This still holds when performing the process for all possible edge-length.

We now have a tree in which

- every child node of a node has the same length and

- the edge weights along any path from the root to a leaf are decreasing by a factor of at least 2.

In a third phase we consider nodes v for which the distance from v to the parent of v is larger than twice the distance to its children. For each child c_j of v we replace the edge (v, c_j) by a sequence of edges and dummy nodes with geometrically decreasing distances. This can be done while only increasing the distance between parent and child by a constant factor.

Finally, we have to ensure that all leaf nodes are on the same level, i. e., have the same hop-distance to the root. Let $\ell_{\max}$ denote the maximum level of a leaf node. For a leaf node v with level less than $\ell_{\max}$ we replace its parent edge by a sequence of edges with geometrically decreasing length until v is on level $\ell_{\max}$. This only increases the distance between v and its parent by a constant factor.

References

[1] AMJAD ABOUD: Correlation clustering with penalties and approximating the recordering buffer management problem. Master's thesis, The Technion - Israel Institute of Technology, Janurary 2008. [Technion TR archive]. 29

[2] HOUMAN ALBORZI, ERIC TORNG, PATCHRAWAT UTHAISOMBUT, AND STEPHEN WAGNER: The k-client problem. *J. Algorithms*, 41(2):115–173, 2001. [doi:10.1006/jagm.2001.1182]. 30

[3] NOA AVIGDOR-ELGRABLI AND YUVAL RABANI: An improved competitive algorithm for reordering buffer management. In *Proc. 21st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pp. 13–21. ACM Press, 2010. 29

[4] REUVEN BAR-YEHUDA AND JONATHAN LASERSON: Exploiting locality: Approximating sorting buffers. In *Proc. 3rd Workshop on Approximation and Online Algorithms (WAOA)*, volume 3879 of *Lecture Notes in Computer Science*, pp. 69–81. Springer, 2005. [doi:10.1007/11671411_6]. 28, 30

[5] YAIR BARTAL: Probabilistic approximations of metric spaces and its algorithmic applications. In *Proc. 37th FOCS*, pp. 184–193. IEEE Comp. Soc. Press, 1996. [doi:10.1109/SFCS.1996.548477]. 29, 42

[6] YAIR BARTAL: On approximating arbitrary metrics by tree metrics. In *Proc. 30th STOC*, pp. 161–168. ACM Press, 1998. [doi:10.1145/276698.276725]. 29

[7] MATTHIAS ENGLERT, DENIZ ÖZMEN, AND MATTHIAS WESTERMANN: The power of reordering for online minimum makespan scheduling. In *Proc. 49th FOCS*, pp. 603–612. IEEE Comp. Soc. Press, 2008. [doi:10.1109/FOCS.2008.46]. 30

[8] MATTHIAS ENGLERT, HEIKO RÖGLIN, AND MATTHIAS WESTERMANN: Evaluation of online strategies for reordering buffers. In *Proc. 5th Internat. Workshop on Efficient and Experimental Algorithms (WEA)*, volume 4007 of *Lecture Notes in Computer Science*, pp. 183–194. Springer, 2006. [doi:10.1007/11764298_17]. 29

[9] MATTHIAS ENGLERT AND MATTHIAS WESTERMANN: Reordering buffer management for non-uniform cost models. In *Proc. 32nd Internat. Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pp. 627–638. Springer, 2005. [doi:10.1007/11523468_51]. 28, 29, 31

[10] JITTAT FAKCHAROENPHOL, SATISH B. RAO, AND KUNAL TALWAR: A tight bound on approximating arbitrary metrics by tree metrics. *J. Comput. System Sci.*, 69(3):485–497, 2004. [doi:10.1016/j.jcss.2004.04.011]. 29, 31

[11] IFTAH GAMZU AND DANNY SEGEV: Improved online algorithms for the sorting buffer problem. In *Proc. 24th Symp. on Theoretical Aspects of Computer Science (STACS)*, volume 4393 of *Lecture Notes in Computer Science*, pp. 658–669. Springer, 2007. [doi:10.1007/978-3-540-70918-3_56]. 29, 30

[12] KAI GUTENSCHWAGER, SVEN SPIEKERMANN, AND STEFAN VOSS: A sequential ordering problem in automotive paint shops. *Internat. J. Production Research*, 42(9):1865–1878, 2004. [doi:10.1080/00207540310001646821]. 29

[13] ROHIT KHANDEKAR AND VINAYAKA PANDIT: Online and offline algorithms for the sorting buffers problem on the line metric. *J. Discrete Algorithms*, 2008. [doi:10.1016/j.jda.2008.08.002]. 28, 29, 30

[14] JENS S. KOHRT AND KIRK PRUHS: A constant factor approximation algorithm for sorting buffers. In *Proc. 6th Latin American Symp. on Theoretical Informatics (LATIN)*, volume 2976 of *Lecture Notes in Computer Science*, pp. 193–202. Springer, 2004. [LATIN:fqbc84923gveuhlr]. 28, 30

[15] JENS KROKOWSKI, HARALD RÄCKE, CHRISTIAN SOHLER, AND MATTHIAS WESTERMANN: Reducing state changes with a pipeline buffer. In *Proc. 9th Internat. Fall Workshop Vision, Modeling, and Visualization (VMV)*, pp. 217–224. Aka GmbH, 2004. 29

[16] HARALD RÄCKE, CHRISTIAN SOHLER, AND MATTHIAS WESTERMANN: Online scheduling for sorting buffers. In *Proc. 10th European Symp. on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pp. 820–832. Springer, 2002. [doi:10.1007/3-540-45749-6_71, ESA:mx7mdle1j62b5n3h]. 28, 29

[17] TOBY J. TEOREY AND TAD B. PINKERTON: A comparative analysis of disk scheduling policies. *Comm. ACM*, 15(3):177–184, 1972. [doi:10.1145/361268.361278]. 28

AUTHORS

Matthias Englert
Department of Computer Science, DIMAP
University of Warwick, Coventry, UK
Matthias.Englert@warwick.ac.uk
http://www.dcs.warwick.ac.uk/~englert

Harald Räcke
Department of Computer Science, DIMAP
University of Warwick, Coventry, UK
H.Raecke@warwick.ac.uk
http://www.dcs.warwick.ac.uk/~harry

Matthias Westermann
Department of Computer Science
University of Bonn, Bonn, Germany
westermann@cs.uni-bonn.de
http://www.i1.cs.uni-bonn.de/staff/marsu

ABOUT THE AUTHORS

MATTHIAS ENGLERT is currently a postdoctoral fellow at the University of Warwick. He graduated, under the supervision of Matthias Westermann, from RWTH Aachen University in 2008. His thesis deals with the design and analysis of online algorithms and is titled "Online Scheduling for Buffering Problems." His current topics of interest include online algorithms, metric embeddings, load balancing, probabilistic input models, and algorithmic game theory.

HARALD RÄCKE graduated from Paderborn University in 2003 under the supervision of Friedhelm Meyer auf der Heide. Currently, he is Assistant Professor in the Centre for Discrete Mathematics and its Applications (DIMAP) at the University of Warwick.

He shared the Best Paper award at FOCS 2002 and STOC 2008. His main interests include the design and analysis of routing and graph partitioning algorithms, the theory of metric embeddings, and algorithmic game theory.

MATTHIAS WESTERMANN received his Ph. D. at the University of Paderborn under the supervision of Friedhelm Meyer auf der Heide. After a postdoctoral year at the International Computer Science Institute in Berkeley, the German Research Foundation awarded him with a research grant for an independent junior research group at TU Dortmund and RWTH Aachen University. Currently, he is a visiting professor at the University of Bonn. Matthias is interested in the design and analysis of efficient algorithms, mostly online and approximation algorithms. Topics of interest include scheduling algorithms, Internet algorithms, distributed and parallel algorithms, and experimental evaluation.