Online and offline algorithms for the Sorting Buffers Problem on the line metric

Rohit Khandekar

IBM T.J. Watson Research Center

Vinayaka Pandit

IBM India Research Laboratory

Abstract

We consider the *sorting buffers* problem. Input to this problem is a sequence of requests, each specified by a point in a metric space. There is a "server" that moves from point to point to serve these requests. To serve a request, the server needs to visit the point corresponding to that request. The objective is to minimize the total distance traveled by the server in the metric space. In order to achieve this, the server is allowed to serve the requests in any order that requires to "buffer" at most k requests at any time. Thus a valid reordering can serve a request only after serving all but k previous requests.

In this paper, we consider this problem on the line metric which is motivated by its application to the disc scheduling problem. We present first approximation algorithms with non-trivial approximation ratios in both online and offline settings. On a line metric with n uniformly spaced points, we give a randomized online algorithm with a competitive ratio of $O(\log^2 n)$ in expectation against an oblivious adversary. In the offline setting, our algorithm yields the first *constant-factor approximation* and runs in quasi-polynomial time $N \cdot n \cdot k^{O(\log n)}$ where N is the total number of requests. Our approach is based on a dynamic program that keeps track of the number of pending requests in each of $O(\log n)$ line segments that are geometrically increasing in length.

Key words: Approximation Algorithms, Online Algorithms, Metric Embedding.

⋆ Preliminary versions of this paper appeared in the proceedings of Symposium on Theoretical Aspects of Computer Science (STACS), 2006 and International Symposium on Algorithms and Computation (ISAAC), 2006.

Email addresses: rkhandekar@gmail.com (Rohit Khandekar), vinayaka.pandit@gmail.com (Vinayaka Pandit).

1 Introduction

The sorting buffers problem (SBP) arises in scenarios where a stream of requests needs to be served. Each request has a "type" and for any pair of types t_1 and t_2, the cost of serving a request of type t_2 immediately after serving a request of type t_1 is known. The input stream can be reordered while serving in order to minimize the cost of type-changes between successive requests served. However, a "sorting buffer" has to be used to store the requests that have arrived but not yet served and often in practice, the size of such a sorting buffer, denoted by k, is small. Thus a legal reordering must satisfy the following property: any request can be served only after serving all but k of the previous requests. The objective in the sorting buffers problem is to compute the minimum cost output sequence which respects this sequencing constraint.

Consider, as an example, a workshop dedicated to coloring cars. A sequence of requests to color cars with specific colors is received. If the painting schedule paints a car with a certain color followed by a car with a different color, then, a significant set-up cost is incurred in changing colors. Assume that the workshop has space to hold at most k cars in waiting. A natural objective is to rearrange the sequence of requests such that it can be served with a buffer of size k and the total set-up cost over all the requests is minimized.

Consider, as another example, the classical disc scheduling problem [13]. A sequence of requests each of which is a block of data to be written on the block at a specified address of the disc. To write the block, the head has to first move to the track corresponding to the address. The time required for accomplishing this is called the *seek time* and can be reliably estimated assuming that the tracks are arranged on a straight line. Once the head has moved to the correct track, the disc has to rotate so that the sector containing the address is right beneath the head. The time required to accomplish this is called as the *rotational latency* and it is a quantity believed to be very difficult to estimate. Therefore, in practice, only seek times are taken into account while optimizing the performance. Therefore, for all practical purposes, the requests can be thought of as corresponding to tracks which are arranged on a straight line with the cost of moving between track i and track j is equal to $|i-j|$. As it is inefficient to visit the tracks in the same order as the requests arrive, one has access to a limited buffer that can store data corresponding to k requests. The goal of disc scheduling is to write the requests in an order that minimizes the total cost of all movements.

As can be observed from the previous examples, usually, the type-change costs satisfy metric properties and hence we formulate the sorting buffers problem on a metric space. Let (V, d) be a metric space on n points. The input to the Sorting Buffers Problem (SBP) consists of a sequence of N requests, the ith

request being labeled with a point $p_i \in V$. There is a server, initially located at a point $p_0 \in V$. To serve ith request, the server has to visit p_i. There is a sorting buffer which can hold up to k requests at a time. In a *legal schedule*, the ith request can be served only after serving at least $i - k$ requests of the first $i - 1$ requests. If a request at point p_j is served after serving a request at point p_i, then a cost of $d(p_i, p_j)$ is incurred. The goal in the SBP is to find a legal schedule that serves all the requests at minimum cost. In the online version of the SBP, the ith request is revealed only after serving at least $i - k$ among the first $i - 1$ requests. In the offline version, on the other hand, the entire input sequence is known in advance. A formal definition of the problem is presented in Section 2.

The car coloring problem described above can be thought of as the SBP on a uniform metric where all the pair-wise distances are identical while the disc scheduling problem corresponds to the SBP on a line metric where all the points lie on a straight line and the distances are given along that line. In this paper, we present offline and online algorithms for the SBP on the line metric, i.e, all our results are applicable to the disc scheduling problem.

1.1 Previous Work

On a general metric, the SBP is known to be NP-hard due to a simple reduction from the Hamiltonian Path problem. The SBP on simpler metrics such as the uniform metric and the uniformly spaced line metric arises naturally in several applications as explained before. The computational complexity of the SBP on both the metrics remains unknown. The only known lower bound on the competitive ratio of online algorithms for the SBP on the line metric is given by Gamzu and Segev [9]. They show a lower bound of 2.154 on the competitive ratio of any deterministic online algorithm for the SBP on the line metric.

The offline version of the sorting buffers problem on any metric can be solved optimally using dynamic programming in $O(N^{k+1})$ time where N is the number of requests in the sequence. This follows from the observation that the algorithm can pick k requests to hold in the buffer from first i requests in $\binom{i}{k}$ ways when the $(i+1)$th request arrives. Therefore, for constant k, the SBP can be solved optimally on any metric space. However, approximation algorithms for the offline version of the SBP for general k have not been studied in the past. This paper is the first work in this direction.

The SBP on the uniform metric (in which all pairwise distances are 1) has been studied before. This problem is interesting only when multiple requests are allowed for the points in the metric space. Räcke et al. [12] presented a deterministic online algorithm, called *Bounded Waste* that has $O(\log^2 k)$ com-

petitive ratio. They also showed that some natural strategies like FIFO, LRU etc. have $\Omega(\sqrt{k})$ competitive ratio. Englert and Westermann [7] considered a generalization of the uniform metric in which moving to a point p from any other point in the space has a cost c_p. They proposed an algorithm called Maximum Adjusted Penalty (MAP) and showed that it gives an $O(\log k)$ approximation, thus improving the competitive ratio of the SBP on uniform metric. Kohrt and Pruhs [11] also considered the uniform metric but with different optimization measure. Their objective was to maximize the reduction in the cost from that of the schedule without a buffer. They presented a 20-approximation algorithm for this variant and this ratio was improved to 9 by Bar-Yehuda and Laserson [1]. As mentioned earlier, it is not known if the offline version of the SBP on the uniform metric is NP-hard.

Charikar and Raghavachari [5] considered a related problem called as the *dial-a-ride problem*. In this problem, the input is a sequence of requests each of which is a source-destination pair in a metric space on n points. Each request needs an object to be transferred from its source to its destination. The goal is to serve all the requests using a vehicle of capacity k so that total length of the tour is minimized. The non-preemptive version requires that once an object is picked, it can be dropped only at its destination while in the preemptive version the objects can be dropped at intermediate locations and picked up later. Charikar and Raghavachari [5] gave approximation algorithms for both preemptive and non-preemptive versions using Bartal's metric embedding result. They gave $O(\log n)$ approximation for the preemptive case and $O(\sqrt{k}\log n)$ approximation for the non-preemptive case. Recently, Gupta et al. [10] gave a different algorithm for the dial-a-ride problem based on an approximation algorithm for the *k-forest problem*. Their algorithm achieves a ratio of $O(\min\{\sqrt{n}, \sqrt{k}\}\log^2 n)$ and easily extends to some generalizations of the dial-a-ride problem. The *capacitated vehicle routing problem* considered by Charikar et al. [4] is a variant in which all objects are identical and hence an object picked from a source can be dropped at any of the destinations. They gave the best known approximation ratio of 5 for this problem and survey the previous results. We observe that techniques for the dial-a-ride problem cannot be extended to the SBP in a straightforward manner. Note that the SBP enforces a sequencing constraint that is not imposed by the dial-a-ride problem. In the SBP, if we are serving the ith request, then it is necessary that at most k requests from first to $(i-1)$th request be outstanding. Whereas, in the dial-a-ride problem, the requests can be served in any order as long as the schedule meets the capacity constraint. Therefore their techniques are not applicable to the SBP.

We present algorithmic results for both the online and offline versions of SBP on the line metric.

1.2.1 Online SBP

Our main result is an $O(\log^2 n)$-competitive online algorithm where n is the number of uniformly spaced points on which request can arrive. In case of disc scheduling, it corresponds to the number of tracks. This is the first approximation algorithm with a non-trivial approximation ratio for the SBP on the line metric. We present a randomized online algorithm with a competitive ratio of $O(\log^2 n)$ in expectation against an oblivious adversary. This algorithm also yields a competitive ratio of $O(\alpha^{-1} \log^2 n)$ if we are allowed to use a buffer of size αk for any $1 \leq \alpha \leq \log n$. Thus, the approximation ratio can be improved if we are allowed slightly larger buffer than the one used by the optimal schedule.

Our algorithm is based on the probabilistic embedding of the line metric into the so-called hierarchical well-separated trees (HSTs) first introduced by Bartal [2]. He showed that any metric on n points can be probabilistically approximated within a factor of $O(\log n \log \log n)$ by metrics on the HSTs [3]. This factor was later improved to $O(\log n)$ by Fakcharoenphol et al. [8]. It is easy to see that the line metric $\{1, \ldots, n\}$ can be probabilistically approximated within a factor of $O(\log n)$ by the metrics induced by binary trees of depth $1 + \log n$ such that the edges in level i have length $n/2^i$. We provide a simple lower bound on the cost of the optimum on a tree metric by counting how many times it must cross a particular edge in the tree. Using this lower bound, we prove that the expected cost of our algorithm is within the factor of $O(\log^2 n)$ of the optimum.

Later we show that most of the natural strategies that one could be tempted to try perform badly. We show that FIFO and Nearest-First have a competitive ratio of $\Omega(k)$. Thus, similar to the SBP on uniform metrics, even on line metric simple heuristics do not work, and one is forced to look for smarter approximation algorithms.

We also show very strong lower bounds on the performance of "memoryless" algorithms. This notion is formalized later in the paper. Specifically, we show that any deterministic algorithm that takes decisions just based on the unordered set of requests which are currently in the buffer has a competitive ratio of $\Omega(k)$. The same proof implies a lower bound of $\Omega(\log n / \log \log n)$ for deterministic memoryless algorithms.

1.2.2 *Offline SBP*

An important step in understanding the structure of the SBP is to develop offline algorithms. We provide such an algorithm. We give a a constant factor approximation algorithm for the offline SBP on a line metric on n uniformly spaced points that runs in quasi-polynomial time: $N \cdot n \cdot k^{O(\log n)}$ where k is the buffer-size and N is the number of input requests. This suggests the existence of polynomial time algorithm with a constant approximation ratio. This is the first constant factor approximation algorithm for this problem on any non-trivial metric space. The approximation factor we prove here is 15. Our algorithm is based on dynamic programming. We show that there is a near-optimum schedule with some "nice" properties and give a dynamic program to compute the best schedule with those nice properties.

1.2.3 *Subsequent work*

Subsequent to our results, other researchers have made very good progress on the SBP. Gamzu and Segev [9] showed that the embedding of the line metric on HSTs is not required and came up with a clever way of partitioning the line to be able to obtain a similar lower bound as we use on the trees. They developed a deterministic algorithm with an approximation ratio of $O(\log n)$. In our earlier paper, we left open the problem of extending our algorithm and analysis to the case of arbitrary metric spaces. Englert et. al [6] made progress on this front and showed a way to exploit the embedding of arbitrary metric spaces on HSTs to obtain an approximation algorithm for general metric spaces as well. They showed an approximation ratio of $O(\log^2 k \log n)$ where k is size of the buffer and n is the number of points over which the requests are made.

1.2.4 *Organization of the paper*

The rest of the paper is organized as follows. In Section 2, we define the sorting buffers problem formally. Section 3 deals with the online version of the sorting buffers problem. Section 3.1 presents our first main result — a polylogarithmic competitive ratio for the line metric. We also present some intuition into why natural strategies like Nearest-First and First-In-First-Out fail to yield a good competitive ratio in Appendix 6.1; while Appendix 6.2 proves that the so-called *memoryless* algorithms are not good either. In Section 4, we present our second main result — a constant approximation for the offline version of the problem on the line metric. Finally, in Section 5, we conclude with some open questions.

2 The Sorting Buffers Problem

Let (V, d) be a metric on n points. The input to the Sorting Buffers Problem (SBP) consists of a sequence of N requests. The ith request is labeled with a point $p_i \in V$. There is a server, initially located at a point $p_0 \in V$. To serve ith request, the server has to visit p_i after its arrival. There is a sorting buffer which can hold up to k requests at a time. The first k requests arrive initially. The $(i+1)$th request arrives after we have served at least $i+1-k$ requests among the first i requests for $i \geq k$. Thus we can keep at most k requests pending at any time. The output is such a legal schedule of serving the requests. More formally, the output is given by a permutation π of $1, \ldots, N$ where the ith request to be served is denoted by $\pi(i)$. Since we can keep at most k requests pending at a time, a legal schedule must satisfy that the ith request to be served must be among first $i+k-1$ requests arrived, i.e., $\pi(i) \leq i+k-1$. The cost of the schedule is the total distance that the server has to travel, i.e., $C_\pi = \sum_{i=1}^{N} d(p_{\pi(i)}, p_{\pi(i-1)})$ where $\pi(0) = 0$ corresponds to the starting point. The goal in SBP is to find a legal schedule π that minimizes C_π where the $(i+1)$th request is revealed only after serving at least $i-k+1$ requests for $i \geq k$.

2.1 The disc model

Overlooking rotational latency, a disc is modeled as an arrangement of tracks numbered $1, \ldots, n$. The time taken to move the head from track i to track j is assumed to be $|i-j|$. The Disc Scheduling problem is the SBP on the line metric space $(\{1, \ldots, n\}, d)$ where $d(i, j) = |i-j|$. It is not known if the offline problem is NP-hard. We argue in Section 3.1.1 that the algorithm that serves the requests in the order they arrive is an $O(k)$-approximation. To the best of our knowledge, no algorithm with a better guarantee was known before.

3 Online SBP

3.1 Algorithm for Sorting Buffers on a Line

At the crux of our algorithm, we use the following simple observation. Imagine that the underlying metric space is partitioned into two sets S and T. Now if k requests at points in T are received while the server is at some point in S, any algorithm with buffer of size at most k must travel a distance of at least $d(S, T) := \min_{s \in S, t \in T} d(s, t)$. In Section 3.1.1, we use this observation to obtain

a lower bound on the cost of the optimum solution for a tree metric. Using this lower bound, in Section 3.1.2, we present a polylogarithmic approximation for binary hierarchically-separated-trees. The algorithm clusters the vertices according to their distances from the server-location and allocates disjoint *sub-buffers* for each cluster. It then serves a cluster if the corresponding sub-buffer overflows. This approach is then extended to the line metric in Section 3.1.3.

3.1.1 A lower bound on OPT *for a tree metric*

Consider first an instance of the SBP on a two-point metric $\{0, 1\}$ with $d(0, 1) = 1$. Let us assume that $p_0 = 0$. There is a simple algorithm that behaves optimally on this metric space. It starts by serving all requests at 0 till it accumulates k requests to 1. It then makes a transition to 1 and keeps serving requests to 1 till k requests to 0 are accumulated. It then makes a transition to 0 and repeats. It is easy to see that this algorithm is optimal. Now consider the First-In-First-Out algorithm that uses a buffer of size 1 and serves any request as soon as it arrives. It is easy to see that this algorithm is $O(k)$-competitive since it makes $O(k)$ trips for every trip of the optimum algorithm.

We use $\mathsf{OPT}(k)$ to denote both the optimum schedule with a buffer of size k and its cost. The following lemma states the relationship between $\mathsf{OPT}(\cdot)$ over different buffer sizes for a two-point metric.

Lemma 1 *For a two-point metric, for any* $1 \leq l \leq k$*, we have* $\mathsf{OPT}(k) \geq \mathsf{OPT}(\lceil k/l \rceil)/2l$.

Proof. Let $p_0 = 0$. By the time $\mathsf{OPT}(\lceil k/l \rceil)$ makes l trips to 1, we know that $\mathsf{OPT}(k)$ must have accumulated at least k requests to 1. Therefore $\mathsf{OPT}(k)$ must travel to 1 at least once. Note that the l (round) trips to 1 cost at most $2l$ for $\mathsf{OPT}(\lceil k/l \rceil)$. We can repeat this argument for every trip of $\mathsf{OPT}(k)$ to conclude the lemma.

Using the above observations, we now present a lower bound on the optimum cost for the SBP on a tree metric.[1] Consider a tree T with lengths $d_e \geq 0$ assigned to the edges $e \in T$. Let $p_1, \ldots, p_N$ denote the input sequence of points in the tree. Refer to Figure 1. Fix an edge $e \in T$. Let L_e and R_e be the two subtrees formed by removing e from T. We can shrink L_e to form a super-node 0 and shrink R_e to form a super-node 1 to obtain an instance of SBP on a two-point metric $\{0, 1\}$ with $d(0, 1) = d_e$. Let LB_e denote the cost of the optimum on this instance. It is clear that any algorithm must spend at least

[1] Recall that a tree metric is a metric on the set of vertices in a tree where the distances are defined by the path lengths between the pairs of points.

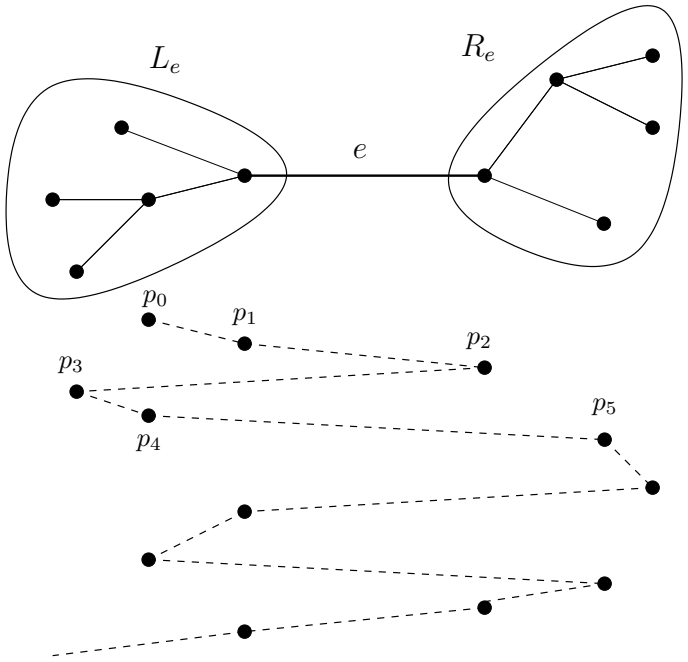

Fig. 1. Lower bound contributed by edge e in the tree

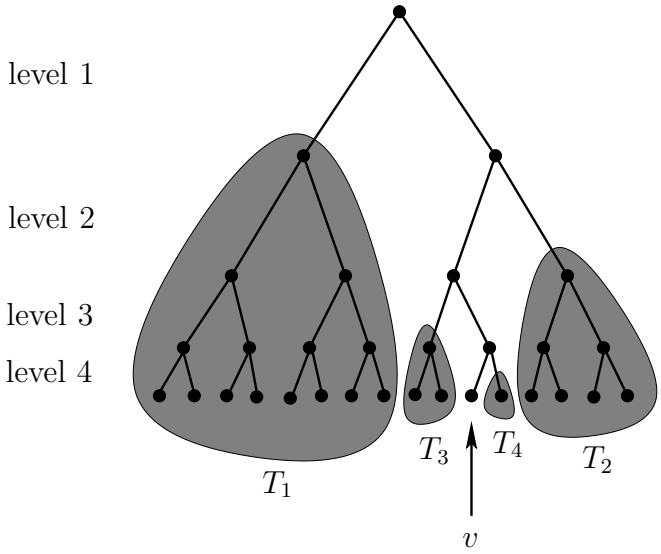

Fig. 2. Partition of the leaves in a phase of the algorithm

LB_e for traveling on edge e. Thus

$$\mathsf{LB} = \sum_{e \in T} \mathsf{LB}_e \tag{1}$$

is a lower bound on the cost OPT of the optimum on the original tree instance. Again, the algorithm that serves the requests in the order they arrive in, is $O(k)$-competitive.

Let $\mathsf{LB}(k)$ denote the above lower bound on $\mathsf{OPT}(k)$, the optimum with a buffer of size k. The following lemma follows from Lemma 1.

Lemma 2 *For any $1 \leq l \leq k$, we have $\mathsf{LB}(k) \geq \mathsf{LB}(\lceil k/l \rceil)/2l$.*

3.1.2 An algorithm on a binary tree

Consider a rooted binary tree T on $n = 2^h$ leaves. The height of this tree is $h = \log n$. The edges are partitioned into levels 1 to h according to their distance from the root; the edges incident to the root are in level 1 while the edges incident to the leaves are in level h. Figure 2 shows such a tree with $n = 8$ leaves. Let each edge in level i have cost $n/2^i$. Consider a metric on the leaves of this tree defined by the path lengths. In this section, we present a deterministic online algorithm for SBP on this metric that has a competitive ratio of $O(\log^2 n)$. Since the First-In-First-Out algorithm has a competitive ratio of $O(k)$, we assume that $k > h = \log n$. We also assume for simplicity that h divides k.

Algorithm. The algorithm goes in phases. Suppose that in the beginning of a phase, the server is present at a leaf v as shown in Figure 2. We partition the leaves other than v into h subsets as shown in the figure. Consider the path P_v from v to the root. Let T_i be the tree hanging to the path P_v at level i for $1 \leq i \leq h$. Let V_i be the set of leaves in T_i. We think of the sorting buffer of size k as being divided into h sub-buffers of size k/h each. We associate the ith sub-buffer with V_i, i.e., we accumulate all the pending requests in V_i in the ith sub-buffer. The algorithm maintains the following invariant.

> Invariant. Each of the h sub-buffers has at most k/h requests, i.e., there are at most k/h pending requests in any V_i.

We input new requests till one of the sub-buffers overflows. Suppose that the $j$the sub-buffer overflows. The algorithm then clears all the pending requests in the subtrees $T_j, T_{j+1}, \ldots, T_h$ by performing an Eulerian tour of the trees $T_j, \ldots, T_h$. At the end of the tour, the head is at an arbitrary leaf of T_j. The algorithm, then, enters the next phase.

To prove the correctness of the algorithm we have to argue that at most k requests are pending at any point in the algorithm.

Lemma 3 *The invariant is satisfied in the beginning of any phase.*

Proof. Initially, the invariant is trivially satisfied. Suppose that it is satisfied in the beginning a phase and that the jth sub-buffer overflows in that phase. The division into trees and sub-buffers changes after the move. However since the server resides in T_j, all the trees $T_1, \ldots, T_{j-1}$ and their corresponding sub-buffers remain unchanged. Also since the algorithm clears all the pending requests in the trees $T_j, \ldots, T_h$, the jth to hth sub-buffers after the move are all empty. Thus the invariant is also satisfied in the beginning of the next phase. ∎

Next we argue that the algorithm is $O(\log^2 n)$ competitive.

Theorem 1 *The total distance traveled by the server in the algorithm is* $O(\mathsf{OPT} \cdot \log^2 n)$.

Proof. For an edge $e \in T$, let $\mathsf{LB}_e(k)$ be the lower bound on $\mathsf{OPT}(k)$ contributed by e as defined in Section 3.1.1. Let $\mathsf{LB}(k) = \sum_e \mathsf{LB}_e(k)$. Let $\mathsf{LB}_e(k/h)$ and $\mathsf{LB}(k/h)$ be the corresponding quantities assuming a buffer size of k/h. We know from Lemma 2 that

$$\mathsf{OPT}(k) \geq \mathsf{LB}(k) \geq \mathsf{LB}(k/h)/2h = \mathsf{LB}(k/h)/2 \log n.$$

To prove the lemma, next we argue that the total cost of the algorithm is $O(\mathsf{LB}(k/h) \cdot \log n)$.

To this end, consider a phase t. Suppose the jth sub-buffer overflows in this phase. Let v be the leaf corresponding to the current position of the server and let u be the vertex on the path from v to the root between the levels j and $j-1$. In this phase, the algorithm spends at most twice the cost of subtree below u. Let e be the parent edge of tree T_j, i.e., the edge that connects T_j to u. Note that the cost of e is $n/2^j$ while the total cost of the subtree below u is $2(\log n - j) \cdot n/2^j = O(\log n \cdot n/2^j)$. With a loss of factor $O(\log n)$, we charge the cost of clearing the requests in $T_j \cup \cdots \cup T_h$ to the cost paid in traversing e in this phase. We say that the phase t transfers a charge of $n/2^j$ to e.

Now fix an edge $e \in T$. Let C_e denote the total charge transferred to e from all the phases. We show that $C_e \leq \mathsf{LB}_e(k/h)$. Refer to Figure 1. Let L_e and R_e be the two subtrees formed by removing e from T. Now C_e is the total cost paid by our algorithm for traversing e in the phases which transfer a charge to e. Note that in these phases, we traverse e to go from L_e to R_e or vice-versa only when there are at least k/h pending requests on the other side. Thus C_e is a at most $\mathsf{LB}_e(k/h)$, the lower bound contributed by e assuming a buffer size of k/h. Thus, $\sum_e C_e \leq \sum_e \mathsf{LB}_e(k/h) = \mathsf{LB}(k/h)$ and the proof is complete. ∎

Lemma 4 *For any $1 \leq \alpha \leq \log n$, there is an $O(\alpha^{-1} \log^2 n)$ competitive algorithm for SBP on the binary tree metric defined above if the algorithm is allowed to use a buffer of size αk.*

The algorithm is similar to the above one except that it assigns a sub-buffer of size $\alpha k/h$ to each of the h subtrees. The proof that it has the claimed competitive ratio is similar to that of Theorem 1 and is omitted.

3.1.3 An algorithm on the line metric

Our algorithm for the line metric is based on the probabilistic approximation of the line metric by a binary tree metric considered in the previous section.

Definition 1 (Bartal [2]) *A set of metric spaces $\mathcal{S}$ over a set of points V, α-probabilistically approximates a metric space M over V, if*

- *every metric space in $\mathcal{S}$ dominates M, i.e., for each $\mathcal{N} \in \mathcal{S}$ and $u, v \in V$ we have $\mathcal{N}(u, v) \geq M(u, v)$, and*
- *there exists a distribution over the metric spaces $\mathcal{N} \in \mathcal{S}$ such that for every pair $u, v \in V$, we have $\mathbb{E}[\mathcal{N}(u, v)] \leq \alpha M(u, v)$.*

Definition 2 *A r-hierarchically well-separated tree (r-HST) is an edge-weighted rooted tree with the following properties.*

- *The weights of edges between a node to any of its children are same.*
- *The edge weights along any path from root to a leaf decrease by at least a*

factor of r.

Bartal [3] showed that any connected edge-weighted graph G can be α - probabilistically approximated by a family r-HSTs where $\alpha = O(r \log n \log \log n)$. Fakcharoenphol, Rao and Talwar [8] later improved this factor to $\alpha = O(r \log n)$. We prove the following lemma for completeness.

Lemma 5 *The line metric on uniformly-spaced n points can be $O(\log n)$ probabilistically approximated by a family of binary 2-HSTs.*

Proof. Assume for simplicity that $n = 2^h$ for some integer h. Let M be a metric on $\{1, \ldots, n\}$ with $M(i, j) = |i - j|$. Consider a binary tree T on $2n$ leaves. Label the leaves from left to right as $l_1, \ldots, l_{2n}$. Partition the edges into levels as shown in Figure 2, i.e., the edges incident to the root are in level 1 and those incident to the leaves are in level $1 + \log n$. Assign a weight of $n/2^i$ to each edge in level i. Now pick r uniformly at random from the set $\{0, 1, \ldots, n-1\}$. Let $\mathcal{N}$ be the metric induced on the leaves $l_{r+1}, l_{r+2}, \ldots, l_{r+n}$ and consider a bijection from $\{1, \ldots, n\}$ to $\{l_{r+1}, \ldots, l_{r+n}\}$ that maps i to l_{r+i}.

It is easy to see that under this mapping, the metric space $\mathcal{N}$ dominates M. Now consider any pair i and $i + 1$. It is easy to see that $\mathbb{E}[\mathcal{N}(l_{r+i}, l_{r+i+1})] = O(\log n)$. By linearity of expectation, we have that for any pair $1 \leq i, j \leq n$, we have $\mathbb{E}[\mathcal{N}(l_{r+i}, l_{r+j})] = O(\log n \cdot |i-j|)$. Therefore this distribution over the binary 2-HSTs forms $O(\log n)$-approximation to the line metric as desired. ∎

It is now easy to extend our algorithm on binary 2-HSTs to the line metric. We first pick a binary 2-HST from the distribution that gives $O(\log n)$ probabilistic approximation to the line metric. Then we run our deterministic online $O(\log^2 n)$-competitive algorithm on this binary tree. It is easy to see that the resulting algorithm is a randomized online algorithm that achieves a competitive ratio of $O(\log^3 n)$ in expectation against an oblivious adversary. It is necessary that the adversary is oblivious to our random choice of the 2-HST metric. Again for $1 \leq \alpha \leq \log n$, we can improve the competitive ratio to $O(\alpha^{-1} \log^3 n)$ by using a buffer of size αk.

3.1.4 Improved Analysis

Previous analysis can be improved to show that the same algorithm is in fact $O(\log^2 n)$ competitive on the line metric. The main idea is to consider the actual distance traveled on the line at the end of each phase instead of using the tree approximation as a black box.

Note that in our algorithm, the disc head moves only at the end of a phase when the buffer for one of the levels overflows. Suppose the buffer for level j overflows at the end of a phase. So, the contribution by the parent edge of the

tree T_j to the lower bound of the tree instance is $n/2^j$. In this move, we clear all the requests belonging to the trees $T_j, T_{j+1}, \ldots, T_{1+\log n}$. Our embedding guarantees that the total distance traveled *on the line metric*, in order to clear all these requests, is $O(n/2^j)$. By adding over all the phases, it is clear that the total distance traveled by the algorithm is $O(LB)$ where LB is the lower bound on the tree instance with buffer size $k/\log n$. This, in turn, implies a randomized $O(\log^2 n)$ competitive ratio for our algorithm against an oblivious adversary. Again, for $1 \leq \alpha \leq \log n$, we can improve the competitive ratio to $O(\alpha^{-1} \log^2 n)$ by using a buffer of size αk.

4 Offline SBP

One natural question to ask in the light of the online algorithm presented in the previous section is, does the offline version admit better approximation algorithms? In this section, we consider the offline SBP on the line metric and our results indicate that much improved approximation ratios might be possible in this case. Our main result is a constant factor approximation algorithm with a quasi-polynomial running time. Considering the likelihood of $\mathcal{NP} \not\subseteq$ DTIME $[n^{\text{polylog } n}]$, it is likely that the offline SBP on the line metric has a constant factor approximation. Our algorithm is based on an exact dynamic program that is quite different from the one indicated in the introduction. It runs in time $N \cdot n \cdot k^{O(\log n)}$.

4.1 Outline of our approach

We start by describing an exact algorithm for the offline SBP on a general metric on n points. As we will be interested in the line metric as in the disc scheduling problem, we use the term "head" for the server and "tracks" for the points. Since the first k requests can be buffered without loss of generality, we fetch and store them in the buffer. At a given step in the algorithm, we define a *configuration* (t, C) to be the pair of current head location t and an n-dimensional vector C that specifies the number of requests pending at each track. Since there are n choices for t and a total of k requests pending, the number of distinct configurations is $O(n \cdot k^n)$. We construct a dynamic program that keeps track of the current configuration and computes the optimal solution in time $O(N \cdot n \cdot k^n)$ where N is the total number of requests. The dynamic program proceeds in N levels. For each level i and each configuration (t, C), we compute the least cost of serving i requests from the first $i + k$ requests and ending up in the configuration (t, C). Let us denote this cost by

$\texttt{DP}[i, t, C]$. This cost can be computed using the relation

$$\texttt{DP}[i, t, C] = \min_{(t', C')} (\texttt{DP}[i-1, t', C'] + d(t', t))$$

where the minimum is taken over all configurations (t', C') such that while moving the head from t' to t, a request at either t' or t in C' can be served and a new request can be fetched to arrive at the configuration (t, C). Note that it is easy to make suitable modifications to keep track of the order of the output sequence.

Note that the high complexity of the above dynamic program is due to the fact that we keep track of the number of pending requests at *each* of the n tracks. We now describe our intuition behind obtaining much smaller dynamic program for the line metric on n uniformly spaced points. Our dynamic program keeps track of the number of pending requests only in $O(\log n)$ segments of the line which are geometrically increasing in lengths. The key observation is as follows: if the optimum algorithm moves the head from a track t to t' (thereby paying the cost $|t - t'|$), a constant factor approximation algorithm can safely move an additional $O(|t - t'|)$ distance and clear all the nearby requests surrounding t and t'. We show that instead of keeping track of the number of pending requests at each track, it is enough to do so for the ranges of length $2^0, 2^1, 2^2, 2^3, \ldots$ surrounding the current head location t. For each track t, we partition the disc into $O(\log n)$ ranges of geometrically increasing lengths on both sides of t. The configuration (t, C) now refers to the current head location t and an $O(\log n)$-dimensional vector C that specifies number of requests pending in each of these $O(\log n)$ ranges. Thus the new dynamic program will have size $N \cdot n \cdot k^{O(\log n)}$.

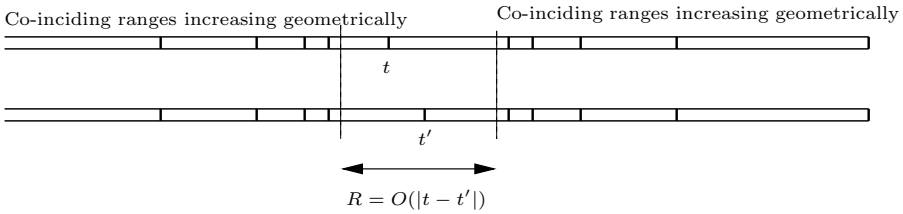

Fig. 3. Division of the line into ranges for tracks t and t'

To be able to implement the dynamic program, we ensure the property that the new configuration around t' should be easily computable from the previous configuration around t. More precisely, we ensure that the partitions for t and t' satisfy the following property: outside an interval of length $R = O(|t - t'|)$ containing t and t', the ranges in the partition for t coincide with those in the partition for t' (see Figure 3). Note however that inside this interval, the two partitions may not agree. Thus when the optimum algorithm moves the head from t to t', our algorithm starts the head from t, clears all the pending requests in this interval and rests the head at t' and updates the configuration from the previous configuration. Since the length of the interval is $O(|t - t'|)$,

our algorithm spends at most a constant factor more than the optimum.

4.2 Partitioning Scheme

Now we define a partitioning scheme and its properties that are used in our algorithm. Let us assume, without loss of generality, that the total number of tracks $n = 2^L$ is a power of two and that the tracks are numbered from 0 to $2^L - 1$ left-to-right. In the following, we do not distinguish between a track and its number. For tracks t and t', the quantity $|t - t'|$ denotes the distance between these tracks which is the cost paid in moving the head from t to t'. We say that a track t is to the right (resp. left) of a track t' if $t > t'$ (resp. $t < t'$).

Definition 3 (landmarks) *For a track t and an integer $p \in [1, L]$, we define pth landmark of t as $\ell_p(t) = (q+1)2^p$ where q is the unique integer such that $(q-1)2^p \leq t < q2^p$. We also define $(-p)$th landmark as $\ell_{-p}(t) = (q-2)2^p$. We also define $\ell_0(t) = t$.*

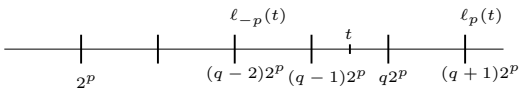

Fig. 4. The pth and $(-p)$th landmarks of a track t

It is easy to see that $\ell_{-n}(t) < \cdots < \ell_{-1}(t) < \ell_0(t) < \ell_1(t) < \cdots < \ell_n(t)$. In fact the following lemma claims something stronger and follows easily from the above definition.

Lemma 6 *Let $p \in [1, L-1]$ and $(q-1)2^p \leq t < q2^p$ for an integer q.*

- *If q is even, then $\ell_{p+1}(t) - \ell_p(t) = 2^p$ and $\ell_{-p}(t) - \ell_{-p-1}(t) = 2^{p+1}$.*
- *If q is odd, then $\ell_{p+1}(t) - \ell_p(t) = 2^{p+1}$ and $\ell_{-p}(t) - \ell_{-p-1}(t) = 2^p$.*

In the following definition, we use the notation $[a, b) = \{t \text{ integer} \mid a \leq t < b\}$.

Definition 4 (ranges) *For a track t, we define a "range" to be a contiguous subset of tracks as follows.*

- *$[\ell_{-1}(t), \ell_0(t) = t)$ and $[\ell_0(t) = t, \ell_1(t))$ are ranges.*
- *for $p \in [1, L-1]$,* **if** *$\ell_{p+1}(t) - \ell_p(t) = 2^{p+1}$ and $\ell_p(t) - \ell_{p-1}(t) = 2^{p-1}$* **then** *$[\ell_p(t), \ell_p(t) + 2^p)$ and $[\ell_p(t) + 2^p, \ell_{p+1}(t))$ are ranges,* **else** *$[\ell_p(t), \ell_{p+1}(t))$ is a range.*
- *for $p \in [1, L-1]$,* **if** *$\ell_{-p}(t) - \ell_{-p-1}(t) = 2^{p+1}$ and $\ell_{-p+1}(t) - \ell_{-p}(t) = 2^{p-1}$* **then** *$[\ell_{-p-1}(t), \ell_{-p-1}(t) + 2^p)$ and $[\ell_{-p-1}(t) + 2^p, \ell_{-p}(t))$ are ranges,* **else** *$[\ell_{-p-1}(t), \ell_{-p}(t))$ is a range.*

The above ranges are disjoint and form a partition of the tracks which we denote by $\pi(t)$.

Note that in the above definition, when the difference $\ell_{p+1}(t) - \ell_p(t)$ and $\ell_{-p}(t) - \ell_{-p-1}(t)$ equals 4 times $\ell_p(t) - \ell_{p-1}(t)$ and $\ell_{-p+1}(t) - \ell_{-p}(t)$ respectively, we divide the intervals $[\ell_p(t), \ell_{p+1}(t))$ and $[\ell_{-p-1}(t), \ell_{-p}(t))$ into two ranges of length 2^p each. For example, in Figure 5, the region between $\ell_{p+2}(t)$ and $\ell_{p+3}(t)$ is divided into two disjoint ranges of equal size.

The following lemma proves a useful relation between the partitions $\pi(t)$ and $\pi(t')$ for a pair of tracks t and t': the ranges in the two partitions coincide outside the interval of length $R = O(|t - t'|)$ around t and t'. As explained in Section 4.1, such a property is important for carrying the information about the current configuration across the head movement from t to t'.

Lemma 7 *Let t and t' be two tracks such that $2^{p-1} \leq t' - t < 2^p$. The ranges in $\pi(t)$ and $\pi(t')$ are identical outside the interval $R = [\ell_{-p}(t), \ell_p(t'))$.*

Proof. First consider the case when $(q-1)2^p \leq t < t' < q2^p$ for an integer q, i.e., t and t' lie in the same "aligned" interval of length 2^p. Then clearly they also lie in the same aligned interval of length 2^r for any $r \geq p$. Thus, by definition, $\ell_r(t) = \ell_r(t')$ for $r \geq p$ and $r \leq -p$. Thus it is easy to see from the definition of ranges that the ranges in $\pi(t)$ and $\pi(t')$ outside the interval $[\ell_{-p}(t), \ell_p(t'))$ are identical.

Consider now the case when t and t' do not lie in the same aligned interval of length 2^p. Since $|t - t'| < 2^p$, they must lie in the adjacent aligned intervals of length 2^p, i.e., for some integer q, we have $(q-1)2^p \leq t < q2^p \leq t' < (q+1)2^p$ (See Figure 5). Let $q = 2^u v$ where $u \geq 0$ is an integer and v is an odd integer.

The following key claim states that depending upon how r compares with the the highest power of two that divides the "separator" $q2^p$ of t and t', either the rth landmarks of t and t' coincide with each other or the $(r+1)$th landmark of t coincides with the rth landmark of t'.

Claim 1 *(1) $\ell_r(t) = \ell_r(t')$ for $r \geq p + u + 1$ and $r \leq -p - u - 1$.*
(2) $\ell_{r+1}(t) = \ell_r(t')$ for $p \leq r < p + u$,
(3) $\ell_{-r}(t) = \ell_{-r-1}(t')$ for $p \leq r < p + u$,
(4) $\ell_{p+u}(t') = \ell_{p+u}(t) + 2^{p+u}$ and $\ell_{p+u+1}(t) - \ell_{p+u}(t) = 2^{p+u+1}$,
(5) $\ell_{-p-u}(t) = \ell_{-p-u-1}(t') + 2^{p+u}$ and $\ell_{-p-u}(t') - \ell_{-p-u-1}(t') = 2^{p+u+1}$,

Proof. The equation 1 follows from the fact that since 2^{p+u} is the highest power of two that divides $q2^p$, both t and t' lie in the same aligned interval of length 2^r for $r \geq p + u + 1$.

The equations 2, 3, 4, and 5 follow from the definition of the landmarks and the fact that t and t' lie in the different but adjacent aligned intervals of length 2^r for $p \leq r < p+u$ (see Figure 5). ∎

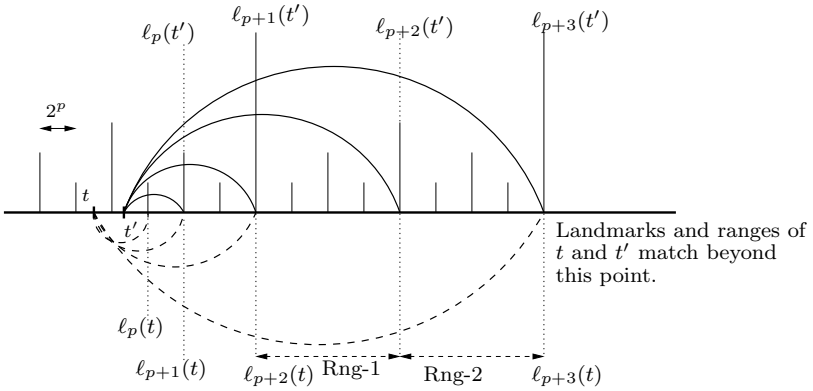

Fig. 5. Landmarks and ranges for tracks t and t' when $q = 4, u = 2$.

Claim 1 implies that all but one landmarks of t and t' coincide with each other. For the landmarks of t and t' that coincide with each other, it follows from the definition of the ranges that the corresponding ranges in $\pi(t)$ and $\pi(t')$ are identical.

The landmarks of t, t' that do not coincide are $\ell_{p+u}(t') = \ell_{p+u}(t) + 2^{p+u}$ and $\ell_{-p-u}(t) = \ell_{-p-u-1}(t') + 2^{p+u}$. But, note that the intervals $[\ell_{p+u}(t), \ell_{p+u+1}(t))$ and $[\ell_{-p-u-1}(t'), \ell_{-p-u}(t'))$ are divided into two ranges each: $[\ell_{p+u}(t), \ell_{p+u}(t) + 2^{p+u})$, $[\ell_{p+u}(t) + 2^{p+u}, \ell_{p+u+1}(t))$ and $[\ell_{-p-u-1}(t'), \ell_{-p-u-1}(t') + 2^{p+u})$, $[\ell_{-p-u-1}(t') + 2^{p+u}, \ell_{-p-u}(t'))$. These ranges match with $[\ell_{p+u-1}(t'), \ell_{p+u}(t'))$, $[\ell_{p+u}(t'), \ell_{p+u+1}(t'))$ and $[\ell_{-p-u-1}(t), \ell_{-p-u}(t))$, $[\ell_{-p-u}(t), \ell_{-p-u+1}(t))$ respectively. This follows again from the Claim 1 and the carefully chosen definition of ranges. Thus the proof of Lemma 7 is complete.

∎

For tracks t and t', where $t < t'$, let $R(t, t') = R(t', t)$ be the interval $[\ell_{-p}(t), \ell_p(t'))$ if $2^{p-1} \leq t' - t < 2^p$. Note that the length of the interval $R(t, t')$ is at most $|\ell_{-p}(t) - \ell_p(t')| \leq 4 \cdot 2^p \leq 8 \cdot |t - t'|$. Thus the total movement in starting from t, serving all the requests in $R(t, t')$, and ending at t' is at most $15 \cdot |t - t'|$.

4.3 The Dynamic Program

Our dynamic program to get a constant approximation for the offline SBP on a line metric is based on the intuition given in Section 4.1 and uses the partition scheme given in Section 4.2. Recall that according to the intuition, when the optimum makes a move from t to t', we want our algorithm to clear all the requests in $R(t, t')$. This motivates the following definition.

Definition 5 *A feasible schedule for serving all the requests is said to be "locally greedy" if there is a sequence of tracks $t_1, \ldots, t_l$, called "landmarks", which are visited in that order and while moving between any consecutive pair of tracks t_i and t_{i+1}, the schedule also serves all the current pending requests in the interval $R(t_i, t_{i+1})$.*

Since the total distance traveled in a locally greedy schedule corresponding to the optimum schedule is at most 15 times that of the optimum schedule, the best locally greedy schedule is a 15-approximation to the optimum. Our dynamic program computes the best locally greedy schedule. For a locally greedy schedule, let a configuration be defined as a pair (t, C) where t is the location of the head and C is an $O(\log n)$-dimensional vector specifying the number of requests pending in each range in the partition $\pi(t)$. Clearly the number of distinct configurations is $O(n \cdot k^{O(\log n)})$.

The dynamic program is similar to the one given in Section 4.1 and proceeds in N levels. For each level i and each configuration (t, C), we compute the least cost of serving i requests from the first $i + k$ requests and ending up in the configuration (t, C) in a locally greedy schedule. Let $\texttt{DP}[i, t, C]$ denote this cost. This cost now can be computed as follows. Consider a configuration (t', C') after serving $i - r$ requests for some $r > 0$ such that while moving from a landmark t' to the next landmark t,

(1) the locally greedy schedule serves exactly r requests from the interval $R(t', t)$,
(2) it travels a distance of D, and
(3) after fetching r new requests, it ends up in the configuration (t, C).

In such a case,

$$\texttt{DP}[i - r, t', C'] + D$$

is an upper bound on $\texttt{DP}[i, t, C]$. Taking the minimum over all such upper bounds, one obtains the value of $\texttt{DP}[i, t, C]$.

Recall that the locally greedy schedule clears all the pending requests in the interval $R(t', t)$ while moving from t' and t and also that the ranges in $\pi(t)$ and $\pi(t')$ coincide outside the interval $R(t', t)$. Thus it is feasible to determine if after serving r requests in $R(t', t)$ and fetching r new requests, the schedule ends up in the configuration (t, C).

The dynamic program, at the end, outputs $\min_t \texttt{DP}[N, t, \mathbf{0}]$ as the minimum cost of serving all the requests by a locally greedy schedule. It is also easy to modify the dynamic program to compute the minimum cost locally greedy schedule along with its cost.

5 Conclusions

Prior to our work, there were no non-trivial algorithms, either online or offline, for the SBP on the line metric. We provided the first approximation algorithms in both the settings. The competitive ratio for the online SBP on the line metric has been improved subsequent to our work. Also, our technique for the line metric has been extended to general metric spaces. It would be interesting to see if a constant factor approximation can be achieved for the online SBP on the line metric. Another point to be noted is that, for the case of uniform metric spaces, the approximation ratio is independent of n, the number of possible points on which requests can come. Instead, it is dependent on k which tends to be much smaller. It would be very interesting if similar approximation ratios can be achieved for the line metric. That would be very compelling for the disc scheduling problem from a practical point of view.

Questions related to the complexity of the SBP are mostly open. It is not known whether the SBP on uniform metric and line metric are NP-hard. Resolving the hardness of these problems remain very challenging. Proving lower bounds on the competitive ratios of online algorithms would also be very interesting. In this paper, we showed lower bounds on deterministic memoryless algorithms. However, such algorithms have very less power. Can we at least prove lower bounds for stronger models?

References

[1] R. Bar-Yehuda and J. Laserson. Exploiting locality: Approximating sorting buffers. *Journal of Discrete Algorithms*, 5(4):729–738, 2007.

[2] Y. Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *IEEE Symposium on Foundations of Computer Science*, pages 184–193, 1996.

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

[4] M. Charikar, S. Khuller, and B. Raghavachari. Algorithms for capacitated vehicle routing. *SIAM Journal of Computing*, 31:665–682, 2001.

[5] M. Charikar and B. Raghavachari. The finite capacity dial-a-ride problem. In *IEEE Symposium on Foundations of Computer Science*, pages 458–467, 1998.

[6] M. Englert, H. Räcke, and M. Westermann. Reordering buffers for general metric spaces. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 556–564, 2007.

[7] M. Englert and M. Westermann. Reordering buffer management for non-

uniform cost models. In *Proceedings of the 32nd International Colloquium on Algorithms, Langauages, and Programming*, pages 627–638, 2005.

[8] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *35th Annual ACM Symposium on Theory of Computing*, pages 448–455, 2003.

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

[10] A. Gupta, M. Hajiaghayi, V. Nagarajan, and R. Ravi. Dial a ride from -forest. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 241–252, 2007.

[11] J. Kohrt and K. Pruhs. A constant approximation algorithm for sorting buffers. In *LATIN 04*, pages 193–202, 2004.

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

[13] A. Silberschatz, P. Galvin, and G. Gagne. *Applied Operating System Concepts*, chapter 13. Mass-Storage Structure, pages 435–468. John Wiley and Sons, first edition, 2000. Disc Scheduling is discussed in Section 13.2.

6 Appendix

6.1 Why Some Natural Strategies Fail for the Online SBP

Many natural deterministic strategies for the online SBP suffer from one of the following drawbacks.

(1) Some strategies block large part of the sorting buffer with requests that are kept pending for a long time. Doing this, the effective buffer size drops well below k, thereby yielding a bad competitive ratio. For example, consider the Nearest-First (also called Shortest-Trip-First (STF)) strategy that always serves the request nearest to the current head location. Suppose that the initial head location is 0. Let the input sequence be $3, \ldots, 3, 1, 0, 1, 0, \ldots$ where there are $k-1$ requests to 3. The STF strategy keeps the $k-1$ requests to 3 pending and serves the 1s and 0s alternatively using an effective buffer of size 1. The optimum schedule, on the other hand, gets rid of the requests to 3 by making a single trip to 3 and uses the full buffer to serve the remaining sequence. It is easy to see that if the sequence of 1s and 0s is large enough, the cost of STF is $\Omega(k)$ times that of the optimum.

(2) Some other strategies, in an attempt to free the buffer slots, travel too far too often. The optimum, however, saves the distant requests and serves

about k of them at once by making a single trip. Consider, for example, the First-In-First-Out (FIFO) strategy that serves the requests in the order they arrive. Suppose again that the initial head location is 0 and the input sequence is a repetition of the following block of requests: $n, 1, \ldots, 1, 0, \ldots, 0$ where there are k requests to 1 and k requests to 0 in each block. The FIFO strategy makes a trip to n for each block while the optimum serves 1s and 0s for $k-1$ blocks and then serves the accumulated $k-1$ requests to n by making a single trip to n. Note that, in doing this, the effective buffer size for the optimum reduces from k to 1. However, for the sequence of k 1s followed by k 0s, having a buffer of size k is no better than having buffer of size 1. It is now easy to see that if $n = k$, then FIFO is $\Omega(k)$ worse than the optimum.

Thus a good strategy must necessarily strike a balance between clearing the requests soon to free the buffer and not traveling too far too often. Obvious combinations of the two objectives, like making decisions based on the ratios of the distance traveled to the number of requests served also fail on similar input instances. We refer the reader to Räcke et al. [12] for more discussion.

6.2 Memoryless Deterministic Algorithms for the Online SBP

We call a deterministic algorithm *memoryless* if it makes its scheduling decisions based purely on the set of pending requests in the buffer and the current position of the server. Such an algorithm can be completely specified by a function ρ such that for every possible (multi-)set S of k requests in the buffer and current position p of the server, the algorithm picks a request $\rho(S, p) \in S$ to serve next. The Nearest-First strategy is an example of a memoryless algorithm.

Theorem 2 *Any memoryless algorithm is $\Omega(k)$-competitive.*

Proof. We consider $k+1$ points $S = \{1, k, k^2, \ldots, k^k\}$ on a line. Consider any deterministic memoryless algorithm A. Suppose the head is initially at 1. We start with a request at each of the points $k, \ldots, k^k$. Whenever A moves the head from location p_i to location p_j, we release a new request at p_i. Thus, at all times, the pending requests and the head location together span all the $k+1$ points in S. Construct a directed graph G on S as vertices as follows. Add an edge from p_i to p_j if from the configuration with the head at p_i (with pending requests at all other points), A moves the head to p_j. Observe that the out-degree of every vertex is one and that G must have a cycle reachable from 1.

Suppose there is a cycle of length two between points $p_i, p_j \in S$ that is reachable from 1. In this case, we give an input sequence as follows. We first follow

the path from 1 to p_i so that the head now resides at p_i. We then give a long sequence of the requests of the form $p_j, p_i, p_j, p_i, \ldots$. A serves p_js and p_is alternatively, keeping the other $k-1$ requests pending in the buffer. The optimum algorithm will instead clear the other $k-1$ requests first and use the full buffer to save an $\Omega(k)$ factor in the cost. Note that this situation is "blocking-the-buffer" (item 1) in the discussion on why some strategies fail.

Suppose, on the other hand, all the cycles reachable from 1 are of length greater than two. Consider such a cycle C on $p_1, p_2, \ldots p_c \in S$ where $c > 2$ and $p_1 > p_2, \ldots, p_c$. Note that the edges (p_1, p_2) and (p_c, p_1) have lengths that are $\Omega(k)$ times the total lengths of all the other edges in C. We now give an input sequence as follows. We first make A bring the head at p_1 and then repeat the following block of requests several times: $p_2, p_3, \ldots, p_c$. For each such block, A makes a trip of C while the optimum serves $p_2, \ldots, p_c$ repeatedly till it accumulates $k-1$ requests to p_1 and then clears all of them in one trip to p_1. Thus overall it saves an $\Omega(k)$ factor in the cost over A. Note that this situation is "too-far-too-often" (item 2) in the discussion on why some strategies fail.

Thus, any deterministic memoryless strategy has a competitive ratio of $\Omega(k)$.

Note that $n = k^k$ in the above example. Thus the lower bound we proved in terms of n is $\Omega(\frac{\log n}{\log \log n})$.