Space-Constrained Interval Selection[*]

Yuval Emek[1], Magnús M. Halldórsson[2,**], and Adi Rosén[3,***]

[1] ETH Zurich, Zurich, Switzerland
emek@tik.ee.ethz.ch

[2] ICE-TCS, School of Computer Science, Reykjavik University, Iceland
mmh@ru.is

[3] CNRS and Université Paris Diderot, France
adiro@lri.fr

Abstract. We study streaming algorithms for the interval selection problem: finding a maximum cardinality subset of disjoint intervals on the line. A deterministic 2-approximation streaming algorithm for this problem is developed, together with an algorithm for the special case of proper intervals, achieving improved approximation ratio of 3/2. We complement these upper bounds by proving that they are essentially best possible in the streaming setting: it is shown that an approximation ratio of $2 - \epsilon$ (or $3/2 - \epsilon$ for proper intervals) cannot be achieved unless the space is linear in the input size. In passing, we also answer an open question of Adler and Azar [1] regarding the space complexity of constant-competitive randomized preemptive online algorithms for the same problem.

1 Introduction

In this paper we consider the *interval selection* problem, namely, finding a maximum cardinality subset of disjoint intervals from a given collection of intervals on the real line. It is well known that this problem has a simple optimal algorithm in the classical setting when the complete set of intervals is given to the algorithm [15]. Here we study this problem in the *streaming* model [17,23], where the input is given to the algorithm as a stream of items (intervals in our case), one at a time, and the algorithm has a limited memory that precludes storing the whole input. Yet, the algorithm is still required to output a feasible solution, with a good approximation ratio.

The motivation for the streaming model stems from applications of managing very large data sets, such as biological data (DNA sequencing), network traffic data, and more. Although some function of the whole data set is to be computed, it is impossible to store the whole input. Depending on the setting, different variants of the streaming model have been considered in the literature, such as the classical streaming model [17] or the so-called *semi-streaming* model

[*] Refer to [9] for a full version of this extended abstract.

[**] Research partially supported by grant 90032021 from the Icelandic Research Fund.

[***] Research partially supported by ANR projects QRAC and ALADDIN.

[12]. Common to all of them is the fact that the space used by the streaming algorithm is linear in some natural upper bound on the size of the output it returns (sometimes, a multiplicative polylogarithmic overhead is allowed).

In many problems considered in the streaming literature, the size of the output is fully determined by some parameter of the input, and thus, one would typically express the space complexity as a function of this parameter (cf. [4,13]). However, in other problems, the size of the output cannot be a priori expressed that way as it depends on the given instance; in such settings it is natural to seek a streaming algorithm whose space complexity is not much larger than the output size of the given instance (cf. [16]). Clearly, as long as the computational model of the streaming algorithm is based on a Turing machine with no distinction between the working tape and the output tape, the size of the output is an inherent lower bound on the required space.

In this paper, we consider a setting where the algorithm is given a stream of real-line intervals, each one defined by its two endpoints, and the goal is to compute a maximum cardinality subset of disjoint intervals (or an approximation thereof). This problem finds many applications, e.g., in resource allocation problems, and it has been extensively studied in the online and offline settings in many variants. We seek algorithms with a good upper bound on the space they use for a given instance, expressed in terms of the size of the output for that specific instance. Typically, we seek algorithms that use space which is at most linear in the size of the output and yet guarantee a good approximation ratio.

Related Work. The offline interval selection problem corresponds to finding a maximum independent set in an interval graph. An optimal greedy algorithm was discovered early [15] and has since been a staple of algorithms textbooks [8,18]. It should be noted that the input can be given in (at least) two different ways: as an intersection graph with the nodes corresponding to the intervals, or as a set of intervals given by their endpoints. This distinction makes little difference in the traditional offline setting, where switching between these representations can be done efficiently. However, it can be important in access- or resource-constrained settings. We choose to study the interval selection problem assuming the latter representation — that is, the input is given as a set of intervals — since we believe that it makes more sense in applications related to the online and streaming settings (most previous works on online interval selection make the same assumption).

The study of space-constrained algorithms goes back at least to the 1980 work of Munro and Paterson on selection and sorting [22]. More recently, the streaming model was developed to capture the processing of massive data-sets that arise in practice [23]. Most streaming algorithms deal with the approximate computation of various statistics, or "heavy hitters", as exemplified by the celebrated paper of Alon, Matias, and Szegedy [4].

A number of classic graph theoretic problems have been treated in the streaming setting, for example, matching problems [20,11], diameter and shortest paths [12,13], min-cut [3], and graph spanners [13]. These were mostly studied under the *semi-streaming* model, introduced by Feigenbaum et al. [12]; in this model, the algorithm is allowed to use $n\log^{O(1)}(n)$ space on an n-vertex graph (i.e., $\log^{O(1)}(n)$ bits per vertex). Closest to our problem, the independent set problem in general sparse graphs (and hypergraphs) was studied in the streaming setting by Halldórsson et al. [16]. Geometric streaming algorithms have also been appearing in recent years, especially dealing with extent and ranges, such as [2].

There is a plethora of literature on interval selection in the online setting. Some papers capture the problem as a call admission problem on a linear network, with the objective of maximizing the number (or weight) of accepted calls. Awerbuch et al. [5] present a strongly $\lceil \log N \rceil$-competitive algorithm for the problem, where N is the number of nodes on the line (corresponding to the number of possible interval endpoints). This yields an $O(\log \Delta)$-competitive algorithm for the weighted case, where Δ is the ratio between the longest to the shortest interval. On the negative side, Awerbuch et al. [5] establish a lower bound of $\Omega(\log N)$ on the competitive ratio of randomized non-preemptive online interval selection algorithms. In the context of the real line, this immediately implies that such algorithms cannot have competitive ratio that does not depend on the length of the input. In fact, Bachmann et al. [6] recently showed that the competitive ratio of randomized non-preemptive online algorithms for interval selection on the real line must be linear in the number of intervals in the input. Preemptive online scheduling has a lower bound of $\Omega(\log \Delta / \log\log \Delta)$ in the weighted case [7]. In comparison, much better results are possible for preemptive online algorithms in the unweighted setting: Adler and Azar [1] devise a 16-competitive algorithm. One way of easing the task of the algorithm is to assume arrival by time, i.e., the intervals arrive in order of left endpoints. This has been treated for different weighted problems [19,24,21,14,10].

Our Results. We give tight results for the interval selection problem in the streaming setting. Our main positive result is a deterministic 2-approximation streaming algorithm that uses space linear in the size of the output (Sect. 3). This is complemented by a matching lower bound (Sect. 4), stating that an approximation ratio of $2-\epsilon$ cannot be obtained by any randomized streaming algorithm with space significantly smaller than the size of the input (which is much larger than the size of the output). The special case of proper interval collections (i.e., collections of intervals with no proper containments) is also considered, for which a deterministic 3/2-approximation streaming algorithm that uses space linear in the output size is presented (deferred to [9]); a matching lower bound on the approximation ratio is established (Sect. 4) for streams of unit intervals (a special case of proper intervals). The upper bounds are extended to *multiple-pass* streaming algorithms: we show that an approximation ratio $1 + 1/(2p-1)$ can be obtained in p passes over the input (deferred to [9]).

In passing, we also answer an open question posed by Adler and Azar [1] in the context of randomized preemptive online algorithms for the interval selection problem. Adler and Azar point out that the decisions made by their online algorithm depend on the whole history (i.e., the input seen so far) and that natural attempts to remove this dependency seem to fail. Consequently, they write (using the term "active call" for an interval in the solution maintained by the online algorithm) that *"it seems very interesting to find out whether there exist constant-competitive algorithms where each decision depends only on the currently active calls and maybe on additional bounded information"*. We answer this question affirmatively by slightly modifying our main algorithm to achieve a randomized preemptive online algorithm that admits constant competitive ratio (slightly improving on that of [1]) and uses space linear in the size of the optimal solution, rather than the size of the input, as the algorithm of Adler and Azar does (deferred to [9]).

2 Preliminaries

We think of the real line $\mathbb{R}$ as stretching from left to right so that an *interval* I contains all points between its left *endpoint* $\text{left}(I)$ and its right endpoint $\text{right}(I)$, where $\text{left}(I) < \text{right}(I)$. Each endpoint can be either *open* (exclusive) or *closed* (inclusive). A *half-open* interval has a closed left endpoint and an open right endpoint. (This is, perhaps, the natural interval type to use in most resource allocation applications.) Observe that the assumption that $\text{left}(I) < \text{right}(I)$ implies that every interval contains an open set (in the topological sense) and that half-open intervals are always well defined.

The interval related notions of *intersection*, *disjointness*, and *containment* follow the standard view of an interval as a set of points. Two intervals I, J *properly* intersect if they intersect without containment; I properly contains J if I contains J and J does not contain I. An interval collection $\mathcal{I}$ is said to be *proper* (and the intervals in the collection, *proper* intervals) if no two intervals in $\mathcal{I}$ exhibit proper containment. The *load* of $\mathcal{I}$ is defined to be $\max_{p \in \mathbb{R}} |\{I \in \mathcal{I} \mid p \in I\}|$.

The *interval selection* problem asks for a maximum cardinality subset of pairwise disjoint intervals out of a given set S of intervals. In the streaming model, the input interval set S is considered to be an ordered set (a.k.a. a *stream*) and the intervals arrive one by one according to that order. The intervals are specified by their endpoints, where each endpoint is represented by a bit string of length b (the same b for all endpoints). This may potentially provide a streaming algorithm with the edge of knowing in advance some bounds on the number of intervals that will arrive and on the number of intervals that can be placed between two existing intervals. However, our algorithms do not take advantage of this extra information and our lower bounds show that it is essentially useless. An optimal solution to a given instance S of the interval selection problem is denoted by $\texttt{Opt}(S)$.

We may sometimes talk about *segments*, rather than intervals, when we want to emphasize that the entities under consideration are not part of the input. Given a set $\mathcal{I}$ of intervals, a *component* (or *connected component*) of $\mathcal{I}$ is a maximal continuous segment in $\bigcup_{I \in \mathcal{I}} I$.

3 The Main Algorithm

Overview. Given a stream S of intervals, our algorithm maintains a collection $A \subseteq S$, referred to as the *actual* intervals, from which the output $\texttt{Alg}(S) =$

$\texttt{Opt}(A)$ is taken. It also maintains a collection V of *virtual* intervals, where each virtual interval is the intersection of two actual intervals that existed in A at some point. The role of the virtual intervals is to filter out undesired intervals from joining A: an arriving interval $I \in S$ joins A if and only if it does not contain any currently maintained virtual or actual interval.

Our algorithm is designed to guarantee that each interval $I \in S$ leaves a *trace* in either A or V, namely, there exists some $J \in A \cup V$ such that $J \subseteq I$. Moreover, if $I, I' \in A$ properly intersect, then $I \cap I' \in V$. This essentially means that an arriving interval is rejected if and only if it contains some previous interval of S or the intersection of two properly intersecting previous intervals in S that have belonged to A.

Following that, it is not difficult to show that the load of the interval collection A is at most 2. Based on a careful analysis of the structure of the (connected) components in A and the locations of the virtual intervals within these components and between them, we can argue that $|V| \leq |A|$. This immediately yields the desired upper bound on the space of our algorithm as $|A| \leq 2 \cdot |\texttt{Opt}(A)|$. The bound on the approximation ratio essentially stems from the observation that $|\texttt{Opt}(S)| \leq |\texttt{Opt}(A \cup V)|$ (a direct corollary of the fact that each interval in S leaves a trace in $A \cup V$) and from the invariant that each actual interval contains at most 2 virtual intervals.

It is interesting to point out that our algorithm is in fact a deterministic preemptive online algorithm that maintains a load-2 interval collection (the collection A). Since the main result of Adler and Azar [1] also relies on such an algorithm, one may wonder if the two algorithms can be compared. Actually, the algorithm of Adler and Azar bases its rejection (and preemption) decisions on similar conditions: an arriving interval is rejected if and only if it contains some previous interval of S or the intersection of two properly intersecting intervals in A. (Adler and Azar use a different terminology, but the essence is very similar.) The difference lies in the latter condition: whereas the algorithm of Adler and Azar considers only the properly intersecting intervals that are currently in A, our algorithm also (implicitly) considers properly intersecting intervals that belonged to A in the past and were preempted since. This seemingly small difference turns out to be crucial as it facilitates our algorithm to use much less memory, thus giving rise to an interesting phenomena: by remembering extra information (i.e., intersecting intervals that belonged to A in the past and are not in A anymore), we actually end up using less memory.

The Algorithm. Consider a stream $S = (I_1, \ldots, I_n)$ of intervals on the real line. It will be convenient to assume that all endpoints are distinct, i.e., $\{\text{left}(I), \text{right}(I)\} \cap \{\text{left}(J), \text{right}(J)\} = \emptyset$ for every two intervals $I, J \in S$. Unless stated otherwise, we will also assume that the intervals mentioned in this section are closed on both endpoints. These two assumptions are lifted in [9].

Our algorithm, denoted $\texttt{Alg}$, maintains a collection $A \subseteq S$ of *actual* intervals and a collection V of *virtual* intervals, where each virtual interval is realized by endpoints of intervals in S. That is, the virtual interval $I \in V$ satisfies $\{\text{left}(I), \text{right}(I)\} \subseteq \{\text{left}(J), \text{right}(J) \mid J \in S\}$. The algorithm initially sets

$A, V \leftarrow \emptyset$. Then, upon arrival of a new interval $I \in S$, `Alg` proceeds according to the policy presented in Algorithm 1.

Algorithm 1. The policy of `Alg` upon arrival of an interval $I \in S$

```
 1: if $\exists J \in A \cup V$ s.t. $J \subseteq I$ then
 2:     reject $I$ and halt
 3: $A \leftarrow A \cup \{I\}$
 4: for all $J \in A$ s.t. $J \supseteq I$ do
 5:     $A \leftarrow A - \{J\}$
 6: for all $J \in V$ s.t. $J \supseteq I$ do
 7:     $V \leftarrow V - \{J\}$
 8: for $p \in \{\text{left}(I), \text{right}(I)\}$ do
 9:     if $\exists J \in V$ s.t. $p \in J$ then
10:         $V \leftarrow V - \{J\} \cup \{I \cap J\}$
11:     else if $\exists J \in A$ s.t. $p \in J$ then
12:         $V \leftarrow V \cup \{I \cap J\}$
13: for all $J \in A$ and $K \in V$ do
14:     if $\text{left}(J) < \text{left}(K) < \text{right}(K) < \text{right}(J)$ then
15:         $A \leftarrow A - \{J\}$
```

Analysis (sketch). We provide here a sketch of the analysis; the detailed version is deferred to [9]. Throughout, we let $1 \leq t \leq n$ denote the time at which `Alg` completed processing interval $I_t \in S$; time $t = 0$ denotes the beginning of the execution. We refer to the period between time $t-1$ and time t as *round* t. The stream prefix $(S_1, \ldots, S_t)$ is denoted by S_t. The collections A and V at time t are denoted by A_t and V_t, respectively, although, when t is clear from the context, we may omit the subscript.

Lemma 1 lies at the core of our analysis: it states that each interval in S leaves some trace in either A or V. This will be employed later on to argue that `Alg`(S) is not much smaller than `Opt`(S).

Lemma 1. *For every interval $I_t \in S$ and for every time $t' \geq t$, there exists some interval $\rho \in A_{t'} \cup V_{t'}$ such that $\rho \subseteq I_t$.*

Lemma 2 — the main lemma regarding the updating phase in lines 8–12 and the resulting structure of the interval collections A and V — states seven invariants maintained by our algorithm; these invariants are proved simultaneously by induction on t, essentially by straightforward analysis of the policy presented in Algorithm 1.

Lemma 2. *For any round $1 \leq t \leq n$, the updating phase satisfies the following two properties:*
(P1) If ρ is added to V in round t, then $\rho \in V_t$.
(P2) If ρ and σ are added to V in round t, then $\rho \cap \sigma = \emptyset$.
Moreover, for any time $0 \leq t \leq n$, the interval collections A and V satisfy the following five properties:

(P3) For every $\rho \in A$ and $\sigma \in V$, if $\rho \cap \sigma \neq \emptyset$, then $\sigma \subset \rho$ with a common endpoint.
(P4) For every $\rho, \sigma \in A$, if $\rho \cap \sigma \neq \emptyset$, then $\rho \cap \sigma \in V$.
(P5) Every point $p \in \mathbb{R}$ is contained in at most 1 virtual interval.
(P6) Every point $p \in \mathbb{R}$ is contained in at most 2 actual intervals.
(P7) There do not exist two actual intervals $\rho, \sigma \in A$ such that $\rho \subseteq \sigma$.

We employ Lemma 2 in order to understand the structure of the components of A and their relations with the intervals in V. To that end, fix some time t and consider an arbitrary component C formed as the union of the actual intervals $\rho_1, \ldots, \rho_k \in A_t$. We denote the leftmost and rightmost points in (the segment) C by $\text{left}(C)$ and $\text{right}(C)$, respectively. Assume without loss of generality that $\text{left}(\rho_i) < \text{left}(\rho_{i+1})$ for every $1 \leq i \leq k-1$. Lemma 2(P6) and (P7) then guarantees that

$$\text{left}(\rho_{i-1}) < \text{left}(\rho_i) < \text{right}(\rho_{i-1}) < \text{left}(\rho_{i+1}) < \text{right}(\rho_i) < \text{right}(\rho_{i+1})$$

for every $2 \leq i \leq k-1$. By Lemma 2(P4), we conclude that $\rho_i \cap \rho_{i+1} \in V_t$ for every $1 \leq i \leq k-1$, while Lemma 2(P3) implies that the segment $[\text{left}(\rho_2), \text{right}(\rho_{k-1})]$ does not intersect with any other virtual interval in V_t. The segment C possibly contains two more virtual intervals at time t: an interval $\sigma_\ell \subseteq [\text{left}(\rho_1), \text{left}(\rho_2))$ and an interval $\sigma_r \subseteq (\text{right}(\rho_{k-1}), \text{right}(\rho_k)]$, but then Lemma 2(P3) guarantees that $\text{left}(\sigma_\ell) = \text{left}(\rho_1) = \text{left}(C)$ and $\text{right}(\sigma_r) = \text{right}(\rho_k) = \text{right}(C)$. An illustration of a component is provided in Fig. 1. There may also exist virtual intervals in between the components of A, but Lemma 3, to be stated soon, essentially shows that their number and structure are fairly limited.

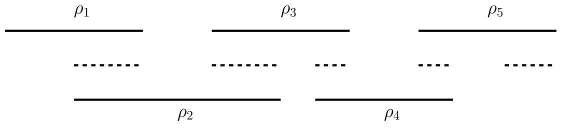

Fig. 1. A component C of A. The solid lines depict the actual interval ρ_i, $i = 1, \ldots, 5$; the dashed lines depict the virtual intervals contained in C.

Let Ψ_t denote the collection of the components of A_t and consider two adjacent components $C_\ell, C_r \in \Psi_t$, where C_ℓ is to the left of C_r. We say that the pair (C_ℓ, C_r) is *solid* at time t if at most one virtual interval in V_t intersects with the segment $[\text{right}(C_\ell), \text{left}(C_r)]$. Lemma 3 states that the pair (C_ℓ, C_r) is always solid.

Lemma 3. *At every time $0 \leq t \leq n$, all pairs of adjacent components in Ψ_t are solid. Moreover, no virtual interval intersects with the segment $(-\infty, \text{left}(C_\ell)]$ nor with the segment $[\text{right}(C_r), +\infty)$, where C_ℓ and C_r are the leftmost and rightmost components in Ψ_t, respectively.*

Lemma 4 is established by combining Lemma 1 and Lemma 3 with a careful accounting of the virtual intervals.

Lemma 4. $|\texttt{Alg}(S_t)| \geq |\texttt{Opt}(S_t)|/2$ *at every time* $0 \leq t \leq n$.

Corollary 1. $|\texttt{Alg}(S)| \geq |\texttt{Opt}(S)|/2$.

It remains to bound the space of our algorithm, showing that it is linear in the length of the bit string representing $\texttt{Alg}(S)$. At each time t, the space of $\texttt{Alg}$ is linear in the length of the bit strings representing A_t and V_t. As $\texttt{Opt}(S_t)/2 \leq \texttt{Alg}(S_t) \leq \texttt{Opt}(S_t)$ for every $0 \leq t \leq n$, and since $\texttt{Opt}(S_t)$ is non-decreasing with t, it is sufficient to show that $|A_t| + |V_t| = O(|\texttt{Alg}(S_t)|) = O(|\texttt{Opt}(A_t)|)$. By Lemma 2(P6), we know that the actual intervals in A_t can be 2-colored such that if two intervals belong to the same color class, then they do not intersect. Thus, $|A_t| \leq 2 \cdot |\texttt{Opt}(A_t)|$ at every time t. On the other hand, Lemma 3 implies that if we count the actual and virtual intervals by scanning the real line from left to right, then the number of virtual intervals never exceeds that of the actual intervals. Therefore, $|V_t| \leq |A_t|$ which concludes our analysis.

4 Lower Bound(s)

In this section we establish lower bounds on the approximation ratio of randomized streaming algorithms for the interval selection problem, establishing the following two theorems.

Theorem 1 (Lower bound for general intervals). *For every real* $\epsilon > 0$*, integers* $k_0, n_0 > 0$*, and subexponential (respectively, sublinear) function* $s : \mathbb{N} \to \mathbb{N}$*, there exist* $k_0 \leq k \leq c \cdot k_0$*, where* c *is a universal constant,* $n > n_0$*, and an interval stream* S *such that (1)* $|S| = n$*; (2)* $|\texttt{Opt}(S)| = k$*; and (3)* $\texttt{Alg}(S) < k(1/2 + \epsilon)$ *for any randomized interval selection streaming algorithm* $\texttt{Alg}$ *with space* $s(kb)$ *(resp., space* $s(nb)$*), where* b *is the length of the bit strings representing the endpoints of* S*.*

Theorem 2 (Lower bound for unit intervals). *For every real* $\epsilon > 0$*, integers* $k, n_0 > 0$*, and subexponential (respectively, sublinear) function* $s : \mathbb{N} \to \mathbb{N}$*, there exist* $n > n_0$*, and a unit interval stream* S *such that (1)* $|S| = n$*; (2)* $|\texttt{Opt}(S)| = k$*; and (3)* $\texttt{Alg}(S) < k(2/3 + \epsilon)$ *for any randomized proper interval selection streaming algorithm* $\texttt{Alg}$ *with space* $s(kb)$ *(resp., space* $s(nb)$*), where* b *is the length of the bit strings representing the endpoints of* S*.*

Our lower bounds are proved by designing a random interval stream S for which every deterministic algorithm performs badly on expectation; the assertion then follows by Yao's principle. (Our construction uses half-open intervals, but this can be easily altered.) Note that under the setting used by our lower bounds, the algorithm is required to output a collection $\mathcal{C}$ of disjoint intervals, and the quality of the solution is then determined to be the cardinality of $\mathcal{C} \cap S$. In other words, the algorithm is allowed to output non-existing intervals (that is, intervals that never arrived in the input), but it will not be credited for them. This, obviously, can only increase the power of the algorithm.

The (k, n)-Gadget. Fix some positive integer m whose role is to bound the space of the algorithm. Our lower bounds rely on the following framework, characterized by the parameters $k, n \in \mathbb{Z}_{>0}$, denoted a (k, n)-*gadget*. Consider an extensive form two-player zero-sum game played between the algorithm (MAX) and the adversary (MIN), depicted by a sequence of k *phases*. Informally, in each phase t, the adversary chooses a permutation $\pi_t \in P_n$, where P_n is the collection of all permutations on n elements, and an index $i_t \in [n]$. The algorithm observes π_t (but not i_t) and produces a *memory image* M_t, i.e., a bit string of length m. The index i_t is handed to the algorithm after the memory image is produced. At the end of the last phase the algorithm tries to *recover* $\pi_t(i_t)$ for $t = 1, \ldots, k$: it outputs some $i_t^* \in [n]$ based on the memory image M_t, index i_t, and all other memory images and indices. For each t such that $i_t^* = \pi_t(i_t)$, the algorithm scores a (positive) point.

More formally, the adversarial strategy is depicted by the choices of the permutations π_t and the indices i_t for $t = 1, \ldots, k$. We commit the adversary to make those choices uniformly at random (so, the adversary reveals its mixed strategy), namely, $\pi_t \in_r P_n$ and $i_t \in_r [n]$ for every t, where all the random choices are independent. The strategy of the algorithm is depicted by the function sequences $\{f_t\}_{t=1}^k$ and $\{g_t\}_{t=1}^k$, where $f_t : P_n \times (\{0,1\}^m \times [n])^{t-1} \to \{0,1\}^m$ and $g_t : \{0,1\}^m \times [n] \times (\{0,1\}^m \times [n])^{k-1} \to [n]$. Let Γ_0 be the empty string and recursively define[1] $\Gamma_t = \Gamma_{t-1} \circ f_t(\pi_t, \Gamma_{t-1}) \circ i_t$. The payoff of the algorithm is the number of indices t, $1 \le t \le k$, such that

$$g_t\left(f_t(\pi_t, \Gamma_{t-1}), i_t, \{f_{t'}(\pi_{t'}, \Gamma_{t'-1}), i_{t'}\}_{t' \neq t}\right) = \pi_t(i_t) .$$

In the language of the aforementioned informal description, the role of the function f_t is to produce the memory image M_t based on the permutation π_t and all previous memory images and indices (whose concatenation is given by Γ_{t-1}). The role of the function g_t is to recover $\pi_t(i_t)$ based on the memory image M_t, index i_t, and all other memory images and indices.

Note that the memory images $M_{t'}$ and indices $i_{t'}$, $t' \neq t$, do not contain any information on the permutation π_t on top of that contained in M_t. In particular, the entropy in $\pi_t(i_t)$ given M_t, i_t, and $\{M_{t'}, i_{t'}\}_{t' \neq t}$ is equal to the entropy in $\pi_t(i_t)$ given M_t and i_t. Therefore, it will be convenient to decompose the domain of the function $g_t : \{0,1\}^m \times [n] \times (\{0,1\}^m \times [n])^{k-1} \to [n]$ so that the $(\{0,1\}^m \times [n])^{k-1}$-part determines which function $\hat{g}_t : \{0,1\}^m \times [n] \to [n]$ is chosen, and then this function $\hat{g}_t$ is used to produce i_t^* based on M_t and i_t. Similarly, we decompose the domain of the function $f_t : P_n \times (\{0,1\}^m \times [n])^{t-1} \to \{0,1\}^m$ so that the $(\{0,1\}^m \times [n])^{t-1}$-part determines which function $\hat{f}_t : P_n \to \{0,1\}^m$ is chosen, and then this function $\hat{f}_t$ is used to produce M_t based on π_t.

We now turn to bound the expected payoff of the algorithm as a function of k, m, and n. The key ingredient in this context is the following lemma, which is essentially a well known fact in slightly different settings; a proof is provided in [9] for completeness.

[1] We use the notation $u \circ v$ to denote the concatenation of the string u to string v.

Lemma 5. *For every real $\alpha > 0$ and integer $n_0 > 0$, there exists an integer $n > n_0$ such that for every two functions $\hat{f} : P_n \to \{0,1\}^m$ and $\hat{g} : \{0,1\}^m \times [n] \to [n]$, where $m = \alpha n \log n$, we have $\mathbb{P}_{\pi \in_r P_n, i \in_r [n]}(\hat{g}(\hat{f}(\pi), i) = \pi(i)) < 2\alpha$.*

Corollary 2. *For every real $\alpha > 0$ and integers $k, n_0 > 0$, there exists an integer $n > n_0$ such that if $m \leq \alpha n \log n$, then the expected payoff of the algorithm (MAX) player in a (k, n)-gadget is smaller than $2\alpha k$.*

The (n, π)-Stack. We now turn to implement a (k, n)-gadget via a carefully designed interval stream. As a first step, we introduce the (n, π)-*stack* construction. Given an integer $n > 0$ and a permutation $\pi \in P_n$, an (n, π)-stack deployed in the segment $[x, y)$, $x < y$, is a collection of n intervals $J_1, \ldots, J_n$ satisfying: (1) all intervals J_i are half open; (2) all intervals J_i have the same length $\text{right}(J_i) - \text{left}(J_i) = \lambda n$, where $\lambda = \frac{y-x}{2n-1/2}$; and (3) $\text{left}(J_i) = x + \lambda(i-1) + \epsilon\pi(i)$ for every $i \in [n]$, where $\epsilon = \lambda/(2n)$. Note that this deployment ensures that $\text{left}(J_n) < \text{right}(J_1)$, hence the half open segment $[\text{left}(J_n), \text{right}(J_1))$ is contained in J_i for every $i \in [n]$. Moreover, the union of the intervals in the stack does not necessarily cover the whole segment $[x, y)$; it is always contained in $[x, y)$, though. The structure of an (n, π)-stack is illustrated in Fig. 2.

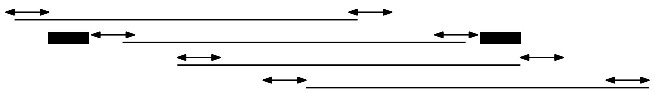

Fig. 2. The relative locations of the intervals in an (n, π)-stack for $n = 4$. The left and right endpoints of interval J_i are located in the segments depicted by the bidirectional arrows whose length is $\lambda/2$. The exact location within this segment is determined by $\pi(i)$. In the construction of the 2-lower bound for general intervals, the bold rectangles correspond to the segments in which the stacks (or auxiliary intervals) identified with the left and right children of the current node are deployed assuming that the good interval is interval J_2 (these segments do not intersect with the segments corresponding to the bidirectional arrows).

The (k, n)-gadget is implemented by introducing k stacks, each corresponding to one phase, and some *auxiliary* intervals; the stack corresponding to phase t is referred to as stack t. The permutation π used in the construction of stack t is π_t. The index i_t will dictate the choice of one *good* interval out of the n intervals in that stack. What exactly makes this interval good will be clarified soon; informally, the algorithm has no incentive to output an interval in a stack unless this interval is good.

The k stacks are used both by the construction of the 2-lower bound for general interval streams and by that of the (3/2)-lower bound for unit intervals. The difference between the two constructions lies in the manner in which these stacks are deployed in the real line, and in the addition of the auxiliary intervals. The details of the 2-lower bound are provided here; those of the (3/2)-lower bound are deferred to [9].

A 2-Lower Bound for General Intervals. The interval stream that realizes the (k, n)-gadget for the 2-lower bound for general intervals is constructed as follows. Assume that $k = 2^\kappa - 1$ for some positive integer κ and consider a perfect binary tree T of depth κ. The k stacks are identified with the internal nodes of T so that stack t precedes stack $t + 1$ in a pre-order traversal of T. (In other words, if stack t is identified with node u and stack t' is identified with a child of u, then $t < t'$.) In addition to the intervals in the stacks, we also introduce $2^\kappa = k + 1$ auxiliary intervals which are identified with the leaves of T; these auxiliary intervals arrive last in the stream. We say that an interval J is *assigned* to node $u \in T$ if J belongs to the stack identified with u or if u is a leaf and J is the auxiliary interval identified with it.

The deployment of the stacks and the auxiliary intervals in $\mathbb{R}$ is performed as follows. Stack 1 (identified with T's root) is deployed in $[0, 1)$. Given the deployment of stack t identified with internal node $u \in T$ in the segment $[x, y)$, we deploy the stacks identified with the left and right children of u in the segments $\sigma_\ell = [x + \lambda(i_t - 3/2), x + \lambda(i_t - 1))$ and $\sigma_r = [x + \lambda(i_t + n - 1/2), x + \lambda(i_t + n))$, respectively, where recall that $\lambda = \frac{y-x}{2n-1/2}$. If the children of u are leaves in T, then we deploy auxiliary intervals in those two segments instead of stacks, that is, one auxiliary interval in σ_ℓ and one in σ_r. Refer to Fig. 2 for an illustration.

The key observation regarding the choice of σ_ℓ and σ_r is that

$$\begin{aligned}&\text{left}(J_{i_t-1}) \leq \text{left}(\sigma_\ell) < \text{right}(\sigma_\ell) \leq \text{left}(J_{i_t}) \quad \text{and}\\&\text{right}(J_{i_t}) \leq \text{left}(\sigma_r) < \text{right}(\sigma_r) \leq \text{right}(J_{i_t+1})\,.\end{aligned}$$

This implies that: (1) the good interval in the stack identified with node $u \in T$ does not intersect with any interval assigned to a descendant of u in T; and (2) a non-good interval in the stack identified with node $u \in T$ contains every interval assigned to a descendant of either the left child of u or the right child of u in T.

The best response of the algorithm would clearly include all the auxiliary intervals in the output, hence it can include an interval J_i of stack t in the output only if it is the good interval of that stack, namely, $i = i_t$. For that purpose, the algorithm has to recover the exact locations of the endpoints of J_{i_t} that implicitly encode $\pi_t(i_t)$. Observing that the endpoints in this construction can be represented by bit strings of length $\log(n) \cdot \log(k)$, Theorem 1 follows by Corollary 2.

References

1. Adler, R., Azar, Y.: Beating the logarithmic lower bound: Randomized preemptive disjoint paths and call control algorithms. J. Scheduling 6(2), 113–129 (2003)
2. Agarwal, P.K., Sharathkumar, R.: Streaming algorithms for extent problems in high dimensions. In: SODA 2010, pp. 1481–1489 (2010)
3. Ahn, K.J., Guha, S.: Graph Sparsification in the Semi-Streaming Model. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 328–338. Springer, Heidelberg (2009)
4. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. 58(1), 137–147 (1999)

5. Awerbuch, B., Bartal, Y., Fiat, A., Rosén, A.: Competitive non-preemptive call control. In: SODA 1994, pp. 312–320 (1994)
6. Bachmann, U.T., Halldórsson, M.M., Shachnai, H.: Online Selection of Intervals and t-Intervals. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 383–394. Springer, Heidelberg (2010)
7. Canetti, R., Irani, S.: Bounding the power of preemption in randomized scheduling. SIAM J. Comput. 27(4), 993–1015 (1998)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press and McGraw-Hill (2009)
9. Emek, Y., Halldórsson, M., Rosén, A.: Space-constrained interval selection (2012), `http://arxiv.org/abs/1202.4326`
10. Epstein, L., Levin, A.: Improved randomized results for the interval selection problem. Theor. Comput. Sci. 411(34-36), 3129–3135 (2010)
11. Epstein, L., Levin, A., Mestre, J., Segev, D.: Improved approximation guarantees for weighted matching in the semi-streaming model. In: STACS 2010, pp. 347–358 (2010)
12. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. Theor. Comput. Sci. 348, 207–216 (2005)
13. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the data-stream model. SIAM J. Comput. 38(5), 1709–1727 (2008)
14. Fung, S.P.Y., Poon, C.K., Zheng, F.: Improved Randomized Online Scheduling of Unit Length Intervals and Jobs. In: Bampis, E., Skutella, M. (eds.) WAOA 2008. LNCS, vol. 5426, pp. 53–66. Springer, Heidelberg (2009)
15. Gavril, F.: Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. SIAM J. Comput. 1(2), 180–187 (1972)
16. Halldórsson, B.V., Halldórsson, M.M., Losievskaja, E., Szegedy, M.: Streaming Algorithms for Independent Sets. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 641–652. Springer, Heidelberg (2010)
17. Henzinger, M.R., Raghavan, P., Rajagopalan, S.: Computing on data streams. In: AMS-DIMACS Series. Special Issue on Computing on Very Large Datasets (1998)
18. Kleinberg, J., Tardos, E.: Algorithm Design. Addison-Wesley (2005)
19. Lipton, R.J., Tomkins, A.: Online interval scheduling. In: SODA 1994, pp. 302–311 (1994)
20. McGregor, A.: Finding Graph Matchings in Data Streams. In: Chekuri, C., Jansen, K., Rolim, J.D.P., Trevisan, L. (eds.) APPROX 2005 and RANDOM 2005. LNCS, vol. 3624, pp. 170–181. Springer, Heidelberg (2005)
21. Miyazawa, H., Erlebach, T.: An improved randomized on-line algorithm for a weighted interval selection problem. J. of Scheduling 7(4), 293–311 (2004)
22. Munro, J.I., Paterson, M.: Selection and sorting with limited storage. Theor. Comput. Sci. 12, 315–323 (1980)
23. Muthukrishnan, S.: Data streams: Algorithms and applications. Foundations and Trends in Theoretical Computer Science 1(2) (2005)
24. Woeginger, G.J.: On-line scheduling of jobs with fixed start and end times. Theor. Comput. Sci. 130(1), 5–16 (1994)