Online Interval Scheduling

Richard J. Lipton[*]
rjl@cs.princeton.edu

Andrew Tomkins[†]
andrewt@cs.cmu.edu

Abstract

We introduce the *online interval scheduling problem*, in which a set of intervals of the positive real line is presented to a scheduling algorithm in order of start time. Upon seeing each interval, the algorithm must decide whether or not to "schedule" it. Overlapping intervals may not be scheduled together. We give a strongly 2-competitive algorithm for the case in which intervals must be one of two lengths, either length 1 or length $k \gg 1$. For the general case in which intervals may have arbitrary lengths, Δ, the ratio of longest to shortest interval, is the important parameter. We give an algorithm with competitive factor $O((\log \Delta)^{1+\epsilon})$, and show that no $O(\log \Delta)$-competitive algorithm can exist. Our algorithm need not know the ratio Δ in advance.

1 Introduction

In the field of on-line scheduling, an algorithm must typically schedule a number of jobs, or tasks, without knowing how long each task will take to complete (*e.g.*, [2]). Recently, however, a new variety of task scheduling problems has been introduced in the domain of *Continuous Media* [5]. Here, a client requests a resource for a certain fixed interval of time, and the admission control algorithm for the resource must decide whether or not to service the request. This differs from more traditional scheduling problems in two ways. First, scheduling an interval represents a commitment to the client so no pre-emption is allowed; and second, requests are made for fixed start and end points, so cannot be scheduled either early or late.

We present algorithms and lower bounds for this problem using the competitive analysis of [7]. The sequence of requests is selected entirely in advance by an adversary who may choose any fixed sequence, but may not create the sequence incrementally based on randomized decisions of the algorithm. Requests are presented to the algorithm in order of start time, and the algorithm must decide whether to schedule each request before receiving the next one. The goal is to keep the resource in use as much of the time as possible.

This model has been studied before in [6], in which it was shown that no deterministic algorithm can achieve good performance. We study randomized algorithms, and show that if all intervals have length either 1 or k for some fixed $k \gg 1$ then there exists a strongly 2-competitive solution. In the case that the intervals have arbitrary lengths, and the ratio of longest to shortest intervals in a particular request sequence is Δ, we give algorithms with competitive factors $O((\log \Delta)^{1+\epsilon})$, for any $\epsilon > 0$. We also give a lower bound of $O(\log \Delta)$ on the competitive factor of any such algorithm. Our algorithms need not know Δ in advance.

Similar problems have been studied in the field of call control [3, 1, 4]. In the language of call control, our algorithms schedule calls on a two-node graph without pre-emption, in which the benefit of a call is its length. Our techniques are useful for designing randomized algorithms if the ratio of longest call to shortest call is not known in advance. Our work may be seen as an extension of deterministic algorithms given in [4] for line graphs. [1] also give randomized

[*]Princeton University, Princeton, NJ. Supported by NSF Grant CCR-9304718.
[†]Carnegie Mellon University, Pittsburgh, PA

call control algorithms; they consider trees, and advance a general paradigm called "Classify and Randomly Select" which is similar in nature to our approach.

The paper is organized as follows. First, we present some definitions, and a more formal description of the model. Next we give algorithms for the case in which intervals have only two distinct lengths. We then introduce a new problem called the *online marriage problem* and give a solution to it, which we use to develop a scheduling algorithm for request sequences with arbitrary length intervals. Next, we give a lower bound for the arbitrary-length problem. Finally, we conclude with some open problems.

1.1 Definitions

An *interval* I is a pair of positive real numbers $\langle \text{start}(I), \text{length}(I)\rangle$. We will sometimes refer to length(I) as $|I|$.

For an interval I and a positive real number r we say $r \in I$ to mean $\text{start}(I) \leq r \leq \text{start}(I) + \text{length}(I)$

A *Problem Instance* is a finite set S of intervals to be scheduled. We assume for simplicity that no two intervals in the set share a start point.

We say $\sigma \subset S$ is a *feasible schedule* if no two intervals of σ overlap.

For a feasible schedule $\sigma \subset S$ we define the *Weight* of σ to be

$$|\sigma| = \sum_{I \in \sigma} \text{length}(I)$$

Let σ_* be some maximum weight feasible schedule of S.

Let $F(S)$ be the collection of all feasible schedules of S.

A *randomized scheduling algorithm* A takes a problem instance S and returns A(S), a feasible schedule of S. We treat A(S) as a random variable over F(S).

The weight of algorithm A on S is the expected weight of the feasible schedule returned by A:

$$W(A,S) = E[W(A(S))]$$

Thus, W(A,S) represents the expected amount of resource utilization when using algorithm A on problem instance S.

A randomized scheduling algorithm A is *online* if the chance of A including an interval I in its feasible schedule depends only on intervals with start times before start(I).

We say that A is *c-competitive* if

$$\forall S : c \cdot W(A,S) \geq |\sigma_*|$$

Algorithm A is *strongly c-competitive* if A is c-competitive, and there is no b-competitive algorithm with $b < c$.

We have defined $W(A,S)$ to be $E[W(A(S))]$. We now define the weight of A on a particular interval $I \in S$ as follows:

$$W_S(A,I) = |I| \cdot \Pr[I \in A(S)]$$

Note that $\Pr[I \in A(S)]$ will depend on S, so $W_S(A,I)$ makes sense only in the context of an entire problem instance S, of which I is a part; hence the subscript W_S.

We now show that the weight of A on S can be computed by adding the weight of A on each interval $I \in S$.

Recall that $F(S)$ is the set of all feasible schedules of S. Recall also that randomized algorithm A returns an element of $F(S)$ drawn randomly from a distribution induced by A's behavior.

LEMMA 1.1. $W(A,S) = \sum_{I \in S} W_S(A,I)$

Proof:

Using linearity of expectation:

$$\begin{aligned} W(A,S) &= E[W(A(S))] \\ &= \sum_{\sigma \in F(S)} (|\sigma| \cdot \Pr[A(S) = \sigma]) \\ &= \sum_{I \in S} (|\, I \,| \cdot \Pr[I \in A(S)]) \\ &= \sum_{I \in S} W_S(A,I) \end{aligned}$$

□

Finally, now that the lemma has shown how to compute the weight of A on a set in terms of

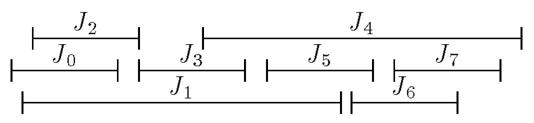


Figure 1: A 2-length problem instance

the individual intervals of the set, we define the weight of A on a subset of S: if $T \subset S$ then

$$W_S(A,T) = \sum_{I \in T} W_S(A,I)$$

Remember that both $W_S(A,I)$ and $W_S(A,T)$ are defined only for a fixed problem instance S.

2 The Virtual Algorithm

Let us begin by restricting our attention to problem instances with only two types of requests, small intervals of length 1 and large intervals of length $k \gg 1$. We say that a problem instance S consisting of only 1-intervals and k-intervals is a *2-length problem instance.* A sample problem instance in this model is shown in Figure 1.

We now present a strongly 2-competitive algorithm called the *Virtual Algorithm* (VA).

We define the algorithm by describing its behavior when presented with 1-intervals and k-intervals. If the resource is in use, do nothing. Otherwise:

- For a 1-interval, flip a fair coin

  Heads: Take the interval

  Tails: Virtually take the interval, but *do no work.* Take no short interval for the next 1 unit of time.

- For a k-interval, take whenever possible.

We wish to show that the Virtual Algorithm is strongly 2-competitive. We show first that the algorithm is 2-competitive, then that no algorithm can have better performance.

2.1 The Virtual Algorithm is 2-competitive

Define a *bucket* to be a set of intervals or parts of intervals. Once again, let σ_* be any maximum weight feasible schedule. We associate a bucket with each interval $J \in \sigma_*$, which we refer to as bucket(J). Initially, all buckets are empty. We have defined the notation $W_S(A,T)$ to mean the weight of A on $T \subset S$. We extend that definition to buckets, which can contain parts of intervals, in the obvious way: For any subinterval I' in a bucket, let the interval I containing I' be called Parent(I'). Then

$$W_S(A, \text{Bucket}) = \sum_{I' \in \text{Bucket}} |I'| \cdot \Pr[\text{Parent}(I') \in A(S)]$$

An interval I *blocks* another interval J if I overlaps the start of J.

The proof proceeds as follows:

1. Place certain intervals, or parts of intervals, from S into buckets.

2. Show that no part of any interval is assigned to more than one bucket.

3. Show that the expected weight of the Virtual Algorithm on the bucket of each $J \in \sigma_*$ is at least half the length of J.

Lemma 2.1. *If intervals of S can be assigned to buckets such that:*

Condition 1: *no part of any interval is placed in more than one bucket, and*

Condition 2: $\forall J \in \sigma_* : W_S(A, \mathit{bucket}(J)) \geq \frac{|J|}{2}$

then A is 2-competitive.

Proof:

$$\begin{aligned} W(A,S) &= \sum_{I \in S} W_S(A,I) \\ &\geq \sum_{J \in \sigma_*} W_S(A, \text{bucket}(J)) \\ &\quad \text{(from Condition 1, above)} \\ &\geq \sum_{J \in \sigma_*} \frac{|J|}{2} \\ &\quad \text{(from Condition 2, above)} \\ &\geq \frac{|\sigma_*|}{2} \end{aligned}$$

□

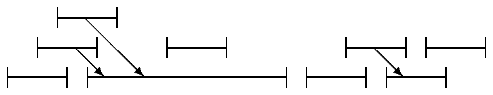


Figure 2: Assigning 1-intervals to buckets

We now show how to assign intervals to buckets. First, each element of σ_* itself is assigned to its own bucket. We now address individually the assignments of 1-intervals $I \notin \sigma_*$ and k-intervals $B \notin \sigma_*$ to buckets.

1-intervals

First, note that any 1-interval can block at most one interval of σ_*. So for every 1-interval I of S, place I in the bucket of the interval of σ_* that it blocks, if any. Otherwise, don't assign it anywhere. In Figure 2, σ_* is shown along the bottom row. The other elements of S are in the upper rows. All elements of σ_* are in their own buckets. Each other element of S has an arrow showing which bucket it is assigned to, if any.

k-intervals

We say that $J \in \sigma_*$ is the *terminal interval* of $I \in S$ if J contains the end point of I. Note that an interval will have at most one terminal interval because otherwise two elements of σ_* would overlap. Some intervals will not have a terminal interval. Also, we say a k-interval B *covers* an interval I if the start and end of I are both contained in B

We now show how a k-interval $B \in S$ is assigned to buckets. Say T is the terminal interval of B. We will assign at least half the length of B to the bucket of T. If T does not exist, the parts of B that would have been assigned to T are not assigned to any bucket.

If B covers an interval J of σ_* we do the following: since J must have length 1, there is a corresponding 1 unit of length of B which covers J. Assign the first half of this length to J, and the second half of it to T.

Assign all unassigned subintervals of B to T.

In Figure 3, we show how a single k-interval B is shared among the elements of σ_*. The Terminal Interval of B is labelled with a T. All subintervals of B which are *not* assigned to T are marked with dashed boxes, and have arrows showing the

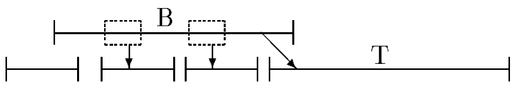


Figure 3: Assigning k-intervals to Buckets

element of σ_* to which they are assigned.

We must show that our buckets obey the two conditions stated in Lemma 2.1. The first condition is that no part of an interval can be assigned to more than one bucket. This follows immediately from the way we assign intervals to buckets:

- 1-intervals can never block more than one interval of σ_*.
- Any part of a k-interval that is assigned to a bucket will be assigned either to the terminal interval or to the overlapping interval of σ_*.

Next we must show the second and final condition required for Lemma 2.1. We restate and prove this condition as Lemma 2.2. Recall that VA is the Virtual Algorithm.

Lemma 2.2. *For all intervals $J \in \sigma_*$,*

$$W_S(VA, bucket(J)) \geq \frac{|J|}{2}$$

Proof: We first show Lemma 2.2 for buckets associated with 1-intervals of σ_*.

Let $I \in \sigma_*$ be any 1-interval of σ_*. We partition $F(S)$, the set of feasible schedules of S, into four disjoint groups of schedules based on the treatment of I in the schedule:

1. The first group comprises those schedules of $F(S)$ with the property that the resource is available (neither taken nor virtually taken) immediately prior to the start of some 1-interval I' that blocks interval I. In this case, $I' \in$ bucket(I), so the expected weight of VA on the entire bucket of I is at least 1/2, since I' will be scheduled with probability 1/2. A schedule from this group will be selected with probability p_1, for some unknown value of p_1.

2. The second group comprises those schedules of $F(S)$ in which the algorithm schedules a k-interval B that completely covers I. A length-1/2 subinterval of B is assigned to I's bucket, according to the rules for assigning

k-intervals. Thus, the weight from bucket(I) for any schedule from this group is at least 1/2. A schedule from this group will be selected with some unknown probability p_2.

3. The third group comprises those schedules of $F(S)$ in which a k-interval is scheduled for which I is the terminal interval, ie, the k-interval ends in I. The weight of the bucket of I for a schedule in this group is at least $k/2$, since at least half the length of the k-interval will be assigned to bucket(I) by the rules for assigning k-intervals. A schedule from this group will be selected with unknown probability p_3.

4. The fourth group comprises those schedules of $F(S)$ in which the resource is neither taken nor virtually taken at the start of I. Note that this case is disjoint from case 1, because in case 1 the resource will be either taken or virtually taken just prior to the start of I. Since I is in its own bucket, the weight of VA on the bucket of I is at least 1/2. A schedule from this group will be selected with probability $p_4 = 1 - p_1 - p_2 - p_3$.

Thus, the gain of the algorithm on bucket(I) is at least

$$p_1 \cdot \frac{1}{2} + p_2 \cdot \frac{1}{2} + p_3 \cdot \frac{k}{2} + p_4 \cdot \frac{1}{2} \geq \frac{1}{2}$$

And since $|I| = 1/2$, we have $W_S(\text{VA}, I) \geq \frac{|I|}{2}$.

Thus the contribution from the bucket of any 1-interval is at least half the length of the 1-interval, so the lemma holds for 1-intervals $I \in \sigma_*$. We must now show that it holds for k-intervals.

Assume $B \in \sigma_*$ is any k-interval in σ_*. Again, we enumerate several cases:

1. The first group comprises those schedules of $F(S)$ in which the resource schedules a k-interval B' overlapping B. Since B is the terminal interval of B', at least half the length of B', $k/2$, belongs to the bucket of B. Thus, the weight of VA on the bucket of B is at least $k/2$. This case occurs with probability p_1.

   Note that if a schedule does not fall into this case then the resource *must* be free at some point between start(B) $-$ 1 and start(B).

2. The second group comprises those schedules of $F(S)$ with the property that the resource is free immediately preceding the start of some 1-interval I that blocks B. With probability 1/2, the 1-interval will be scheduled, and with probability 1/2, the 1-interval will be virtually scheduled, allowing us to schedule either B or another k-interval in bucket(B). The weight of the bucket of B in this case is thus $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot k \geq k/2$. The case occurs with probability p_2.

3. The third group comprises those schedules of $F(S)$ which leave the resource free (that is, neither taken nor virtually taken) immediately prior to the start of B, or if B is the terminal interval of some k-interval $B' \in S$, immediately prior to the start of B'. Since all of B and at least half the length of B' must be assigned to the bucket of B, the weight of VA on the bucket of B must be at least $\frac{k}{2}$. This occurs with probability $p_3 = 1 - p_1 - p_2$.

All schedules that could be chosen by the virtual algorithm fall into one of these three groups.

So the total weight of VA on the bucket of B is at least

$$p_1 \cdot \frac{k}{2} + p_2 \cdot \frac{k}{2} + p_3 \cdot \frac{k}{2} = \frac{k}{2}$$

Since $\mid B \mid = k$, we see that the weight of VA on the bucket of B is at least $\frac{k}{2}$.

Thus for any interval $J \in \sigma_*$, whether short or long, the weight of VA on the bucket of J is at least $\frac{|J|}{2}$. This completes the proof of Lemma 2.2. □

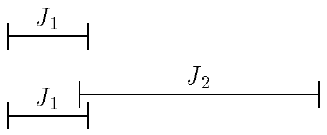
Lemma 2.2 shows us that we have fulfilled the conditions of Lemma 2.1, and so have shown that the Virtual Algorithm is 2-competitive. We now show that no algorithm can perform better.

2.2 No Algorithm can be better than 2-competitive

Consider the following 2 problem instances:

Instance 1:

Instance 2:



Let $p_1 = \Pr[\text{B takes } J_1]$.

If $p_1 < 1/2$ then B is not 2-competitive, so we can assume $p_1 \geq 1/2$. Thus $p_2 = \Pr[\text{B takes } J_2] \leq 1/2$, so B can be no better than 2-competitive on Instance 2. □

As one technical caveat, we are assuming that algorithms are not told the value of k in advance, so the probabilities p_1 and p_2 may not depend on k.

This completes the proof that no algorithm may have better competitive ratio than the Virtual Algorithm, which gives the following theorem:

Theorem 2.1. *The Virtual Algorithm is strongly 2-competitive.*

3 An $O(n^{1+\epsilon})$-competitive algorithm for the Online Marriage Problem

We now turn to a related problem whose solution will allow us to extend the Virtual Algorithm to intervals of arbitrary length.

3.1 The Online Marriage Problem

Imagine the following game played by a player and a host:

Host: Would you like \$2?	Player: No
Host: Would you like \$4?	Player: No
Host: Would you like \$8?	Player: No
Host: Would you like \$16?	Player: Yes

The host picks a number n in advance, which is kept secret from the player. The game is played until either the player says "yes" (in which case the player gets the money) or the player refuses n requests (in which case the player gets nothing).

More formally, say we are given a finite geometric sequence $[b, b^2, \ldots, b^n]$. We seek a randomized algorithm A which, when offered $\$b^k$, flips a coin and says "yes" or "no" with probability p_k.

For convenience, we shall define a sequence c_i similar to p_i as follows: $c_i = p_i \prod_{j=1}^{i-1}(1-p_j)$

Thus, p_i is the probability that b^i is taken, given that all previous offers are not taken. Likewise, c_i is simply the probability that b^i is taken.

We define the gain G of algorithm A, as a function of n, as follows:

$$G(A, n) = \sum_{i=1}^{n} c_i b^i$$

3.2 An Algorithm Based on the Zeta Function

Recall that the *zeta function* is defined as follows:

$$\zeta(d) = \sum_{n=1}^{\infty} \frac{1}{n^d}$$

and note that for $d > 1$, $\zeta(d)$ converges.

Consider the algorithm which flips coins such that $c_i = \frac{1}{i^{1+\epsilon}} \frac{1}{\zeta(1+\epsilon)}$. We shall call this algorithm the $(1+\epsilon)$-*algorithm*. Note that the c_i's represent a probability distribution over the b^i's and so should sum to 1:

$$\sum_{i=1}^{\infty} c_i = \frac{1}{\zeta(1+\epsilon)} \sum_{i=1}^{\infty} \frac{1}{i^{1+\epsilon}} = \zeta(1+\epsilon) \cdot \frac{1}{\zeta(1+\epsilon)} = 1$$

In order to present an algorithm that actually flips coins, we must back-solve from our c_i sequence for the p_i sequence. Inductively, given the values of $p_1 \ldots p_{i-1}$, and the value for c_i, we see that

$$p_i = \frac{c_i}{\prod_{j=1}^{i-1}(1-p_j)}$$

The following lemma provides a simple competitive bound on the $(1+\epsilon)$-Algorithm.

Lemma 3.1. *The $(1+\epsilon)$-Algorithm is $O(n^{1+\epsilon})$-competitive*

Proof: If the sequence has length n, then clearly the greatest possible gain of any algorithm is b^n. But the $(1+\epsilon)$ algorithm will take the b^n offer with probability c_n, so will have gain of at least $c_n b^n = O(\frac{b^n}{n^{1+\epsilon}})$.□

4 Scheduling Algorithms for Arbitrary Length Intervals

We now turn to the problem of scheduling requests from problem instances S that may contain arbitrary length intervals.

Recall that the intuition for the Virtual Algorithm was the following:

Virtual Algorithm Intuition: If we reject a short interval I, schedule no other short interval that begins during I.

We now introduce the *Marriage Algorithm*, and we take the following approach to extending to intervals of arbitrary lengths:

Marriage Algorithm Intuition: If we reject an interval I, schedule no other interval that begins during I unless it's twice as long as I.

If the resource is free and a request arrives for an interval I then with some probability we service the request; otherwise, we virtually service the request. This means that we leave the resource free, but until the end of I we never even consider scheduling an interval less than twice the length of I. Thus, all elements of a sequence of intervals can be considered for scheduling only if each is twice the length of the previous one; that is, if the sequence increases exponentially, as in the marriage problem of the previous section.

We now give a more formal presentation of the algorithm, using the sequence of probabilities $[p_i]$ defined in the previous section:

Define the active interval I_{active} to be the interval which was most recently either taken or virtually taken. Initially, this interval is the empty interval, which contains no points. The *depth* of the active interval is some positive integer calculated during the algorithm.

Given a new interval I, decide as follows:

1. If I starts outside I_{active} set depth to 1, set I_{active} to I, and take I with probability p_{depth}, otherwise virtually take I.
2. If I starts within I_{active}, and I_{active} is taken, do nothing.
3. If I starts within I_{active}, and I_{active} is virtually taken, and $|I| < 2|I_{\text{active}}|$ then do nothing.
4. If I starts within I_{active}, I_{active} is virtually taken, and $|I| \geq 2|I_{\text{active}}|$, then set I_{active} to I, increment depth, and take I with probability p_{depth}, otherwise virtually take I.

THEOREM 4.1. *If the ratio of longest to shortest intervals in S is Δ then the Marriage Algorithm is $O((\log \Delta)^{1+\epsilon})$-competitive.*

Proof: For simplicity in this proof, let us assume that all "free space" has been removed from S. That is, start times have been modified to remove any time periods with no pending requests, without changing lengths, relative start orderings, or overlaps. This operation clearly does not change the performance of the algorithm in any way.

For a set of intervals $S_i \subset S$, define $|S_i|$ to be the distance between the start point of the first interval and the end point of the last-ending interval of S_i.

From the beginning of S construct the longest possible set of overlapping intervals $I_1 \ldots I_m$ with the property that I_1 is the first interval in S and $\text{start}(I_j) < \text{start}(I_{j+1})$ and $2 * \text{length}(I_j) \leq \text{length}(I_{j+1})$. Call this set S_1. By "longest possible set," we mean that the construction maximizes $|S_1|$.

We now wish to begin the same process after the end of S_1 to create S_2. But we wish to eliminate "interference" from S_1. Recall that an interval I is given the opportunity to be scheduled with some probability unless either the network is already taken (I is blocked), or the network is virtually taken by an interval more than half as long (I is virtually blocked).

Some intervals will have start points overlapping S_1, but will extend beyond the end of S_1. We look for the first interval starting beyond the end of S_1 that could never be virtually blocked by any of these overlapping intervals. That is, we look for the first interval J starting outside S_1 such that if I overlaps $\text{start}(J)$ then either I begins outside S_1 or $2 * \text{length}(I) < \text{length}(J)$. Once we have located J, we use it as the first interval of S_2, using the same construction as for S_1. We continue this process until the end of S.

We break S into regions corresponding to the S_i's and analyze the performance of the marriage algorithm on each region. An interval is placed in the region of a particular S_i if its start point occurs after the end of all intervals in S_{i-1} but before the end of the last interval of S_i.

We now require the following Lemma to show that the Marriage Algorithm performs well on each S_i. We will then complete the Theorem by showing that the S_i's together are at least $1/7^{th}$ as long as σ_*.

LEMMA 4.1. *The weight of the Marriage Algorithm on the region of S_i is at least $\frac{c_{\log \Delta}}{4}|S_i|$.*

Proof:

Fix i. Let I be the last interval of S_i. Let σ be a feasible schedule. Knowing that the marriage algorithm generated σ and knowing the entire problem instance S allows us to determine the state of the algorithm at various points of σ. For instance, consider the start point of I. We can identify this point in σ. Imagine moving back in time along the schedule σ until we reach the endpoint of some interval in σ. Call this interval L.

Since we know that the resource was free immediately after L, we know the state of the marriage algorithm, so we can simulate the algorithm from the endpoint of L onwards. The interval $N \in S$ with the next start point would be presented to MA. A p_1-biased coin would be flipped. For the particular run of the algorithm that generated σ, we note that if $N \in \sigma$ then the coin must have come up heads; otherwise the coin must have come up tails. Whichever is the case, we can continue our simulation, determining the outcome of coin flips by checking to see if the interval in question appears in σ.

Eventually we will be presented with the start point of I. At this point, we end the simulation. We can look back through the simulation to determine when the resource was last free (neither taken nor virtually taken). Immediately after the last free point, MA would have flipped a p_1-biased coin for some interval J, and the resource would remain either taken or virtually taken until the start of I. We focus our attention on interval J, which we refer to as the *leading interval* of interval I. For some schedules, I will be its own leading interval.

So intuitively, the leading interval of I is the first interval for which a p_1-biased coin was flipped, as we proceed backwards from the start point of I.

We now break $F(S)$, the set of all feasible schedules of S, into groups that share the same leading interval of I. We shall call these groups *partitions*. So all schedules in a particular partition have the same leading interval.

Now, consider a fixed partition P with leading interval J. For a particular schedule $\sigma \in P$, MA might have flipped heads for J, in which case $J \in \sigma$, or might have flipped tails in which case $J \notin \sigma$. If MA flipped tails, J would have been virtually taken, and another interval might have had the opportunity to be scheduled.

For any possible outcome of the coin flips, there will be some schedule in P corresponding to that outcome. Consider the schedule which results if, when faced with J, MA had actually flipped tails. And from that point on, MA flipped tails for every single coin flip it made. Then one of two cases would occur:

1. MA would flip a coin for interval I.
2. MA, when presented with interval I, would not flip a coin because it had virtually scheduled some I' with length at least $\frac{|I|}{2}$.

If partition P lies in case 1, we would expect a random schedule from P to contain I with probability at least $c_{\log\Delta}$. If partition P lies in case 2, we would expect a random schedule from P to contain I' with probability at least $c_{\log\Delta}$. This is because $c_{\log\Delta}$ is a lower bound on the probability that MA, when starting at the leading interval of P, flips all tails until faced with I (or I'), and then flips heads.

Informally, each partition represents all possible outcomes of a particular online marriage game – the "offers" made to the player are the intervals for which coins would be flipped, beginning with the leading interval of the partition and continuing through either I or I'.

Since $2|I'| > |I|$, we expect to schedule at least $c_{\log\Delta}\frac{|I|}{2}$ from every partition. Intervals I and I' both lie within the region associated with S_i. This is true for I because $I \in S_i$, and for I' because I' virtually blocks an element of S_i (I), so could not also overlap an interval in S_{i-1}. Thus, the expected contribution from the region of S_i is at least $c_{\log\Delta}\frac{|I|}{2}$.

Since each interval of S_i must be at least twice as long as the previous one, and each one overlaps the previous one, and I is the last interval of S_i, it must be the case that $|I| \geq \frac{|S_i|}{2}$.

So in both cases above we expect to acquire at least

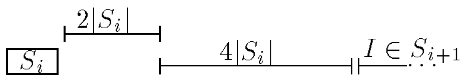
$$\frac{c_{\log\Delta}}{4}|S_i|$$

from a particular partition. This is true for all partitions. Since no schedule lies in two partitions, and the partitions cover $F(S)$, we can conclude that the expected weight of the algorithm on the region of S_i must be at least $\frac{c_{\log\Delta}}{4}|S_i|$. This completes the proof of Lemma 4.1. □

We now complete the proof of Theorem 4.1.

We wish to show that the distance from the end of S_i to the beginning of S_{i+1} cannot be more than $6|S_i|$. Note that no interval whose start point lies within S_i can be longer than $2|S_i|$, or it could be added to S_i. So any interval starting after S_i of length at least $4|S_i|$ would satisfy the conditions to be a valid start interval for S_{i+1}.

The following picture shows how to achieve the maximum distance between the S_i's. Note that if the first interval were any longer it would be part of S_i, and if the second were any longer it could never be virtually blocked, so it would be the start interval of S_{i+1}.



Thus, the greatest possible length between S_i and S_{i+1} is $2|S_i|$ for an interval extending beyond S_i plus $4|S_i|$ for an invalid start interval starting just before the end of this overlapping interval. So the space between S_i and S_{i+1} must be less than $6|S_i|$. Thus, the S_i's represent at least $1/7^{th}$ of the total distance of S. We now apply Lemma 4.1 to analyze the performance of the Marriage Algorithm (MA) on S. Recall that we defined the region associated with an S_i as the set of intervals whose start points lie after the end of all intervals in S_{i-1} but before the end of the last interval of S_i. Call this region $R(S_i)$.

$$\begin{aligned} W(\mathrm{MA}, S) &\geq \sum_i W_S(\mathrm{MA}, R(S_i)) \\ &\geq \sum_i \frac{c_{\log\Delta}}{4}|S_i| \\ &\geq \frac{c_{\log\Delta}}{4}\frac{|S|}{7} \end{aligned}$$

And since $|\sigma_*|$ is clearly bounded by $|S|$, this completes the proof of Theorem 4.1. □

5 A Lower Bound for the General Problem

Consider the set σ_n consisting of intervals $I_0, \ldots, I_n$ and constructed as follows:

$$\forall k : \mathrm{start}(I_k) = k/n, \ \mathrm{length}(I_k) = 2^k$$

The start times guarantee that all the intervals of σ_n overlap, and that they are ordered by length. The best possible choice is I_n.

For any algorithm as $n \to \infty$, we can think of the distribution $\{c_k\}$ over the positive integers, where $c_k = \Pr[\text{Algorithm takes } I_k]$.

Note that Δ, the ratio of shortest to longest elements in the sequence, is 2^n. So by the results of the previous section, we can achieve a competitive ratio of $O(n^{1+\epsilon})$. We now show that substantial improvement of this result is not possible.

THEOREM 5.1. *There does not exist an algorithm which is $O(n)$-competitive on σ_n.*

Proof:

Assume $\exists k \forall n \sum_{i=0}^{n} c_i 2^i \geq k\frac{2^n}{n}$ for some distribution c_i. That is, assume there is an $O(n)$-competitive algorithm.

We consider the sequence $[c_1, \ldots, c_n]$ for some fixed n, and we take partial sums of $2\log n$ consecutive elements:

$$\sum_{i=d(2\log n)}^{(d+1)(2\log n)} c_i$$

We shall show that any set of c_i's achieving the competitive bound will not form a valid distribution. We proceed by lower-bounding the sums described above. First, from the assumption:

$$\begin{aligned} \sum_{i=0}^{(d+1)(2\log n)} c_i 2^i &\geq \frac{k 2^{(d+1)(2\log n)}}{(d+1)(2\log n)} \\ &= k\frac{(n^2)^{d+1}}{(d+1)(2\log n)} \\ &= (n^2)^d \cdot \frac{kn^2}{(d+1)(2\log n)} \end{aligned}$$

and

$$\begin{aligned}\sum_{i=0}^{d(2\log n)} c_i 2^i &\leq 2^{d(2\log n)} \sum_{i=0}^{d(2\log n)} c_i \\ &\leq 2^{d(2\log n)} = (n^2)^d\end{aligned}$$

So there must be some k' such that

$$\sum_{i=d(2\log n)}^{(d+1)(2\log n)} c_i 2^i \geq \frac{k'(n^2)^{d+1}}{(d+1)(2\log n)}$$

and

$$2^{(d+1)(2\log n)} \sum_{i=d(2\log n)}^{(d+1)(2\log n)} c_i \geq \frac{k'(n^2)^{d+1}}{(d+1)(2\log n)}$$

so finally,

$$\sum_{i=d(2\log n)}^{(d+1)(2\log n)} c_i \geq \frac{k'}{(d+1)(2\log n)}$$

Note that k' can be chosen to be valid for all n and d beyond a certain n_0. For instance, $k' = k/2$ will have this property.

Summing all these partial series of c_i's over d ranging from 1 to $n/(2\log n)$, we get:

$$\begin{aligned}\sum_{i=2\log n}^{n} c_i &\geq k'\left(\frac{1}{4\log n} + \frac{1}{6\log n} + \cdots + \frac{1}{n}\right) \\ &\geq \frac{k'}{2\log n}\left(\sum_{i=1}^{n/2\log n} \frac{1}{i} - 1\right) \\ &\geq \frac{k'}{2\log n}\left(\log(\frac{n}{2\log n}) - 1\right) \\ &\geq \frac{k'}{2\log n}(\log n - \log\log n^2 - 1) \\ &\geq k'\left(\frac{1}{2} - \frac{\log\log n^2 + 1}{2\log n}\right) \\ &\geq k'/4\end{aligned}$$

So we have a lower bound for $\sum_{i=2\log n}^{n} c_i$. We can of course perform the substitution $n \leftarrow \log n$, which would yield $\sum_{i=2\log\log n}^{\log n} c_i \geq k'/4$. Note that the c_i's from this new sum are disjoint from those of the previous sum. We can continue this process of taking logarithms until we reach a constant:

$$\sum_{i=1}^{n} c_i \geq \frac{k'}{4}\log^* n$$

And clearly, for any value of k', we can choose n large enough that $\log^* n > 4/k'$, yielding $\sum c_i > 1$, a contradiction. This proves Theorem 5.1. □

6 Conclusions and Open Problems

We have introduced the *interval scheduling problem*, and shown the following results:

1. A strongly 2-competitive algorithm for 2-length problem instances.
2. An $O((\log\Delta)^{1+\epsilon})$-competitive algorithm for scheduling intervals of arbitrary lengths.
3. An $O(\log\Delta)$ lower bound on the competitive ratio for scheduling intervals of arbitrary lengths.

We have also introduced the *online marriage problem*, presented one solution, and shown how to use this problem to solve our scheduling problem for arbitrary length intervals, with no prior knowledge of the min and max lengths.

This work suggests several directions for further research:

Convergence of Series. What is the exact relation between convergent series and scheduling with arbitrary length intervals? Also, since we have given upper and lower bounds with a gap between them, what is the tight bound? We have been able to narrow the bounds somewhat, but have not determined a tight bound.

Requests for fractions of a resource. Is it possible to service requests that require only a fraction of the resource? For instance, a request could be of the form "I require 10% of the bandwidth of the network between 1:30 and 3:00." The same goal would apply: maximize overall utilization of the network. We believe that techniques of [1] could be used in conjunction with our algorithms to solve a wide range of extensions.

Algorithms for more general release times. One could imagine a model in which clients could make requests for 30 minutes of resource time, to be delivered within the next 10 minutes. Thus, the interval would be allowed to slide slightly to accommodate other users. Are there competitive algorithms for this more general model? The goal would be to achieve good competitive ratios with respect to the best schedule that fulfills all requests by their deadlines.

Other applications for the online marriage problem. We presented this problem because we needed the results for scheduling with arbitrary length intervals. Are there any other applications of the problem, perhaps in a more general form?

Distributed applications such as Network Routing. Is it possible to use algorithms such as these in a distributed setting, perhaps to guarantee throughput over a distributed network against worst-case traffic?

7 Acknowledgements

Thanks to Merrick Furst for his support and many useful comments. Thanks also to Chris Colby and Anne Gattiker for their careful readings of preliminary versions of the manuscript.

References

[1] B. Awerbuch, Y. Bartal, A. Fiat, and A. Rosén. Competitive non-preemptive call control. In *Symposium on Discrete Algorithms*, 1993.

[2] A. Feldmann, J. Sgall, and S-H. Teng. Dynamic scheduling on parallel machines. In *32nd Foundations of Computer Science*, 1991.

[3] J. Garay and I. Gopal. Call preemption in communications networks. In INFOCOM92*, Florence, Italy*, 1992.

[4] J. Garay, I. Gopal, S. Kutten, Y. Mansour, and M. Yung. Efficient on-line call control algorithms. In *2nd Annual Israel Conference of Computing and Systems, Netania, Israel*, 1993.

[5] R. Govindan and D. Anderson. Scheduling and IPC mechanisms for continuous media. Technical Report UCB/CSD 91/622, Berkeley, 1992.

[6] D. Long and M. Thakur. Scheduling real-time disk transfers for continuous media applications. In *Twelfth IEEE Symposium on Mass Storage Systems*, pages 227–232, April, 1993.

[7] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.