Lookahead Scheduling in a Real-Time Context

Ben Coleman and Weizhen Mao
Department of Computer Science, College of William & Mary, Williamsburg, VA 23187-8795, USA
{*coleman,wm*}*@cs.wm.edu*

Abstract

The goal of a resource assignment problem is to satisfy a sequence of requests for a limited number of resources. Traditionally, online algorithms have been used to solve these problems, however there has been interest recently in semi-online algorithms, specifically lookahead algorithms. We explore the advantages of lookahead when applied to job scheduling, a representative resource assignment problem. We show results in three areas. First, we develop an original model for lookahead job scheduling on multiple processors in a real-time framework. Second, we show competitive analysis of some scheduling algorithms developed using the lookahead model. Third, we describe in detail a lookahead algorithm for a two-processor system, show that the algorithm can make each scheduling decision in near-constant time, and evaluate the average-case performance of the algorithm through simulation. Our results demonstrate that lookahead algorithms consistently produce better schedules than online algorithms with little cost in addition to a built-in lookahead mechanism.

Keywords: Job Scheduling, Online Algorithms, Lookahead Algorithms

1 A Lookahead Model

In our job scheduling problem, we have n jobs to schedule on m processors. Associated with each job J_j is a processing time p_j and an arrival time a_j. The goal is to create a schedule that executes all n jobs so as to minimize the total completion time, $\sum_{j=1}^{n} C_j$, where C_j is the completion time of J_j in the schedule.

A scheduling algorithm must decide when to execute each job and on which processor. An offline algorithm knows everything about the jobs when making decisions. It knows when jobs will arrive and the processing time of each job. Using this information, an offline algorithm can create an optimal schedule. On the other hand, an online algorithm considers the jobs in the order of arrival and schedules each job without knowledge of future jobs.

We are interested in algorithms that differ from offline and online algorithms in two ways. First, we utilize lookahead. Unlike an offline algorithm that knows everything about future jobs or an online algorithm that knows nothing about future jobs, a lookahead algorithm has knowledge of some future jobs. We feel that this middle ground represents a more realistic situation. Consider a doctor who responds to patients' requests for an office visit. The doctor is unable to know all such requests that will occur in the future, however, she has access to an appointment book which records all requests in the near future, say, up to next month. In general, lookahead is used to find out future requests and then allocate resources in such a way to optimize the quality of service. The second difference relates to the algorithm's sense of time. Offline and online algorithms operate outside time. Both are free to schedule jobs at any time, so long as it is not earlier than the arrival time of the job. As a result, the running time of these algorithms is rarely considered. However, in a real-time environment, once a job arrives, the time used to make scheduling decisions affects the time the job begins execution. Thus, the amount of time used to make decisions cannot be ignored.

Because of these differences, a lookahead algorithm represents a more realistic approach that falls between overly optimistic offline algorithms and overly pessimistic online algorithms. To implement a lookahead algorithm, we need a variety of components. First, we require a wait queue, Q_W, to hold jobs that have arrived but are not yet scheduled. To incorporate lookahead, we need a second queue, Q_{LA} to hold information about some jobs that will arrive in the near future. An algorithm is lookahead-k if the size of Q_{LA} is k, thus can hold the next k immediately in-coming jobs. The central component of the algorithm is the scheduler which is responsible for making decisions. When a processor becomes idle, the scheduler decides, based on the contents of Q_W and Q_{LA} at the time, whether to wait for the first job in Q_{LA} to arrive or to execute the first job in Q_W on the idle processor. We will discuss the principles of the design of an efficient and smart scheduler later.

Although many previous scheduling results use lookahead [1, 4], the definitions differ. Our work is a continuation of [2], which shows the benefits of lookahead on a one-processor scheduling problem.

2 Competitive Analysis

Competitive analysis compares the worst-case performance of an online or semi-online algorithm with that of an optimal offline algorithm. An algorithm

is said to be c-competitive if for any instance it produces a solution that is always within c times the optimal. The ratio is said to be tight if there is an instance that obtains the stated value.

Consider the problem defined earlier of scheduling n jobs on m processors. We study a lookahead-k algorithm, LAk, in which the lookahead queue Q_{LA} contains the next k immediately arriving jobs and the wait queue Q_W contains jobs that have arrived but not yet been scheduled in the non-decreasing order of their processing times. That Q_W is ordered by non-decreasing processing times is based on the easy fact that if Q_{LA} is empty at the time the optimal schedule is produced by scheduling the jobs in Q_W using the rule of Shortest Job First.

When a processor becomes idle, the scheduler is called to make a decision, which can be to schedule the first job in Q_W on the idle processor or to make the processor wait for the first job in Q_{LA} to arrive. To make the locally best decision, the scheduler assumes that no more jobs will come other than those already in Q_{LA}, then determines the best remaining schedule of the jobs in Q_W and Q_{LA} with the minimum total completion time, and finally make a decision consistent with that schedule, i.e., the scheduler will choose to execute the first job in Q_W if the best schedule executes at the same time and it will choose to wait if the best schedule waits at the same time. Note that the naive implementation of LAk which exhausts all possible remaining schedules to determine the locally best decision is not efficient. We will present a near-constant implementation in the next section.

We have studied the competitive performance of this LAk algorithm and proved the following:

Theorem 1. *The competitive ratio for LAk is at least $\frac{2}{(k+1)(k+2)}\left(\frac{n}{m}+\frac{1}{2}k(k+1)\right)$, where m is the number of processors and n is the number of jobs.*

Proof: To prove a lower bound, we need to simply show an instance that achieves the stated ratio. Since the competitive ratio measures the worst-case performance of an algorithm, an existing instance with a given bound proves the lower bound.

The proof uses the instance where m long jobs with processing time L arrive at time 0, another $k \cdot m$ long jobs with processing time L arrive at time ϵ and $l \cdot m$ short jobs with processing time 1 arrive at the 2ϵ.

In the optimal schedule, all m processors wait from time 0 to 2ϵ and then execute all short jobs before executing any long jobs. In the LAk schedule, all m long jobs with arrival time 0 will be executed before any short jobs. The second batch of longs jobs will be schedule last. It is easy to verify that the ratio of the total completion time of the optimal schedule to the total completion time of the LAk schedule is $\frac{2}{(k+1)(k+2)}\left(\frac{n}{m}+\frac{1}{2}k(k+1)\right)$. □

We conjecture that the established lower bound is also an upper bound, thus indicating that $\frac{2}{(k+1)(k+2)}\left(\frac{n}{m}+\frac{1}{2}k(k+1)\right)$ is the tight competitive ratio for LAk. The reason for this conjecture is twofold. One, when $k=0$ and $m=1$, LAk is in fact the Shortest Job First algorithm (SJF) in a single-processor system. It has been proved [3] that the competitive ratio for the algorithm in this special case is n, matching the general lower bound in Theorem 1 with $k=0$ and $m=1$. Two, when $k=1$ and $m=1$, LAk is in fact a lookahead-1 algorithm in a single-processor system. It has been proved [2] that that the competitive ratio for the algorithm in this special case is $\frac{1}{3}(n+1)$, matching the general lower bound in Theorem 1 with $k=1$ and $m=1$. Although these consistencies do not prove the upper bound, they do provide strong evidence to support our conjecture.

Assuming the bound for LAk is tight, an interesting fact comes out of the mathematics. First, note that in general, the bigger the size of k, the smaller the competitive ratio. Intuitively, as k approaches infinity, we have the optimal offline algorithm. In fact, when $k \geq \sqrt{\frac{2}{c-1}\cdot\frac{n}{m}}$ for some fixed constant c, we get $\frac{2}{(k+1)(k+2)}\left(\frac{n}{m}+\frac{1}{2}k(k+1)\right) \leq \frac{2}{(k+1)(k+2)}\left(\frac{1}{2}(c-1)k^2+\frac{1}{2}k(k+1)\right) = \frac{k^2}{(k+1)(k+2)}(c-1)+\frac{k}{k+2} < (c-1)+1 = c$. This indicates that the competitive ratio for LAk becomes bounded by the constant c for large k.

3 The LA1 Algorithm for the Two-Processor System

The most important component of our lookahead model is the scheduler. In this section, we describe an efficient scheduler for the LA1 algorithm for a system with two processors, P_1 and P_2. As mentioned earlier, the scheduler is responsible for making decisions and is called in two situations, when a processor becomes idle, and when a job arrives and there is already an idle processor. If either of Q_W and Q_{LA} is empty, the decision to be made by the scheduler is trivial. Now assume that both Q_W and Q_{LA} are not empty. Specifically, we assume that

- The wait queue Q_W contains l jobs, $J_1, \ldots, J_l$, all of which have arrived but have not been scheduled, with $p_1 \leq \cdots \leq p_l$.
- The lookahead queue, Q_{LA}, contains the next incoming job J_{l+1} (called the lookahead job) with processing time p_{l+1} and arrival time a_{l+1}.

As pointed out earlier, when the scheduler has to make a decision, it always assumes that no jobs will arrive other than the lookahead job and then tries

to make the locally best decision at the time, with the hope that a series of locally optimal decision will eventually lead to an optimal schedule at the end. In the previous section, we described a naive but expensive method that exhausts all possible remaining schedules of the jobs in Q_W and Q_{LA} to make each decision optimally. Here we are interested in an efficient algorithm, ideally constant-time, that can make a decision based on the contents in Q_W and Q_{LA}, the status of P_1 and P_2, and the current time.

We first consider some cases in which the choice to schedule J_1, the shortest and first job in Q_W, is obvious. Without loss of generality, assume P_1 is idle at time t and we need to make a scheduling decision. Also, let t' be the time when P_2 becomes idle and let $\Delta = a_{l+1} - t > 0$ be the amount of time before the lookahead job arrives. In each of the following cases, the scheduler executes J_1 rather than waiting for J_{l+1} to arrive.

- $t \geq t'$: When both processors are idle, it does not make sense to make both wait.
- $p_1 \leq \Delta$: J_1 can fit into Δ without increasing the completion times of other jobs.
- $a_{l+1} \geq t'$: In this case, J_{l+1} will arrive after t'. If P_1 is kept idle, then at time t' both processors are idle and we have the first case.
- $p_{l+1} \geq p_1$: If the algorithm waits, then when J_{l+1} arrives, it will be placed in the queue somewhere behind J_1. Therefore, the jobs will be scheduled in the same order regardless of whether or not the algorithm waits. It follows that starting the sequence of jobs earlier produces a better schedule.
- $p_{l+1} < p_1$ and $\Delta + p_{l+1} \geq p_1$: We leave the reasoning for this case to the reader to figure out.

When none of the above hold, we have $t < t'$, $p_1 > \Delta$, $a_{l+1} < t'$, $p_{l+1} < p_1$, and $\Delta + p_{l+1} < p_1$. The decision whether or not to wait requires further consideration. We break the remaining possibilities into four cases:

- Case 1: $t + p_1 > t'$ and $t + \Delta + p_{l+1} < t'$.
- Case 2: $t + p_1 > t'$ and $t + \Delta + p_{l+1} \geq t'$.
- Case 3: $t + p_1 \leq t'$ and $t + \Delta + p_{l+1} + p_1 \geq t'$.
- Case 4: $t + p_1 \leq t'$ and $t + \Delta + p_{l+1} + p_1 < t'$.

We define three job sets based on the waiting schedule, which is the schedule leaving P_1 idle from t to a_{l+1}. Set A contains all jobs on P_1 which start after a_{l+1} and complete before t'. Note that A may be empty. Set B contains jobs on P_1 following A. Let J_b be the first, and therefore the shortest job in B. Finally, set C contains jobs on P_2 which start after t'. Note that because the jobs are scheduled in the order of non-decreasing processing times, jobs $B = \{J_b, J_{b+2}, J_{b+4} \ldots\}$ will be scheduled on P_1 and jobs $C = \{J_{b+1}, J_{b+3}, J_{b+5} \ldots\}$ will be scheduled on P_2. It follows that we have $|B| = |C|$ or $|B| = |C|+1$.

The following four lemmas (one for each case) show that even in these more complex situations, the scheduling decision can be made by only considering a small portion of the known jobs. Each lemma assumes $t < t'$, $p_1 > \Delta$, $a_{l+1} < t'$, $p_{l+1} < p_1$, and $\Delta + p_{l+1} < p_1$. In the omitted proofs, the difference between the waiting schedule (the remaining schedule of jobs in Q_W and Q_{LA} that makes P_1 wait at t) and the non-waiting schedule (the remaining schedule of jobs in Q_W and Q_{LA} that schedules a job on P_1 at t) is calculated. If the difference is non-negative, the scheduler should choose to schedule, otherwise it should choose to wait. Define A to be the largest set of the first few consecutive jobs in the sequence $J_{l+1}, J_1, J_2, \ldots, J_l$ such that $\sum_A p_j \leq t' - a_{l+1}$, and identify the next job following the longest job in A in the sequence to be J_b. Let $x = |A|$, $y = |B| = \lceil \frac{l+1-x}{2} \rceil$, and $z = |C| = \lfloor \frac{l+1-x}{2} \rfloor$.

Lemma 1. *Suppose $t+p_1 > t'$ and $t+\Delta+p_{l+1} < t'$. The scheduler chooses to schedule J_1 on P_1 at t if and only if $(1+y)\Delta + (y-z-1)\min\{t'-t, p_1-p_{l+1}\} + (y-z)(t+p_{l+1}-t') \geq 0$.*

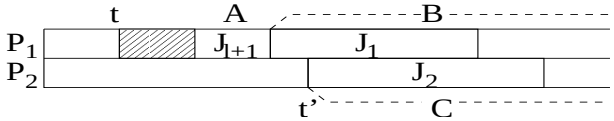

Figure 1

Lemma 2. *Suppose $t+p_1 > t'$ and $t+\Delta+p_{l+1} \geq t'$. The scheduler chooses to schedule J_1 on P_1 at t if and only if $y\Delta + (z-y)\min\{t'-t, p_1-p_{l+1}\} \geq 0$.*

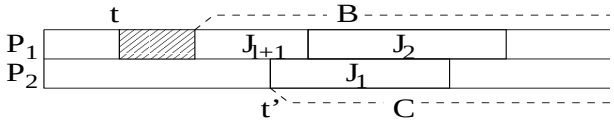

Figure 2

Lemma 3. *Suppose $t+p_1 \leq t'$ and $t+\Delta+p_{l+1}+p_1 \geq t'$. The scheduler chooses to schedule J_1 on P_1 at t if and only if $(1+y)\Delta + p_{l+1} - p_1 + (z-y+1)\max\{0, t'-t-p_{l+1}-p_b\} \geq 0$.*

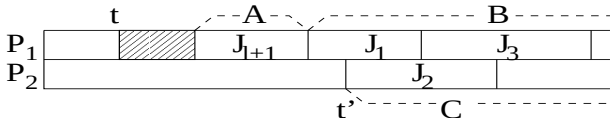

Figure 3

Lemma 4. *Suppose $t+p_1 \leq t'$ and $t+\Delta+p_{l+1}+p_1 < t'$. The scheduler chooses to schedule J_1 on P_1 at t if and only if $(x+y)\Delta + p_{l+1} - p_1 + (z-y+1)\max\{0, t'-t-\sum_A p_j - p_b\} \geq 0$.*

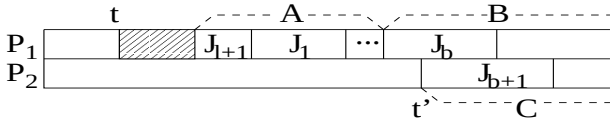

Figure 4

Figures 1 through 4 correspond to Lemmas 1 through 4. Each shows the schedule produced when the scheduler waits for J_{l+1} to arrive. The sets A, B, and C are labeled in each figured as well as the relevant jobs. When considering the possible non-waiting schedules, an important fact (not proven here) is that all the jobs in A and B except the first stay together (in most cases the first job remains as well). That is, the jobs keep the same order and are not split up on different processors. This fact allows us to calculate the difference between the waiting and non-waiting schedules.

Together, these four lemmas and the five "obvious" cases specify the algorithm used by the scheduler. In the pseudo-code that follows, "schedule" means that J_1 will be executed on the idle processor starting at time t. Accordingly, "wait" means that the processor will be idle from t to a_{l+1}.

Algorithm for each decision making step in LA1
  if Q_W is empty then return
  if Q_{LA} is empty then schedule
  else if $t \geq t'$ or $p_1 \leq \Delta$ or $a_{l+1} \geq t'$ or $p_{l+1} \geq p_1$
      or $\Delta + p_{l+1} \geq p_1$ then schedule
  else if Lemma 1 holds then schedule
  else if Lemma 2 holds then schedule
  else if Lemma 3 holds then schedule
  else if Lemma 4 holds then schedule
  else wait

4 Analysis of Algorithm

An important feature of our algorithm is that it makes each scheduling decision in near-constant time. This allows the scheduler to operate in a real-time context because it can make instantaneous decisions and not delay the execution of any waiting jobs.

The first five "obvious" cases can be calculated by only considering the first job in Q_W, the lookahead job and the completion times on the two processors, t and t'. Further, in the first three lemmas $|A|$ is zero or one and all the parameters in the conditions can be obtained easily. Therefore, the conditions of these lemmas can be evaluated in constant time. In the fourth lemma, $|A|$ is some unknown value greater than 1 that we need in order to compute y and z. In the worst case, A may contain all the jobs in Q_W and Q_{LA}. Therefore, $|A| = \mathcal{O}(l+1) = \mathcal{O}(n)$, where n is the number of jobs to be scheduled. Given this pessimistic result, we are interested in the average size of A, or $E[|A|]$.

Theorem 2. $E[|A|] \leq 3$.

Proof: We assume that the average job inter-arrival time is μ_a and the average job processing time is μ_p. Suppose processor P_1 is idle at time t. Let J_0 be the last job scheduled on P_2 with starting time t^0 and completion time $t' = t^0 + p_0$. Without loss of generality, assume $t' > t$. Consider the waiting schedule in Figure 5.

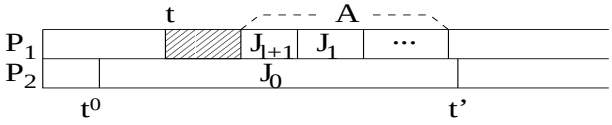

Figure 5

Note that $0 \leq |A| \leq l+1$. Also, for the moment, we ignore J_{l+1} and consider only those jobs from Q_W that fit in the time span $t' - t$. Let this set of jobs be A'. We wish to construct a set of jobs larger than A'. First, we will look at how many of the individual jobs would fit between $t' - t$. Clearly this number is at least as large as $|A'|$. Second, we will maximize the time available. If $|A'|$ is large, we would expect $t' - t$ to be large as well. For a given p_0, this time span is maximized when $t = t^0$. However, note that at time t^0, J_0 was the smallest job in Q_W. Therefore, any job in A' must have arrived after time t^0 (and before time t). Consequently, as t approaches t^0, we minimize the amount of time for jobs to arrive. Rather than considering some average position of t, assume that jobs can arrive at any time between t^0 and t', as though $t = t'$. Let X be the number of jobs that arrive between t^0 and t'. Also, assume these jobs must have processing time smaller than $t' - t^0$, as though $t = t^0$. These assumptions include at least as many jobs as those in A'. Therefore,

$$E[|A'|] \leq Pr\{p_j < t' - t^0\} \cdot E[X]$$

Recall that $p_0 = t' - t^0$. Any job could be J_0, and therefore the average size of p_0 is μ_p. Since the inter-arrival times are on average μ_a, we have

$$E[|A'|] \leq Pr\{p_j < \mu_p\} \cdot \tfrac{\mu_p}{\mu_a}$$

Now consider J_{l+1} and the original set of interest, A. Note that if $a_{l+1} + p_{l+1} \geq t'$, J_{l+1} completes after t' and $|A| = 0$. Otherwise, we have at most $1 + |A'|$ jobs that will fit in the time span $t' - t$. Therefore,

$$E[|A|] \leq Pr\{a_{l+1} + p_{l+1} < t'\} \left[1 + Pr\{p_j < \mu_p\} \cdot \tfrac{\mu_p}{\mu_a}\right]$$

Note that $\rho = \mu_p / \mu_a$ is the traffic intensity of the system. In a realistic two-processor system we expect $\rho \leq 2$, otherwise jobs arrive faster than they can be executed. Therefore, the average size of A is less than the probability J_{l+1} will a fit in the time span $t' - t^0$ plus a fraction of the traffic intensity. It follows that $E[|A|] \leq 3$. □

Because $|A|$ is relatively small, the condition in Lemma 4 can be checked in near-constant time. Since this is the most time consuming case in the algorithm, we conclude that any scheduling decision can be made in near-constant time.

5 Simulation Results

With the goal of gaining quantitative information

about the performance of a lookahead algorithm, we modeled our two-processor system using next-event simulation. The two algorithms used in the simulations were SJF (or equivalently, LA0), as a benchmark for comparison, and LA1. The SJF algorithm maintains the wait queue sorted by processing time with the shortest job at the head of the queue. When a processor becomes idle, the first job in the queue is scheduled on that processor. The LA1 algorithm is as described earlier.

For all simulations, inter-arrival times were assumed to be Exponentially distributed. Results were gathered when the processing times were taken from Uniform, Triangle, Erlang, Exponential and Hyper-Exponential distributions. For most results, $n = 25$ but we also considered cases where n was as large as 200. Finally, the traffic intensity, or ratio between the arrival rate and the service rate, was varied between 1.5 and 2.5 (with a traffic intensity of 2 representing system saturation).

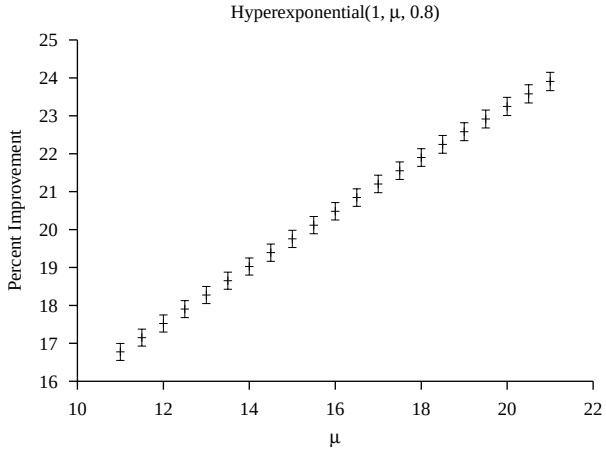

Figure 6

Figure 6 shows the percent improvement of the LA1 algorithm over the SJF algorithm when service times were taken from a Hyper-Exponential distribution. When $\mu = 10$ the traffic intensity is 1.5 and when $\mu = 22$ the traffic intensity is 2.5. Thus the simulation considers conditions from under utilization to overly saturated.

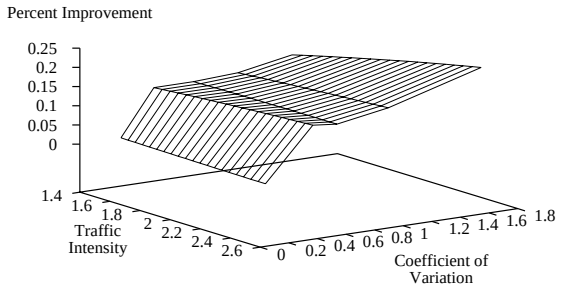

Figure 7

In order to meaningfully compare the various service time distributions, we use the unit-less coefficient of variation, the standard deviation over the mean. In the context of job lengths, larger values of the coefficient imply an increase in the likelihood that a long job is followed by a short job. Figure 7 summarizes our simulation results. From left to right, the service time distributions are Erlang, Triangle, Uniform, Exponential and Hyper-Exponential. For all runs, the lookahead algorithm produced better schedules, but when the variation of job sizes was the largest, the most improved schedules were produced. This implies that the greatest improvement occurs when long jobs are delayed by short jobs.

We also studied the state of the system when lookahead occurred. We found that

- The number of times lookahead is utilized is directly proportional to the percent improvement.
- Traffic intensity affects the average queue size which in turn determines the frequency that lookahead is utilized.
- The longer the algorithm is willing to wait, the better the improvement.
- Long jobs followed by short jobs create opportunities for lookahead to be utilized.
- In general, larger coefficients of variation produce larger percent improvements.

6 Future Research

For future research, we are interested in proving the tight competitive ratio for LAk, applying lookahead to other resource assignment problems, and developing alternative performance analysis methods suitable for lookahead algorithms.

References

[1] P. Keskinocak. Online algorithms with lookahead: A survey. ISYE Working Paper, 1999.

[2] W. Mao and R. K. Kincaid. A look-ahead heuristic for scheduling jobs with release dates on a single machine. *Computers and Operations Research*, 21(10):1041–1050, 1994.

[3] W. Mao, R. K. Kincaid, and A. Rifkin. On-line single machine scheduling algorithms, chapter 8. In S. Nash and A. Sofer, editors, *The Impact of Emerging Technologies on Computer Science and Operations Research*, pages 157–173. Kluwer Academic Publisher, 1995.

[4] S. Phillips and J. Westbrook. On-line algorithms: Competitive analysis and beyond. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 10. CRC Press, Boca Raton, 1999.