451: Online Algorithms

G. Miller, K. Sutner
Carnegie Mellon University
2020/12/01

Algorithm Analysis

- Direct Analysis
  Determine asymptotic time/space complexity (worst case, average, amortized). Maybe pin down dominant term.
- Complexity Theory
  Use complexity classes (time, space, probabilistic) for soft upper/lower bounds by establishing membership and hardness.
- Competitive Ratio
  Compare the relative performance of resource limited algorithms, in particular against the optimal one (even though we may not know exactly what it is).

Offline	A traditional algorithm that has access to the whole instance.
Offline	An algorithm that reads the input in a sequential manner, essentially reading from a stream. Actions are taken while the input is read.

So, progress is incremental: one cannot just store all the input, and then run an offline algorithm. There is a general class of a so-called streaming algorithms that imposes memory constraints to prevent shenanigans (compute a sketch of the stream), but may allow for multiple scans.

We will here deal with online algorithms that take some action every time an input item is read, in the absence of any information about the remainder of the input. The goal is to compare them to offline algorithms.

Goals

- We need to develop upper and lower bounds and hope they match.
- Ideally, we want to identify optimal algorithms (if they exist).

As usual, there will be a distinction between deterministic and randomized algorithms.

Of course, in the RealWorld$^{\mathrm{TM}}$ one can measure actual performance over many inputs and then pick the algorithm that appears to do best. Still, it won't hurt to have some idea why things work out the way they do.

Terminology

Say we are trying to solve some minimization problem. Let $\mathcal{A}$ be some online algorithm that produces an approximate solution to the problem.

Write

$$\begin{aligned} \textsf{opt}(I) &= \text{optimal solution for } I \\ \mathcal{A}(I) &= \text{solution produced by } \mathcal{A} \text{ for } I \end{aligned}$$

We are interested in the competitive ratio

$$\rho(\mathcal{A}) = \limsup_{|I|\to\infty} \frac{\mathcal{A}(I)}{\textsf{opt}(I)}$$

In other words, we are looking for the least ρ such that

$$\mathcal{A}(I) \leq \rho\, \textsf{opt}(I) + o(\textsf{opt}(I))$$

for sufficiently large instances I. Some authors prefer a constant instead of the o-term. If the constant is 0 we are dealing with the strict competitive ratio.

Example: Ski Rental

You are vacationing at Ischgl to ski, for a long time.

You can rent skis for r Schilling, or buy a pair for b Schilling. Every morning you have to decide to either rent or buy. Unfortunately, bad weather might prevent you from getting on the slopes.

Question: What is the best strategy to minimize total cost?

To simplify matters slightly, assume $r = 1$ and b is integral.

So renting for k–1 good days and then buying on day k has total cost $k - 1 + b$.

Optimal Solution

The optimal, offline solution is fairly simple:

$$\begin{aligned} &\text{rent on good days} && \iff && g < b \\ &\text{buy on day 1} && \iff && g \geq b \end{aligned}$$

Hence opt$(g, b) = \min(g, b)$.

Of course, in the online version g is not known, the weather demon is in charge of g.

We want an algorithm $\mathcal{A}$ that guarantees $\mathcal{A}(b) \leq \rho \min(g, b)$ for some hopefully small ρ and all g.

Where's the Stream?

The algorithm knows b, the cost of buying. It then reads a bit stream

$$b_1 b_2 b_3 \dots$$

where $b_i = 1$ means the weather is good, bad otherwise.

So, if $b_i = 0$ there is nothing to do (you are in Ischgl, after all).

For $b_i = 1$ there is a decision to be made, but note that if you already bought at day $j < i$ there is again no decision.

But then we might as well assume that the sequence looks like

$$\underbrace{111\dots111}_{g}\,0000\dots$$

where the weather demon controls g (we ignore the length of the vacation). $g = \infty$ is a special case.

Strategy

Here is a fairly natural cheapskate strategy: try to avoid buying until the last possible moment, aka better-late-than-never.

- Rent for b–1 days, then buy (assuming there are enough good days).

Analysis:

If $g < b$ we have $\mathcal{A}(b) = g = \text{opt}(g, b)$.

If $g \geq b$, $\mathcal{A}(b) = 2b - 1$ but $\text{opt}(g, b) = b$.

So our competitive ratio is $(2 - 1/b)$, slightly better than 2 for b fixed, but tends to 2 as b grows.

Burning Question: Is there a better strategy?

It Depends

If the strategy is deterministic, no improvement is possible.

To see why, think about the weather demon trying to make you as unhappy as possible. Never buying is a bad idea . . .

Case 1: buy at day $k < b$.

weather	bad after day k
optimal	rent k times
actual	$k - 1 + b \geq 2k$

Case 2: buy at day $k \geq b$.

weather	bad after day k
optimal	buy day 1
actual	$k - 1 + b \geq (2 - 1/b)\, b$

To do better against the weather demon we need to randomize our strategy.

We have to make sure the adversary is sufficiently weak, otherwise we are essentially back in the deterministic situation:

- the demon knows your probabilistic strategy, **but**
- does not know the random bits used,
- does not know the actual choices.

What would a reasonable randomized strategy look like?

Following the better-late-than-never paradigm, we could still pick a purchase day $k < b$ at random, using a probability distribution over $[b–1]$. Note that is not so clear how the probabilities should be chosen.

Case 1: $g < b$.

The optimal solution here is g. The expected cost of the randomized strategy is

$$C = \sum_{k=1}^{g} (k - 1 + b) p_k + \sum_{k=g+1}^{b} g\, p_k$$

Case 2: $g \geq b$.

The optimal solution here is b. The expected cost of the randomized strategy is

$$C' = b + \sum_{k=1}^{b} (k - 1)\, p_k$$

We would like to find probabilities that minimize ρ such that $C \leq \rho\, g$ and $C' \leq \rho\, b$.

LP to the Rescue

This seems rather messy, but note that we can collect computational evidence by solving a linear programming problem for the probabilities and ρ.

With a lot more work one finds the solution

$$p_i = \rho/b\,(1 - 1/b)^{b-i}$$

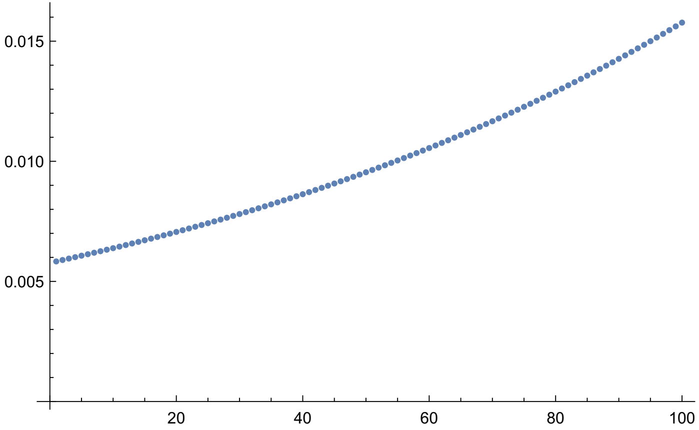
$$\rho = 1/(1 - (1 - 1/b)^b)$$

So the competitive ratio of our randomized algorithm approaches

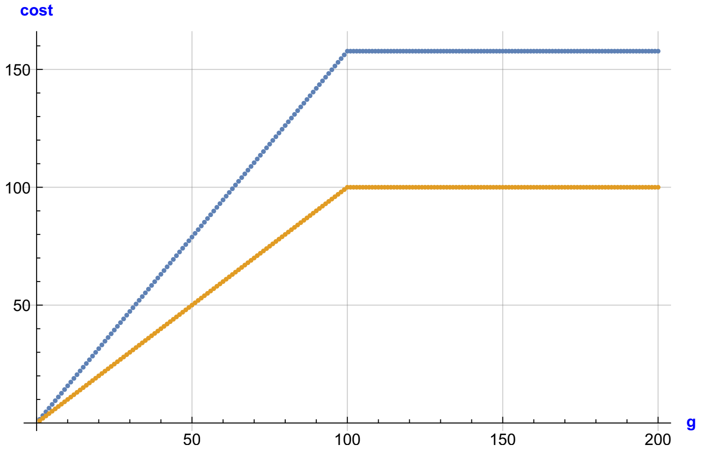
$$\frac{e}{e-1} \approx 1.5819 < 2$$

as b gets large.

$b = 100, \rho = 1.577$



$b = 100$ Cost



Recall a problem from the splay tree lecture: we are given a list L of length m, and we have to support a sequence of access operations. Wlog L is a permutation of $[m]$ and the cost of accessing r is the position of r in L. We would like to minimize the access costs of a sequence of n requests.

Here are some plausible strategies:

Do Nothing	Do not move any element.
Move-To-Front	Move the accessed element into position 1.
Single Swap	Exchange the accessed element with its left neighbor.
Frequency Count	Order the list according to access frequencies observed so far (breaking ties somehow).

We showed that MTF has cost at most twice the optimal static list (sorted by actual frequencies over the whole run).

Clearly a static list is not optimal for offline algorithms. So how do these strategies compare to the best offline algorithm?

Claim:
For Do-Nothing, Single-Swap, Frequency-Count the competitive ratio is $\Omega(n)$.

This is shown by constructing bad request sequences. On the other hand, MTF performs fairly well:

Theorem

Move-To-Front is 4-competitive, and this bound is tight.

So we have MTF versus $\mathcal{B}$, the best offline algorithm.

Write cost^M and cost^B for the number of operations performed by the two methods on a sequence of m requests r_i, $i \in [n]$, $r_i \in [m]$. MTF produces a sequence of permutations of $[m]$

$$\sigma_1, \sigma_2, \ldots, \sigma_{n+1}$$

where σ_i is the state before r_i is processed: $\sigma_i(r)$ is the position of item r in the list, the cost of accessing r. Wlog $\sigma_1 = Id$.

Similarly $\mathcal{B}$ produces $\tau_1, \tau_2, \ldots, \tau_{n+1}$.

The cost of round i of MTF is access plus move-to-front by transposition:

$$\text{cost}^M(i) = 2\,\sigma_i(r_i) - 1$$

For $\mathcal{B}$ we have access cost $\tau_i(r_i)$ but we don't know how much time is spent on moving things around.

To deal with this problem, we consider inversions. Recall that for a single permutation π an inversion is a pair $s < t$, such that $\pi(s) > \pi(t)$. A bi-inversion is a pair $s < t$ that

$$\text{is an inversion in } \sigma \quad \text{xor} \quad \text{is an inversion in } \tau$$

This is a slightly strange measure of similarity between σ and τ. It ranges from 0 to $\binom{m}{2}$. Try to characterize the permutation pairs that maximize the number of bi-inversions.

We can use the number of bi-inversions to define a potential function:

$$\Phi_i = 2 \text{ number of bi-inversions wrto } \sigma_i \text{ and } \tau_i$$

$$\text{cost}_\Phi^M(i) = \text{cost}(i) + \Phi_i - \Phi_{i-1}$$

where $\Phi_0 = 0$.

A Glitch

In terms of amortized analysis,

$$\mathsf{cost}^M = \mathsf{cost}^M_\Phi - \Delta\Phi \leq \mathsf{cost}^M_\Phi,$$

so it suffices to show that

$$\mathsf{cost}^M_\Phi \leq 4\,\mathsf{cost}^B = 4\,\mathsf{opt}$$

The problem here is that we don't know all of cost^B: one part is access, which is $\tau_i(r_i)$, but we have no description of τ_{i+1}.

The easy case is when $\tau_i = \tau_{i+1}$: there is no extra cost. We'll handle this first and then deal with the other possibility.

No τ Change

Suppose $\tau' = \tau$ when processing r. Clearly we only have to consider the inversion status of pairs involving r. Call the other item s.

If s is to the right of r in σ nothing changes, so assume it's to the left.

Moving r up front flips the inversion status of s, r wrto σ. Here is the effect on $\Delta\Phi$.

	$s < r$	$s > r$
$\tau : s\text{–}r$	$+1$	$+1$
$\tau : r\text{–}s$	-1	-1

Let $\alpha[\beta]$ be the number of s in row 1[2], so $\alpha + \beta = \sigma(r) - 1$. Hence

$$\begin{aligned} \text{cost}^M_\Phi(i) &= 2\sigma(r) - 1 + \Delta\Phi \\ &= \big(2(\alpha+\beta)+1\big) + 2\big(\alpha - \beta) \\ &= 4\alpha + 1 \leq 4\,\text{cost}^B(i) \end{aligned}$$

For the last step note that $\text{cost}^B(i) \geq \alpha + 1$.

General Case

But what if $\tau \neq \tau'$?

Then $\mathcal{B}$ also performs a number of swaps in its list after the access operation.

Each swap adds 1 to the direct cost of $\mathcal{B}$, and changes the potential at most by 2, so the previous bound still holds.

Done.

To get a lower bound on the competitive ratio we need to unearth more information about the mysterious algorithm $\mathcal{B}$.

Lemma

$\mathcal{B}(\boldsymbol{r}) \leq m + \binom{m}{2} + (m+1)(n-1)/2$

Proof.

First define an apparently brain-dead algorithm $\mathcal{A}_\pi$ for any permutation π of $[m]$: it sets $\sigma_i = \pi$ for all $i \geq 2$ and then just pays the cost of access in π:

$$\mathsf{cost}^{\mathcal{A}_\pi} \leq m + \binom{m}{2} + \sum_{i \geq 2} \pi(r_i)$$

To average over all permutations, note that

$$\sum_{\pi \in \mathfrak{S}} \pi(r) = \sum_{i} i\,(m-1)! = \frac{1}{2} m(m+1)\,(m-1)!$$

$$\begin{aligned} 1/m! \sum_{\pi \in \mathfrak{S}} \mathsf{cost}^{\mathcal{A}_\pi} &\leq 1/m! \left(\sum_{\pi} m + \tbinom{m}{2} + \textstyle\sum_{i \geq 2} \pi(r_i) \right) \\ &= m + \tbinom{m}{2} + 1/m! \textstyle\sum_{i,\pi} \pi(r_i) \\ &= m + \tbinom{m}{2} + 1/(2\, m!) \textstyle\sum_{i} m(m+1)\,(m-1)! \\ &= m + \tbinom{m}{2} + 1/2(m+1)(n-1) \end{aligned}$$

$\square$

This is quite intuitive: on average access should cost about $m/2$.

4 is Sharp

Next question: what is the worst an adversary can do to MFT? Well, always choose the last element in the list as the next request. Total cost is $(2m-1)n$.

Theorem

The competitive ratio of MTF approaches 4 as m goes to infinity.

Proof.

By the last lemma we need to consider

$$\lim_n \frac{(2m-1)n}{m+\binom{m}{2}+1/2(m+1)(n-1)} = 4 - 6/(m+1)$$

□

Computer storage forms a natural hierarchy according to speed. Say, we have an SSD with N pages of memory, and a cache in RAM with a capacity of k pages. We have to process a sequence of n requests for pages.

For simplicity only charge for page misses: we get a request for a page currently not in the cache.

We may safely assume that $k < N$ and n is sufficiently large, so at some point the cache fills up (if that never happens, there really is no issue). From that point on, whenever a cache miss occurs, we remove one page from the cache before bringing in the new one (page replacement algorithm).

Question:
Which page should be evicted so as to minimize the total number of faults?

Natural Strategies

Natural deterministic online strategies are

FIFO	Remove the oldest page.
LRU	Remove least recently used page.
LFU	Remove the page least frequently used.

Of course, since the algorithm is online we don't know the overall frequencies, only the currently observed ones of pages in the cache.

As a practical matter, LRU seems to be far and away the best of these. Note that it is far from clear what the average input here looks like (locality properties).

Optimal Offline Algorithm

Suppose we know the full sequence of page requests. Here is what intuitively ought to be the best strategy: greedily grab as much reprieve as possible.

FIF Furthest-in-the-Future: remove the page that appears as late as possible in the sequence of future requests (maybe never).

Theorem

FIF is the optimal offline algorithm for paging.

A general proof is tricky, let's just take a look at the toy model when $N = k + 1$. When FIF evicts page p, the next fault will happen when p is requested again. So the cost of FIF is essentially n/k. Could we beat this?

Adversary from Hell

The worst an adversary can do is to always request a page that is currently not in the cache. Note that $N = k + 1$ is already enough for this purpose. This is called an adaptive adversary and is not particularly realistic: your user program does not obstruct the OS in this manner.

For example, suppose we run FIFO on

$$1, 2, \ldots, k, \overline{k}, 1, 2, \ldots, k, \overline{k}, 1, 2, \ldots, k, \overline{k}, 1, 2, \ldots$$

Then every request is a miss by design; beyond the first k we always have to evict a page. FIF will remove $k, k-1, k-2, \ldots$ which is easily seen to be optimal.

This implies that the competitive ratio of FIFO cannot be better than k in the limit. The same argument works for LRU.

Oblivious Adversary

A more realistic and relevant type of adversary is oblivious. It may know our caching strategy, but it

- does not know the random bits used,
- does not know the actual choices made.

One can show the following, see below for a special case.

Theorem

Suppose a randomized paging algorithm $\mathcal{A}$ has competitive ratio ρ against oblivious adversaries. Then $\rho \geq H_k \approx \ln k$, the kth harmonic number.

Blocks

Suppose we have some online algorithm $\mathcal{A}$ and our requests are

$$\boldsymbol{r} = r_1, r_2, \ldots, r_n$$

We can partition $\boldsymbol{r}$ into disjoint blocks

$$\boldsymbol{r} = B_1, B_2, \ldots, B_\beta$$

marching from left to right as follows: each block is a maximum contiguous sequence of requests in which exactly k distinct pages are requested. Note that the blocks depend on the algorithm as well as $\boldsymbol{r}$.

E.g., starting with an empty cache, the next request after B_1 causes the first eviction.

Sometimes it is slightly more convenient to assume that the algorithm starts with a full cache (the same if we compare two methods). In the limit, this makes no difference.

But Why?

Recall that we are dealing with page replacement algorithms only.

It is not unreasonable to expect such algorithms to have a natural persistence property:

> Suppose a page is placed into cache during block B.
> Then it will stay there for the duration of the block.

This has a nice consequence: instead of dealing with the whole block, we can just focus on the first occurrence of each page. In essence, we can assume that a block simply has length k.

Example: LRU

Consider the blocks $B_1, \dots, B_\beta$ defined by LRU.

Claim 1: LRU faults at most k times per block.

Use the persistence property: once a page is placed into the cache, it cannot be evicted until k other pages get requested. But that will happen in the next block.

Claim 2: FIF has at least $\beta - 1$ faults.

The first request p in B_2 produces a page fault since B_1 is maximal. After it is handled, the cache contains p and $k-1$ other pages. Done by induction.

So the competitive ratio is at most

$$\frac{k\beta}{\beta - 1} \to k \quad \text{for} \quad \beta \to \infty$$

It follows LRU has competitive ratio k.

Here is a simple randomized paging algorithm MARK that is surprisingly good in the theoretical sense. There are other versions, this one is probably the simplest one.

Any page in the cache may be marked or unmarked; it is marked when it first enters the cache. Say p is the next request.

- p not in cache:
  - if all pages are marked, unmark everybody
  - randomly remove an unmarked page
  - insert p marked

So we try to keep pages that have been inserted recently, a method vaguely reminiscent of LRU.

Note that MARK has persistence: to get evicted a page has to be unmarked, and that happens in the next block.

Let's use the blocking idea from above again: k different requests per block. At the beginning of a block, all pages in the cache are unmarked.

Let C be the cache contents at the beginning of block B. Call a page requested during B old/new if it is/isn't in C. Of course, an old page may be gone by the time it is requested. Let ν be the number of new pages during B.

The ν new requests all lead to evictions, the real question is what happens with old requests p: this time there are two possibilities

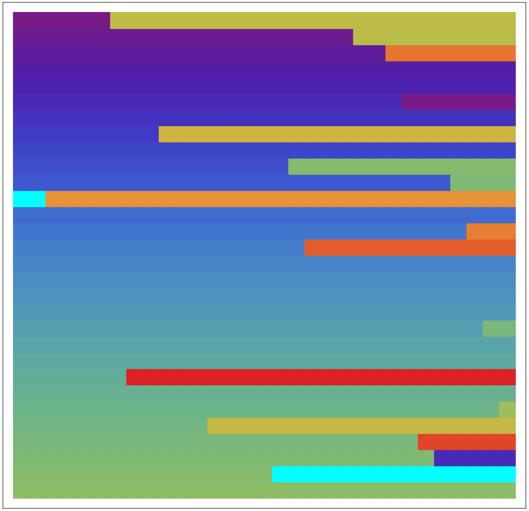
- p is still there, no eviction necessary,
- p has been evicted (unfortunately), and needs to be reinserted.

$k = 6$, initial cache $[k]$, requests $p = 7, 8, 9$, so $\nu = 3$.

Extra miss: on $p = 2$ at time 5.

	time						
	0	1	2	3	4	5	6
p	-	7	6	8	5	2	4
cache	6		6				
	5				5		
	4						**9**
	3					**2**	
	2			**8**			
	1	**7**					

$k = 30$, $\nu = 11$. One of the 6 kick-out-and-reinsert cases is highlighted in cyan.



We may safely assume that all the new requests come first, that can only drive up the number of old page misses.

Now consider step $\nu + 1$: the probability of a miss is ν/k and the expected number of misses is $\nu + \nu/k$.

At step $\nu + 2$, the probability of a miss is

$$(\nu + \nu/k)/k = \nu(k+1)/k^2 \leq \nu/(k-1),$$

and so on, and so forth.

Summing the expected total cost of all (new + old) requests is at most

$$\nu + \nu \sum_{i=0}^{k-\nu-1} \frac{1}{k-i} \leq \nu H_k$$

Optimality?

Write ν_i for the new requests in block B_i. The vague idea behind the next claim is that we can amortize good performance by the optimal algorithm by looking at adjacent blocks. More precisely, define

$$\delta_i = \#\text{ elements in the MARK, but not the FIF, cache}$$

a sort of potential function.

Claim: FIF has at least $1/2 \sum \nu_i$ faults.

Write c_i for the cost of FIF during block B_i (this a MARK block). We will show that

$$c_i \geq \max(\nu_i - \delta_i,\ \delta_{i+1})$$

$c_i \geq \nu_i - \delta_i$

During block B_i, the FIF cache differs in at most δ_i places from the MARK cache, so it must have at least $\nu_i - \delta_i$ faults.

$c_i \geq \delta_{i+1}$

The k pages requested during block B_i must be in the cache when block B_{i+1} starts; furthermore, each such page was in the FIF cache at some point during the block. But the FIF cache at the beginning of block B_{i+1} has only δ_{i+1} of these pages, so there were at least that many evictions.

Putting Things Together

Max is a bit awkward, so lets use the average instead, $(\nu_i - \delta_i + \delta_{i+1})/2$. Summing over all blocks we get a lower bound for the total cost of FIF:

$$1/2 \left(\sum \nu_i + \delta_\beta - \delta_0 \right)$$

We have $\delta_0 = 0$, and δ_β is small compared to the sum, so we get a lower bound of $1/2 \sum \nu_i$

It follows that our marking algorithm is $2H_k = O(\log k)$ competitive.

Again, this is against an oblivious adversary, and indeed optimal. There is no defense against an adaptive adversary.