CS264: Beyond Worst-Case Analysis
Lecture #4: Parameterized Analysis of Online Paging[*]

Tim Roughgarden[†]

January 19, 2017

1 Preamble

Recall our three goals for the mathematical analysis of algorithms: the Explanation Goal, the Comparison Goal, and the Design Goal. Recall that, for the online paging problem, traditional competitive analysis earns a pretty bad report card on the first two goals. First, the competitive ratios of all online paging algorithms are way too big to be taken seriously as predictions or explanations of empirical performance. Second, while competitive analysis does identify the least recently used (LRU) policy as an optimal online algorithm, it also identifies less laudable policies (like first-in first-out (FIFO) or even flush-when-full (FWF)) as optimal.

Last lecture introduced resource augmentation guarantees. This approach made no compromises about worst-case analysis, but rather changed the benchmark — restricting the offline optimal algorithm to a smaller cache, which of course can only make competitive ratios smaller. The primary benefit of resource augmentation is much more meaningful and interpretable performance guarantees, which is good progress on the Explanation Goal. A key drawback is that it failed to differentiate between the LRU policy and the FIFO and FWF policies. Intuitively, because the empirical superiority of LRU appears to be driven by the properties of "real-world" data, namely locality of reference, we can't expect to separate LRU from other natural paging algorithms (like FIFO) without at least implicitly articulating such properties.

The goal of this lecture is to parameterize page sequences according to a natural measure of locality, to prove parameterized upper and lower bounds on the performance of natural paging policies, and (finally!) to prove a sense in which LRU is strictly superior to FIFO (and FWF). As a bonus to this progress on the Comparison Goal, we'll obtain good absolute performance guarantees that represent a big step forward for the Explanation Goal.

[*]©2014–2017, Tim Roughgarden.

[†]Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

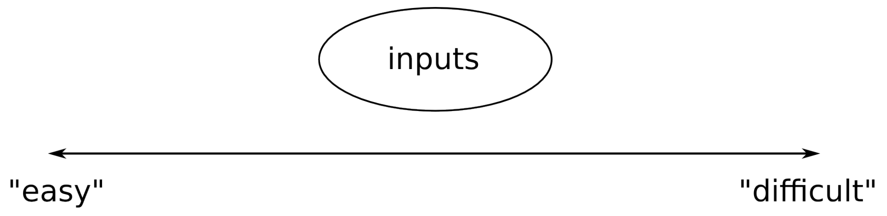

Figure 1: A parameterization of the inputs of a problem.

2 Parameterized Analysis

2.1 How It Works

We pause to explain what we mean by "parameterized analysis." We have in mind a two-step program. The first step is to impose a natural parameterization of all inputs. This parameter should model the intuitive "easiness" or "difficulty" of an input — perhaps merely with respect to one algorithm, or ideally in some more general sense (Figure 1). Such parameters are also called "condition numbers" in some contexts.

The second step is to analyze the performance of an algorithm as a function of the chosen parameter. This gives a much more fine-grained analysis of algorithm performance than traditional worst-case analysis, which typically parameterizes inputs solely by their size.

The best-case scenario is that the chosen parameter is independent of the description of the algorithm, and is referenced only in the algorithm's analysis. For example, the LRU algorithm is perfectly well defined on all inputs; we want a quantitative measure of locality not to design a new algorithm, but rather to articulate properties of data that are common in real instances and sufficient to prove good performance guarantees for LRU. There are also cases, however, where it makes sense to design algorithms whose behavior depends explicitly on the value of a parameter (e.g., see Homework #3).

2.2 Example: Running Time of the Kirkpatrick-Seidel Algorithm

The running time analysis of the Kirkpatrick-Seidel (KS) algorithm for computing the maxima of a point set (Lecture #2) is an exemplary parameterized analysis. Our goal in that lecture was not parameterized analysis per se, but rather a proof that the KS algorithm is instance-optimal in a natural sense. We noted in that lecture that a prerequisite for an instance-optimality result is a tight, up to a constant factor, input-by-input upper bound on the performance of the instance-optimal algorithm. This motivated us to step through a sequence of increasingly fine-grained parameterizations of 2D Maxima instances. First, parameterizing solely by the number n of input points, the worst-case running time of the KS algorithm is $O(n \log n)$, and no better upper bound that depends solely on n is possible. This was not good enough for an input-by-input tight upper bound, so we considered bounds parameterized by both the input size n and the output size h. We discussed an upper bound of $O(n \log h)$ on the running time of the KS algorithm on every input. This was still not good enough for a tight input-by-input upper bound, so we finally used a quite detailed parameterization (see Lecture #2) that characterizes the running time of the KS algorithm, up to a constant factor, on every input. This parameter is $\Theta(n)$ on the "easiest" inputs, $\Theta(n \log n)$ on the "hardest" inputs, and in general can be anywhere in between. It is independent of the description of the KS algorithm, and used only to identify which instances are "easy" for it and which ones are "hard."

2.3 Why Bother?

The first reason to work hard to perform a detailed parameterized analysis of an algorithm is to extract advice about when — meaning on which types of inputs — you should use the algorithm. (Note that different algorithms will generally fare differently for different parameterizations of inputs.) A simple example occurs in graph algorithms, where running times are generally phrased as a function of both the number n of vertices and the number m of edges. For many graph problems, the best algorithm to use can depend on the edge density of the graph, with very different algorithms being appropriate for sparse versus dense graphs.[1]

A second motivation for parameterized analysis is that it offers a two-step approach to explain the good "real-world" performance of an algorithm. The first step is to give a parameterized analysis of the algorithm — identifying "easy" inputs on which the algorithm is guaranteed to do well. The second step is to argue, either empirically or mathematically, that real-world instances tend to be "easy" according to this parameter. For example, if the parameter is easy to compute and canonical benchmark instances are available, one can just check if the benchmarks qualify as easy or not. A different approach is to posit some model of data (e.g., a distribution or generative model) and prove (mathematically or by simulation) that typical instances are easy. For example, this two-step approach is a good way to think about smoothed analysis (see future lectures); more imminently, we'll apply it this lecture to give a satisfying theoretical explanation of the empirical superiority of the LRU algorithm.

3 Quantifying Locality: The Working Set Model

We return to the model of online paging introduced last lecture. Our goal is to (i) parameterize page request sequences according to their "amount of locality" and (ii) show that the LRU algorithm's performance is excellent, both in an absolute sense and compared to other online algorithms, for page sequences that exhibit a high degree of locality. So how should we measure locality?

There is not a unique answer to this question (see Homework #2), but a tried-and-true one will serve us well. Denning [2] defined the *working set model*, which is parameterized by a function f from the positive integers $\mathbb{N}$ to $\mathbb{N}$, which describes how many distinct page requests we might see in a window of a given length. Formally, we say that the request

[1]For instance, for the all-pairs shortest paths problem, iterated invocations of Dijskstra's algorithm is usually a good idea in sparse graphs, while the Floyd-Warshall algorithm can be faster in dense graphs.

sequence σ *conforms to* f if for every positive integer n and every set of n consecutive page requests in σ, there are requests for at most $f(n)$ *distinct* pages. Intuitively, the "smaller" f is, the more it imposes locality on request sequences that conform to it.

For example, if $f(n) = n$ for all n, then every request sequence conforms to f. If $f(2) = 1$, then only the constant request sequences conform to f. Canonical functions that you might want to keep in mind are $f(n) \approx \sqrt{n}$, where (for example) only windows of size at least 100 contain requests for at least 10 distinct pages, and $f(n) \approx 1 + \log_2 n$, where only windows of size at least 1000 contain requests for at least 10 distinct pages.

The main point of this lecture is a very satisfying theorem about online paging algorithms that earns good grades on our Prediction and Comparison Goals. There are two differences between today's model and last lecture's model that enable this result. The first good idea is, rather than parameterizing the performance of an online algorithm solely by the cache size k (as in last lecture), we will parameterize performance also by the function f. While it's generally best if parameters are simple (like a single number), we'll parameterize here by an entire function. This seems necessary given that we're also simultaneously parameterizing by the cache size k, with the relevant values of f depending on k.

This parameterization is a really good idea, but it's not enough. The reason is that the competitive ratio of every deterministic online algorithm remains at least k, even after restricting to page sequences that conform to a function f. This is true for essentially every non-trivial function f; see Homework #2. We reiterate that you should never let stupid examples get in the way of a good idea. The counterexamples in Homework #2 don't imply that parameterizing performance by f is a bad idea, just that we'll need to change the model in a second way as well.

Rather than studying the relative performance of online paging algorithms (i.e., their competitive ratios), we'll focus on their absolute performance, meaning the page fault rate $\frac{\#\text{ faults}}{|\sigma|}$. Recall that you've seen this performance metric before — in the second case of Young's theorem, last lecture. Conceptually, using an absolute performance measure is promising because it eschews comparison with an optimal offline algorithm. It is conceptually unsatisfying to use an offline algorithm — which is not an option on the table — to compare online algorithms. We saw last lecture how comparing performance to such a strong benchmark can blur the distinctions between different algorithms, with all reasonable (and some unreasonable) paging algorithms having the same competitive ratio.[2] Here, we'll just directly compare the page fault rates of different online algorithms, without worrying about what a hypothetical offline algorithm would do. Finally, guarantees on the page fault rate are often more meaningful to practitioners than guarantees on the competitive ratio, anyway.[3]

[2]That said, this approach can be useful, especially if compelling positive results exist.

[3]To argue that being close to the offline optimal algorithm is useful, one should also argue that the latter is good in some absolute sense.

4 Statement of the Main Result

Having finalized the model, we can now state the main result of this lecture, which is due to Albers, Favrholdt, and Giel [1]. The theorem statement includes two terms — "concave" and the key parameter "$\alpha_f(k)$" — that will be defined shortly. (Intuitively, the more locality imposed by f, the smaller the parameter $\alpha_f(k)$.) The conceptual content of the result can be fully appreciated even before reading these definitions.

Theorem 4.1 ([1])

(a) For every concave function f, for every cache size $k \geq 2$, and for every deterministic online algorithm A, the worst-case page fault rate (over sequences that conform to f) is at least $\alpha_f(k)$.

(b) For every concave function f, and for every cache size $k \geq 2$, the worst-case page fault rate (over sequences that conform to f) of the LRU algorithm is at most $\alpha_f(k)$.

(c) There exists a concave function f, a cache size $k \geq 2$, and a request sequence σ that conforms to f such that the page fault rate of the FIFO algorithm on σ is strictly larger than $\alpha_f(k)$.

Parts (a) and (b) of the theorem are in some ways analogous to the first two results from last lecture — a worst-case lower bound on the performance of every deterministic online paging algorithm, and a matching worst-case upper bound for the LRU algorithm. The results here are more satisfying, for two reasons. First, parts (a) and (b) of Theorem 4.1 prove the worst-case optimality of the LRU algorithm (in terms of the page fault rate) for every function f and cache size k, while last lecture only proved this (for the competitive ratio) for every cache size k. Thus Theorem 4.1 proves the optimality of LRU in a more fine-grained sense than the results of last lecture. Second, as we'll see below when we define $\alpha_f(k)$, the absolute performance guarantee in Theorem 4.1 is directly meaningful for many functions f and cache sizes k, in contrast to the disappointingly big competitive ratios we were stuck with last lecture.

And it gets better: part (c) of Theorem 4.1 rigorously separates, for the first time in these lectures, the LRU algorithm from the FIFO algorithm. The former is optimal among online algorithms for every f and k, while the latter is not (and the FWF algorithm is even worse).

We conclude this section by defining the undefined terms in Theorem 4.1. Consider a function f; see Figure 2 for a concrete example. Observe that there is no loss of generality in assuming that $f(n+1) \leq f(n)+1$ for every n — every sequence with at most $f(n)$ distinct pages in every window of size n automatically has at most $f(n)+1$ distinct pages in every window of size $n+1$. Similarly, we can assume that $f(n+1) \geq f(n)$ for every n. Thus, the top row of the "function table" (Figure 2) is a bunch of 1s, followed by a bunch of 2s, followed by a bunch of 3s, and so on. We can completely describe such a function by giving, for each $j \in \mathbb{N}$, the number m_j of (consecutive) values of n for which $f(n) = j$. For example, in Figure 2, $m_1 = 1$, $m_2 = 1$, $m_3 = 2$, and $m_4 = 3$.

$f(n)$	1	2	3	3	4	4	4	5	$\cdots$
n	1	2	3	4	5	6	7	8	$\cdots$

Figure 2: A concave function, with $m_1 = 1$, $m_2 = 1$, $m_3 = 2$, $m_4 = 3, \ldots$

In this lecture, by a *concave* function, we mean a function f with $m_1 \leq m_2 \leq m_3 \leq \cdots$.[4] For the rest of this lecture, we consider only concave functions f.[5] This is not a worrisome restriction. Concavity is largely consistent with empirical data [2]. Also, even for data sets where the empirically observed working set function $\hat{f}$ is not concave, one can choose any concave function f with $\hat{f}(n) \leq f(n)$ — only more sequences conform to f than to $\hat{f}$ — and invoke Theorem 4.1(b) with f to obtain a performance guarantee for LRU on the original data.

The key parameter $\alpha_f(k)$ is defined for $k \geq 2$ as

$$\alpha_f(k) = \frac{k-1}{f^{-1}(k+1) - 2}, \tag{1}$$

where $f^{-1}(m)$ is the smallest value of n for which $f(n) = m$. For example, in Figure 2, $f^{-1}(3) = 3$, $f^{-1}(4) = 5$, and $f^{-1}(5) = 8$. A good way to interpret $f^{-1}(m)$ is as a lower bound on the window size if you know the window contains requests for at least m distinct pages.

Revisiting our example functions f above, if $f(n) = n$ for all n, then $\alpha_f(k) = 1$ for all k. (Since this f allows all page request sequences, we know from last lecture that we are stuck with a worst-case page fault rate of 1.) If $f(n) \approx \sqrt{n}$, then $f^{-1}(m) \approx m^2$ and so $\alpha_f(k) \approx 1/k$. Note that this is a pleasingly small fault rate even for modest-size caches. If $f(n) \approx 1 + \log n$, then $\alpha_f(k) \approx k/2^k$, which is small every for tiny caches.

We conclude that, for many reasonable choices of f and k, the page fault rate guarantees offered by Theorem 4.1 are so small as to be meaningful even when taken at face value.

5 Proof of Theorem 4.1

We now prove Theorem 4.1, one part at a time. We assume throughout this proof that $f(2) = 2$ (and so $m_1 = 1$); if $f(2) = 1$ then only the constant request sequences conform to f and every reasonable paging algorithm has fault rate 0. We also assume throughout that $k \geq 2$; if $k = 1$ (and $f(2) = 2$) then every deterministic online paging algorithm has worst-case fault rate 1 over sequences that conform to f (why?).

To prove part (a) — the page fault rate lower bound for every deterministic algorithm — fix f, $k \geq 2$, and a deterministic online paging algorithm A. Assume that the number N of pages is $k + 1$, so that at any given time there is a unique page missing from a (full) cache.

[4]This is a non-standard definition of a concave function, but is in the same spirit of "diminishing returns."

[5]The keen reader should verify that this concavity assumption is only used in the proof of part (a) of Theorem 4.1.

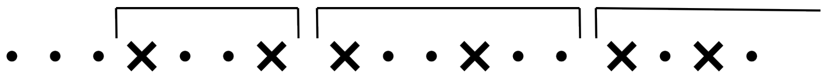

Figure 3: Blocks of $k-1$ faults, for $k=3$.

The plan is to copy the proof of the lower bound on the competitive ratio of deterministic algorithms from last lecture. We can't do this directly, as the sequences used there might well request N distinct pages in a window of length N, thereby failing to conform to f. We reuse that sequence as best we can, while inserting duplicate page requests to create a sequence that conforms to f.

Formally, the sequence σ consists of an arbitrarily large number ℓ of *phases*, with each phase comprising $k-1$ blocks. For $j = 1, \ldots, k-1$, the jth block of a phase consists of m_{j+1} consecutive requests for the same page p_j, where p_j is the unique page missing from the algorithm A's cache at the start of the block. Thus, by the choice of the p_j's, A incurs a page fault on the first request of a block, and not on any of the other (duplicate) requests of that block. Thus, algorithm A suffers exactly $k-1$ page faults per phase.

To compute A's page fault rate on σ, we need to compute the length of a phase. It is $m_2 + m_3 + \cdots + m_k$; since $m_1 = 1$ this is $(\sum_{j=1}^{k} m_j) - 1$. Recalling the definition of the m_j's, this is $(f^{-1}(k+1) - 1) - 1 = f^{-1}(k+1) - 2$.

The final step of the proof of Theorem 4.1(a) is to check that the sequence σ does indeed conform to f; we defined the page duplications so that this would be true, and leave the formal proof to Homework #2.[6]

Moving on to part (b) — a matching upper bound for the LRU algorithm — fix k and f and consider a sequence σ that conforms to f. Our fault rate target $\alpha_f(k)$ is a major clue to the proof (recall (1)): we should be looking to partition the sequence σ into blocks of length at least $f^{-1}(k+1) - 2$ such that each block has at most $k-1$ faults. Guided by the latter property, consider groups of $k-1$ consecutive faults of LRU on σ. Each such group defines a *block*, beginning with the first fault of the group, and ending with the page request that immediately precedes the beginning of the next group of faults (see Figure 3).

Claim: Consider a block other than the first or last. Consider the page requests in this block, together with the requests immediately before and after this block. These requests are for at least $k+1$ distinct pages.

The claim immediately implies that every block contains at least $f^{-1}(k+1) - 2$ requests; since there are $k-1$ faults per block, this proves that the page fault rate is at most $\alpha_f(k)$ (ignoring the additive error due to the first and last blocks), proving Theorem 4.1(b).

We proceed to the proof of the Claim. We note that, in light of part (c) of the theorem, it is essential that we use in the proof properties of the LRU algorithm not shared by FIFO. Fix your favorite block, other than the first or last, and let p be the page requested immediately

[6]This step is where we use the assumption that f is concave, and also that we omitted a block of length m_1 from the construction of a phase.

$f(n)$	1	2	3	3	4	4	5	5
n	1	2	3	4	5	6	7	$\cdots$

Figure 4: Function used to construct a bad page sequence for FIFO.

prior to this block. Note that this request could have been a page fault, or not (cf., Figure 3). In any case, p is in the cache when this block begins. Consider the $k-1$ faults contained in the block, together with the kth fault that occurs immediately after the block. We consider three cases.

First, if the k faults occur on distinct pages that are all different from p, then clearly we have identified our $k+1$ distinct requests (p and the k faults). For the second case, suppose that two of the k faults are for the same page $q \neq p$. How could this have happened? After the first fault on q, q was brought into the cache. Because we're running the LRU algorithm, the only way q can get evicted is if there are k requests for distinct pages other than q after this page fault. This gives us our $k+1$ distinct page requests (q and the k other distinct requests between the two faults on q). For the final and most interesting case, suppose that one of these faults is on the page p. Because p was requested just before the first of these faults, the LRU algorithm, subsequent to this request and prior to evicting p, must have received requests for k distinct pages other than p. These requests, together with that for p, give the desired $k+1$ distinct page requests. Since these three cases cover all the possibilities, this completes the proof of the claim and of Theorem 4.1(b).[7]

Finally, we prove part (c) of Theorem 4.1. Many different choices of f and k show that the page fault rate of the FIFO algorithm can be greater than LRU's worst-case bound of $\alpha_f(k)$; we just provide one simple example. Take $k = 4$, and suppose there are 5 pages, $\{0, 1, 2, 3, 4\}$. The function f is shown in Figure 4; since there are only five pages, we can assume that $f(n) = 5$ for all $n \geq 7$. Observe that this function is concave, and that

$$\alpha_f(k) = \frac{4-1}{f^{-1}(4+1)-2} = \frac{3}{5}.[8]$$

Part (b) of the theorem implies that LRU's page fault rate is at most 60% when $k = 4$, for every request sequence that conforms to f. We now exhibit a sequence where the page fault rate of FIFO is higher.

The sequence consists of an arbitrarily large number of identical blocks of eight page requests:

$$10203040 \quad 10203040 \quad 10203040 \quad \cdots \quad 10203040.$$

[7]Observe that the arguments in the first two cases apply to many different paging algorithms, including FIFO. It is the third case that really relies on properties of the LRU algorithm. To see why the argument in the third case fails for the FIFO algorithm, suppose that p was already in the cache when it was requested just prior to the block. (If this request resulted in a page fault, then the argument continues to work for FIFO.) Unlike LRU, this request does not "reset p's clock"; if it was brought into the cache long ago, FIFO might well evict p on the block's very first fault.

[8]The page fault rate is high because f is close to linear in the relevant range; there are also bad examples for FIFO in which $\alpha_f(k)$ is small.

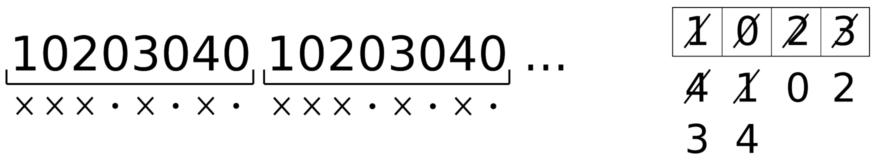

Figure 5: Sequence (left) and cache (right) for FIFO on the given input with cache size 4.

The intuition for why this sequence is worse for FIFO than LRU is clear: LRU will never fault on page 0, except for the very first time it is brought into the cache, whereas FIFO will incur the occasional page fault on it. To see that FIFO's fault rate is bigger than $\alpha_f(k) = .6$, we need to trace its behavior; see Figure 5.

Assume that FIFO begins with the empty cache. Then, in the first block, the first request results in a page fault. The cache is initially populated by 1, 0, 2, and 3; the page fault on 4 results in 1's eviction (since it was the first to be brought in). All 8 requests are faults except for the second, third, and fourth requests to 0.

In the second block, the request to 1 results in a page fault. At this point, the page 0 was originally brought into the cache earlier than its other inhabitants (2, 3, and 4); thus, FIFO evicts it. This causes a page fault on the 0, resulting in 2's eviction; on the 2, resulting in 3's eviction; and on the 4, resulting in 1's eviction. The fault pattern of the second block is the same as that of the first (even though the cache did not start empty this time), and it concludes with same cache contents as before. Thus, this pattern continues for every block, resulting in a fault rate of

$$\frac{5}{8} = 62.5\% > 60\% = \alpha_f(k).$$

This completes the proof of Theorem 4.1.

Remark 5.1 (FIFO not much worse than LRU) The fault rate of FIFO in the proof of Theorem 4.1(c) is worse than $\alpha_f(k)$, but not by much (62.5% vs. 60%). In fact, FIFO's fault rate is never too much more than $\alpha_f(k)$; see Homework #2. This is somewhat consistent with empirical results, where in many cases LRU is better than FIFO, but not by a lot.

References

[1] S. Albers, L. M. Favrholdt, and O. Giel. On paging with locality of reference. *Journal of Computer and System Sciences*, 70(2):145–175, 2005.

[2] P. J. Denning. The working set model for program behavior. *Commuications of the ACM*, 11(5):323–333, May 1968.