CS264: Beyond Worst-Case Analysis
Lecture #1: Introduction and Motivating Examples*

Tim Roughgarden†

January 10, 2017

1 Preamble

We begin in Sections 2–4 with three examples that clearly justify the need for a course like this, and for alternatives to the traditional worst-case analysis of algorithms. Our discussion is brief and informal, to provide motivation and context. Later lectures treat these examples, and many others, in depth.

2 Caching: LRU vs. FIFO

Our first example demonstrates how the worst-case analysis of algorithms can fail to differentiate between empirically "better" and "worse" algorithms. The setup should be familiar from your systems courses, if not your algorithms courses. There is a small fast memory (the "cache") and a big slow memory. For example, the former could be an on-chip cache and the latter main memory; or the former could be main memory and the latter a disk. There is a program that periodically issues read and write requests to data stored on "pages" in the big slow memory. In the happy event that the requested page is also in the cache, then it can be accessed directly. If not — on a "page fault" a.k.a. "cache miss" — the requested page needs to be brought into the cache, and this requires evicting an incumbent page. The key algorithmic question is then: *which page should be evicted?*

For example, consider a cache that has room for four pages (Figure 1). Suppose the first four page requests are for a, b, c, and d; these are brought into the (initially empty) cache in succession. Perhaps the next two requests are for d and a — these are already in the cache and so no further actions are required. The party ends when the next request is for page e — now one of the four pages a, b, c, d needs to be evicted to make room for it. Even worse, perhaps the subsequent request is for page f, requiring yet another eviction.

*©2014–2017, Tim Roughgarden.

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

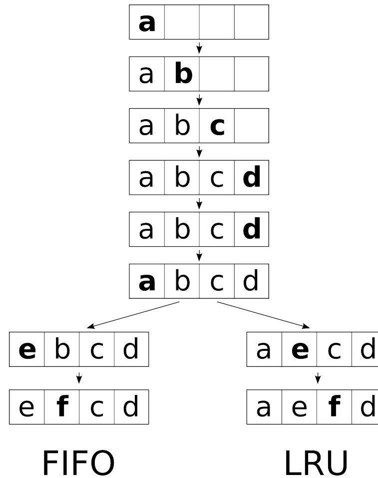

Figure 1: FIFO and LRU under the page request sequence a, b, c, d, d, a, e, f.

Evicting a page is exasperating because, for all you know, it's about to be requested again and you'll just have to bring it right back into the cache. As a thought experiment, if you were clairvoyant and knew the future — the entire sequence of page requests — then it is intuitive that evicting the page requested *furthest in the future (FIF)*, thereby delaying regret for the eviction as long as possible, is a good idea. Indeed, Bélády proved in the 1960s that the FIF algorithm achieves the minimum-possible number of page faults — some are inevitable (like the two in Figure 1), but FIF incurs no more than these. This is a classic optimal greedy algorithm.[1]

Without clairvoyance about the future page requests, in practice we resort to heuristic cache eviction policies. For example, the *first-in first-out (FIFO)* policy evicts the page that has been in the cache the longest. In Figure 1, FIFO would evict page a to make room for e, and b for f. Another famous caching policy is *least recently used (LRU)*, which evicts the page whose most recent request is further in the past. LRU behaves differently than FIFO in the example in Figure 1, evicting b for e and c for f (since a was requested right before e).

You should have learned in your systems classes that the LRU policy kicks butt in practice, and is certainly much better than the FIFO policy. Indeed, cache policy designers generally adopt LRU as the "gold standard" and strive to simulate it with minimal overhead.[2] There is strong intuition for LRU's superior empirical performance: request sequences generally possess *locality of reference*, meaning that a page requested recently is likely to be requested again soon. For example, if the pages contain the code of a program, and the program spends a lot of its time in loops, then a typical page request will be the same as some recent request. Another way to think about LRU is that it is attempting to simulate the optimal FIF algorithm, using the assumption that the near future will look like the recent past.

[1]Can you prove its optimality? It's not so easy.

[2]In most systems, explicitly keeping track of the most recent request to every page in the cache is too expensive to implement.

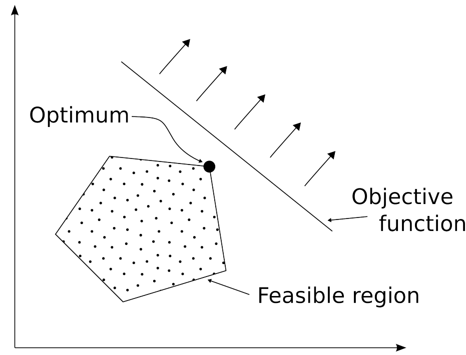

Figure 2: Visualization of a linear programming instance with $n = 2$.

Research Challenge 1 Develop theory to explain LRU's empirical superiority.

Challenge 1 is surprisingly non-trivial; the most satisfying solutions to date are only from the 2000s, and we will cover them in a few lectures. One immediate challenge is that FIFO and LRU are in some sense incomparable, in that each outperforms the other on some request sequences. (They are incomparable in a quite strong sense; see Homework #3.) For example, in Figure 1, if the request after f is for page c, then LRU looks like a bad idea in hindsight. If the following request is for a, then FIFO looks like the wrong choice.

One prerequisite for addressing Challenge 1 is to at least implicitly propose a model of data. After all, we believe LRU is better than FIFO because of properties of real-world data — locality of reference — and only mathematical models that reflect this have a chance of differentiating between them.

3 Linear Programming

This section reviews a famous example where worst-case analysis gives a wildly inaccurate prediction of the empirical performance of an algorithm. Recall that in a *linear programming* problem, the goal is to maximize a linear function ($\max \mathbf{c}^T\mathbf{x}$) subject to linear constraints (s.t. $\mathbf{A}\mathbf{x} \le \mathbf{b}$), or equivalently over an intersection of halfspaces. Here $\mathbf{c}$ and $\mathbf{x}$ are n-vectors; $\mathbf{b}$ is an m-vector; and $\mathbf{A}$ is an $m \times n$ matrix. In Figure 2, $n = 2$.

In practice, the linear programming algorithm of choice is called the *simplex method*. This is an ancient algorithm (1940s) but, amazingly, despite lots of advances in linear programming, suitably optimized versions of the simplex method remain the most commonly used algorithms in practice. The empirical performance of the simplex method is excellent, typically running in time linear in the number of variables (i.e., in n). What does worst-case analysis have to say about the algorithm?

Theorem 3.1 ([4]) *The worst-case running time of the simplex method is exponential in n.*

To add insult to injury, there are other linear programming algorithms that run in worst-case polynomial time but have far worse empirical performance than the simplex method.

The ellipsoid method [3], while beautiful and extremely useful for proving "polynomial-time in principle" results, is the most egregious of these.

Theorem 3.1 shows that worst-case analysis is an unacceptable way to analyze linear programming algorithms — it radically mischaracterizes the empirical performance of a great algorithm and, as a result, implicitly recommends (empirically) inferior algorithms to superior ones.

Research Challenge 2 Develop theory to explain the empirical performance of the simplex method.

This is another tough problem, with the most satisfying explanation coming from the invention of "smoothed analysis" in 2001. The take-away from this theory is that the simplex method provably runs in polynomial time on "almost all" instances, in a very robust sense — sort of like average-case analysis, but on steroids. We'll cover some of the greatest hits of smoothed analysis in future lectures.

4 Clustering

Our final motivating example concerns clustering problems. The goal here is "unsupervised learning," meaning finding patters in unlabeled data. The point of running a clustering algorithm is to identify "meaningful groups" of data points. For example, maybe you have mapped a bunch of images to Euclidean space (using bitmaps or some features), and the hope is that a clustering algorithm identifies groups that correspond to "pictures of cats" and "pictures of dogs."

Identifying "meaningful groups" is an informal goal, and some choice of formalization is necessary prior to tackling it by computer. A common approach is to specify a numerical objective function defined on clusterings — perhaps you've heard of k-means, k-median, correlation clustering, etc. — and then optimize it. What does worst-case analysis say about this approach?

Meta-Theorem 4.1 *Every standard approach of modeling clustering as an optimization problem results in an NP-hard problem.*

Thus, assuming $P \neq NP$, no polynomial-time algorithm solves any of these clustering-motivated optimization problems, at least not exactly on every instance.

In practice, however, fast clustering algorithms often find meaningful clusterings. This fact suggests the following challenge for theory.

Research Challenge 3 Prove that clustering is hard only when it doesn't matter.

In other words, there's no need to dispute Meta-Theorem 4.1 — we can agree that there are computationally intractable instances of clustering problems. But we only *care* about instances of clustering for which a "meaningful solution" exists — and in such special cases, it's plausible that the input contains extra clues, footholds that an algorithm can exploit to

quickly compute a good solution. This has been very active topic in algorithms and machine learning research during this decade, and we'll survey the highlights of this literature in a couple of weeks.

5 Provable Bounds in Machine Learning

The clustering challenge above is really a special case of a broader issue, which has become a very hot research topic over the past few years.

Research Challenge 4 Understand the unreasonable effectiveness of machine learning heuristics.

For example, in the training phase for supervised learning, why does gradient descent so often avoid getting stuck in local optima of non-convex problems? Why does stochastic gradient descent need so few passes to arrive at a good solution? Why do alternating minimization algorithms like the EM algorithm work so well for solving computationally hard problems? You may well have encountered similar mysteries in your own work.

The first goal of research on this challenge is to develop models and conditions under which such machine learning heuristics can be proved to work well. The second goal is to use such models to guide the design of new and better algorithms. We'll see a success story of this type in a different unsupervised learning problem (topic modeling).

6 Course Information

See course webpage: vitercik.github.io/bwca

7 Analysis of Algorithms: What's the Point?

Why bother trying to mathematically analyze the performance of algorithms? Having motivated the need for alternative algorithm analysis frameworks, beyond traditional worst-case analysis, let's zoom out and clarify what we want from such alternative frameworks.

7.1 Goals of Analyzing Algorithms

We next list several well-motivated and conceptually distinct reasons why we want rigorous methods to reason about algorithm performance. This course covers a lot of different definitions and models, and whenever we introduce a new way of analyzing algorithms or problems, it's important to be clear on which of these goals we're trying to achieve.

Goal 1 (Performance Prediction) The first goal is to explain or predict the empirical performance of algorithms. Frequently this goal is pursued for a fixed algorithm (e.g., the simplex method); and, sometimes, with a particular set of or distribution over inputs in mind (e.g., "real-world instances").

One motivation of Goal 1 is that of a natural scientist — taking an observed phenomenon like "the simplex method is fast" as ground truth, and seeking a transparent mathematical model that explains it. A second is that of an engineer — you'd like a theory that advises you whether or not an algorithm will perform well for a problem that you need to solve. Making sense of Goal 1 is particularly difficult when the performance of an algorithm varies wildly across inputs, as with the running time of the simplex method.

Goal 2 (Identify Optimal Algorithms) The second goal is to rank different algorithms according to their performance, and ideally to single out one algorithm as "optimal."

At the very least, given two algorithms A and B for the same problem, we'd like a theory that tells us which one is "better." The challenge is again that an algorithm's performance varies across inputs, so generally one algorithm A is better than B on some inputs and vice versa (recall the FIFO and LRU caching policies).

Note that Goals 1 and 2 are incomparable. It is possible to give accurate advice about which algorithm to use (Goal 2) without accurately predicting (absolute) performance — preserving the relative performance of different algorithms is good enough. Goal 1 is usually pursued only for a single or small set of algorithms, and algorithm-specific analyses do not always generalize to give useful predictions for other, possibly better, algorithms.

Goal 3 (Design New Algorithms) Guide the development of new algorithms.

Once a measure of algorithm performance has been declared, the Pavlovian response of most computer scientists is to seek out new algorithms that improve on the state-of-the-art with respect to this measure. The focusing effect triggered by such yardsticks should not be underestimated. For an analysis framework to be a success, it is not crucial that the new algorithms it leads to are practically useful (though hopefully some of them are). Their role

as “brainstorm organizers” is already enough to justify the proposal and study of principled performance measures.

7.2 Summarizing Algorithm Performance

Worst-case analysis is the dominant paradigm of mathematically analyzing algorithms. For example, most if not all of the running time guarantees you saw in introductory algorithms used this paradigm.[3]

To define worst-case analysis formally, let $\text{cost}(A, z)$ denote a *cost measure*, describing the amount of relevant resources that algorithm A consumes when given input z. This cost measure could be, for example, running time, space, I/O operations, the solution quality output by a heuristic (e.g., the length of a traveling salesman tour), or the approximation ratio of a heuristic. We assume throughout the entire course that the cost measure is given; its semantics vary with the setting.[4]

If $\text{cost}(A, z)$ defines the “performance” of a fixed algorithm A on a fixed input z, what is the “overall performance” of an algorithm (across all inputs)? Abstractly, we can think of it as a vector $\{\text{cost}(A, z)\}_z$ indexed by all possible inputs z. Most (but not all) of the analysis frameworks that we'll study summarize the performance of an algorithm by compressing this infinite-dimensional vector down to a simply-parameterized function, or even to a single number. This has the benefit of making it easier to compare different algorithms (recall Goal 2) at the cost of throwing out lots of information about the algorithm's overall performance.

Worst-case analysis summarizes the performance of an algorithm as its maximum cost — the ℓ_∞ norm of the vector $\{\text{cost}(A, z)\}_z$:

$$\max_{\text{inputs } z} \text{cost}(A, z). \tag{1}$$

Often, the value in (1) is parameterized by one or more features of z — for example, the worst-case running time of A as a function of the length n of the input z.

7.3 Pros and Cons of Worst-Case Analysis

Recall from your undergraduate algorithms course the primary benefits of worst-case analysis.

1. *Good worst-case upper bounds are awesome.* In the happy event that an algorithm admits a good (i.e., small) worst-case upper bound, then to first order the problem is completely solved. There is no need to think about the nature of the application domain or the input — whatever it is, the algorithm is guaranteed to perform well.

[3]Two common exceptions, which use average-case analysis, are the running time analysis of deterministic Quicksort on a random input array, and the performance of hash tables with random data.

[4]Figuring out the right cost measure to optimize can be tricky in practice, of course, but this issue is application-specific and outside the scope of this course.

2. *Mathematical tractability.* It is usually easier to establish tight bounds on the worst-case performance of an algorithm than on more nuanced summaries of performance. Remember that the utility of a mathematical model is not necessary monotone increasing in its verisimilitude — a more accurate mathematical model is not helpful if you can't deduce any interesting conclusions from it.

3. *No data model.* Worst-case guarantees make no assumptions about the input. For this reason, worst-case analysis is particularly sensible for "general-purpose" algorithms that are expected to work well across a range of application domains. For example, suppose you are tasked with writing the default sorting subroutine of a new programming language that you hope will become the next Python. What could you possibly assume about all future applications of your subroutine?

The motivating examples of Sections 2–4 highlight some of the cons of worst-case analysis. We list these explicitly below.

1. *Overly pessimistic.* Clearly, summarizing algorithm performance by the worst case can overestimate performance on most inputs. As the simplex method example shows, for some algorithms, this overestimation can paint a wildly inaccurate picture of an algorithm's overall performance.

2. *Can rank algorithms inaccurately.* Overly pessimistic performance summaries can prevent worst-case analysis from identifying the right algorithm to use in practice. In the caching problem, it cannot distinguish between FIFO and LRU; for linear programming, it implicitly suggests that the ellipsoid method is superior to the simplex method.

3. *No data model.* Or rather, worst-case analysis corresponds to the "Murphy's Law" data model: whatever algorithm you choose, nature will conspire against you to produce the worst-possible input. This algorithm-dependent way of thinking does not correspond to any coherent model of data.

   In many applications, the algorithm of choice is superior precisely because of properties of data in the application domain — locality of reference in caching problems, or the existence of meaningful solutions in clustering problems. Traditional worst-case analysis provides no language for articulating domain-specific properties of data, so it's not surprising that it cannot explain the empirical superiority of LRU or various clustering algorithms. In this sense, the strength of worst-case analysis is also its weakness.

Remark 7.1 Undergraduate algorithms courses focus on problems and algorithms for which the strengths of worst-case analysis shine through and its cons are relatively muted. Most of the greatest hits of such a course are algorithms with near-linear worst-case running time — MergeSort, randomized QuickSort, depth- and breadth-first search, connected components, Dijkstra's shortest-path algorithm (with heaps), the minimum spanning tree algorithms of

Kruskal and Prim (with suitable data structures), and so on. Because every correct algorithm runs in at least linear time for these problems (the input must be read), these algorithms are close to the fastest-possible on every input. Similarly, the performance of these algorithms cannot vary too much across different inputs, so summarizing performance via the worst case does not lose too much information.

Many of the slower polynomial-time algorithms covered in undergraduate algorithms, in particular dynamic programming algorithms, also have the property that their running time is not very sensitive to the input. Worst-case analysis remains quite accurate for these algorithms. One exception is the many different algorithms for the maximum flow problem — here, empirical performance and worst-case performance are not always well aligned (see e.g. [1]).

We should celebrate the fact that worst-case analysis works so well for so many fundamental computational problems, while at the same time recognizing that the cherrypicked successes highlighted in undergraduate algorithms can paint a potentially misleading picture about the range of its practical relevance.

7.4 Report Card for Worst-Case Analysis

Recall Goals 1–3 — how does worst-case analysis fare? On the third goal, it has been tremendously successful. For a half-century, researchers striving to optimize worst-case performance have devised thousands of new algorithms and data structures, many of them useful. On the second goal, worst-case analysis earns a middling grade — it gives good advice about which algorithm or data structure to use for some important problems (e.g., shortest paths), and bad advice for others (e.g., linear programming). Of course, it's unreasonable to expect any single analysis framework to give accurate algorithmic advice for every single problem that we care about. Finally, worst-case analysis effectively punts on the first goal — taking the worst case as a performance summary is not a serious attempt to predict or explain empirical performance. When worst-case analysis does give an accurate prediction, because of low performance variation across inputs (Remark 7.1), it's more by accident than by design.

8 Algorithm Design vs. Algorithm Analysis

There is a strong analogy between the organization of this course and that of most undergraduate algorithms courses. In an undergrad course like CS161, the primary goal is to develop a toolbox for algorithm *design*.[5] You learn that there is no "silver bullet" — no single algorithmic idea will solve every computational problem that you'll ever encounter. There are, however, a handful of powerful design techniques that enjoy wide applicability: divide and conquer, greedy algorithms, dynamic programming, proper use of data structures, etc. It's a bit of an art to figure out which techniques are best suited for which problems, but through practice you can hone this skill. Finally, these algorithm design techniques are taught largely through representative — and typically famous and fundamental — problems

[5] And also teach some necessary evils, like the vocabulary of asymptotic analysis.

and algorithms. This kills two birds with one stone: students both sharpen their ability to apply a given technique through repeated examples, and the examples are problems or algorithms that every card-carrying computer scientist would want to know, anyways. For example, divide-and-conquer is taught using the fundamental problem of sorting, and students learn famous algorithms like MergeSort and QuickSort. Similarly, greedy algorithms are taught using the minimum spanning tree problem, and the algorithms of Kruskal and Prim. Dynamic programming via shortest-path problems, and the algorithms of Bellman-Ford and Floyd-Warshall. And so on.

The goals and philosophy of CS264 are comparable in many respects; the primary difference is that the high-level goal is to supply you with a toolbox for algorithm *analysis*, and specifically ways to rigorously compare different algorithms. We'll cover a large number of analysis frameworks (instance optimality, resource augmentation, parameterized analysis, distributional analysis, smoothed analysis, etc.). Once again, no single way of analyzing algorithms is best for all computational problems, and it's not always clear which analysis framework is best suited for which problem. But the lectures and homework will introduce you to many such frameworks, each applied in several examples. And to maximize the value of these lectures, whenever possible we choose as examples problems and algorithms that are interesting in their own right (online paging, linear programming, clustering, machine learning heuristics, secretary problems, local search, etc.).

9 Instance Optimality

We conclude the lecture with the concept of *instance optimality*, which was first emphasized by Fagin, Lotem, and Naor [2].[6] This concept is best understood in the context of Goal 2, the goal of ranking algorithms according to performance. For starters, suppose we have two algorithms for the same problem, A and B. Under what conditions can we declare, without any caveats, that A is a "better" algorithm than B? One condition is that A *dominates* B in the sense that

$$\text{cost}(A, z) \leq \text{cost}(B, z) \tag{2}$$

for *every* input z. If (2) holds, there is no excuse for ever using algorithm B. Conversely, if (2) fails to hold in either direction, then A and B are incomparable and choosing one over the other requires a trade-off across different inputs.

Similarly, under what conditions can we declare an algorithm A "optimal" without inviting any controversy? Perhaps the strongest-imaginable optimality requirement for an algorithm A would be to insist that it dominates *every* other algorithm. This is our initial definition of instance optimality.

Definition 9.1 (Instance Optimality — Tentative Definition) An algorithm A for a

[6] This paper was the winner of the 2014 ACM Gödel Prize, a "test of time" award for papers in theoretical computer science.

problem is *instance optimal* if for every algorithm B and every input z,

$$\text{cost}(A, z) \le \text{cost}(B, z).$$

If A is an instance-optimal algorithm, then there is no need to reason about what the input is or might be — using A is a "foolproof" strategy.

Definition 9.1 is extremely strong — so strong, in fact, that only trivial problems have instance-optimal algorithms in the sense of this definition.[7] For this reason, we relax Definition 9.1 in two, relatively modest, ways. First, we modify inequality (2) by allowing a (hopefully small) constant factor on the right-hand side. Second, we impose this requirement only for "natural" algorithms B. Intuitively, the goal of this second requirement to avoid getting stymied by algorithms that have no point other than to foil the theoretical development, like algorithms that "memorize" the correct answer to a single input z. The exact definition of "natural" varies with the problem, but it will translate to quite easy-to-swallow restrictions for the problems that we study in the homework and in the next lecture.

Summarizing, here is our final definition of instance optimality.

Definition 9.2 (Instance Optimality — Final Definition) An algorithm A for a problem is *instance optimal with approximation c with respect to the set $\mathcal{C}$ of algorithms* if for every algorithm $B \in \mathcal{C}$ and every input z,

$$\text{cost}(A, z) \le c \cdot \text{cost}(B, z),$$

where $c \ge 1$ is a constant, independent of B and z.

Even this relaxed definition is very strong, and for many problems there is no instance-optimal algorithm (with respect to any sufficiently rich set $\mathcal{C}$); see Homework #1. Indeed, for most of the problems that we study in this course, we need to make additional assumptions on the input or compromises in our analysis to make progress on Goals 1–3. There are, however, a few compelling examples of instance-optimal algorithms, and we focus on these in the next lecture and on Homework #1.

References

[1] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[2] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003. Preliminary version in *PODS '01*.

[3] L. G. Khachiyan. A polynomial algorithm in linear programming. *Soviet Mathematics Doklady*, 20(1):191–194, 1979.

[7] Talk is cheap when it comes to proposing definitions — always insist on compelling examples.

[4] V. Klee and G. J. Minty. How Good is the Simplex Algorithm? In O. Shisha, editor, *Inequalities III*, pages 159–175. Academic Press Inc., New York, 1972.