CS5330: Randomized Algorithms

Experimental Randomized Algorithms

Due: March 8, 2019, 11:59pm

When designing a new randomized algorithm, there is an interplay between theory and experiment. From the theory, you can get insights about what might work. (For example, if you sample about $\log n$ times, you will get a reasonable estimate—for a variety of different types of problems.) However, some processes are difficult to analyze. In those cases, it is often useful to run experiments! Experiments can provide a variety of insights:

- Does the algorithm seem to do what you expected?
- Is it's performance reasonable? Does the performance in practice match the asymptotic in theory? How big are the constant factors?
- What should you be trying to prove? If you discover that the performance seems better than expected, perhaps there is a stronger theorem you can prove! If the performance is worse than expected, perhaps that means there is something wrong with your model.
- Experiments can help you to analyze your algorithm theoretically. Perhaps you discover that some property you did not expect holds.
- What modification/variants might perform *even better*? Perhaps you discover that by changing the algorithm in small ways, it works a lot better! This might suggest that there is a new and better algorithm to design.

The goal in this assignment is to experiment with a standard randomized algorithm and see what you can learn about it. Below you will find three different choices for the base algorithm—choose one. (You may also suggest an alternative; come talk to me.) After selecting the algorithm, there are then three parts:

1. Implement the basic algorithm in a standard (and efficient) manner and run baseline experiments. These experiments will provide a point of comparison for any modifications you make later. These experiments should establish both absolute performance as well as gauge how the algorithm scales.
2. Run experiments to answer the primary question I have asked. Experiment with the variations suggested here. Compare the results against the baseline experiments. Try to come up with a conjecture for *why* you are seeing the results you are seeing.
3. Pose a couple further questions about the algorithm/data structure in question. (I have suggested some possible secondary questions, but you are encouraged to think of your own.) These questions could be about variations in the algorithm, or about how the algorithm would perform in a different context (i.e., a different type of experiment). Ideally, these would be questions that have potentially theoretical answers, but are hard to analyze directly. Run

experiments to answer your questions, and develop conjectures as to what might (or might not) be true about your variations.

For the purpose of this assignment, think of yourself as an experimental scientist: ask a question, develop a hypothesis for what might be going on, and then run some experiments to answer your question and test your hypothesis. Then, develop some conjectures that explain the data you have seen. (Ideally, you could use your data to pose some theoretical conjectures which we could try to prove!)

You may notice that some algorithms may appear to have a more difficult implementation or more components to the primary question; in that case, you may choose to address only one or two secondary questions. On the other hand, if the primary question is easy to answer, then you might want to address a few more secondary questions. Overall, however, I am not particularly interested in *quantity* as I am in the questions you choose to ask. Interesting questions that yield insight into the algorithm and data structure are much more valuable than thousands of mundane experiments that reveal nothing new.

In the following three sections, I discuss three different problems, along with some experimental questions you may choose to answer about them:

- Multi-pivot QuickSort
- Cuckoo Hashing
- Backoff and Contention Resolution

Choose one, and submit a report on your results explaining what you have found about the algorithm you have chosen. Please write your report as a well-organized, self-contained document (i.e., so it can be read by anyone independent of this class). Please structure your report around the questions you are trying to answer, the experimental evidence you have accumulated, and the answers you have found.

Caveats:

You may need to read a little bit elsewhere to learn a litte bit more about these algorithms—however, you should not need much more background than we have discussed in this class already. (Of course, in your writeup do cite anything you read.) The goal of this assignment is *not* to read everything that has been done on a topic—it is to ask your own questions and run experiments to answer them. (Please do not submit a report that consists primarily of explaining experiments that others have already done.)

Also, notice that running good experiments is itself a difficult challenge. There are several things that can go wrong during an experiment. A few examples:

- The thing you are really measuring is not actually the thing you want to be measuring. Sometimes the performance may be dominated by unexpected costs (or mistakes) in your code, which does not allow you to observe the thing you are interested in. An example of this I saw recently involved trying to measure the performance of a parallel data structure; the experiment involved running 10,000 operations on the data structure. After each operation, the thread incremented a counter. Strangely, no matter the input to the data structure, the results were the same. It turns out that the performance of the counter was dominating the performance! The contention on the counter was the bottleneck, and the performance of the data structure was negligible compared to the cost of the counter (specifically, the hidden fact that it was acquiring a lock on the counter). Beware that you are actually measuring the performance of your data structure and not something else.

- Sometimes the performance depends a lot on what you choose to optimize (and what you do not). If there is something repeated many times in an inner loop, it may come to dominate the performance, and your results may depend heavily on whether your implementation of that inner function is naive or efficient.

- The test data makes a huge difference on the results. For example, many experiments typically run on random data. For example, you might choose a graph at random on which to test your coloring algorithm. However, random graphs have very different properties than real world graphs and your results will not tell you much about the real performance of the algorithm! Unless you are explicitly trying to test average-case performance, be careful in using randomly generated input data. (Similarly, beware of patterns in your data, e.g., it contains only even numbers, it contains only ages at most 80, etc.)

- The size of the input data makes a big difference. There are many algorithms that behave relatively differently on small examples and big examples. For example, an algorithm may seem reasonable on small data, but turn out to be way too slow on large data. Or alternatively, the algorithm may have big "startup" oerheads that result in seemingly slow running time for small data, but scale well and so do well with larger data sets. Be sure to test your algorithm on a variety of different sized inputs.

1 QuickSort

QuickSort is perhaps the most surprising example in the last ten years of experimental evidence leading to significant new theoretical insights—and significant improvement. QuickSort was invented in 1959 by Tony Hoare and has served as the standard sorting algorithm for most of computer science ever since. There have been a variety of optimizations since it was first invented (e.g., on how to do the partitioning and how to do the recursion), but the basic algorithm has stayed the same: choose a random pivot, partition the graph, and recurse on the two halves.

In 2009, a revolution occurred: Vladimir Yaroslavskiy proposed using *two pivots* instead of one. The theorists observed that this change yielded no real improvement: the running time remains $O(n \log n)$ with high probability (and in expectation). Why bother? Yaroslavskiy responded with experiments: two pivots are indeed faster than one!

This (experimental) discovery led to a much closer analysis of QuickSort, carefully counting the number of comparisons and thinking harder about caching effects. There has been a significant amount of work over the last several years analyzing why two pivots may be better than one, and attempting to understand what is really going on.

And it has also had a significant practical impact: multi-pivot QuickSort has now become the standard implementation of QuickSort across a variety of platforms!

Implement: As a first step, implement QuickSort and 2-pivot QuickSort as efficiently as you can and take some baseline measurements of performance. (For both, you should think hard about the right way to implement partitioning, which may have a significant impact on performance.) Part of the challenge of this problem is developing an efficient implementation.

Primary question: How many pivots is optimal? Why do more pivots help? Why do too many pivots hurt? In the process, you will likely want to count the number of comparisons and the number of swaps (or array accesses). If you implement partitioning efficiently, the asymptotic performance should remain the same, hence to answer these questions you will need to pay careful attention to constant factors.

Possible secondary questions: There are a variety of possible further questions you might ask, and you are encouraged to think of your own questions. Some possibilities are briefly listed here:

- What if the array has many duplicate elements? How does that change the situation? (How does that change your partitioning algorithm?) Does that change the optimal number of pivots? (Real data almost always has many duplicate elements.)
- Real data is often already sorted or almost sorted (with only a few items out of place). QuickSort is not particularly efficient on almost sorted data. Is multi-pivot QuickSort any better or worse on almost sorted data? does that change the optimal number of pivots?
- One theory as to why multi-pivot QuickSort is better has to do with caching effects. Unfortunately, it is hard to experiment with changing the size of a hardware cache. However,

you can simulate a software cache and vary the size. Does the size of the cache change the optimal number of pivots?

- Does the size of the key being sorted matter? For example, if you are sorting an array of strings (where the strings may be reasonably large) instead of an array of integers, does it change the results (e.g., the optimal number of pivots)?
- What abolut random binary search trees? If multi-pivot QuickSort is better than single pivot QuickSort, and if QuickSort is (essentially) equivalent to constructing a random binary search tree (where a random node is chosen for the root of the tree, the remaining keys are divided among the two subtrees, and the construction proceeds recursively), then would you expect a random k-ary search tree (i.e., with degree k instead of 2) to perform better than a random binary search tree? What about a treap with degree > 2? (Is it even possible to build such a thing efficiently? I don't know!) If more than one pivot helps, you would expect larger degree to help!

You are encouraged to think of your own questions to ask (and answer) about multipivot QuickSort and its implications.

Note on experiments: Do try to run experiments on a variety of inputs—not just random permutations—including inputs with duplicates, inputs that are nearly sorted (or reverse sorted), and inputs that derive from real data (e.g., all the words in the plays be Shakespeare on Project Gutenburg).

2 Hashing

One of the revolutions in hashing in the last ten years has been multichoice hashing: instead of simply hashing an item to single bucket, choose two or more buckets for that item. For example, in Cuckoo Hashing, there are two hash functions and each item is always placed in one of the two buckets indicated (kicking out other items as needed). There are several advantages to these schemes: They tend to yield faster queries in the worst-case, guaranteeing worst-case constant queries in the case of Cuckoo Hashing. They tend to be useful for solving other problems, like load balancing. And they tend to allow for a variety of variations that can improve performance in certain scenarios. For example, in Cuckoo Hashing, you can vary the number of items that can be stored in a bucket, as well as the number of hash functions used, as well as the order in which the Cuckoo Graph is searched.

For this question, we are going to think of a hash table with chaining. The table consists of a single large array $A[0, \ldots, m-1]$ of size m. (If you are going to insert m items, choose $m \geq n$.) For each item to be inserted, there are d hash functions $h_0, \ldots, h_{d-1}$ and the item will *always* be inserted at one of those d locations. There are two variants for the hash functions:

- **H1:** Each hash function maps to a random location in the range $[0, \ldots, m-1]$.
- **H2:** The array is divided into pieces of size m/d. Each hash function maps to a random location in the range $[0, \ldots, m/d-1]$, and hash function h_j maps to the jth segment of the array (i.e., if $h_j(x) = \ell$, then item x is assigned to bucket $j(m/d) + \ell$).

Given these two ways to choose the hash functions, you can use Cuckoo Hashing to decide which location to use: always put item x in the bucket indicated by $h_1(x)$. If the bucket at $h_1(x)$ exceeds some threshold τ, then take one (or more) item(s) from that bucket and move it/them to another bucket. E.g., if item y is in the same bucket as x, and if $h_a(y) = h_1(x)$, then move y to $h_{a+1}(y)$. (How do you choose which item y to move? You might want to choose the item y for which the value of a is smallest! Or maybe try choosing randomly? Or maybe try something else!)

For example, Cuckoo hashing (as we saw it in class) uses option H2 with $d = 2$ to specify hash functions (i.e., each each function maps to a separate array, which can be seen as two parts of one big array) and uses the Cuckoo rule with a threshold of $\tau = 1$ (i.e., if there is > 1 item in a bucket, it kicks one out).

Implement: Implement a Cuckoo Hash Table using both options for how to choose hash functions (i.e., both H1 and H2).

Primary question: Which performs better, H1 or H2? How should you choose which item to kick out? What threshold τ should you choose? How many hash functions should you use (i.e., what is the best choice of d)? How does performance change with different array sizes?

For your experiments, you will have to compare different costs (e.g., insertion, searching for items in the table, searching for items not in the table) for different types of data sets. You may also

want to consider several different metrics, e.g., average cost vs. worst-case cost of an operation. Another key choice you will have to make is which hash function to use. (Note: please do not only use randomly generated data; you may well want to find some real data, e.g., all the words in the plays by Shakespeare on Project Gutenberg.)

Possible secondary questions: There are a variety of possible further questions you might ask, and you are encouraged to think of your own questions. Some possibilities are briefly listed here:

- How does performance improve if you add a small *stash*, i.e., a secondary hash table that you can put items in when a problem occurs. For example, in Cuckoo hashing, you might choose to put items in the stask if they force a rebuild, or if they lead to $\Theta(\log n)$ length paths during insertion. Or you might put an item in the stash if a bin is getting too big. How big a stash do you need?
- How does the choice of hash function impact performance? (And does it change the answers to any of the questions previously answered?) For example, how does tabulation hashing compare to a simple modular hashing scheme? How do both compare to a real random function? How do these compare to the hash functions that are often used in practice?

3 Backoff Protocols

One of the most frequent uses of randomization in the real world is contention resolution: more than one device or process wants to access some resource, and only one can use it at a time. With a little bit of randomization, we can efficiently share the resource. A classic example is the 802.11 WiFi protocol's distributed coordination function, which uses *exponential backoff* to resolve collisions. Another common example is locks in concurrent systems, where backoff protocols can be used to minimize spinning.

For the purpose of discussion, I am going to imagine a collection of devices trying to access a shared resource (e.g., a wireless channel or a printer or whatever). We will assume that devices have synchronized clocks and that time proceeds in steps: 1, 2, 3, ... In each time step, a device can attempt to claim the resource. If it is the only device to try to use the resource in that step, then it succeeds and completes its task. If more than one device tries to use the resource at the same time, then they all fail.

The basic idea in a backoff protocol is that each device maintains a window W that determines how much it is going to backoff initially. Initially, when a device receives a request to claim the resource, it chooses a random time t in the window W. It waits $t-1$ time steps and then tries to claim the resource in time step t. If it fails, then it silently waits until the end of the window. At that point, it increases its window size W and repeats the procedure.

Depending on how you modify the window size W, you get a variety of different backoff protocols with different properties:

- *Linear backoff:* On every retry, set $W = W + c$, for some small constant c.
- *Binary Exponential backoff:* On every retry, double the window size: set $W = 2W$. Thus if the window starts out as size $W = 1$, then after k tries, it has a window of size 2^k. (Using techniques from Problem Set 3, you should be able to provide an upper bound on how long it will take n devices that all start at the same time to each get a turn to access the resource.)
- *General Exponential backoff:* On every retry, set $W = \alpha W$ for some constant $\alpha > 1$.
- *Log-backoff:* On every retry, set $W = W(1 + 1/\log(W))$. (Notice that the window doubles approximately after $\log W$ failures.)
- *Log-log-backoff:* On every retry, set $W = W(1 + 1/\log\log(W))$.
- *Backoff-backon:* Begin by setting $k = 1$. If $W > 1$, then set $W = W/2$. Otherwise, if $W = 1$, then set $W = 2^k$ and set $k = k + 1$.
- *Polynomial backoff:* After r tries, set $W = (r + 1)^\alpha$ for some constant α. (For example, if $\alpha = 2$, this is quadratic backoff.)

You can probably invent several more! (Notice that typicaly the protocol does not know the value of n, i.e., how many competitors there are for the resource.)

Implement: Your first task is to build a simple simulator so that you can test out different backoff protocols. To begin with, assume that n devices all begin at time step 0 and all of them want to access the resource. For each of the backoff protocols above, determine on average how long it takes for all the devices to gain access to the resource. Also, measure how many times each device has to retry (on average). Finally, look at how the performance scales with n.

Primary question: The main question is which backoff protocol works best under which circumstances. (You may choose to focus on a smaller set of backoff protocols that seem to have the best performance, and you may include variations that you have invented.) One way to evaluate this is to consider what happens when n processes all begin the protocol at the same time $t = 0$.

Now look at what happens under other arrival models:

- In each time step, one new device arrives with probability λ. (Examine different choices of λ.) This captures regular but slow arrivals.
- In each time step, the probability distribution on the number of devices that arrive is Gaussian. This captures slightly bursty arrivals. (Often Gaussian distributions are seen as reasonable models of reality.)
- In each time step, with probability $1/\sqrt{n}$, a cluster of $\sqrt{n}$ new devices arrive. Otherwise, with constant probability 1 device arrives. (This captures very bursty traffic.)

What is the worst case arrival distribution you can come up with? Which backoff protocol do you think works best? (Notice that when a device arrives at a later time $t > 0$, it begins with it usual initial window of size $W = 1$. Thus different devices will have different sized windows at the same time.)

Possible secondary questions: There are a variety of possible further questions you might ask, and you are encouraged to think of your own questions. Some possibilities are briefly listed here:

- Sometimes, it is not important that *every* device gets a chance to access the resource. Perhaps it is only important that *one* device gets to use the resource. (For example, if the devices are trying to broadcast a *DANGER* message, it suffices that one of them succeeds so that we all hear the warning!) Or alternatively, perhaps it is only important that half the devices get to use the resource. (We can tolerate some of the devices just failing.) Does this change which protocol you would recommend?
- Sometimes once a resource has been acquired, it takes more than one time slot to finish the task. For example, if I am using a backoff protocol to claim a wireless channel, then once I successful gain access to the channel, I will continue to broadcast for several time slots until I have finished sending my packet. Assume that it takes k time slots to finish using your resource. (The value k may be a random variable.) If one process has claimed the channel and a second process interrupts before the job is complete, then the process may have to start again! How does that change your protocol? How does that change your results?

- Some tasks are more important than others. If each task has a priority, how would that change your protocol? How would that change your results?
- Sometimes, attempting to claim the access is very expensive. Assume that your goal is to minimize the number of times each device tries and fails to acquire the resource. How does that change your results?
- Perhaps when there is a collision and more than one device tries to use the resource, then the resource has to be "reset" and cannot be used for the following k time slots. (This makes a collision very expensive!) How does this change your results?