CS5234: Combinatorial and Graph Algorithms Lecture 1

Introduction and How to Sketch a Graph

Lecturer: Seth Gilbert *August 11, 2016*

Abstract

Today, we begin with an overview of the goals of this module, and discuss the (tentative) plan for the semester. We then begin by looking at three related problems: "sketching" a graph, sampling from a stream, and finding the connected components in a dynamic graph.

1 Introduction

The goal of this semester is to better understand how we deal with graphs (and datasets) that are very, very big. We will explore how to adapt classical algorithms, introducing a variety of tools for building efficient algorithms that can handle data at scale.

What is different in this module? As an undergraduate, you take several modules covering basic algorithms and data structures; you study the basic problems of computer science, e.g., searching, sorting, trees, and graphs; and you learn the classical solutions to these problems, e.g., binary search, quicksort, red-black trees, Dijkstra's algorithm, etc. Most of what you learn in those classes was invented/discovered prior to 1985 (to choose an arbitrary date), and they form an indispensable toolset for the computer scientist.

Eventually, you reach a point where you are solving your own problems, building your own systems, writing your own code, and developing your own algorithms. And often those classical algorithms are no longer entirely sufficient.

- *Scale:* The problems are bigger. A graph with 1000 nodes is easy to deal with. A graph with one billion nodes is not. In many ways, this question of scale motivates much of what we are talking about this semester.
- *Where is the data?* The data is often not provided in a simple and accessible format. It may be distributed among different servers. It may be provided on-line as a stream of information. There may be noise in the dataset.
- *Dynamic world:* The data is no longer static. For example, instead of working with a fixed, unchanging graph, the graph may change over time. It is no longer sufficient to simply solve a problem once.
- *Context matters:* The data comes from somewhere, and that matters. Does it represent a social network or a wireless network or a game? There are often special properties and structure of the data that you can leverage.

Let me give you an example that I hope illustrates the flavor of the questions we will think about this semester. Let's take the problem of finding a minimum spanning tree (MST): given a weighted, undirected graph $G = (V, E)$, find a spanning tree of the graph that is of minimum weight.

- *Algorithms 101:* In an introductory algorithms class, you typically learn about two algorithms to find an MST: Prim's Algorithms and Kruskal's Algorithm. Both of these algorithms solve the problem efficiently in $O(m \log n)$ time, where m is the number of edges and n is the number of nodes. This seems pretty fast—until you want to process the Facebook graph. Facebook has approximately 1.7 billion users with about 300 friends/user. Assuming your algorithm takes exactly $m \log n$ time (i.e., ignoring constant factors and overhead), and assuming you can execute 2.5 billion instructions per second (e.g., 2.5GHz), and ignoring all the overhead associated with memory/disk access (e.g., paging/caching), this would take about two hours. In practice, you would be lucky if it finished in 20 hours.

- *Special structure:* However, maybe we can do better than Prim's or Kruskal's. Where did the graph come from? Is it from a social network? Is it planar? Can we leverage any of these properties to build a faster algorithm? For example, if the graph is planar, we can build an MST in $O(n)$ time.
- *Randomization:* Can we come up with a faster algorithm than Prim's or Kruskal's for general graphs, i.e., without relying on special structure? Yes! Using randomizatinon, we can develop an MST algorithm that runs in $O(m)$ time. In theory, that would reduce the two hours above to about four minutes! (Yes, at these scales, log factors matter, even though we often ignore them.) In practice, it is not as clear how much improvement this yields, since the randomized algorithm is significantly more complicated. There is a good open question here: can we find a sufficiently simple $O(m)$ MST algorithm to make it practical?
- *Approximation:* If $O(m)$ is not fast enough, perhaps we can find an even faster algorithm, if we can live with an approximate solution. While we need linear time to find an exact, perfect solution, we can find a "good enough" estimate in less time. Amazingly, we can estimate the the weight of the MST of a graph with maximum degree d and maximum edge weight w in approximately $O(dw \log(dw))$ time—no matter how big the graph is!
- *Streaming:* What if you have only limited access to the graph data? Instead of random access, you get to start at the beginning and read through all the edges in some order: $e_1, e_2, e_3, \ldots, e_m$. That is, you get to see each edge exactly once. (Often you do not have free access to the data due to the way in which it is stored, whether remotely, on disk, distributed, or whatever. This type of access pattern is often better for cache use as well, since you access the data in order.)After reading through all the edges, you have to output an MST. Without storing the entire graph, can you still find an (approximate) MST? How much space do you need?
- *Dynamic:* What if the MST is changing, i.e., you are adding and deleting edges (e.g., Facebook friends) over time? Again, imagine that the data is arriving in a stream as a sequence of requests to add and remove edges from the graph. When all the requests are completed, your job is to calculate an MST of the final graph.
- *Caching:* As data gets larger, memory performance becomes a limiting feature. The data is too big to store in memory. The graph is now stored on disk, and transfering it to memory is expensive. How do you find an MST while minimizing the disk access costs?
- *Parallel/GPU:* As the graph gets big, it becomes increasingly impractical to run your algorithm on a single core of a single processor. What advantages can we get from parallelism? Can we take advantage of many cores to find a faster solution? What about GPUs?
- *Distributed/MapReduce:* An alternative pradigm for speeding up your computation is to use a distributed cluster of machines. One of the dominant paradigms for implementing this is MapReduce (e.g., as in Hadoop). How fast can we find an MST using MapReduce?

The goal this semester to develop a set of tools you can use to adapt graph algorithms these types of scenarios, when the graph is very large and you are no longer able to simply apply classical solutions.

2 Graph Connectivity

Today, we will start with a simple problem: graph connectivity. Imagine you have a very large graph with billions of edges. The graph is sufficiently large that you cannot store it all in main memory. You want to decide whether the graph is connected, the number of connected components, etc. (Next week, we will extend some of these ideas to the more general problem of finding a minimum spanning tree.)

More formally, assume you are given an unweighted, undirected graph $G = (V, E)$. We want to decide whether it is connected, i.e., for every $u, v \in V$, is there a path in the graph from u to v? If it is not connected, then determine how many connected components it has. And, perhaps, you would also like to be able to answer questions about whether pairs of nodes are connected, e.g., questions of the form, "Are nodes u and v connected in G or not?")

Traditionally, this is one of the easiest of graph problems to answer. Simply run a depth-first-search (or breadth-first-search) on the graph, identifying each of the connected components. If the graph has n nodes and m edges, then the running time is $O(n + m)$. However, once the graph is sufficiently large, even a "simple" algorithm like this may become too expensive.

We will explore three variants of this problem that demonstrate some of the challenges we will face this semester:

1. *Fast estimation:* Can we decide *faster* whether the graph is connected? And can we estimate the number of connected components without having to explore the entire graph? We will see how to test if a graph with degree d is connected (or almost connected) in time $O(1/\epsilon^2 d)$ time, where ϵ is an error parameter. Similarly, we will see how to approximate the number of connected components in time $O(d/\epsilon^3)$.

2. *Streaming updates:* Instead of being given the graph (which is too large to store), we are given a stream of *updates* adding and removing edges from the graph. We will show how to maintain a data structure using only $O(n \log^3(n))$ space (i.e., much smaller than the $O(n + m)$ space needed to store the whole graph) that will allow us to answer questions about connectivity.

3. *Distributed data:* Imagine that the nodes of the graph are users, distributed around the world. Each user has an address book identifying their friends (i.e., the edges in the graph). You are allowed to retrieve a small amount of information from each user, and then need to calculate whether the graph is connected. The goal is to minimize the amount of information sent to you by each user. Notice that this problem seems very difficult: there may be only one edge connecting two components of the graph; one of the two users must send you information about that edge. And yet neither user knows which is the critical edge. How is it possible for them to send you anything less than their entire neighbor lists? (In fact, this will use the exact same solution as the streaming algorithm.)

These problems involve some neat algorithmic techniques, and they show how even very simple problems can become quite interesting in different settings. We will also see how to build on these algorithms for connectivity to solve the (approximate) minimum spanning tree problem.

3 Streaming Data

Let us focus now on data streams. Imagine that you are receiving a stream of input data $S = \langle s_1, s_2, s_3, \ldots \rangle$. Each s_i represents some new information, and you are allowed to process each s_i once. (Some streaming algorithms are designed for multiple passes, i.e., you are allowed to repeat the stream S multiple times. Today, we will focus on single-pass algorithms.)

This type of streaming model captures several common situations:

- *Server handling requests:* Imagine you are implementing a server that is processing requests that are sent to it over the network. Each element in the stream is a request.

- *Data on external storage:* Imagine reading a very large dataset on disk (or on a distributed storage system). The entire dataset is so large that it cannot fit in memory. Retrieving the data is (relatively) slow. In order to minimize access costs, the goal is simply to read the data (in order) once. (Reading the data in-order maximizes cache-performance, and also allows pre-fetching to work well.)

- *Sensor data:* Imagine you have a sensor network continuously collecting data on the world. This data is created as a stream over time that you need to process as it arrives.

There are several examples of streaming problems:

- The stream is a sequence of temperature readings from a sensor $S = \langle t_1, t_2, \ldots, t_k \rangle$. You want to calculate the average temperature, $(1/k) \sum(t_i)$. Or, perhaps you want to calculate the average of the temperature readings in the past hour.
- The stream is a sequence of grades from all the students at NUS. You want to calculate the median grade, as well as the 80^{th} percentile.
- The stream is a sequence of names of all the children born in China in the last year. You want to identify the 10 most common names.

The simplest solution to any streaming problem is simply to store all the input data in the stream, as it arrives. Once the stream is complete, you can simply perform the requisite computation using a simple classical algorithm. This, however, may not be feasible since the size of the stream may be very large. The goal is to solve the problem at hand using a minimum amount of space.

Typically, if you want to solve a streaming problem exactly (with no error), you will need to store a very large amount of information. Thus we often relax our requirements, instead being satisfied with an approximate solution.

Here we discuss a few common streaming problems in more detail, before proceeding to develop a streaming algorithm for one particular problem that will be useful in the context of graph connectivity.

3.1 Counting Distinct Elements

A classical streaming problem involves counting the number of distinct elements in a stream. For example, imagine the stream $S = \langle s_1, s_2, \ldots \rangle$ where each $s_i \in [1, n]$, i.e., is an integer between 1 and n. The goal is to determine how many distinct integers are in the stream. For example, if the stream consists of $S = \langle 4, 5, 4, 7, 4, 8, 4 \rangle$, then the number of distinct elements is 4.

If the goal is to find the exact number of distinct elements, then it requires $\Omega(n)$ space, i.e., there is no way to solve the problem without storing a lot of information about the stream. However, we can find an estimate of the number of distinct elements much more efficiently: using $O(\log(1/\delta) \log^2 n/\epsilon^2)$ we can guarantee that with probability at least $(1 - \delta)$, our estimate is within a factor of $(1 + \epsilon)$ of the right answer. (This solution uses the technique of Flajolet and Martin. In fact, we can do even better.)

Imagine an n-dimensional vector $v = [v_0, v_1, \ldots, v_n]$ where v_i is the number of times that i appears in the stream. For example, the stream $S = \langle 4, 5, 4, 7, 4, 8, 4, 7 \rangle$ represents the vector $v = [0, 0, 0, 4, 1, 0, 2, 1]$.

In this case, counting the number of distinct elements in the stream is equivalent to finding $||v||_0$, i.e., the $L0$-norm of v. Thus this is often referred to as $||L||_0$ estimation.

In general, there exist good streaming algorithms for other norms as well, e.g., for $||L||_1$ estimation (i.e., the sum of the absolute value of the entries) and $||L||_2$ estimation (i.e., the sum of the values squared). (These algorithms work in more general models that allow negative numbers as vector entries.)

3.2 Index Query

Imagine instead that the goal is to answer queries of the form, "how many times does '4' appear in the stream?" That is, we want to know the number of times a given value appears. Equivalently, we want to estimate the value of v_i for each i.

In fact, often we assume the stream allows both increments and decrements. That is, each s_i in the stream is either of the form:

- $(x, +1)$, which adds one to v_x, or

- $(x, -1)$, which subtracts one from v_x.

For example, imagine the stream consists of

$$S = \langle (4,+1), (5,+1), (4,-1), (5,+1), (7,+1), (7,-1), (7,+1), (7,+1), (7,+1) \rangle,$$

where we indicate "increment" with a '+' and "decrement" with a '-'. In this case, the final vector looks like $v = [0,0,0,0,2,0,3]$; a query of '4' returns 0 and a query of '5' returns 2.

3.3 L0-Sampling

Sometimes, instead of finding the number of distinct elements, all we need is to identify at least one of the elements in the stream. That is, we want an algorithm that allows us to choose one of the elements uniformly at random (or *almost* uniformly at random) from the stream. For a stream

$$S = \langle 4,4,5,4,4,4,4,5,7,7,4,7,4 \rangle,$$

the goal is to return each of the values $\{4,5,7\}$ with probability $1/3$.

As before, the more interesting case is where the stream allows both increments and decrements. In this case, the goal is to choose an element uniformly at random from the non-zero elements of the resulting vector. For example, in the stream

$$S = (4,+1), (5,+1), (4,-1), (5,+1), (7,+1), (7,-1), (7,+1), (7,+1), (7,+1),$$

elements 5 and 7 should each be selected with probability $1/2$. (Element 4 should not be selected because $v_4 = 0$ in the final vector.)

3.4 Graph Sketches

A common solution to many streaming problems is to use a "sketch." A *sketch* is simply a small representation of a larger dataset. Given some large dataset D, a sketch is a function f such that $f(D)$ is significantly smaller than D, and yet captures some of the essential properties of D. (We will not be overly formal about our definition of a sketch here.) That is, a sketch is a small summary of a dataset.

Often, streaming algorithms use sketches. As you see the stream $D = \langle d_1, d_2, \ldots \rangle$, you can construct the sketch $f(D)$. You can then use the sketch to compute whatever queries are needed on the stream.

We often want one additional property from a sketch: the ability to merge to sketches. Imagine you have two datasets: D_1 and D_2. Then we want some mechanism to merge sketch $f(D_1)$ and sketch $f(D_2)$ to construct a new sketch $f(D_1 + D_2)$. (Here $D_1 + D_2$ refers to the joint dataset that contains both D_1 and D_2.)

A sketch with this mergable property can be easily used to sketch a stream. Imagine you have seen

$$S = \langle s_1, s_2, s_3, s_4 \rangle,$$

and you have already calculated $f(S)$. When you see the next element in the stream, you can calculate $f(\langle s_5 \rangle)$, and then merge the two sketches $f(S)$ and $f(\langle s_5 \rangle)$ to calculate the sketch of the entire stream $f(\langle s_1, s_2, s_3, s_4, s_5 \rangle)$.

We will be particularly interested in *linear* sketches, i.e., where we can simply add the two sketches $S(D_1) + S(D_2)$ to produce the sketch of the joint dataset $S(D_1 + D_2)$.

We are going to construct a *graph sketch* that provides a succinct summary of a graph—or a subgraph. More precisely, given a set of nodes V and a set of edges E that may or may not connect nodes in V, the sketch function will yield a small poly-logarithmic sized sketch that can be used to identify one outgoing edge from the graph—if any exists—or the output 'NONE' if all the edges are either contained within V or outside V.

4 Building an $L0$-Sampler

In this section, we will describe how to build an $L0$-sampler. Assume that we are processing a stream $S = \langle s_1, s_2, \ldots, s_T \rangle$ of T items, where each $s_j = (x_j, a_j)$; each x_j is an integer in $[1, n]$ and each a_j is either $+1$ or -1. (Obviously, we can generalize to arbitrary amounts, but for today we are only interested in incrementing and decrementing by one.)

Given a stream S, let v be the vector constructed by following all the increment and decrement instructions in the stream. We define $v_j = \sum_{i:x_i=j} a_i$, i.e., the jth position in the vector is the sum of the a_i where $x_i = j$. We will also assume that at all points in the stream, every component in the vector is $\leq n$.

Finally, we fix some error probability $\epsilon > 0$. Our goal is to build an $L0$-Sampler that is correct with probability at least $1 - \epsilon$.

We will say that a stream is s-sparse if the resulting vector v has at most s non-zero entries. (That is, there are at most s values in $[1, n]$ for which the increments and decrements do not cancel out.)

Our first goal is to build a mechanism for compressing (and recovering) sparse vectors. That is, given a vector v that is s-sparse, we want a mechanism for storing it in much less than $\Theta(n)$ space, while still being able to recover the complete vector correctly. This problem of sparse vector recovery is actually immensely important in a variety of contexts, e.g., it forms the basis for compressive sensing.

4.1 Simple 1-Sparse Sampler

Assume, for now, that the stream S is 1-sparse. In this case, our goal is to identify *the single* non-zero element in the stream. If the stream S is *not* 1-sparse, we should return an error. For example, if the vector $v = [0, 0, 0, -7, 0, 0, 0]$ then we should return 4 (i.e., the index of the non-zero element in the vector). If the vector is $v = [0, 2, 0, 7, 1, 0, 0]$, then we should return an error. If the vector is empty (i.e., all zeros), then we should indicate that the vector is empty.

There is a very simple strategy for building a 1-sparse sampler that uses $O(\log n)$ space and correctly detects errors (i.e., empty or overfull vectors) with probability at least $1 - \epsilon/n^2$. (We choose the error probability to be ϵ/n^2 as this will be useful later.) We maintain to integer variables to represent the stream:

$$\begin{aligned} weight(S) &= \sum_j (a_j) \\ sum(S) &= \sum_j (x_j \cdot a_j) \end{aligned}$$

Recall that the stream consists of items (x_j, a_j), where x_j is the index of the vector and a_j is either 1 or -1, depending on whether the index is being incremented or decremented. Thus the $weight(S)$ is simply the sum of the values of the vector; in $\sum(S)$, the values are weighted by their position in the vector. For example, imagine the following stream:

$$S = \langle (5, +1), (3, +1), (5, -1), (2, +1), (6, +1), (9, +1), (1, +1), (1, +1), (6, +1) \rangle$$

Notice that this stream is not 1-sparse, and it represents the vector $[2, 1, 1, 0, 0, 2, 0, 0, 1]$. In this case, $weight(S) = 2 + 1 + 1 + 2 + 1 = 7$, and $sum(S) = 2 * 1 + 1 * 2 + 1 * 3 + 2 * 6 + 1 * 9 = 28$.

The weight and the sum can be calculated easily as the requests are streamed, simply adding the next element in the stream to each. Also this sampler is really a linear sketch: if for two streams S_1 and S_2 we have calculated $weight(S_1)$ and $weight(S_2)$, then we can clearly conclude that $weight(S_1 + S_2) = weight(S_1) + weight(S_2)$; the same holds for the sum.

Assume vector v is 1-sparse, and assume that i is the (unique) index of the vector v that is non-zero. Then all the values associated with other indices cancel out, leaving simply $weight(S) = v_i$ and $sum(S) = i \cdot v_i$. We can thus identify the non-zero index by calculating $sum(S)/weight(S)$.

There is one remaining problem: if the stream is not 1-sparse, it may return an incorrect answer. If the vector is empty (i.e., all zero), then the sum and weight will both be zero. Thus it is easy to identify an all-zero vector. However, if the vector is not 1-sparse, then this strategy may return an incorrect answer. In the example above (which is not 1-sparse), recall that the $sum(S) = 28$ and the $weight(S) = 7$, and so the sampler might return index 4. However, the index $v_4 = 0$, and so that would be an invalid answer for the sampler.

We need a fingerprint that can ensure that the result is correct. Let P be a sufficiently large prime (e.g., $> n^3/\epsilon$). Choose z randomly in $[0, P-1]$. As the algorithm progresses, along with maintaining the weight and sum, also maintain:

$$fingerprint(S) = \sum_j (a_j \cdot z^{x_j}) \mod P$$

When the stream terminates, we calculate:

$$\begin{aligned} i &= sum(S)/weight(S) \\ error &= fingerprint(S) - (weight(S) \cdot z^i \mod P) \end{aligned}$$

If $error \neq 0$, then we report an error (i.e., the vector v is not 1-sparse). Otherwise, we return i.

Notice that if the vector is 1-sparse, then it is always correct: in the fingerprint, all the components not associated with index i cancel out, and hence the fingerprint is correct.

Claim 1 *If the vector v is 1-sprase, the algorithm above returns the correct index with probability 1.*

It remains to argue that when the vector is not 1-sparse, then an error is reported, most of the time.

Claim 2 *If v is not 1-sparse, the probability of detecting a failure is at least $1 - \epsilon/n^2$.*

Proof Assume, for a moment, that we do not choose z until after the stream is complete. Instead, we simply store $fingerprint$ as a polynomial in z. (Of course, this is not an efficient way to store the $fingerprint$. We are simply thinking about this for the purpose of analysis.) Notice, then, that if all the terms do not cancel out (i.e., if v is not identically zero and if v is not 1-sparse), then $error$ is a polynomial in z of degree at most n. (If v is 1-sparse, then all the terms do cancel and $error$ is identically zero.)

Since it is a polynomial of degree at most n, we know that this polynomial has at most n roots, i.e., at most n values of z where $error = 0$. Since we chose z at random from the range $[0, P-1]$, the probability that we randomly selected one of those zeros is n/P.

Since we chose $p > n^3/\epsilon$, this ensures that the probability that we fail to detect an error is at most $\epsilon n/n^3 \leq \epsilon/n^2$. □

Finally, we examine the space used by this sampler:

Claim 3 *The total space used by the 1-sparse sampler is $O(\log(n/\epsilon))$.*

The maximum value for $weight$ is n^2, the maximum value for sum is n^3, and the maximum value for $fingerprint$ is n^3/ϵ. (Recall that each index of the vector can be at most n.) Each of these requires at most $5\log(n/\epsilon)$ bits, and so the total space needed is $O(\log(n/\epsilon))$.

4.2 s-Sparse Sampler

Next, we generalize the 1-sparse sampler above to the case where the vector v is s-sparse for some fixed value s. In this case, we want the following behavior:

- If the vector is s-sparse, then it returns one index i that is non-zero.
- If the vector is not s-sparse but contains at least one non-zero index, then the sampler either returns one index i that is non-zero, or it returns an error.
- If the vector contains no non-zero indices (i.e., it is empty), then it returns empty.

The s-sparse sampler should operate correctly according to these results (and not return an index that is not non-zero) with probability at least $1 - \epsilon/n^2$.

In fact, the s-sampler that we will describe is more powerful: it can fully reconstruct any s-sparse vector, returning the value of *every* non-zero index. If the vector is s-sparse, it will return *all* the non-zero indices.

Assume we already know how to construct a sampler for 1-sparse vectors, as above. We refer to such a sampler as an S^1 sampler.

Choose k random hash functions $h_\ell : [1, n] \to [1, 2s]$ that maps integers in the range $[1, n]$ to integers in the range $[1, 2s]$. We will assume, for today, that h is a perfectly random function, i.e., for each i, $h_\ell(i)$ is mapped uniformly at random to a value in the range $[1, 2s]$. And we assume that we can store each hash function in $O(\log n)$ space. In reality, if the hash function were perfectly random, then it would require much more space! (For example, it would take $O(n \log s)$ space to store such a hash function.) However, we need only limited independence and so there do exist (provably) good hash functions that both use little space and also provide enough randomness.

We now construct $2sk$ of the S^1 samplers: $S^1_{\ell,r}$ where $\ell \in [1, k]$ and $r \in [1, 2s]$. For each item in the sequence $s_j = (x_j, a_j)$, we map the item s_j to the samplers: $S^1_{1,h_1(x_j)}, S^1_{2,h_2(x_j)}, \ldots, S^1_{k,h_k(x_j)}$. That is, item s_j in the stream is mapped to exactly k different samplers, each determined by the proper hash function.

Notice that each hash function effectively partitions the stream into $2s$ smaller sub-streams. Since there are k different hash functions, the stream is partitioned in k different ways. A given sampler $S^1_{\ell,r}$ is given a substream of the original stream containing all the indices that h_ℓ maps to r. (It is important, however, that a given index in the vector is always mapped to the same substream.)

The initial stream is s sparse. We claim that for every index i at which v is non-zero, at least one of the sub-streams is 1-sparse with good probability. That is, at least one of the sampler $S^1_{\ell,r}$ where $h_\ell(i) = r$ is 1-sparse—meaning that i is the only non-zero index mapped to r by h_ℓ.

Claim 4 *If v is s-sparse, then for every index i at which v is non-zero, there exists some $\ell \in [1, k]$ such that the sub-stream assigned to $S^1_{\ell,h_\ell(i)}$ is 1-sparse, with probability at least $1 - s/2^k$.*

Proof Fix an index i and a hash function ℓ. Recall that there are at most $s - 1$ other non-zero indices in v. Hence there are at most $s - 1$ of the S^1 samplers in the set $S^1_{\ell,\cdot}$ that have at least one non-zero index in $[1, n]$ mapped to them. (Here the set $S^1_{\ell,\cdot} = \{S^1_{\ell,r} : r \in [1, 2s]\}$.) Thus, the probability of a "collision" (i.e., hashing i to an S^1 sampler that has some other non-zero index mapped to it) is at most $(s - 1)/2s < 1/2$. That is, with probability at least $1/2$, the sub-stream mapped to the sampler $S^1_{\ell,h_\ell(i)}$ is 1-sparse.

There are k such hash function, each of which is independent, and hence the probability that they all fail is at most $1/2^k$. Taking a union bound over the s non-zero indices, the lemma follows with probability at least $1 - s/2^k$. □

We now fix $k = 2 \log (n/\epsilon)$, and show that this yields the desired performance:

Claim 5 *With probability at least $1 - \epsilon/n$, the s-sparse sampler behaves as follows: if the vector v is s-sparse, it returns exactly the set of non-zero indices in v; if the vector v is not s-sparse, it either returns an error or it returns some subset of the non-zero indices in v. This holds assuming s is a constant $< \sqrt{n}$ and $\epsilon > 1/2^{\sqrt{n}}$.*

Proof If the stream is s-sparse, by the previous claim, we know that with probability at least $1 - s/2^k$, each non-zero index is assigned to a 1-sparse sampler that handles a 1-sparse substream. Therefore, in this case, all non-zero indices will be discovered.

There are $2ks$ 1-samplers, so we also know that with probability at least $1 - (2ks\epsilon/n^2)$, all the 1-sparse samplers operated correctly. This implies that they will not return any indices that are *not* non-zero.

Combining these two results, we see that the claim holds with probability at least $1 - (s/2^k + (2ks\epsilon/n^2))$ (by a union bound).

Since we chose $k = 2\log(n/\epsilon)$, this yields a probability of error at most $\epsilon^2 s/n^2 + 4s\log(n/\epsilon)\epsilon/n^2$. Thus, we can bound the probability of error with ϵ/n (as long as s is a constant sufficiently less than n, and ϵ is not exponentialy small in n). □

Claim 6 *The S^s sampler uses space at most $O(s\log^2(n/\epsilon))$.*

Notice that since the s-Sampler just consists of a fixed size array of 1-Samplers, and the 1-Samplers are linear sketches, we can conclude that the s-Sampler is also a linear sketch of the stream.

One other note: there is no guarantee here that if the stream is *not* s-sparse, then it returns an error. It guarantees that it only returns correct non-zero indices (because the 1-samplers are correct). However, by random chance, some of the 1-samplers may be properly 1-sparse even if the overall stream is not s-sparse. For our purposes today, this does not matter. However, if you want to return an error if the vector is not s-sparse, you can do a similar fingerprint check as in the 1-sampler. This fingerprint can rely on the fact that if the recovery procedure completed correctly, it should return no more than s non-zero indices, and those should be *all* the non-zero indices.

It is, though, useful to be able to distinguish an empty vector from a non-empty vector. In this case, that is not hard: the vector is empty if and only if the S^1 samplers are all empty.

4.3 General Sampler

At this point, we know how to build a sampler as long as the vector being constructed is sparse. However, the vector v may not be sparse. In fact, we have no idea the density of the vector v: it may be very sparse or very dense. The solution to this problem is sampling!

Intuition. Let us try guessing how many non-zero entries the vector has. For example, what if we think that the vector has about k non-zero entries? Then imagine we choose each index of the vector with probability $1/k$, ignoring the others. Let v_k be the reduced vector that we get from sub-sampling the indices of the original vector v. If the vector v contains anywhere from k to sk entries, then we would expect v_k to have somewhere between 1 and s entries. We could then use an s-Sampler on the resulting vector v_k to identify one element.

Building the sampler. Therefore, we construct the following $L0$-sampler:

- For $k = 1, 2, 4, \ldots, n$, let h_k be a random (hash) function that maps each value in the range $[1, n]$ to $[0, 1]$ where the probability that $h(i)$ maps to 1 is $1/k$. (Alternatively, you might imagine that h_k maps each integer input to the range $[1, k]$ uniformly at random.) As before, we assume that such perfectly random hash functions exist, and that we can store them efficiently, e.g., in $O(\log n)$ space per hash function.
- We instantiate $\log n$ s-sparse samplers: $S_1^s, S_2^s, \ldots, S_{\log n}^s$. Here we let $s = \Theta(\log(1/\epsilon))$. Later in the analysis, the constant will be fixed more precisely.
- For each item (x_i, a_i) in the stream, for each $k = 2^j$, if $h_k(x_i) = 1$, then we process (x_i, a_i) with sampler S_j^s. Notice that each item in the stream may be processed by all $\log n - 1$ samplers, depending on the random hash functions. We refer to the vector processed by sampler S_k^s as v_k.

- Process (x_i, a_i) with sampler S_1^s. Every element is processed by this sampler.

To find a non-zero index, examine all the s-sparse samplers and take all the indices that are returned. Choose one of these uniformly at random.

Claim 7 *The total space for the $L0$ sampler is $O(s \log^3(n/\epsilon)) = O(\log^4(n/\epsilon))$.*

Analysis. It remains to show that it works correctly, i.e., that with probability at least $1 - \epsilon$, we correctly identify a randomly chosen index from the vector. For today, we will skip the analysis that the index is chosen *uniformly* from the non-zero entries. All we need for today is that we can identify *some* index. However, it should be relatively clear based on symmetry that each index in the vector is equally likely.[1]

In particular, we will show that with probability at least $1 - \epsilon$, at least one of the vectors v_k processed by a sampler S_k^s is s-sparse and contains at least one non-zero index. If that is true, then we know from the analysis if the s-sampler that it will properly identify at least one non-zero index.

As part of the analysis, we will need to use a few tail bounds that are common in algorithms involving sampling. These are known as Chernoff bounds. Imagine you have a set of $0/1$ random variables $x_1, x_2, \ldots, x_n$ where each x_i is 1 with probability p and 0 otherwise, for some constant p. Let $X = \sum_i (x_i)$. Notice that the expected value $\mathrm{E}[X] = pn$. We define $\mu = \mathrm{E}[X]$.

Then for any error δ, $0 \leq \delta \leq 1$, the following two bounds hold:

$$\begin{aligned} \Pr[X \geq (1+\delta)\mu] &\leq e^{-\delta^2 \mu/3} \\ \Pr[X \leq (1-\delta)\mu] &\leq e^{-\delta^2 \mu/2} \leq e^{-\delta^2 \mu/3} \end{aligned}$$

That is, the sum of the independent random variables is close to the mean. In our case, we will choose $\delta = 1/2$ and combine the two bounds, relying on the following fact:

$$\Pr[|X - \mu| \geq \mu/2] \leq 2e^{-\mu/12}$$

We can now prove the key claim:

Claim 8 *With probability at least $1 - \epsilon/2$, there exists a $k = 2^\ell$ such that the vector v_k is s-sparse.*

Proof Fix some constant $c = 12 \ln(4/\epsilon)$, and note that $c \geq 2$. Recall earlier we chose $s = \Theta(\log(1/\epsilon))$. We can now be more precise: fix $s = 3c$.

Our first step is to identify a choice of k, where k is a power of 2, such that v has at least ck and at most $2ck$ non-zero entries. Let z be the number of non-zero entries in v.

First, if $z \leq s$, then we choose $k = 1$. In this case, we are done and the lemma is proved: with probability 1, vector v_1 is already s-sparse.

Assume $z > s$, i.e., $z > 2c$. Define $k = 2^{\lfloor \log(z/c) \rfloor}$. That is, k is the largest power of 2 that is less than z/c. Notice that:

- $z/c > 2$, and hence $k \geq 2$.
- v has at least kc non-zero entries, since $c2^{\lfloor \log(z/c) \rfloor} \leq c(z/c) \leq z$.

[1] A somewhat handwavy and imprecise argument might precede as follows: imagine calculating the probability that some index i is chosen—it will be some complicated expression involving the probability that i is the only non-zero index mapped to a given S^1 sampler. Notice that this expression will be the same for each index. Since exactly one index is selected in the end, and since they each have the same probability of being chosen, we conclude that there is a uniform random choice among the non-zero indices.

- v has at most $2kc$ non-zero, since $2c2^{\lfloor \log(z/c) \rfloor} \geq 2c(z/(2c)) \geq z$. Notice this follows because $\lfloor \log(z/c) \rfloor \geq \log(z/c) - 1 \geq \log(z/(2c))$.

From this, we conclude that vector v has between $[kc, 2kc]$ non-zero entries. Also, notice that k is a power of 2 (by choice).

We now want to calculate the probability that the vector v_k is s-sparse and contains at least one non-zero entry. That is, What is the probability that vector v_k has ≥ 1 and $\leq s$ non-zero entries?

Here we can directly use the Chernoff bound from above. Let $x_j = h_k(j)$ be a random variable. We want to chose that $X = \sum(x_j)$ is at least 1 and at most s. Notice that here $\mu = \mathrm{E}[X] = z/k$, where $c \leq z/k \leq 2c$. The Chernoff bound shows that:

$$\begin{aligned} \Pr[|X - z/k| \geq z/(2k)] &\leq 2e^{-(z/k)(1/12)} \\ &\leq 2e^{-(1/12)c} \\ &\leq 2e^{-\ln(4/\epsilon)} \\ &\leq \epsilon/2 \end{aligned}$$

Then, from the Chernoff bound, we have shown that with probability at least $1 - \epsilon/2$, the vector v_k has:

- at least $z/k - z/(2k) = z/(2k) \geq c/2 \geq 1$ non-zero entries, and
- at most $z/k + z/(2k) = (3/2)(z/k) \leq 3c \leq s$ non-zero entries.

That is, we conclude that v_k is s-sparse and contains at least 1 non-zero entry. □

Since there is an s-sparse sampler that processes an s-sparse vector v_k with probability at least $1 - \epsilon/2$, and all the $\log n$ samplers work correctly with probability at least $1 - (\log(n)\epsilon)/n$, we conclude that with probability at least $1 - \epsilon$, the $L0$-sampler works correctly (for $n \geq 4$).

Claim 9 *With probability at least $1 - \epsilon$, the L0-sampler returns a (random) non-zero index of the vector v.*

Another aspect of the analysis is the cost of updating and querying the sampler:

Claim 10 *Each query or update of the L0-sampler has cost $O(\log^2(n/\epsilon))$*

Proof Each $L0$-sampler consists of $\log n$ s-samplers. Querying or updating an L0-sampler require, in the worst-case, querying or updating each of the s-samplers (in the worst-case).

To query or update an s-sampler consists of querying or updating $O(\log(n/\epsilon))$ 1-samplers. Each 1-sampler can be queried or updates in $O(1)$ time. Hence the total cost of query or update is $O(\log^2(n/\epsilon))$ in the worst-case. □

Finaly, notice that the L0-sampler is, again, really just a linear sketch: each of the S^s samplers used in the construction is a linear sketch, and hence if we have two $L0$ samplers, we can add them together. A given $L0$-sampler consists of $\log n$ s-samplers, and each s-sampler contains $O(\log^2(n/\epsilon))$ 1-samplers, each of which can be merged in $O(1)$ time. Thus any two L0-samplers can be merged in $O(\log^3(n/\epsilon))$ time.

To summarize, we have shown the following:

Theorem 11 *Given $\epsilon < 1$, there exists an L0-sampler that uses space $O(\log^4(n/\epsilon))$ and identifies a (random) non-zero index of the vector v with probability at least $1-\epsilon$. Each query or access to the L0-sampler has cost $O(\log^2(n/\epsilon))$. Two L0-samplers can be merged in time $O(\log^3(n/\epsilon)$.*