Sketching String Similarities

Yuchong Zhang
University of Toronto
yuchongz.zhang@mail.utoronto.ca

***Abstract*—This expository article serves as a pedagogical introduction to three sketching algorithms used for estimating similarities between strings. Although some considerations of problems that arise from implementions are discussed, the emphasis will be placed on exploring the ideas behind these algorithms rather than their real-world performance. The intended audience is undergraduates with background in Probability and Computer Science. The three selected algorithms extend the same basic idea to overcome increasing difficulties to estimate similarity measures that are increasingly better, and the main hope is to provide the reader with an intuitive understanding of them. Efforts are made to keep the notations simple and explanations intuitive.**

***Index Terms*—string similarity, sketching algorithms**

I. Introduction

In general, a measurement of the similarity between two strings is simply a function γ that takes any two strings S_1, S_2 as inputs and outputs $\gamma(S_1, S_2) \in [0, 1]$. Intuitively, $\gamma(S_1, S_2)$ indicates what "percentage" of S_1, S_2 are the same, with $\gamma(S_1, S_2) = 1$ meaning two strings are the "same", and $\gamma(S_1, S_2) = 0$ meaning they are "completely distinct". The problem of measuring string similarities is foundational to many real-world applications. For example, quantifying the similarity between two genome sequences is central to many problems in computational biology, and matching similar text documents is useful for search engine or database queries. There are various ways of measuring string similarities and corresponding algorithms to compute them exactly. For example, it is well-known that the Levenshtein distance between two strings can be computed via Dynamic Programming.

Despite many improvements, these algorithms require storage space and computation time that are often too computationally expensive in practice. For instance, computing the Levenshtein distance takes approximately quadratic time and linear space in the length of two inpute strings, while massive data such as genomes often consist of billions of letters, which in many cases are too large to be stored in memory. Moreover, massive strings are constantly being produced, making it impractical to analyze them all. Theoretical progress has indicated that it might be hopeless to compute certain string similarities exactly using much less resources. For example, the Levenshtein editing distance is likely not computable in less than quadratic time [1]. Thus, people are motivated to look for compromised solutions: would it be possible to use minimal resources to compute the similarities *approximately*, with some room for errors?

One possible solution is streaming and sketching algorithms. On a high-level, these are algorithms that process the input strings in a stream of little "pieces" while maintaining and updating some compact summary of the inputs, called *sketches*. The small pieces are not stored cumulatively, and once all pieces in the stream are processed, the algorithm uses only information in the sketches to approximate the similarity within acceptable accuracies and a small chance of failing to do so. Of course, depending on the particular similarity measurement, there will be different strategies for obtaining and utilizing the sketches. But the goal is the same: *to use only limited space and time to maintain the sketch and achieve reasonable accuracy with a high chance of success.*

In this article, we will explore ideas behind three such algorithms that correspond to different measures of string similarities. Section II covers the classical MinHash algorithm for estimating Jaccard Similarity. Section III discusses the author's own algorithm for estimating Weighted Jaccard Similarity. And finally section IV introduces Order MinHash, which provides estimates for Levenshtein editing distance. The reader will see that the basic idea behind the simple MinHash can be extended to overcome the increasing estimation difficulty posed by increasingly "refined" measurements of string similarities.

II. MinHash: the classical algorithm for Jaccard Similarity Estimation

A. Jaccard Similarity

Throughout this article, S, S_1 and S_2 will be arbitrary strings over some alphabet of letters, and $k \geq 1$ will be some integer.

- A k-mer of S is simply a substring of S with length k.
- $N_k(S)$ denotes set of all k-mers of S.

We will regard k-mers as the little pieces of a string. They can be think of as a generalized version of the individual letters (since when $k = 1$, a k-mer is simply a letter). We consider k-mers because they are important in DNA sequencing, where a DNA sequence can be viewed as a string over the 4-letter alphabet $\{A, T, C, G\}$ (the four nucleotides). Also, k-mers retain some (limited)information about the groupings and order of certain letters. We will always consider "overlapping" k-mers. For instance, in $ATCG$, both AT and TC are 2-mers. In general, if $|S| = n$, then $|N_k(S)| = n - k + 1$.

For any two sets A, B, the *Jaccard Index* of A, B is defined as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

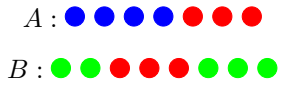
This is a simple way of measuring similarity between two sets, and by treating S_1, S_2 as sets of their k-mers, we get the *Jaccard Similarity* between S_1, S_2, which is defined as:

$$J_k(S_1, S_2) = J(N_k(S_1), N_k(S_2)) = \frac{|N_k(S_1) \cap N_k(S_2)|}{|N_k(S_1) \cup N_k(S_2)|}$$

Jaccard Similarity might seem like a rather crude way of measuring similarities, but it is good enough in many applications, and most importantly, there is a robust sketching algorithm MinHash for estimating it.

B. MinHash: key idea

MinHash is first discussed by Broder in his 1997 seminal paper [2]. The basic idea is simple. Suppose we have two sets A, B that share some elements between them, as follows:



where the red balls indicate elements in $A \cap B$, and $A \cup B$ is the set consisting of balls of all colors. Consider a bijection $h : A \cup B \to \{1, \ldots, |A \cup B|\}$ that "randomly" assigns unique labels to the balls. And consider the following experiment: we apply h to A, record the ball a that gets the minimum label, then apply h to B, record the ball b that gets the minimum label. What is the probability that $a = b$?

It is clear that if h assigns the smallest label to a green ball, then b would be a green ball and a cannot equal b because A has no green ball. Similarly, $a \neq b$ if h assigns the smallest label to a blue ball. However, if a red ball gets the smallest label, then because A, B both contain all red balls, a must equal b. If we assume *any ball in $A \cup B$ has a equal chance of getting the minimum label*, then clearly

$$\mathbb{P}[a = b] = \frac{\text{\# of red balls}}{\text{\# of all balls}} = \frac{|A \cap B|}{|A \cup B|} = J(A, B)$$

which is the Jaccard Index $J(A, B)$. How does this help us find $J(A, B)$? We have constructed an event (namely $a = b$) whose probability of happening is $J(A, B)$, where the uncertainty comes from h. To estimate $J(A, B)$, we just need to estimate $\mathbb{P}[a = b]$. To do this, simply repeat the experiment n times and count the number of times m that $a = b$ occurred. Then $\frac{m}{n} \approx \mathbb{P}[a = b] = J(A, B)$. Intuitively, the more times we repeat the experiment, the closer $\frac{m}{n}$ should be to $J(A, B)$. (This can be made formal easily by the Law of Large Numbers.) Depending on how accurate we want $\frac{m}{n}$ to be, we can decide how many times the experiment should be run.

This is the key idea behind MinHash. In the context of estimating string similarity, we regard S_1, S_2 as sets of k-mers $N_k(S_1), N_k(S_2)$, and the Jaccard Similarity $J_k(S_1, S_2) = J(N_k(S_1), N_k(S_2))$, so we need the "balls" to be all the k-mers, and the function h would be a hash function of the strings. In reality, the randomness of h comes from *randomly* drawing h from a whole class $\mathcal{H}$ of functions that (deterministically) maps elements of $A \cup B$ to numbers, called a class of hash functions. Note that in the analysis above, $\mathbb{P}[a = b] = J(A, B)$ rests on the assumption that any element in $A \cup B$ has an equal chance of getting the minimum label:

$$\forall x, y \in A \cup B, \mathbb{P}[x = \min] = \mathbb{P}[y = \min] = \frac{1}{|A \cup B|}$$

This is a very non-trivial property that $\mathcal{H}$ needs to satisfy, and if $\mathcal{H}$ does satisfy this, then we say $\mathcal{H}$ is *min-wise independent*. Great efforts have been made to construct such hash functions that can work in practice, and the theory behind them can be found in [3] [4]. For our purposes, we will simply assume that we have a class $\mathcal{H}$ of min-wise independent hash functions that map k-mers to numbers. The MinHash algorithm for strings are as follows:

Algorithm 1 MinHash: sketching

1) Draw $h \in \mathcal{H}$ at random. Slide a window of length k across S_1, S_2 one letter at a time and apply h to the k-mer in that window.
2) As the window slides, maintain a, b, the minimum hash values of k-mers in S_1 and S_2, respectively.
3) Return 1 if $a = b$, 0 otherwise.

Algorithm 2 MinHash: estimation

1) Run R independent copies of the sketching part, and let δ_j be the return value of the jth copy.
2) Output $\hat{J} = \frac{1}{R} \sum_{j=1}^{R}$ as the estimation of $J_k(S_1, S_2)$.

To summarize, the event that the two hash values collide has the key property that it happens with probability equal $J_k(S_1, S_2)$. This allows us to repeat the sketching part like an experiment and emprically observe how often the event occurs, thus obtaining an estimate of its probability. This idea will be crucial to the algorithm in Section III that estimates a "better" similarity measurement than Jaccard Similarity. But before we go there, let us consider some practical problems that arise from implementing MinHash, which will also serve as motivations to improving the algorithm in Section III.

III. Estimation of Weighted Jaccard Similarity via Random Sampling

A. Weighted Jaccard Similiarity

In some sense, the robustness of MinHash comes from the simplicity of Jaccard Similarity. The shortcoming is that by treating S_1, S_2 as sets of k-mers and use $J_k(S_1, S_2)$, Jaccard Similarity ignores information such as k-mer frequencies and the order in which k-mer appear, both of which can greatly affect string similarities. A perhaps trivial example that illustrates this is the following:

$$S_1 = 00000000000010000000000000000000$$

$$S_2 = 00000000000000000000000000000000$$

Intuitively, we would like to say that S_1 and S_2 are very similar because they are both almost all zeroes. However, because the single "1" in S_1 produces k extra k-mers for S_1, the Jaccard Similarity

$$J_k(S_1, S_2) = \frac{1}{k+1} \leq 0.5$$

One way to capture the reason we think these extra k-mers should not affect the similarity between S_1, S_2 is that each of them occurs "rarely" (only once in S_1) while the k-mer of all 0's occurs many times. This motivates us to find similarity measure that takes into account k-mer frequencies. One such measurement can be obtained by simple extension from the Jaccard Similarity.

For k-mer σ and string S, let $N_k(\sigma, S)$ be the number of times σ appears in S (0 if not present), and the *Weighted Jaccard Similarity* between S_1, S_2 is defined by:

$$J_k^w(S_1, S_2) := \frac{\sum_\sigma \min\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}}{\sum_\sigma \max\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}}$$

The idea behind this definition is that if σ appears l times in both S_1, S_2, then these l occurrences should be treated as the same, but if σ appears more times in S_1 than in S_2, then those additional occurrences will contribute to S_1. A figure that illustrates the differences between various similarity measurements will be included in Section IV when we discuss OrderMinHash.

To see the that $J_k^w(S_1, S_2)$ is an extension from $J_k(S_1, S_2)$, consider appending to each k-mer σ at certain position its *occurrence number* i, where i is the number of times σ appears in S up to and include its position. If we let $T_k(S)$ denote the set of indexed k-mers of S, then it can be easily verified that

$$J_k^w(S_1, S_2) = J(T_k(S_1), T_k(S_2)) = \frac{|T_k(S_1) \cap T_k(S_2)|}{|T_k(S_1) \cup T_k(S_2)|}$$

This relation means that in some sense, Weighted Jaccard Similarity is simply Jaccard Similarity viewed through another lens. This will allow us to apply the same key idea behind MinHash to develop a sketching algorithm that estimates Weighted Jaccard Similarity.

B. Motivation

Although MinHash has many variants, the fundamental idea behind it is to construct an event, in that case, the collision of minimum hash values in S_1, S_2, whose probability of happening equals $J_k(S_1, S_2)$, the quantity MinHash tries to estimate. Once we have this event, we can estimate the probability of it happening by repeating experiments, thus obtaining an estimate of $J_k(S_1, S_2)$. We will try to follow the same idea to come up with a sketching algorithm that estimates the Weighted Jaccard Similarity. We know that

$$\begin{aligned} J_k^w(S_1, S_2) &= \frac{\sum_\sigma \min\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}}{\sum_\sigma \max\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}} \\ &= \frac{|T_k(S_1) \cap T_k(S_2)|}{|T_k(S_1) \cup T_k(S_2)|} \end{aligned}$$

So we need an event whose probability of happening equals $\frac{|T_k(S_1) \cap T_k(S_2)|}{|T_k(S_1) \cup T_k(S_2)|}$. Of course, we also need the event to be easy to "simulate" so that running the experiments repeatedly can be done using little resources. An obvious event whose probability of happening equals $J_k^w(S_1, S_2)$ is for a uniform random sample of indexed k-mer (σ, i) in $T_k(S_1) \cup T_k(S_2)$ to fall in $T_k(S_1) \cap T_k(S_2)$. The problem is that in order to perform such sampling, we need access to the indexed k-mer sets $T_k(S_1)$ and $T_k(S_2)$, which may not always be available. More specifically, in order to sample from $T_k(S_1) \cup T_k(S_2)$, we need the input strings S_1, S_2 to contain information about all the k-mer indexes. Pictorially, we need S_1, S_2 to have the following forms:

$$\begin{aligned} S_1 &= \ldots(\sigma, 1)\ldots(\sigma, 2)\ldots(\sigma, l)\ldots(\sigma, m)\ldots \\ S_2 &= \ldots(\sigma, 1)\ldots(\sigma, 2)\ldots(\sigma, l)\ldots \end{aligned}$$

which would allow us to somehow "pair up" the first $l = \min\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}$ occurrences of σ one by one and treat each pair as only "one occurrence" to obtain $T_k(S_1) \cup T_k(S_2)$. In other words, *certain pairs* σ should not be distinguished.

But in reality, the input strings may well not be in this form and simply appear as sequences of letters:

$$\begin{aligned} S_1 &= \ldots\sigma\ldots\sigma\ldots\sigma\ldots\sigma\ldots \\ S_2 &= \ldots\sigma\ldots\sigma\ldots\sigma\ldots \end{aligned}$$

Clearly, we can distinguish all occurrences of σ in S_1 from those in S_2, and distinguish all occurrences of σ in S_1 by their (absolute) position in the string. But this would treat all occurrences of σ as distinct, and we would have no way of knowing which pairs of σ in S_1, S_2 should be treated as the same. Note that going through the strings to compute all the occurrence numbers would be computationally infeasible since we assumed the k-mer sets are huge and input strings massive.

Although we do not directly have access[1] to $T_k(S_1)$ and $T_k(S_2)$, by treating all k-mer occurrences as distinct, we can still "sample" an indexed k-mer in the following way:

1) Randomly sample a position from all the available positions in S_1 or S_2 where a k-mer might reside.
2) If σ is the k-mer at that position, we can go through the string to compute the occurrence number of m of this particular σ by counting how many times σ appears in the string up to its position.

Moreover, once (σ, m) is obtained, and the random position from the first step is in, say, S_1, then it directly follows from the definition of indexed k-mers that we can easily check whether (σ, m) is in $T_k(S_1) \cap T_k(S_2)$ by going through S_2 and count whether σ appears at least m times in S_2.

This method provides a randomly sampled indexed k-mer (σ, m), and an easy way of checking whether it is in $T_k(S_1) \cap$

[1]However, if for some reason we do have access to $T_k(S_1), T_k(S_2)$, we can actually estimate an even better similarity measurement. This will be discussed in Section IV.

$T_k(S_2)$. The problem is that we did not directly sample (σ, m) from $T_k(S_1) \cup T_k(S_2)$. Remember that (σm) is determined by randonly sampling from all available positions where a k-mer might occur. In other words, the true sample space is the set of all k-mer occurrences treated as distinct, which has size $\sum_\tau N_k(\tau, S_1) + N_k(\tau, S_2)$, as opposed to $T_k(S_1) \cup T_k(S_2)$, which has size $\sum_\tau \max\{N_k(\tau, S_1), N_k(\tau, S_2)\}$. So the actual sampling process is not a uniform sampling from $T_k(S_1) \cup T_k(S_2)$, which means the probability that $(\sigma, i) \in T_k(S_1) \cap T_k(S_2)$ is likely not $J_k^w(S_1, S_2)$.

But there is a silver line: we know exactly how larger the true sample space is compared to $T_k(S_1) \cup T_k(S_2)$. So we might hope that we can somehow use this information to transform the probability $\mathbb{P}[(\sigma, i) \in T_k(S_1) \cap T_k(S_2)]$ into $J_k^W(S_1, S_2)$. If this is true, then as long as the process of transformation does not introduce too much error, we can still indirectly estimate $J_k^w(S_1, S_2)$ by estimating the aforementioned probability. As we will see below, this hope is indeed true.

C. The Algorithm

Now we present the algorithm based on the idea discussed in the previous section and confirm that it indeed works.

Algorithm 3 $f(S_1, S_2, k)$: experiment of random sampling

1) Sample a position i uniformly at random from all possible positions in S_1, S_2 where a k-mer resides.
2) Let $S :=$ the string containing position i, $S' :=$ the other string. $\sigma :=$ the k-mer at position i in S.
3) Pass through S, find $m \leftarrow$ number of occurrences of σ in S up to and including position i.
4) Pass through S', check if σ appears at least m times in S'. Return 1 if true, 0 otherwise.

Algorithm 4 $g(S_1, S_2, k)$: repeat experiments f to estimate $J_k^w(S_1, S_2)$

1) Run R independent copies of $f(S_1, S_2, k)$. $\delta_j =$ return value of the jth copy.
2) $\hat{p} = \frac{1}{R} \sum_{j=1}^R \delta_j$.
3) Output $\frac{\hat{p}}{2-\hat{p}}$ as the estimate of $J_k^w(S_1, S_2)$.

We now show that the algorithm indeed estimates $J_k^w(S_1, S_2)$. As previously discussed, f is the indicator for the event that the random sample (σ, m) falls in $T_k(S_1) \cap T_k(S_2)$. Because (σ, m) is not directly sampled from $T_k(S_1 \cup T_k(S_2)$, $\mathbb{P}[f \text{ returns } 1]$ will not be $J_k^w(S_1, S_2)$. But we can compute $\mathbb{P}[f \text{ returns } 1]$ explicitly as follows: fix any k-mer σ, and without loss of generality, suppose S_1, S_2 have the following form:

$$S_1 = \sigma \ldots \sigma \ldots \sigma \ldots \sigma \ldots \sigma \ldots$$
$$S_2 = \sigma \ldots \sigma \ldots \sigma \ldots \sigma \ldots$$

where σ and σ mark the last occurrence of σ in S_1, S_2, respectively. Then by definition, σ is the $N_k(\sigma, S_2)$th copy of σ in S_2. Let σ be the $N_k(\sigma, S_2)$th copy of σ in S_1.

Clearly, the event f returns 1 and the sampled k-mer in step 2 of f equals σ happens if and only if the sample is a σ that falls in the following region:

$$S_1 = \sigma \ldots \sigma \ldots \sigma \ldots \sigma \ldots \sigma \ldots$$
$$S_2 = \sigma \ldots \sigma \ldots \sigma \ldots \sigma \ldots$$

The circled region above contains $2 \cdot N_k(\sigma, S_2) = 2\min\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}$ occurrences of σ. As argued before, the true sample space is equivalently all k-mer occurrences treated distinct and thus has size $\sum_\tau N_k(\tau, S_1) + N_k(\tau, S_2)$. Because the sampling in step 1 of f is uniform sampling, this means

$$\mathbb{P}[f \text{ returns } 1 \text{ and sample is } \sigma] = \frac{2\min\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}}{\sum_\tau N_k(\tau, S_1) + N_k(\tau, S_2)}$$

By marginalizing out σ, we get:

$$\begin{aligned} p &= \mathbb{P}[f \text{ returns } 1] \\ &= \sum_\sigma \mathbb{P}[f \text{ returns } 1 \text{ and sample is } \sigma] \\ &= \sum_\sigma \frac{2\min\{N_k(\sigma, S_1), N_k(\sigma, S_2)\}}{\sum_\tau N_k(\tau, S_1) + N_k(\tau, S_2)} \\ &= \frac{2\sum_\tau \min\{N_k(\tau, S_1), N_k(\tau, S_2)\}}{\sum_\tau N_k(\tau, S_1) + N_k(\tau, S_2)} \end{aligned}$$

In comparison:

$$J_k^w(S_1, S_2) = \frac{\sum_\tau \min\{N_k(\tau, S_1), N_k(\tau, S_2)\}}{\sum_\tau \max\{N_k(\tau, S_1), N_k(\tau, S_2)\}}$$

which is different from p, as expected. But we can easily transform:

$$\begin{aligned} p &= \frac{2\sum_\tau \min\{N_k(\tau, S_1), N_k(\tau, S_2)\}}{\sum_\tau N_k(\tau, S_1) + N_k(\tau, S_2)} \\ &= \frac{2\sum_\tau \min\{N_k(\tau, S_1), N_k(\tau, S_2)\}}{\sum_\tau \min\{N_k(\tau, S_1), N_k(\tau, S_2)\} + \max\{N_k(\tau, S_1), N_k(\tau, S_2)\}} \end{aligned}$$

and solve

$$\begin{aligned} \frac{p}{2-p} &= \frac{\sum_\tau \min\{N_k(\tau, S_1), N_k(\tau, S_2)\}}{\sum_\tau \max\{N_k(\tau, S_1), N_k(\tau, S_2)\}} \\ &= J_k^w(S_1, S_2) \end{aligned}$$

thus justifying the estimator $\hat{p} = \frac{1}{R} \sum_{j=1}^R \delta_j$ in $g(S_1, S_2, k)$.

D. Accuracy Analysis

Let us analyze the accuracy of the estimation. First note that

$$\mathbb{E}[\hat{p}] = \frac{1}{R} \sum_{j=1}^R \mathbb{E}[\delta_j] = p$$

So $\hat{p}$ is an unbiased estimator of the true probability p that f returns 1. We proved that p satisfies

$$\frac{p}{2-p} = J_k^w(S_1, S_2)$$

So we can hope $\frac{\hat{p}}{2-\hat{p}}$ will not be too far from $J_k^w(S_1, S_2)$ if $\hat{p}$ is not too far from p. Indeed:

$$\begin{aligned}|\frac{\hat{p}}{2-\hat{p}} - J_k^w(S_1,S_2)| &= |\frac{p}{2-p} - \frac{\hat{p}}{2-\hat{p}}| \\ &= \frac{2|p-\hat{p}|}{(2-p)(2-\hat{p})} \\ &\leq 2|p-\hat{p}| \\ &\text{(Since } 1 < 2-p, 2-\hat{p} < 2.\text{)}\end{aligned}$$

Thus if $\hat{p}$ is within ϵ of p, then $\frac{\hat{p}}{2-\hat{p}}$ will be within 2ϵ of $J_k^w(S_1, S_2)$. Recall from the introduction the goal we want to achieve with respect to accuracy: we want the probability that the estimator $\frac{\hat{p}}{2-\hat{p}}$ is far from $J_k^w(S_1, S_2)$ to be small. We can easily formalize this notion as follows. First, compute

$$\begin{aligned}\text{Var}[\hat{p}] &= \text{Var}[\frac{1}{R}\sum_{j=1}^{R}\delta_j] \\ &= \frac{1}{R^2}\sum_{j=1}^{R}\text{Var}[\delta_j] \text{ (by independence)} \\ &= \frac{1}{R}(\mathbb{E}[\delta_1^2] - \mathbb{E}[\delta_1]^2) \\ &= \frac{p-p^2}{R}\end{aligned}$$

Then by Chebyshev's inequality, for $\epsilon > 0$:

$$\begin{aligned}&\mathbb{P}[|\hat{p} - \mathbb{E}[\hat{p}]| \geq \epsilon] \\ &\qquad = \mathbb{P}[|\hat{p} - p| \geq \epsilon] \\ &\qquad \leq \frac{\text{Var}[\hat{p}]}{\epsilon^2} \\ &\qquad = \frac{p-p^2}{R\epsilon^2}\end{aligned}$$

Combine this with $|\frac{\hat{p}}{2-\hat{p}} - J_k^w(S_1, S_2)| \leq 2|p-\hat{p}|$, we get:

$$\begin{aligned}\mathbb{P}[|\frac{\hat{p}}{2-\hat{p}} - J_k^w(S_1,S_2)| < \epsilon] &\geq \mathbb{P}[|\hat{p}-p| < \epsilon/2] \\ &= 1 - \mathbb{P}[|\hat{p}-p| \geq \epsilon/2] \\ &\geq 1 - \frac{4(p-p^2)}{R\epsilon^2}\end{aligned}$$

From the above bound, it is clear that given a fixed ϵ, we can make the probability that our estimate falls within ϵ of the true $J_k^w(S_1, S_2)$ by running the "experiment" $f(S_1, S_2, k)$ enough times, which conforms with our intuition from Section II-B.

E. Resource analysis and streaming model considerations

Recall from Section I that apart from the accuracy requirement, we also need to make sure that the algorithm does not take too much computing resources, primarily time and space. Since g simply runs R independent copies of f, the resources it takes should be approximately R times that of f, or possibly the same as f, depending on whether we run the copies of f in parallel. We can divide f into 3 major steps:

1) Sample a k-mer σ at a random position i.
2) Pass through one of S_1, S_2 once to compute the occurrence number m of σ.
3) Pass through another string once to check if $(\sigma, m) \in T_k(S_1) \cap T_k(S_2)$.

Along each pass, we need to constantly compare the sample k-mer with another k-mer in the string. Since comparing two k-mers takes $O(k)$ time and the total number of k-mers along the way is $O(|S_1| + |S_2|)$, f takes $O(k(|S_1| + |S_2|))$ time to execute steps 2) and 3). If we know the length of the two strings $|S_1|, |S_2|$, then sampling a random position is equivalent to simply generating a random integer i in the range $[0, |S_1| - k] \cup [|S_1|, |S_1| + |S_2| - k]$, which can be easily done using $O(1)$ time and space. However, the algorithm needs to retrieve the k-mer at position i. And the time it does this is dependent upon how the input strings S_1, S_2 are stored.

First, there is the perhaps trivial situation when S_1, S_2 are already stored in Random Access Memory, which means we can access the letter at any position in a string and retrive the k-mer at position i in $O(1)$ time. In this situation, it is often true that information about $|S_1|, |S_2|$ is easily accessible (for example, Strings in Python keep track of their own lengths, which can be found in $O(1)$ time). In this situation, f only needs to pass through S_1, S_2 once and the runtime would be $O(k(|S_1| + |S_2|))$. If $|S_1|, |S_2|$

However, this situation assumes that we must already have enough memory to store S_1, S_2, which seems to defeat the point of using probablistic sketching. What about the more common streaming model, where the input strings appear as little pieces in an incoming stream?

Without loss of generality, we can assume the strings appear as streams of k-mers because otherwise we can simply wait until enough letters have arrived to assemble a k-mer. In this model, we need to perform uniform sampling of a k-mer occurrence from an incoming stream with an unknown number of items.

Of course, there is one easy and trivial solution:

1) Maintain 2 counters and go through the stream once to find out $|S_1|, |S_2|$.
2) Sample a random integer as the position i, go through the stream again to obtain the k-mer at position i.
3) Go through the stream for a third time to do part(2) of f.

This method does not increase the asymptotic runtime of f (still $O(k(|S_1| + |S_2|))$. But ideally we want to restart the stream as few times as possible because it might be expensive in practical situations (for example, sending large amount of data over the internet), and worse still, in some cases we only have access to the stream once!

We can a common technique called Reservoir Sampling to reduce the number of passes required for sampling to one. In general, Reservoir Sampling can sample without replacement l items from a stream of unknown length n such that each one of the n items has probability l/n of ending up in the final sample. Since f only require sampling one position

with probability $1/n$, we can use the most basic version of Reservoir Sampling, which goes as follows:

- Start the stream and keep the first item.
- As the ith item in the stream arrives, update the sample to be the ith item with probability $1/i$ and do nothing with probability $1 - 1/i$.

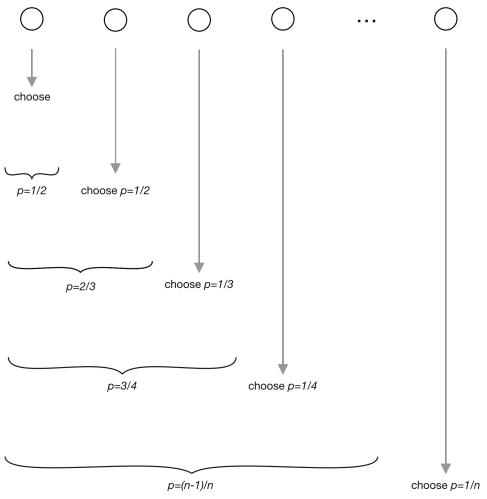


Fig. 1. The process of Reservoir Sampling.

It is easy to see that

$$\forall t, \mathbb{P}[\text{select item } t] = \frac{1}{t} \cdot \frac{t}{t+1} \cdot \frac{t+1}{t+2} \cdots \frac{n-1}{n} = \frac{1}{n}$$

So with Reservoir Sampling, the experiment function f can be implemented as:

Algorithm 5 f+(S_1, S_2, k): Improvements via Reservoir Sampling.

1) Use Reservoir Sampling to sample a k-mer σ (and its position) uniformly at random.
2) Restart the stream again. Count the occurrence number m of σ in the string from which it is sampled.
3) Count whether the other string contains at least m copies of σ, and return 0 or 1 accordingly.

Despite the improvement, this implementation still requires two passes through the stream, which is less than ideal and greatly limits the applicability of the algorithm. The author has been attempting to try to reduce the number of passes needed to 1, a discussion of which will be provided in the next section.

Finally, it is clear that f only needs to store the sample k-mer σ ,another k-mer in the stream to compare with σ, and a constant number of counters. So f requires only $O(k)$ space.

F. An ongoing attempt to further improve f

Let us explore some possible directions to improve f.

Recall the three major steps of f:

1) Sample a k-mer σ at a random position i.
2) Pass through one of S_1, S_2 once to compute the occurrence number m of σ.
3) Pass through another string once to check if $(\sigma, m) \in T_k(S_1) \cap T_k(S_2)$.

The difficulty of reducing the required number of passes through the stream by f arises from the fact that in the incoming stream model, we need one pass to execute step 1) (i.e., obtaining the sketch σ and i), and another pass to finish steps 2) and 3) (i.e., utilizing the sketch). In comparison, once MinHash obtains its sketch, namely the two minimum hash values, we no longer need information regarding the input strings to use the sketch (the two minimum hash values can be compared in $O(1)$ time).

Obviously, in order to reduce the number of passes, we need to somehow perform one of the steps above without passing through the entire input strings. For example, f only requires one pass when S_1, S_2 are both stored in RAM, because in that case step 1) reduces to generating random index and string indexing. This is of course unfeasible in the incoming stream model, so we are motivated to consider steps 2) and 3).

One possible idea is the following. Since the occurrence number m of an indexed k-mer (σ, m) is simply an integer within the range $[1, N_k(\sigma, S)]$, one might think that if σ and its frequency $N_k(\sigma, S)$ are given, then a random sample of an indexed k-mer can be obtained by randomly generating an integer in the range $[1, N_k(\sigma)]$. Of course, we might be given all the k-mer frequencies, and we assumed that finding them by brute force is not feasible. But there is a sketching algorithm called Count-min sketch [7] that can serve as a "frequency table" with limited space and time, and of course, some chances of errors.

Without getting into the details, Count-min sketch consumes a stream of items, and at any time, we can make a query to find out the approximate frequency of any item σ it has seen so far. The approximate frequency might be larger than the true frequency of that item, but it will not be smaller. One can tune the parameters of Count-min sketch to make the probability of the approximate frequency being too far from the true frequency low. If we combine Count-min sketch with the Reservoir sampling technique, we obtain the following algorithm:

Algorithm 6 f++(S_1, S_2, k): Improvements via Count-min sketch

1) Maintain two separate Count-min sketches for S_1, S_2
2) Feed incoming k-mers to Count-min sketch while performing Reservoir Sampling in parallel. Update Count-min sketch accordingly.
3) Once stream ends, Reservoir Sampling yields a k-mer σ and its position i. Let $S :=$ string in which σ is sampled, $S' :=$ the other string.
4) Read the approximate frequency N of $N_k(\sigma, S)$ from Count-min sketch for S. Generate random integer m from range $[1, N]$.
5) Read the approximate frequency N' of $N_k(\sigma, S')$ from Count-min sketch for S'. Return 1 or 0 based on whether $m \leq N'$.

Basically, rather than performing steps 2) and 3) by passing through the stream again, we shifted the burden to Count-min sketch. The "improvement" seems promising because if we assume that assume the Count-min sketch outputs the exact k-mer frequencies (i.e., $N = N_k(\sigma, S)$ and $N' = N_k(\sigma, S')$ in the algorithm). Then it turns out that the probability of f++ returning 1 is the same quantity as before:

$$\frac{2\sum_\tau \min\{N_k(\tau, S_1), N_k(\tau, S_2)\}}{\sum_\tau N_k(\tau, S_1) + N_k(\tau, S_2)}$$

The current difficulty lies in measuring how the errors of Count-min sketch affects the true probability of f++ returning 1. This exploration is still ongoing and the author hopes to come up with explicit error bounds.

IV. OrderMinHash: Extending MinHash to Editing Distance [6]

A. Editing Similarity

Although Weighted Jaccard Similarity takes into account k-mer frequencies, it still discards the information about the order in which k-mers appear, which can be crucial. For example, if we have a massive k-mer set and S_1, S_2 both contain every k-mer exactly once, then $J_k^w(S_1, S_2)$ would be 1, but certainly the k-mer orders can be drastically different. One measurement that takes k-mer orders into account is a normalized version of the Levenshtein Editing Distance.

The (Levenshtein) Editing Distance as follows. An *indel* (short for insertion and deletion) is the operation of inserting or deleting a single letter to either one of S_1, S_2, and a *mismatch* is the operation of substituting a single letter in either string with another letter. The Editing Distance between S_1, S_2 is the minimum number of indels or mismatches necessary to transform S_1 into S_2. Note that the editing distance really measures how disimilar S_1, S_2 are in the sense that the larger the distance is, the more disimilar S_1, S_2 are. To obtain a similarity measurement, we simply normalize as follows:

$$Ed(S_1, S_2) = 1 - \frac{\text{Editing Distance}}{\max\{|S_1|, |S_2|\}}$$

To see how Editing Similarity is sensitive to both k-mer frequencies and orders, consider the case where S_1, S_2 1) share a good amount of k-mers in common, 2) those common k-mers have roughly the same frequencies, and 3) they appear in the same relative order:

$$S_1 = abctabcxyzbnm12$$

$$S_2 = abcWabc356xyz00bnm$$

If this is the case, then to transform S_1 into S_2, we can "align" those k-mers together, like aligning the 3-mers by their color in the example above. Once we do that, we can leave the aligned parts alone, and only modifying the unaligned parts via indels or mismatches. The intuition is that if the aligned parts are *large enough*, then modifying the remaining parts should not take too many operations.

Note that in order for the aligned parts to be "large" enough, all 3 conditions must all be satisfied. This high-bar set by editing distance means that the important part for an algorithm that estimate $Ed(S_1, S_2)$ is for it to be sensitive to strings that have large aligned parts.

B. Order MinHash: idea

Although the Editing Similarity above retains more information than (Weighted) Jaccard Similarity, it is much harder to estimate. The idea behind MinHash of "simulating" an event with certain probability can also be extended here, although a much stronger assumption about the input strings is required. In order for the event probability to reflect the Editing Similarity, it must be sensitive to both k-mer frequencies and order.

To account for k-mer frequencies, we will use the same method as before by considering indexed k-mer sets $N_k(S_1), N_k(S_2)$, and as previously noted, we will assume access to $N_k(S_1), N_k(S_2)$. i.e., the k-mers in S_1, S_2 are already indexed by their occurrence numbers.

Order MinHash is more or less based on the following simple idea. Recall the 3 conditions discussed in the previous section for $Ed(S_1, S_2)$ to be high. First, sharing a good amount of k-mers with similar frequencies means that if we randomly draw a set A of k-mers (counting repeats) from S_1 and another set B of k-mers from S_2, then there's a good chance that the two sets share a number of k-mers in common, or maybe even most of k-mers in common. Secondly, having k-mers that appear in the same relative order means that if we sort the k-mers in A, B according to their original orders in S_1 and S_2, then the results have a good chance of being largely aligned.

To implement this idea, consider the set of permutations $\mathcal{W}$ on the indexed k-mer set $T_k(S_1) \cup T_k(S_2)$. Apart from its occurrence number, we can obviously the (absolute) position of a k-mer. With these information, the Order MinHash (OMH) sketch is defined as follows:

Algorithm 7 OMH sketch

1) Draw a permutation $\pi \in \mathcal{W}$ unformly at random.
2) Pass through S_1, S_2 and permute the indexed k-mers along the way. Let $\pi_{k,l}(S_1) :=$ the l smallest indexed k-mers of S_1 under π. Similarly obtain $\pi_{k,l}(S_2)$.
3) Sort $\pi_{k,l}(S_1)$ and $\pi_{k,l}(S_2)$ according to the absolute positions of the indexed k-mers in S_1, S_2.
4) Return 1 if $\pi_{k,l}(S_1) = \pi_{k,l}(S_2)$, 0 otherwise.

The collison of $\pi_{k,l}(S_1), \pi_{k,l}(S_2)$ seems to happen only when the three conditions discussed previously are satisfied to a great degree, which intuitively means $Ed(S_1, S_2)$ is high (only a small number of modifications is required to completely align S_1, S_2), otherwise collision seems unlikely. Mathematically, this is formalized by saying that there exists functions $p_1, p_2 : \mathbb{R} \to [0, 1]$, such that the collision probability $p = \mathbb{P}[\pi_{k,l}(S_1) = \pi_{k,l}(S_2)]$ satisfies

$$Ed(S_1, S_2) \geq t_1 \implies p \geq p_1(t_1)$$

and

$$Ed(S_1, S_2) \leq t_2 \implies p \leq p_2(t_2)$$

In other words, high/low editing similarity means high/low chance of collision, as our intuition suggests. Note that although this relation between p and $Ed(S_1, S_2)$ is weaker than what we got with algorithms in previous sections, we can still get an estimate of the upper/lower bounds of $Ed(S_1, S_2)$ by estimating p via repeated experiments.

The proof for these two results are essentially formalizations of the idea discussed before: in order for $Ed(S_1, S_2)$ to be high, the "unaligned" segments between S_1, S_2 cannot be too large, resulting in higher chance that the sorted sketches would align. The details can be found in [6].

REFERENCES

[1] Backurs A., Indyk P. (2015) Edit distance cannot be computed in strongly subquadratic time (unless SETH is false) In: Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing, STOC '15. ACM, New York, NY, USA, pp. 51–58.

[2] A. Z. Broder, "On the resemblance and containment of documents," in Compression and Complexity of Sequences 1997. Proceedings. IEEE, 1997, pp. 21–29.

[3] A. Z. Broder, "Min-Wise Independent Permutations", in Journal of Computer and System Sciences 60, 2000, pp. 630-659.

[4] M. Thorup, "High Speed Hashing for Integers and Strings", 2020.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] Marçais G, DeBlasio D, Pandey P, and Kingsford C. (2019). Locality-sensitive hashing for the edit distance. Bioinformatics 35, i127–i135.

[7] G Cormode, S Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications", 2005, Journal of Algorithms 55 (1), 58-75