Approximate Similarity Search Under Edit Distance Using Locality-Sensitive Hashing

Samuel McCauley
Williams College, Williamstown, MA, USA

Abstract

Edit distance similarity search, also called approximate pattern matching, is a fundamental problem with widespread database applications. The goal of the problem is to preprocess n strings of length d, to quickly answer queries q of the form: if there is a database string within edit distance r of q, return a database string within edit distance cr of q.

Previous approaches to this problem either rely on very large (superconstant) approximation ratios c, or very small search radii r. Outside of a narrow parameter range, these solutions are not competitive with trivially searching through all n strings.

In this work we give a simple and easy-to-implement hash function that can quickly answer queries for a wide range of parameters. Specifically, our strategy can answer queries in time $\widetilde{O}(d3^r n^{1/c})$. The best known practical results require $c \gg r$ to achieve any correctness guarantee; meanwhile, the best known theoretical results are very involved and difficult to implement, and require query time that can be loosely bounded below by 24^r. Our results significantly broaden the range of parameters for which there exist nontrivial theoretical bounds, while retaining the practicality of a locality-sensitive hash function.

2012 ACM Subject Classification Information systems → Nearest-neighbor search; Theory of computation → Pattern matching

Keywords and phrases edit distance, approximate pattern matching, approximate nearest neighbor, similarity search, locality-sensitive hashing

Digital Object Identifier 10.4230/LIPIcs.ICDT.2021.21

Funding This research was supported in part by the European ResearchCouncil under the European Union's 7th Framework Programme (FP7/2007–2013) / ERC grant agreement no. 61433, by the VILLUM Foundation grant 16582, and by a Zuckerman STEM Postdoctoral Fellowship.

Acknowledgements I would like to thank Thomas Ahle, Martin Aumüller, Tobias Christiani, Tsvi Kopelowitz, Rasmus Pagh, Ely Porat, Francesco Silvestri, and Shikha Singh for many helpful comments and discussions.

1 Introduction

For a large database of items, a *similarity search* query asks which database item is most similar to the query. This leads to a basic algorithmic question: how can we preprocess the database to answer these queries as quickly as possible?

Similarity search is used frequently in a wide variety of applications. Unfortunately, for databases containing high-dimensional items, algorithm designers have had trouble obtaining bounds that are significantly faster than a linear scan of the entire database. This has often been referred to as the "curse of dimensionality." Recent work in fine-grained complexity has begun to explain this difficulty: achieving significantly better than linear search time would contradict the strong exponential time hypothesis [2, 14, 35].

However, these queries can be relaxed to *approximate similarity search* queries. For an approximation factor c, we want to find a database item that is at most a c factor less similar than the most similar item.

© Samuel McCauley;
licensed under Creative Commons License CC-BY 4.0
24th International Conference on Database Theory (ICDT 2021).
Editors: Ke Yi and Zhewei Wei; Article No. 21; pp. 21:1–21:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Approximate similarity search is fairly well-understood for many metrics; see [3] for a survey. For example, in Euclidean space we have theoretical upper bounds [4, 12], fast implementations [5, 17, 21, 25, 27, 37], and lower bounds for a broad class of algorithms [4]. Many of these results are based on *locality-sensitive hashing* (LSH), originally described in [20]. A hash is locality-sensitive if similar items are likely to share the same hash value.

When a database contains text items, a natural notion of similarity is edit distance: how many character inserts, deletes, and replacements are required to get from the query string to a database string? In fact, edit distance similarity search is frequently used in computational biology [22, 24, 33], spellcheckers [7, 38], computer security (in the context of finding similarity to weak passwords) [28], and many more applications; see e.g. [6].

Surprisingly, finding an efficient algorithm for approximate similarity search under edit distance remains essentially open. Known results focus on methods for exact similarity search (with $c = 1$), which incur expensive query times, and on embeddings, which require very large – in fact superconstant – approximation factors c.

However, recent work provides a potential exception to this. The CGK embedding [8] is simple and practical, and embeds into Hamming space with stretch $O(r)$ – in particular, it does well when the distance between the closest strings is fairly small. EmbedJoin, a recent implementation by Zhang and Zhang [39], showed that the CGK embedding performs very well in practice. EmbedJoin first embeds each string into Hamming space using the CGK embedding. Then, the remaining nearest neighbor search[1] is done using the classic bit sampling LSH for Hamming distance. Each of these steps – both the CGK embedding and the bit sampling LSH – is repeated several times independently. This method gave orders of magnitude better performance than previous methods. Furthermore, their results greatly outperformed the worst-case CGK analysis.

Thus, several questions about using CGK for edit distance similarity search remained. Zhang and Zhang used several CGK embeddings, performing a sequence of Hamming distance hashes for each – can these two steps be combined into a single method to improve performance? Meanwhile, their tests focused on practical datasets; is it possible to provide worst-case bounds for this method, ensuring good performance for any dataset?

In this paper we answer these questions in the affirmative. In doing so, we give the first locality-sensitive hash for edit distance with worst-case guarantees.

1.1 Results

The main result of our paper is the first locality-sensitive hash for edit distance. We analyze the performance of this hash when applied to the problems of approximate similarity search and approximate nearest neighbor search, obtaining time bounds that improve on the previously best-known bounds for a wide range of important parameter settings.

Let n be the number of strings stored in the database. We assume that all query strings and all database strings have length at most d. We assume $d = O(n)$ and the alphabet size is $O(n)$.[2]

[1] Zhang and Zhang investigated similarity *joins*, in which all similar pairs in a set are returned, rather than preprocessing for individual nearest neighbor queries. However, their ideas can be immediately generalized.

[2] Usually d and the alphabet size are much smaller. If this assumption does not hold, it is likely that a completely different approach will be more successful: for example, if $d = \mathrm{poly}(n)$, then the method used to calculate the edit distance between two strings becomes critically important to the query time.

Our first result analyzes the time and space required by our LSH to solve the approximate similarity search problem. This data structure works for a fixed radius r: for each query, if there exists a database point within distance r, we aim to (with good probability) return a database point within distance cr. We use $\widetilde{O}(f(n))$ as a shorthand for $O(f(n) \cdot \text{polylog} f(n))$.

▶ **Theorem 1.** *There exists a data structure answering Approximate Similarity Search queries under Edit Distance in $\widetilde{O}(d3^r n^{1/c})$ time per query, and $\widetilde{O}(d3^r n^{1+1/c})$ preprocessing time and space.*

We also give a data structure that answers queries where the distance r to the closest neighbor is not known during preprocessing. We call this the approximate *nearest* neighbor search problem.

▶ **Theorem 2.** *There exists a data structure answering Approximate Nearest Neighbor Search queries under Edit Distance in $\widetilde{O}(d3^r n^{1/c})$ time per query and $\widetilde{O}(dn^2)$ preprocessing time and space.*

Implications for Related Problems. Our results lead to immediate bounds for similarity join, where all close pairs in a database are computed; see e.g. [34, 39, 40].

Much of the previous work on approximate similarity search under edit distance considered a variant of this problem: there is a long text T, and we want to find all locations in T that have low edit distance to the query q. Our results immediately apply to this problem by treating all d-length substrings of T as the database of items.

Frequently, practical situations may require that we find all of the neighbors with distance at most r, or (similarly) the k closest neighbors. See e.g. [1] for a discussion of this problem in the context of LSH. Our analysis immediately applies to these problems. However, if there are k desired points, the running times given in Theorems 1 and 2 increase by a factor k.

1.2 Comparison to Known Results

In this section, we give a short summary of some key results for edit distance similarity search. We focus on algorithms that have worst-case query time guarantees. We refer the reader to [39, 40] as good resources for related practical results, and [6, 26, 31] for a more extensive discussion of related work on the exact problem (with $c = 1$).

Exact Similarity Search Under Edit Distance. Exact similarity search under edit distance (i.e. with $c = 1$) has been studied for many years. We focus on a breakthrough paper of Cole, Gottlieb, and Lewenstein that achieved space $O(n5^r(1.5r + \log n)^r/r!)$ and query time $O(d + 6^r(1.5r + \log n)^r/r!)$ [15].[3] We will call this structure the CGL tree. These bounds stand in contrast to previous work, which generally had to assume that the length of the strings d or the size of the alphabet $|\Sigma|$ was a constant to achieve similar bounds. Later work has improved on this result to give similar query time with linear space [9].

Before comparing to our bounds, let us lower bound the CGL tree query time – while this gives a lower bound on an upper bound (an uncomfortable position since we are not specifying its exact relationship to the data structure), it will be helpful to get a high-level idea of how the performance of the CGL tree compares to our results. Using Sterling's approximation, and dropping the $+d$ term, we can simplify the query time to

[3] These bounds are a slight simplification of the actual results using the AM-GM inequality.

$\widetilde{O}((6e/r)^r(1.5r+\log n)^r) \leq \widetilde{O}((9e)^r(1+(\log n)/(1.5r))^r)$. From this final equation, we can see that even for very small n, the guaranteed query time is at least $(9e)^r > 24^r$; if $\log n \gg 1.5r$ it can become much worse.

Comparing the $(9e)^r$ term with our query time of $\widetilde{O}(d3^r n^{1/c})$, it seems that which is better depends highly on the use case – after all, we're exchanging a drastically improved exponential term in r for a polynomial term in n.

However, there is reason to believe that our approach has some significant advantages. First, for c bounded away from 1, with moderate n and small d, the CGL query time rapidly outpaces our own even for small r. Let's do a back-of-the-envelope calculation with some reasonable parameters – we ignore constants here, but note that slight perturbations in r easily make up for such discrepancies. If we have 400k strings of 500 characters[4] with $c = 1.5$, $6^r(1.5r + \log n)^r/r! \geq d3^r n^{1/c}$ for $r > 4$. In other words, even for very small search radii and fairly large n (where the CGL tree excels), the large terms in the base of r can easily overcome a polynomial-in-n term. Second, the constants in the CGL tree seem to be unfavorable: the CGL tree uses beautiful but nontrivial data structures for LCA and LCP that may add to the constants in the query time. In other words, it seems likely that the CGL tree is most viable for even smaller values of r than the above analysis would indicate.

We suspect that these complications are part of the reason why state-of-the-art practical edit distance similarity search methods are based on heuristics or embeddings, rather than tree-based methods (see e.g. [40]).

Approximate Similarity Search Under Edit Distance. Previous results for approximate similarity search with worst-case bounds used either product metrics, or embeddings into L_1.

In techniques based on *product metrics*, each point is mapped into several separate metrics. The distance between two points is defined as their maximum distance in any of these metrics. Using this concept, Indyk provided an extremely fast (but large) nearest-neighbor data structure requiring $O(d)$ query time and $O(n^{d^{1/(1+\log c)}})$ space for any $c \geq 3$ [19].

Embedding into L_1. Because there are approximate nearest neighbor data structures for L_1 space that require $n^{1/c+o(1)}$ time and $n^{1+1/c+o(1)}$ space,[5] an embedding into L_1 with stretch α leads to an approximate nearest neighbor data structure with query time $n^{\alpha/c+o(1)}$ for $c > \alpha$.

A long line of work on improving the stretch of embedding edit distance into L_1 ultimately resulted in a deterministic embedding with stretch $\exp(\sqrt{\log d}/\log\log d)$ [32].

More recently, the CGK embedding parameterized by r instead of d, giving an embedding into Hamming space[6] with stretch $O(r)$ [8]. However, the constants proven in the CGK result are not very favorable – the upper limit on overall stretch given in the paper is $2592r$ (though this may be improvable with tighter random walk analysis). Thus, using the CGK embedding, and then performing the standard bit sampling LSH for Hamming distance on the result, gives an approximate similarity search algorithm with query time $n^{2592r/c+o(1)}$ so long as $c > 2592r$. We describe in detail how our approach improves on this method in Appendix 1.3.

[4] These are the parameters of the UniRef90 dataset from the UniProt Project `http://www.uniprot.org/`, one protein genome dataset used as an edit distance similarity search benchmark [39, 40]; other genomic datasets have (broadly) similar parameters.

[5] This can be improved to $n^{1/(2c-1)+o(1)}$ and $n^{1+1/(2c-1)+o(1)}$ time and space respectively using data-dependent techniques, and can be further generalized to other time-space tradeoffs; see [4].

[6] Hamming space and L_1 have essentially the same state-of-the-art similarity search bounds.

Zhang and Zhang [39] implemented a modified and improved version of this approach. Their results far outperformed the above worst-case analysis. Closing this gap between worst-case analysis and practical performance is one contribution of this work.

There is a lower bound of $\Omega(\log d)$ for the stretch of any embedding of edit distance into L_1 [23]. This implies that embedding into L_1 is a hopeless strategy for $c < \log d$, whereas we obtain nontrivial bounds even for constant c. Thus, for this parameter range, using a locality-sensitive hash is fundamentally more powerful than embedding into L_1.

Locality-Sensitive Hashing. An independent construction of an LSH for edit distance was given by Marçais et al. [29]. Their work uses a fundamentally different approach, based on an ordered min-hash of k-mers. Their results include bounds proving that the hash is locality-sensitive; however, they do not place any worst-case guarantees on the gap between the probability that close points collide and the probability that far points collide.

Exponential search cost. To our knowledge, a trivial brute force scan is the only algorithm for approximate similarity search under edit distance whose worst-case cost is not exponential in the search radius r. While we significantly improve this exponential term, removing it altogether remains an open problem. A recent result of Cohen-Addad et al. gave lower bounds showing that, assuming SETH, there exist parameter settings such that cost exponential in r is required for any edit distance similarity search algorithm [14]. Their results do not immediately imply that the exponential-in-r term in our query time is necessary (since the $n^{1/c}$ term satisfies their lower bound for sufficiently small c); however, this may give some indication as to why removing this exponential term has proven so challenging.

1.3 Technical Overview and Comparison to the CGK Embedding

Our hash function follows the same high-level structure as the CGK embedding [8]. In fact, our hash reduces to their embedding by omitting the appended character \$, and setting $p_a = 1/2$ and $p_r = 0$ (these parameters are defined in Section 3).

However, our hash has two key differences over simply using the CGK embedding to embed into Hamming space, and then using bit sampling. These differences work together to allow us to drastically improve the $n^{2592r/c+o(1)}$ bound we obtained in Section 1.2.

First, we modify p_a; that is, we modify the probability that we stay on a single character x_i of the input string for multiple iterations. Second, we combine the embedding and bit sampling into a single step – this means that we can take the embedding into account when deciding whether to sample a given character.

Combining into one step already gives an inherent improvement. After embedding, we do not want to sample a "repeated" character – this is far less useful than sampling a character the last time it is written, after the hash has attempted to align them. Thus, we only sample a character (with probability $1 - p_r$) the last time that character is written.

However, the significant speedup comes from using *repeated* embeddings – in short, at a high level, each LSH in our approach consists of a single CGK embedding with a single bit sampling LSH. If a single embedding is used, the performance of the algorithm as a whole has the expected stretch of that single embedding as a bottleneck. As a result, the expected stretch winds up in the exponent of n, and c must be at least as large as the expected stretch to guarantee correctness. By repeatedly embedding, our bounds instead depend (in a sense) on the *best*-case stretch over the many embeddings.

These repeated embeddings is where these two differences – modifying p_a and integrating into a single LSH – act in concert. A back-of-the-envelope calculation implies that a CGK embedding will have stretch[7] 2 with probability at least $1/4^r$. This analysis seems difficult to tighten: if we perform 4^r embeddings, how well will we do in the cases that don't have $O(1)$ stretch? Meanwhile, any constant loss in the analysis winds up in the exponent of n. Overall, with the CGK embedding as a black box, a full analysis would require an analysis (with tight constants) detailing the probability that an embedding has any given stretch. Instead, by combining these approaches in a single LSH, we can instead model the entire problem as a single random walk in a two-dimensional grid.

Overall, a combined approach gives better worst-case performance, and a unified (and likely simpler) framework for analysis.

2 Model and Preliminaries

We denote the alphabet used in our problem instance as Σ. We use two special characters $\perp$ and \$, which we assume are not in Σ. The hash appends \$ to each string being hashed; we call a string \$-terminal if its last character is \$ and it does not contain another \$.

We index into strings using 0-indexed subscripts; x_0 is the first character of x and x_i is the $i+1$st character. We use $x[i]$ to denote the prefix of x of length i; thus $x[i] = x_0 \dots x_{i-1}$. Finally, we use $x \circ y$ to denote the concatenation of two strings x and y, and $|x|$ to denote the length of a string x.

2.1 Edit Distance

Edit distance is defined using three operations: inserts, deletes, and replacements. Given a string $x = x_1 x_2 \dots x_d$, inserting a character σ at position i results in a string $x' = x_1 \dots x_{i-1} \sigma x_i \dots x_d$. Replacing the character at position i with σ results in $x' = x_1 \dots x_{i-1} \sigma x_{i+1} \dots x_d$. Finally, deletion of the character at position i results in $x' = x_1 \dots x_{i-1} x_{i+1} \dots x_d$. We refer to these three operations as *edits*. The edit distance from x to y is defined as the smallest number of edits that must be applied to x to obtain y. We denote this as $\mathrm{ED}(x, y)$.

2.2 Model and Problem Definition

In this paper we solve the approximate similarity search problem under edit distance, which can be defined as follows.

▶ **Definition 3** (Approximate Similarity Search Under Edit Distance)**.** *Given a set of n strings S and constants c and r, preprocess S to quickly answer queries of the form, "if there exists a $y \in S$ with $\mathrm{ED}(q, y) \le r$, return a $y' \in S$ with $\mathrm{ED}(q, y') \le cr$ with probability $> 1/10$."*

The above is sometimes called the approximate *near* neighbor problem. The constant $1/10$ is arbitrary and can be increased to any desired constant without affecting our final bounds.

Oftentimes, we want to find the nearest database item to each query rather than parameterizing explicitly by r.

---

[7] To be more precise, with probability $1/4^r$ one string with distance r from the query will have embedded Hamming distance r, while all strings with distance x will have embedded Hamming distance $\ge x/2$.

▶ **Definition 4** (Approximate Nearest Neighbor Search Under Edit Distance)**.** *Given a set of n strings S and a constant c, preprocess S to quickly answer queries of the form, "for the smallest r such that there exists a $y \in S$ with* $\mathrm{ED}(q, y) \leq r$*, return a $y' \in S$ with* $\mathrm{ED}(q, y') \leq cr$ *with probability* $> 1/10$*."*

For most previous LSH-based approaches, efficient Nearest Neighbor Search algorithms follow immediately from Approximate Similarity Search algorithms using the black box reduction of Har-Peled, Indyk, and Motwani [18]. However, the exponential dependence on r in our bounds requires us to instead use a problem-specific approach.

2.3 Locality-Sensitive Hashing

A hash family is *locality sensitive* if close elements are more likely to hash together than far elements. Locality-sensitive hashing is one of the most effective methods for approximate similarity search in high dimensions [4, 10, 18, 20].

▶ **Definition 5** (Locality-Sensitive Hash)**.** *A hash family $\mathcal{H}$ is (r, cr, p_1, p_2)-sensitive for a distance function $d(x, y)$ if*

- *for all x_1, y_1 such that $d(x_1, y_1) \leq r$,* $\Pr_{h \in \mathcal{H}}(h(x_1) = h(y_1)) \geq p_1$*, and*
- *for all x_2, y_2 such that $d(x_2, y_2) \geq cr$,* $\Pr_{h \in \mathcal{H}}(h(x_1) = h(y_1)) \leq p_2$*.*

Some previous work (i.e. [11, 16]) has a stricter definition of locality sensitive hash: it requires that there exists a function f such that $\Pr(h(x) = h(y)) = f(d(x, y))$. Our hash function does not satisfy this definition; the exact value of x and y is necessary to determine their collision probability (see Lemma 16 for example).

A Note on Concatenating Hashes. Most previous approaches to nearest neighbor search begin with an LSH family that has $p_1, p_2 = \Omega(1)$. A logarithmic number of independent hashes are concatenated together so that the concatenated function has collision probability $1/n$. This technique was originally developed in [20], and has been used extensively since; e.g. in [1, 4, 13].

However, in this paper, we use a single function each time we hash. We directly set the hash parameters to achieve a desirable p_1 and p_2 (in particular, we want $p_2 \approx 1/n$). This is due to the stray constant term in Lemma 15. While our hash could work via concatenating several copies of a relatively large-probability[8] LSH, this would result in a data structure with larger space and slower running time. One interesting implication is that, unlike many previous LSH results, our running time is not best stated with a parameter $\rho = \log p_1 / \log p_2$ – rather, we choose our hashing parameters to obtain the p_1 and p_2 to give the best bounds for a given r, c, and n.

3 The Locality-Sensitive Hash

Each hash function from our family maps a string x of length d with alphabet Σ to a string $h(x)$ with alphabet $\Sigma \cup \{\perp\}$ of length $O(d + \log n)$. The function scans over x one character at a time, adding characters to $h(x)$ based on the current character of x and the current length of $h(x)$. Once the function has finished scanning x, it stops and outputs $h(x)$.

At a high level, for two strings x and y, our hash function can be viewed as randomly guessing a sequence of edits T, where $h(x) = h(y)$ if and only if applying the edits in T to x obtains y. Equivalently, one can view the hash as a random walk through the dynamic

[8] Although less than constant – Lemma 14 and the assumption that $p \leq 1/3$ implies $p_1 \leq (1/3)^r$.

programming table for edit distance, where matching edges are traversed with probability 1, and non-matching edges are traversed with a tunable probability $p \le 1/3$. We discuss these relationships in Section 4.1.

Note the contrast with the CGK embedding, which uses a similar mechanism to guess the *alignment* between the two strings for each mismatch, rather than addressing each edit explicitly. This difference is key to our improved bounds; see Section 1.3.

Parameters of the Hash Function. We parameterize our algorithm using a parameter $p \le 1/3$. By selecting p we can control the values of p_1 and p_2 attained by our hash (see Lemmas 14 and 15). We will describe how to choose p to optimize nearest neighbor search performance for a given r, c, and n in Section 4.3. We split p into two separate parameters p_a and p_r defined as $p_a = \sqrt{p/(1+p)}$ and $p_r = \sqrt{p}/(\sqrt{1+p} - \sqrt{p})$. Since $p \le 1/3$, we have $p_a \le 1/2$ and $p_r \le 1$. For the remainder of this section, we will describe how the algorithm behaves using p_a and p_r. The rationale behind these values for p_a and p_r will become clear in the proof of Lemma 13 – in short, our choice of p_a and p_r ensures that each type of edit is guessed with the same probability.

Underlying Function. Each hash function in our hash family has an *underlying function* that maps each *(character, hash position)* pair to a pair of uniform random real numbers: $\rho : \Sigma \cup \{\$\} \times \{1, \ldots, 8d/(1-p_a) + 6 \log n\} \to [0,1) \times [0,1)$.[9] We discuss how to store these functions and relax the assumption that these are real numbers in Section 4.5.

The only randomness used in our hash function is given by the underlying function.[10] In particular, this means that two hash functions h_1 and h_2 have identical outputs on all strings if their underlying functions ρ_1 and ρ_2 are identical. Thus, we pick a random function from our hash family by sampling a random underlying function. We use $h_\rho(x)$ to denote the hash of x using underlying function ρ.

The key idea behind the underlying function is that the random choices made by the hash depend only on the current character seen in the input string, and the current length of the output string. This means that if two strings are aligned – in particular, if the "current" character of x matches the "current" character of y – the hash of each will make the same random choices, so the hashes will stay the same until there is a mismatch. This is the "oblivious synchronization mechanism" used in the CGK embedding [8].

3.1 How to Hash

A hash function h is selected from the family $\mathcal{H}$ by sampling a random underlying function ρ. We denote the hash of a string x using ρ as $h_\rho(x)$. The remainder of this section describes how to determine $h_\rho(x)$ for a given x and ρ.

To hash x, the first step is to append \$ to the end of x to obtain $x \circ \$$. We will treat $x \leftarrow x \circ \$$ as the input string from now on – in other words, we assume that x is \$-terminal. Let i be the current index of x being scanned by the hash function. We will build up $h_\rho(x)$ character-by-character, storing intermediate values in a string s. The hash begins by setting $i = 0$, and s to the empty string.

[9] Adding \$ to the alphabet allows us to hash past the end of a string – this helps with edits that append characters.

[10] In fact, the underlying function is a generalization of the random string used in the CGK embedding.

Algorithm 1 Calculating $h_\rho(x)$.

1: $i \leftarrow 0$
2: Create an empty string s
3: **while** $i < |x|$ and $|s| < 8d/(1-p_a) + 6\log n$ **do**
4: $(r_1, r_2) \leftarrow \rho(x_i, |s|)$
5: **if** $r_1 \leq p_a$ **then**
6: Append $\perp$ to s
7: **else if** $r_2 \leq p_r$ **then**
8: Append $\perp$ to s
9: $i \leftarrow i+1$
10: **else**
11: Append x_i to s
12: $i \leftarrow i+1$
13: **return** s

$x = abc$
$h(x) = \perp a \perp\perp\perp\perp$
$y = bac$
$h(y) = \perp a \perp\perp\perp\perp$
$z = cba$
$h(z) = c\perp\perp a\$$

x_i \ $\lvert s\rvert$	0	1	2	3	4	5
a	$(0.1, 0.7)$	$(0.9, 0.6)$	$(0.1, 0.7)$	$(0.6, 0.8)$	$(0.2, 0.3)$	$(0.5, 0.6)$
b	$(0.6, 0.3)$	$(0.8, 0.3)$	$(0.8, 0.2)$	$(0.9, 0.4)$	$(0.1, 0.1)$	$(0.1, 0.5)$
c	$(0.7, 0.6)$	$(0.5, 0.9)$	$(0.1, 0.9)$	$(0.2, 0.8)$	$(0.7, 0.4)$	$(0.4, 0.6)$
$\$$	$(0.1, 0.4)$	$(0, 0.1)$	$(0.1, 0.3)$	$(0.8, 0.7)$	$(0.9, 0.5)$	$(0.6, 0)$

Figure 1 Example of how to hash three strings x, y, and z with underlying function ρ_1 (given in the table on the right). We use $\Sigma = \{a, b, c\}$ and $p = 1/8$, so $p_a = 1/3$ and $p_r = 1/2$. For simplicity, we round the values of ρ_1 to the first decimal place, and only give ρ_1 for $|s| \leq 5$.

The hash function repeats the following process while $i < |x|$ and[11] $|s| < 8d/(1-p_a) + 6\log n$. The hash first stores the current value of the underlying function based on x_i and $|s|$ by setting $(r_1, r_2) \leftarrow \rho(x_i, |s|)$. The hash performs one of three actions based on r_1 and r_2; in each case one character is appended to the string s. We name these cases a *hash-insert*, *hash-replace*, and *hash-match*.

- **If** $r_1 \leq p_a$**, hash-insert:** append $\perp$ to s.
- **If** $r_1 > p_a$ **and** $r_2 \leq p_r$**, hash-replace:** append $\perp$ to s and increment i.
- **If** $r_1 > p_a$ **and** $r_2 > p_r$**, hash-match:** append x_i to s and increment i.

When $i \geq |x|$ or $|s| \geq 8d/(1-p_a) + 6\log n$, the hash stops and returns s as $h_\rho(x)$.

We provide pseudocode for this method in Algorithm 1, and an example hash in Figure 1.

4 Analysis

In this section we show how analyze the hash given in Section 3, proving Theorems 1 and 2.

We begin in Section 4.1 with some structure that relates hash collisions between two strings x and y with sequences of edits that transform x into y. We use this to bound the probability that x and y collide in Section 4.2. With this we can prove our main results in Sections 4.3 and 4.4. Finally we discuss how to store the underlying functions in Section 4.5.

[11] The requirement $|s| < 8d/(1-p_a) + 6\log n$ is useful to bound the size of the underlying function in Section 4.5. We show in Lemma 6 that this constraint is very rarely violated.

4.1 Interpreting the Hash

In this section, we discuss when two strings x and y hash (with underyling function ρ) to the same string $h_\rho(x) = h_\rho(y)$.

We define three sequences to help us analyze the hash. In short, the *transcript* of x and ρ lists the decisions made by the hash function as it scans x using the underlying function ρ. The *grid walk* of x, y, and ρ is a sequence based on the transcripts (under ρ) of x and y – it consists of some edits, and some extra operations that help keep track of how the hashes of x and y interact. Finally, the *transformation* of x, y, and ρ is a sequence of edits based on the grid walk of x, y, and ρ.

Using these three sequences, we can set up the basic structure to bound the probability that x and y hash together using their edit distance. We use these definitions to analyze the probability of collision in Section 4.2.

Transcripts. A *transcript* is a sequence of hash operations: each element of the sequence is a hash-insert, hash-replace, or hash-match. Essentially, the transcript of x and ρ, denoted $\tau(x, \rho)$, is a log of the actions taken by the hash on string x using underlying function ρ.

We define an index function $i(x, k, \rho)$. The idea is that $i(x, k, \rho)$ is the value of i when the kth hash character is written when hashing x using underlying function ρ.

We set $i(x, 0, \rho) = 0$ for all x and ρ. Let $(r_{1,k}, r_{2,k}) = \rho(x_{i(x,k,\rho)}, k)$. We can now recursively define both $\tau(x, \rho)$ and $i(x, k, \rho)$. We denote the kth character of $\tau(x, \rho)$ using $\tau_k(x, \rho)$.

- If $r_{1,k} \le p_a$, then $i(x, k+1, \rho) = i(x, k, \rho)$, and $\tau_k(x, \rho) =$ hash-insert.
- If $r_{1,k} > p_a$ and $r_{2,k} \le p_r$, then $i(x, k+1, \rho) = i(x, k, \rho) + 1$, and $\tau_k(x, \rho) =$ hash-replace.
- If $r_{1,k} > p_a$ and $r_{2,k} > p_r$, then $i(x, k+1, \rho) = i(x, k, \rho) + 1$, and $\tau_k(x, \rho) =$ hash-match.

A transcript $\tau(x, \rho)$ is *complete* if $|\tau(x, \rho)| < 8d/(1 - p_a) + 6 \log n$.

▶ **Lemma 6.** *For any string x of length d,* $\Pr_\rho[\tau(x, \rho)$ *is complete*$] \ge 1 - 1/n^2$.

Proof. If $\tau(x, \rho)$ has ℓ hash-insert operations, then $|\tau(x, \rho)| \le d + \ell$. We bound the probability that $\ell > 7d/(1 - p_a) + 6 \log n$.

For each character in x, we can model the building of $\tau(x, \rho)$ as a series of independent coin flips. On heads (with probability p_a), ℓ increases; on tails the process stops. Thus we expect $1/(1 - p_a)$ hash-insert operations for each character of x, and at most $d/(1 - p_a)$ hash-insert operations overall.

Using standard multiplicative Chernoff bounds (i.e. [30, Exercise 4.7]), the probability that $\ell > 7d/(1 - p_a) + 6 \log n$ is at most $\exp(-(6d(1 - p_a) + 6 \log n)/3) < 1/n^2$. ◀

Grid Walks. A *grid walk* $g(x, y, \rho)$ for two strings x and y and underlying function ρ is a sequence that helps us examine how $h_\rho(x)$ and $h_\rho(y)$ interact – it is a bridge between the transcript of x, y and ρ, and the transformation induced by x, y, and ρ (which is a sequence of edits). We formally define the grid walk, and discuss how it corresponds to a random walk in a graph. This graph is closely based on the dynamic programming table for x and y.

The grid walk is a sequence of length $\max\{|\tau(x, \rho)|, |\tau(y, \rho)|\}$. The grid walk has an alphabet of size 6: each character is one of $\{\texttt{insert}, \texttt{delete}, \texttt{replace}, \texttt{loop}, \texttt{match}, \texttt{stop}\}$. At a high level, `insert`, `delete`, and `replace` correspond to string edits – for example, `insert` corresponds to the index of x being incremented while the index of y stays the same (as if we inserted the corresponding character into y). `loop` corresponds to both strings writing $\perp$

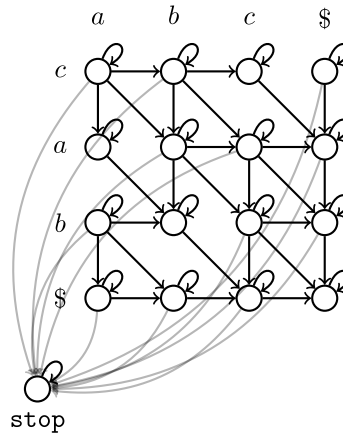

Figure 2 This figure shows $G(x, y)$ for $x = abc\$$ and $y = cab\$$. For clarity, all edge labels are ommited and `stop` edges are partially transparent.

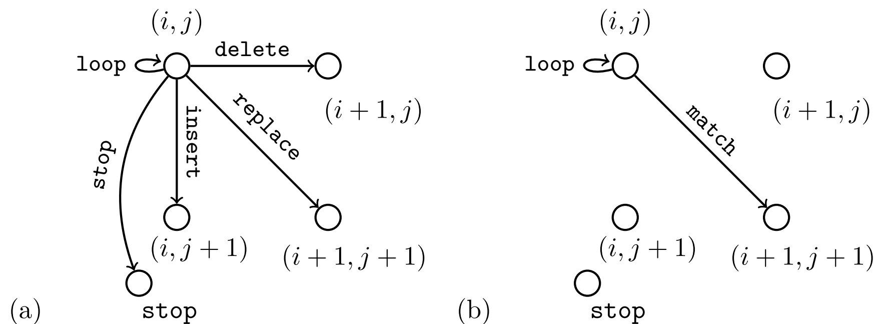

Figure 3 The edges for a single node (i, j) with $i < |x| - 1$ and $j < |y| - 1$. (a) represents the edges if $x_i \neq y_j$; (b) represents the edges if $x_i = y_j$.

without increasing i; the process "loops" and we continue with nothing changed except the length of the hash. `match` corresponds to the case when both hashes simultaneously evaluate the same character – after a sequence of `loop` operations, they will `match` by both writing out either the matching character or $\perp$ to their respective hashes. `stop` is a catch-all for all other cases: the strings write out different characters, the hashes are no longer equal, and the analysis stops.

We define a directed graph $G(x, y)$ to help explain how to construct the walk. Graph $G(x, y)$ is a directed graph with $|x||y| + 1$ nodes, corresponding roughly to the dynamic programming table between x and y. We label one node as the *`stop` node*. We label the other $|x||y|$ nodes using two-dimensional coordinates (i, j) with $0 \leq i < |x|$, and $0 \leq j < |y|$.

We now list all arcs between nodes. We label each with a grid walk character; this will be useful for analyzing $g(x, y, \rho)$. Consider an (i, j) with $0 \leq i < |x| - 1$ and $0 \leq j < |y| - 1$. For any (i, j) with $x_i \neq y_j$, we place five arcs:

- a `delete` arc from (i, j) to $(i + 1, j)$,
- a `replace` arc from (i, j) to $(i + 1, j + 1)$,
- an `insert` arc from (i, j) to $(i, j + 1)$,
- a `loop` arc from (i, j) to (i, j), and
- a `stop` arc from (i, j) to the `stop` node.

Table 1 This table defines a grid walk for non-matching characters in strings x and y, given the corresponding transcripts.

$\tau_k(x,\rho)$	$\tau_k(y,\rho)$	$g_k(x,y,\rho)$
hash-replace	hash-replace	`replace`
hash-replace	hash-insert	`delete`
hash-insert	hash-replace	`insert`
hash-insert	hash-insert	`loop`
hash-match	-	`stop`
-	hash-match	`stop`

These arcs are shown in Figure 3a. For any (i,j) with $x_i = y_j$, we place two edges: a `match` arc from (i,j) to $(i+1,j+1)$, and a `loop` arc from (i,j) to (i,j); see Figure 3b.

The rightmost and bottommost nodes of the grid are largely defined likewise, but arcs that lead to nonexistent nodes instead lead to the `stop` node.[12] For $0 \le j < |y|-1$ there is an `insert` arc from $(|x|-1,j)$ to $(|x|-1,j+1)$ a `stop` arc, `delete` arc, and `replace` arc from $(|x|-1,j)$ to the `stop` node, and a `loop` arc from $(|x|-1,j)$ to $(|x|-1,j)$. For $0 \le i < |x|-1$, there is a `delete` arc from $(i,|y|-1)$ to $(i+1,|y|-1)$, a `stop` arc, an `insert` arc, and a `replace` arc from $(i,|y|-1)$ to the `stop` node, and a `loop` arc from $(i,|y|-1)$ to $(i,|y|-1)$. Finally, node $(|x|-1,|y|-1)$ has a `loop` arc to $(|x|-1,|y|-1)$. See Figure 2.

The stop node has (for completeness) six self loops with labels `match`, `insert`, `replace`, `delete`, `loop`, and `stop`.

We now define the grid walk $g(x,y,\rho)$. We will use $G(x,y)$ to relate $g(x,y,\rho)$ to $h_\rho(x)$ and $h_\rho(y)$ in Lemmas 7 and 8.

We determine the kth character of $g(x,y,\rho)$, denoted $g_k(x,y,\rho)$, using $\tau_k(x,\rho)$ and $\tau_k(y,\rho)$, as well as $x_{i(x,k,\rho)}$ and $y_{i(y,k,\rho)}$. For $k > \min\{|\tau(x,\rho)|, |\tau(y,\rho)|\}$, $g_k(x,y,\rho) =$ `stop`.

If $x_{i(x,k,\rho)} \neq y_{i(y,k,\rho)}$, we define $g_k(x,y,\rho)$ using Table 1.

If $x_{i(x,k,\rho)} = y_{i(y,k,\rho)}$, then $\tau_k(x,\rho) = \tau_k(y,\rho)$. If $\tau_k(x,\rho) = \tau_k(y,\rho)$ is a hash-insert, then $g_k(x,y,\rho) =$ `loop`; otherwise, $g_k(x,y,\rho) =$ `match`.

We say that a grid walk is *complete* if both $\tau(x,\rho)$ and $\tau(y,\rho)$ are complete. We say that a grid walk is *alive* if it is complete and it does not contain `stop`.

The next lemma motivates this definition: the grid walk defines a path through the grid corresponding to the hashes of x and y.

▶ **Lemma 7.** *Consider a walk through $G(x,y)$ which at step i takes the edge with label corresponding to $g_i(x,y,\rho)$. Assume k is such that the prefix $g(x,y,\rho)[k]$ of length k is alive. Then after k steps, the walk arrives at node $(i(x,k,\rho), i(y,k,\rho))$.*

Proof. Our proof is by induction on k. We prove both that the walk arrives at node $(i(x,k,\rho), i(y,k,\rho))$, and that the walk is well-defined: the next character in $g(x,y,\rho)$ always corresponds to an outgoing edge of the current node.

For the base case $k = 0$ the proof is immediate, since $(i(x,0,\rho), i(y,0,\rho)) = (0,0)$. Furthermore, node $(0,0)$ has an outgoing `match` edge if and only if $x_0 = y_0$ (otherwise it has an outgoing `insert`, `delete`, and `replace` edge); similarity, $g_0(x,y,\rho) =$ `match` only if $x_0 = y_0$ (the rest of the cases follow likewise).

[12] Since x and y are \$-terminal, these nodes never satisfy $x_i = y_j$ except at $(|x|-1, |y|-1)$

Assume that after $k-1$ steps, the walk using $g(x,y,\rho)[k-1]$ arrives at node $(i(x,k-1,\rho), i(y,k-1,\rho))$. We begin by proving that the walk remains well-defined. We have $g_{k-1}(x,y,\rho) = \texttt{match}$ only if $x_{i(x,k-1,\rho)} = y_{i(y,k-1,\rho)}$; in this case $(i(x,k-1,\rho), i(y,k-1,\rho))$ has an outgoing `match` edge. We have $g_{k-1}(x,y,\rho) = \texttt{insert}$ (or `delete` or `replace`) only if $x_{i(x,k-1,\rho)} \neq y_{i(y,k-1,\rho)}$; again, node $(i(x,k-1,\rho), i(y,k-1,\rho))$ has the corresponding outgoing edge. All nodes have outgoing `loop` and `stop` edges.

Now we show that after k steps, the walk using $g(x,y,\rho)[k]$ arrives at node $(i(x,k,\rho), i(y,k,\rho))$. We split into five cases based on $g_{k-1}(x,y,\rho)$ (if $g_{k-1}(x,y,\rho) = \texttt{stop}$ the lemma no longer holds).

- `replace`: We have $\tau_k(x,\rho) =$ hash-replace, and $\tau_k(y,\rho) =$ hash-replace. Thus, $i(x,k,\rho) = i(x,k-1,\rho)+1$ and $i(y,k,\rho) = i(y,k-1,\rho)+1$. In $G(x,y)$, the edge labeled `replace` leads to node $(i(x,k-1,\rho)+1, i(y,k-1,\rho)+1)$.
- `match`: We have $\tau_k(x,\rho) =$ hash-replace, and $\tau_k(y,\rho) =$ hash-replace. Thus, $i(x,k,\rho) = i(x,k-1,\rho)+1$ and $i(y,k,\rho) = i(y,k-1,\rho)+1$. In $G(x,y)$, the edge labeled `match` leads to node $(i(x,k-1,\rho)+1, i(y,k-1,\rho)+1)$.
- `delete`: We have $\tau_k(x,\rho) =$ hash-replace, and $\tau_k(y,\rho) =$ hash-insert. Thus, $i(x,k,\rho) = i(x,k-1,\rho)+1$ and $i(y,k,\rho) = i(y,k-1,\rho)$. In $G(x,y)$, the edge labeled `insert` leads to node $(i(x,k-1,\rho)+1, i(y,k-1,\rho))$.
- `insert`: We have $\tau_k(x,\rho) =$ hash-insert, and $\tau_k(y,\rho) =$ hash-replace. Thus, $i(x,k,\rho) = i(x,k-1,\rho)$ and $i(y,k,\rho) = i(y,k-1,\rho)+1$. In $G(x,y)$, the edge labeled `insert` leads to node $(i(x,k-1,\rho), i(y,k-1,\rho)+1)$.
- `loop`: We have $\tau_k(x,\rho) =$ hash-insert, and $\tau_k(y,\rho) =$ hash-insert. Thus, $i(x,k,\rho) = i(x,k-1,\rho)$ and $i(y,k,\rho) = i(y,k-1,\rho)$. In $G(x,y)$, the edge labeled `loop` leads to node $(i(x,k-1,\rho), i(y,k-1,\rho))$. ◀

With this in mind, we can relate grid walks to hash collisions.

▶ **Lemma 8.** *Let x and y be any two strings, and ρ be any underlying function where both $\tau(x,\rho)$ and $\tau(y,\rho)$ are complete.*

Then $h_\rho(x) = h_\rho(y)$ if and only if $g(x,y,\rho)$ is alive. Furthermore, if $h_\rho(x) = h_\rho(y)$ then the path defined by $g(x,y,\rho)$ reaches node $(|x|,|y|)$.

Proof. *If direction*: Assume that $h_\rho(x) = h_\rho(y)$; we show that the path defined by $g(x,y,\rho)$ is alive and reaches $(|x|,|y|)$.

First, $g(x,y,\rho)$ must be alive: $g_k(x,y,\rho) = \texttt{stop}$ only when $x_{i(x,k,\rho)} \neq y_{i(y,k,\rho)}$ and either $\tau_k(x,\rho) =$ hash-match or $\tau_k(y,\rho) =$ hash-match, or when $k > \min\{|\tau(x,\rho)|, |\tau(y,\rho)|\}$. Since $x_{i(x,k,\rho)}$ (resp. $y_{i(y,k,\rho)}$) is appended to the hash on a hash-match, this contradicts $h_\rho(x) = h_\rho(y)$. Furthermore, we must have $|\tau(x,\rho)| = |\tau(y,\rho)|$ because $|\tau(x,\rho)| = |h_\rho(x)| = |h_\rho(y)| = |\tau(y,\rho)|$.

Since $\tau(x,\rho)$ and $\tau(y,\rho)$ are complete, $i(x,|\tau(x,\rho)|-1,\rho) = |x|$ and $i(y,\tau(y,\rho)-1,\rho) = |y|$. Thus, by Lemma 7, the walk reaches $(|x|,|y|)$.

Only If direction: We show that if $h_\rho(x) \neq h_\rho(y)$ then $g(x,y,\rho)$ is not alive. Let k be the smallest index such that the kth character of $h_\rho(x)$ is not equal to the kth character of $h_\rho(y)$. At least one of these characters cannot be $\perp$; thus either $\tau_k(x,\rho) =$ hash-match, or $\tau_k(y,\rho) =$ hash-match. If $x_{i(x,k,\rho)} \neq y_{i(x,k,\rho)}$, then $g_k(x,y,\rho) = \texttt{stop}$ and we are done. Otherwise, $x_{i(x,k,\rho)} = y_{i(y,k,\rho)}$; thus $\tau_k(x,\rho) = \tau_k(y,\rho)$, and the kth character of both $h_\rho(x)$ and $h_\rho(y)$ is $x_{i(x,k,\rho)} = y_{i(y,k,\rho)}$. But this contradicts the definition of k. ◀

We now bound the probability that the grid walk traverses each edge in $G(x,y)$.

▶ **Lemma 9.** *Let x and y be any two strings, and for any $k < 8d/(1-p_a) + 6\log n$ let E_k be the event that $i(x,k,\rho) < |x|$, $i(y,k,\rho) < |y|$, and $x_{i(x,k,\rho)} \neq y_{i(y,k,\rho)}$. Then if* $\Pr_\rho[E_k] > 0$, *the following four conditional bounds hold:*

$$\Pr_\rho[g_k(x,y,\rho) = \texttt{loop} \mid E_k] = p_a^2$$
$$\Pr_\rho[g_k(x,y,\rho) = \texttt{delete} \mid E_k] = p_a(1-p_a)p_r$$
$$\Pr_\rho[g_k(x,y,\rho) = \texttt{insert} \mid E_k] = p_a(1-p_a)p_r$$
$$\Pr_\rho[g_k(x,y,\rho) = \texttt{replace} \mid E_k] = (1-p_a)^2 p_r^2.$$

Proof. We have $|\tau(x,\rho)| > k$ and $|\tau(y,\rho)| > k$ from E_k. Thus:
- $\Pr_\rho(\tau_k(x,\rho) = \text{hash-insert} \mid E_k) = p_a$
- $\Pr_\rho(\tau_k(x,\rho) = \text{hash-replace} \mid E_k) = (1-p_a)p_r$
- $\Pr_\rho(\tau_k(x,\rho) = \text{hash-match} \mid E_k) = (1-p_a)(1-p_r)$.

The respective probabilities for $\tau_k(y,\rho)$ hold as well. Combining these probabilities with Table 1 gives the lemma. ◀

Transformations. We call a sequence of edits for a pair of strings x and y *greedy* if they can be applied to x in order from left to right, and all operations are performed on non-matching positions. We formally define this in Definition 10. With this in mind, we can simplify a sequence of edits for a given x and y, with the understanding that they will be applied greedily.

A *transformation* is a sequence of edits with position and character information removed: it is a sequence consisting only of `insert`, `delete`, and `replace`. We let $T(x,y)$ be the string that results from greedily applying the edits in T to x when x does not match y. We say that a transformation is *valid* for strings x and y if the total number of `delete` or `replace` operations in T is at most $|x|$, and the total number of `insert` or `replace` operations in T is at most $|y|$. The following definition formally defines how to apply these edits.

▶ **Definition 10.** *Let x and y be two \$-terminal strings, and let T be a transformation that is valid for x and y.*

If T is empty, $T(x,y) = x$. Otherwise we define $T(x,y)$ inductively. Let $T' = T[|T|-1]$ be T with the last operation removed, let $\sigma = T_{|T|-1}$ be the last operation in T, and let i be the smallest index such that the ith character of $T'(x,y)$ is not equal to y_i. Position i always exists if $T'(x,y) \neq y$ because x and y are \$-terminal; otherwise $i = 0$.[13]

We split into three cases depending on σ. If $\sigma = \texttt{insert}$, we obtain $T(x,y)$ by inserting y_i at position i in $T'(x,y)$. If $\sigma = \texttt{delete}$, we obtain $T(x,y)$ by deleting the ith character of $T'(x,y)$. Finally, if $\sigma = \texttt{replace}$, we obtain $T(x,y)$ by replacing the ith character of $T'(x,y)$ with y_i.

We say that a transformation T *solves* x and y if T is valid for x and y, $T(x,y) = y$, and for any $i < |T|$, the prefix $T' = T[i]$ satisfies $T'(x,y) \neq y$.

A classic observation is that edit distance operations can be applied from left to right, greedily skipping all matches. The following lemma shows that this intuition applies to transformations. Since Definition 10 does not allow characters to be appended onto the end of x, we use the appended character \$ to ensure that there is an optimal transformation between any pair of strings.

[13] The case where i is reset to 0 is included for completeness and will not be used in the rest of the paper. It only occurs when x is first transformed into y, and then a sequence of redundant edits (such as an equal number of inserts and deletes) are performed.

▶ **Lemma 11.** *Let x and y be two strings that do not contain \$. Then if* $\mathrm{ED}(x, y) = r$*,*
- *there exists a transformation T of length r that solves $x \circ \$$ and $y \circ \$$, and*
- *there does not exist any transformation T' of length $< r$ that solves $x \circ \$$ and $y \circ \$$.*

Proof. We prove a single statement implying the lemma: if $\widehat{T}$ is the shortest transformation that solves $x \circ \$$ and $y \circ \$$, then $|\widehat{T}| = \mathrm{ED}(x, y)$.

Let $\sigma_1, \dots \sigma_r$ be the sequence of edits applied to $x \circ \$$ to obtain $\widehat{T}(x \circ \$, y \circ \$)$ in Definition 10. These operations apply to increasing indices i because $\widehat{T}$ is the shortest transformation satisfying $\widehat{T}(x \circ \$, y \circ \$)$. Let σ_i be the last operation that applies to an index $i < |x|$. Let $\widehat{y} = \widehat{T}[i+1](x \circ \$, y \circ \$)$ be the string obtained after applying the operations of $\widehat{T}$ through σ_i. Clearly, x is a prefix of $\widehat{y}$. We claim that because $\widehat{T}$ is the shortest transformation, the operations in $\widehat{T}$ after σ_i must be $|\widehat{y}| - |x| - 1|$ `insert` operations. Clearly there must be at least $|\widehat{y} - |x| - 1|$ operations after σ_i because i is increasing and only one character in $\widehat{y}$ matches the final character \$ of x. By the same argument, if $\widehat{T}$ has any `insert` or `replace` operations it cannot meet this bound.

With this we have $\mathrm{ED}(x, y) \le |\widehat{T}|$ because we can apply $\sigma_1, \dots, \sigma_i$, followed by $|\widehat{y} - |x| - 1|$ insert operations to x to obtain y. This totals to $|\widehat{T}|$ operations overall.

We also have $\mathrm{ED}(x, y) \ge |\widehat{T}|$ by minimality of $\widehat{T}$ because any sequence of edits applied to x that obtains y will obtain $y \circ \$$ when applied to $x \circ \$$. ◀

For a given x, y, and ρ, we obtain the *transformation induced by x, y, and ρ*, denoted $\mathcal{T}(x, y, \rho)$, by removing all occurrences of `loop` and `match` from $g(x, y, \rho)$ if $g(x, y, \rho)$ is alive. Otherwise, $\mathcal{T}(x, y, \rho)$ is the empty string.

In Lemma 12 we show that strings x and y collide exactly when their induced transformation T solves x and y. This can be seen intuitively in Figure 2 – the grid walk is essentially a random walk through the dynamic programming table.

▶ **Lemma 12.** *Let x and y be two distinct strings and let $T = \mathcal{T}(x, y, \rho)$. Then $h_\rho(x) = h_\rho(y)$ if and only if T solves x and y.*

Proof. *If direction*: Assume T solves x and y. Since $x \ne y$, T must be nonempty; thus $g(x, y, \rho)$ is alive. By Lemma 8, $h_\rho(x) = h_\rho(y)$.

Only If direction: Assume $h_\rho(x) = h_\rho(y)$; by Lemma 8 $g(x, y, \rho)$ is alive.

Let $g(x, y, \rho)[k]$ be the prefix of $g(x, y, \rho)$ of length k, and let T^k be $g(x, y, \rho)[k]$ with `loop` and `match` removed. We prove by induction that $T^k(x[i(x, k, \rho)], y[i(y, k, \rho)]) = y[i(y, k, \rho)]$. This is trivially satisfied for $k = 0$.

Assume that $T^{k-1}(x[i(x, k-1, \rho)], y[i(y, k-1, \rho)]) = y[i(y, k-1, \rho)]$. We split into five cases based on the kth operation in $g(x, y, \rho)$.

- `match`: We must have $x_{i(x,k-1,\rho)} = y_{i(y,k-1,\rho)}$ and $T^k = T^{k-1}$. Furthermore, $i(x, k, \rho) = i(x, k-1, \rho) + 1$ and $i(y, k, \rho) = i(y, k-1, \rho) + 1$. Thus $T^k(x[i(x, k, \rho)], y[i(y, k, \rho)]) = T^{k-1}(x[i(x, k-1, \rho)], y[i(y, k-1, \rho)]) \circ x_{i(x,k-1,\rho)} = y[i(y, k, \rho)]$.
- `insert`: We have $i(x, k, \rho) = i(x, k-1, \rho)$ and $i(y, k, \rho) = i(y, k-1, \rho) + 1$. Thus, $T^{k-1}(x[i(x, k, \rho)], y[i(y, k, \rho)]) = y[i(y, k-1, \rho)]$ and $y[i(y, k, \rho)]$ differ only in the last character. Then $T^k(x[i(x, k, \rho)], y[i(y, k, \rho)]) = T^{k-1}(x[i(x, k-1, \rho)], y[i(y, k-1, \rho)]) \circ y_{i(y,k,\rho)-1} = y[i(y, k, \rho)]$.
- `replace`: We have $i(x, k, \rho) = i(x, k-1, \rho) + 1$ and $i(y, k, \rho) = i(y, k-1, \rho) + 1$. Thus, $T^{k-1}(x[i(x, k, \rho)], y[i(y, k, \rho)]) = y[i(y, k-1, \rho)] \circ x_{i(x,k,\rho)-1}$ and $y[i(y, k, \rho)]$ differ only in the last character. By definition, the final character of $T^{k-1}(x[i(x, k, \rho)], y[i(y, k, \rho)]$ is replaced with $y_{i(y,k,\rho)-1}$, obtaining $T^k(x[i(x, k, \rho)], y[i(y, k, \rho)]) = y[i(y, k-1, \rho)] \circ y_{i(y,k,\rho)-1} = y[i(y, k, \rho)]$.

- `delete`: We have $i(x,k,\rho) = i(x,k-1,\rho)+1$ and $i(y,k,\rho) = i(y,k-1,\rho)$. Thus, $T^{k-1}(x[i(x,k,\rho)], y[i(y,k,\rho)]) = y[i(y,k-1,\rho)] \circ x_{i(x,k,\rho)-1}$ and $y[i(y,k,\rho)]$ differ only in the last character (which is deleted). Then $T^k(x[i(x,k,\rho)], y[i(y,k,\rho)]) = T^{k-1}(x[i(x,k-1,\rho)], y[i(y,k-1,\rho)]) = y[i(y,k,\rho)]$.
- `loop`: We have $i(x,k,\rho) = i(x,k-1,\rho)$, $i(y,k,\rho) = i(y,k-1,\rho)$, and $T^k = T^{k-1}$. We immediately obtain $T^k(x[i(x,k,\rho)], y[i(y,k,\rho)]) = y[i(y,k,\rho)]$.

By Lemma 8, $g(x,y,\rho)$ reaches node $(|x|,|y|)$, so the above shows that with $k = |g(x,y,\rho)|$, $T(x,y) = y$. ◀

We are finally ready to prove Lemma 13, which forms the basis of our performance analysis.

▶ **Lemma 13.** *For any \$-terminal strings x and y, let T be a transformation of length t that is valid for x and y. Then*

$$p^t - 1/n^2 \le \Pr_\rho[T \text{ is a prefix of } \mathcal{T}(x,y,\rho)] \le p^t.$$

Proof. Define a *grid sequence* to be a sequence of grid walk operations.

Let G_T be the set of all grid sequences g such that g does not contain `stop`, and deleting `loop` and `match` from g results in a transformation T_g such that T is a prefix of T_g. Then by definition, if T is a prefix of $\mathcal{T}(x,y,\rho)$ then $g(x,y,\rho) \in G_T$; furthermore, if $g(x,y,\rho) \in G_T$ and $g(x,y,\rho)$ is complete, then T is a prefix of $\mathcal{T}(x,y,\rho)$.

We begin by proving that $\Pr_\rho[g(x,y,\rho) \in G_T] = p^t$. We prove this by induction on t; $t = 0$ is trivially satisfied. We assume that for any transformation T' of length $|T'| = t-1$, we have $\Pr_\rho[g(x,y,\rho) \in G_{T'}] = p^{t-1}$, and prove the above for any T with $|T| = t$.

Let σ be the last operation in T, and let $T' = T[|T|-1]$ be T with σ removed. Thus, $g(x,y,\rho) \in G_T$ only if there exist (possibly empty) grid sequences g' and g'' satisfying

- $g' \in G_{T'}$,
- g'' consists of `loop` and `match` operations concatenated onto the end of $g' \circ \sigma$, ending with a `match` operation if g'' is nonempty, and
- $g(x,y,\rho)$ consists of zero or more `loop` operations concatenated onto g''.

By definition of conditional probability,

$$\Pr[g(x,y,\rho) \in G_T] = \sum_{g'} \left(\Pr[g' \in G_{T'}] \cdot \sum_{g''} \Pr[g'' \in G_T \mid g' \in G_{T'}] \Pr[g(x,y,\rho) \in G_T \mid g'' \in G_T] \right).$$

We bound these terms one at a time.

Clearly there is only one g' satisfying the conditions, which can be obtained by taking the prefix of $g(x,y,\rho)$ before the final `insert`, `delete`, or `replace` operation. By the inductive hypothesis, $\sum_{g'} \Pr[g' \in G_{T'}] = p^{t-1}$.

We now bound $\Pr[g'' \in G_T \mid g' \in G_{T'}]$. The conditional means that we can invoke Lemma 9 (as $\Pr[E_k] = p^{t-1} > 0$).

We have $\Pr[g'' \in G_T \mid g' \in G_{T'}] = \Pr[g'' \in G_T \mid g' \circ \sigma \in G_T] \Pr[g' \circ \sigma \in G_T \mid g' \in G_{T'}]$.

We split into two cases depending on σ. Recall that $p_r = p_a/(1-p_a)$. Since T is valid, if $\sigma =$ `delete` or $\sigma =$ `replace` we cannot have $i(x,k,\rho) = |x|-1$; similarly if $\sigma =$ `insert` or $\sigma =$ `replace` we cannot have $i(y,k,\rho) = |y|-1$. Then by Lemma 9, if $\sigma =$ `delete` or $\sigma =$ `insert`, $\Pr(g' \circ \sigma \in G_T \mid g' \in G_{T'}) = p_a(1-p_a)p_r = p_a^2$. Similarly, if $\sigma =$ `replace`, $\Pr(g' \circ \sigma \in G_T \mid g' \in G_{T'}) = (1-p_a)^2 p_r^2 = p_a^2$. For any k such that $i(x,k,\rho) = i(y,k,\rho)$, $g_k(x,y,\rho) \neq$ `stop` by definition; meanwhile, if $i(x,k,\rho) \neq i(y,k,\rho)$ then $g_k(x,y,\rho) \neq$ `match`. Thus, $\sum_{g''} \Pr(g'' \in G_T \mid g' \circ \sigma \in G_T) = 1$.

Finally we bound $\Pr[g(x,y,\rho) \in G_T \mid g'' \in G_T]$. Let ℓ be the number of operations concatenated onto g'' to obtain $g(x,y,\rho)$. Then by Lemma 9,

$$\Pr[g(x,y,\rho) \in G_T \mid g'' \in G_T] = \sum_{\ell} p_a^{2\ell} = 1/(1-p_a^2).$$

Multiplying the above bounds, we have $\Pr[g(x,y,\rho) \in G_T] = p^{t-1}p_a^2/(1-p_a^2)$. Noting that $p = p_a^2/(1-p_a^2)$, we obtain $\Pr[g(x,y,\rho) \in G_T] = p^t$.

We have that if T is a prefix of $\mathcal{T}(x,y,\rho)$ then $g(x,y,\rho) \in G_T$; thus

$$\Pr_{\rho}[T \text{ is a prefix of } \mathcal{T}(x,y,\rho)] \le p_t.$$

Meanwhile, T is a prefix of $\mathcal{T}(x,y,\rho)$ if $g(x,y,\rho) \in G_T$ and $g(x,y,\rho)$ is complete. By the inclusion-exclusion principle,

$$\begin{aligned}\Pr[T \text{ is a prefix of } \mathcal{T}(x,y,\rho)] &= \Pr[g(x,y,\rho) \in G_T] + \Pr[g(x,y,\rho) \text{ is complete}] - \\ &\Pr[g(x,y,\rho) \in G_T \text{ or } g(x,y,\rho) \text{ is complete}] \ge p^t + \Pr[g(x,y,\rho) \text{ is complete}] - 1.\end{aligned}$$

We have that $\Pr[g(x,y,\rho) \text{ is complete}] = 1 - \Pr[\tau(x,\rho) \text{ or } \tau(y,\rho) \text{ is not complete}]$. By union bound and Lemma 6, $\Pr[g(x,y,\rho) \text{ is complete}] \ge 1 - 2/n^2$. Substituting, $\Pr[T \text{ is a prefix of } \mathcal{T}(x,y,\rho)] \ge p^t - 2/n^2$. ◀

4.2 Bounds on Collision Probabilities

We can now bound the probability that two strings collide.

▶ **Lemma 14.** *If x and y satisfy $\mathrm{ED}(x,y) \le r$, then $\Pr_\rho(h_\rho(x) = h_\rho(y)) \ge p^r - 2/n^2$.*

Proof. Because $\mathrm{ED}(x,y) \le r$, by Lemma 11 there exists a transformation T of length r that solves x and y. By Lemma 13, h induces T on x and y (which is sufficient for $h(x) = h(y)$ by Lemma 12) with probability $p^r - 2/n^2$. ◀

The corresponding upper bound requires that we sum over many possible transformations.

▶ **Lemma 15.** *If x and y satisfy $\mathrm{ED}(x,y) \ge cr$, then $\Pr_\rho(h_\rho(x) = h_\rho(y)) \le (3p)^{cr}$.*

Proof. Let $\mathcal{T}$ be the set of all transformations that solve x and y. By Lemma 12 and Lemma 13,

$$\Pr_{h\in\mathcal{H}}(h(x) = h(y)) = \sum_{T\in\mathcal{T}} p^{|T|}.$$

Thus, we want to find the $\mathcal{T}$ (for the given x and y) that maximizes this probability.

Since all pairs $T_1, T_2 \in \mathcal{T}$ solve x and y, there is no pair $T_1, T_2 \in \mathcal{T}$ such that T_1 is a prefix of T_2. Thus, $\mathcal{T}$ can be viewed as the leaves of a trie of branching factor at most 3, where each leaf has depth at least cr.

We show that without loss of generality all leaves are at depth cr. Consider a leaf T_1 at the maximum depth of the trie $i > cr$, and its siblings T_2 and T_3 if they exist. Collapse this leaf and its siblings, replacing them instead with a leaf T_p corresponding to their parent in the trie; call the resulting set $\mathcal{T}'$. Since we have added a transformation of length $i-1$ and removed at most three of length i, this changes the total cost of $\mathcal{T}$ by at least $p^{i-1} - 3p^i$; this is positive since $p \le 1/3$. Repeating this process results in a set $\mathcal{T}_M$ with all nodes at depth cr, where $\mathcal{T}_M$ gives larger collision probability than the original set $\mathcal{T}$.

There are at most 3^{cr} transformations in $\mathcal{T}_M$, each of length cr. Thus $\Pr_{h\in\mathcal{H}}(h(x) = h(y)) \le 3^{cr}p^{cr}$. ◀

The following special case is not used in our similarity search bounds, but may be useful in understanding performance on some datasets. In short, strings that do not have any matching characters achieve better performance bounds. It would be interesting to see if this analysis can be extended to other special cases.

▶ **Lemma 16.** *Let x and y be two \$-terminal strings with $\mathrm{ED}(x,y) \geq cr$ such that for all $i < |x| - 1$ and $j < |y| - 1$, $x_i \neq y_j$. Then* $\Pr_\rho(h_\rho(x) = h_\rho(y)) \leq (2p/(1-p))^{cr}$.

Proof. Let $\widehat{x}$ and $\widehat{y}$ be arbitrary \$-terminal strings of length cr with no other characters in common. We use grid walks on $G(\widehat{x},\widehat{y})$ to reason about grid walks on $G(x,y)$.

Let $GR(i,j)$ be the set of all grid walks reaching node (i,j) in $G(\widehat{x},\widehat{y})$. Let $W(i,j) = \Pr[g(\widehat{x},\widehat{y},\rho) \in GR(i,j)]$. We have $W(0,0) = 1$.

Clearly, $GR(i,j)$ is a subset of $GR(i-1,j) \cup GR(i-1,j-1) \cup GR(i,j-1)$. In fact, using a case-by-case analysis essentially identical to that of Lemma 13,

$$W(i,j) \leq p \cdot W(i-1,j) + p \cdot W(i-1,j-1) + p \cdot W(i,j-1).$$

We take $W(i^*,-1) = 0 = W(-1,j^*)$ for all i^* and j^* so that we can state this recursion without border cases.

We show by induction that if $\max i,j = \ell$, then $W(i,j) \leq (2p/(1-p))^\ell$. This is already satisfied for $\ell = 0$.

Assume that the induction is satisfied for all $W(i^*,j^*)$ with $\max\{i^*,j^*\} = \ell - 1$. For all (i,j) such that $\max\{i,j\} = \ell$, at most two of $(i-1,j-1)$, $(i-1,j)$, and $(i,j-1)$ have max $\ell - 1$; the remaining pair has max ℓ. Thus

$$W(i,j) \leq p\left(\frac{2p}{1-p}\right)^{\ell-1} + p\left(\frac{2p}{1-p}\right)^{\ell-1} + p\left(\frac{2p}{1-p}\right)^{\ell} \leq \left(\frac{2p}{1-p}\right)^{\ell}$$

All grid walks in $G(x,y)$ that go through $(|x|-1,|y|-1)$ must be in $GR(|x|-1,|y|-1)$. Since we must have $\max\{|x|,|y|\} = cr + 1$, the proof is complete. ◀

4.3 Final Running Time for Approximate Similarity Search

In this section, we describe how to get from our LSH to an algorithm satisfying Definition 3, proving Theorem 1.

Space and Preprocessing. To preprocess, we first pick $R = \Theta(1/p_1)$ underlying hash functions $\rho_1, \rho_2, \ldots, \rho_R$. For each string x stored in the database, we calculate $h_{\rho_1}(x), \ldots, h_{\rho_R}(x)$, and store them in a dictionary data structure for fast lookups (for example, these can be stored in a hash table, where each $h_\rho(x)$ has a back pointer to x). We set $1/p_1 = 3^r n^{1/c}$ (see the discussion below), leading to space $\widetilde{O}(d3^r n^{1+1/c})$.

We store the underlying functions $\rho_1, \rho_2, \ldots \rho_R$ so they can be used during queries. In Section 4.5, we show that these functions can be stored in $\widetilde{O}(|\Sigma|dR)$ space, which is a lower order term if $|\Sigma| = O(n)$.

In the common case that $|\Sigma| = O(n/d)$, the underlying functions are cheap to store, and we can further decrease the space. For each x, we can store a random $\log n$-bit hash of $h_\rho(x)$ for all ρ, rather than the full hash string of length $\Theta(d)$. This gives a space bound of $\widetilde{O}(3^r n^{1+1/c} + dn)$.

Queries. For a given query q, we calculate $h_1(q), h_2(q), \ldots, h_R(q)$. For each database string x that collides with q (i.e. for each x such that there exists an i with $h_{\rho_i}(q) = h_{\rho_i}(x)$), we calculate $\mathrm{ED}(x, q)$. We return x if the distance is at most cr. After repeating this for all R underlying functions, we return that there is no close point.

Correctness of the data structure follows from the definition of p_1: if $\mathrm{ED}(q, x) \le r$, then after $\Theta(1/p_1)$ independent hash functions, q and x collide on at least one hash function with constant probability.

The cost of each repetition is the cost to hash, plus the number of database elements at distance $> cr$ that collide with q. The cost to hash is $O(d/(1-p_a) + \log n)$ by definition, and the cost to test if two strings have distance at most cr is $O(dcr)$ by [36]. The number of elements with distance $> cr$ that collide with q is at most np_2 in expectation. Thus our total expected cost can be written

$$O\left(\frac{1}{p_1}\left(\frac{d}{1-p_a} + \log n + (dcr)np_2\right)\right).$$

This can be minimized (up to a factor $O(\log n)$) by setting $p_2 = 1/ncr$ (recall that $p_a \le 1/2$).

Thus, we set $p_2 = 1/ncr$, which occurs at $p = 1/(3(ncr)^{1/cr})$. Using this value of p, we get $p_1 \ge p^r = \Omega(1/(r3^r n^{1/c}))$.

Putting this all together, the expected query time is $\widetilde{O}(d3^r n^{1/c})$.

4.4 Approximate Nearest Neighbor

In this section we generalize Section 4.3 to prove Theorem 2. Let $R = \{i \in \{1, \ldots, d\} \mid 3^i n^{1/c} \le n\}$. We build $O(\log n)$ copies of the data structure described in Section 4.3 for each $r^* \in R$.

Queries. We iterate through each $r^* \in R$ in increasing order, querying the data structure as described above. If we find a string at distance at most cr^* we stop and return it. If we reach an r^* such that $3^{r^*} n^{1/c} > n$, we simply scan through all strings to check which is the closest.

Assume the actual nearest neighbor is at distance r. By Chernoff bounds, we succeed with high probability when $r^* = r$; that is, we return a string at distance at most cr. Thus, the cost is at most $\sum_{r^*=1}^{r} \widetilde{O}(d3^{r^*} n^{1/c}) = \widetilde{O}(d3^r n^{1/c})$ with high probability.

Space. We build $O(\log n)$ copies of each data structure; thus the total space is $\sum_{r=1}^{r^*} \widetilde{O}(d3^r n^{1+1/c}) = \widetilde{O}(dn^2)$ by definition of r^*. We obtain preprocessing time $\widetilde{O}(dn^2)$ immediately.

4.5 Storing Underlying Functions

Our algorithm uses a large number of fully-random, real-number hashes; this causes issues with the space bounds since we need to store each hash. In this section we relax this assumption.

We modify ρ to hash to a uniformly random element of the set $\{0, \epsilon, 2\epsilon, \ldots, 1\}$. Since the domain of each ρ has size $O(|\Sigma|(d + \log n)$, this means that each ρ can be stored in $O(|\Sigma| \log(1/\epsilon)(d + \log n))$ bits of space.

Intuitively, setting $\epsilon = 1/n$ should not affect our query bounds, while still retaining the space bounds of Theorems 1 and 2. We show this in Lemma 17.

▶ **Lemma 17.** *With p_a and p_r increased by $\epsilon = 1/n$, and assuming $d = O(n)$, if x and y satisfy $\mathrm{ED}(x, y) \le r$, then $\Pr(h(x) = h(y) \ge \Omega(p^r - 2/n^2)$. If x' and y' satisfy $\mathrm{ED}(x', y') \ge cr$ then $\Pr((h(x) = h(y)) \le O((3p)^{cr})$.*

Proof. For simplicity, we let $\widehat{p_a} = p_a + \epsilon$ and $\widehat{p_r} = p_r + \epsilon$.

Since $p_1 = \Omega(1/(r3^r n^{1/c})$, we have (omitting constants for simplicity) $p = 1/(3n^{1/rc})$. Therefore, $p_a = \sqrt{1/(1 + 3n^{1/rc})} \gg 1/n$, and thus $p_r = p_a/(1 - p_a) \gg 1/n$. Thus, $p_a < \widehat{p_a} < p_a(1 + 1/n)$ and $p_r < \widehat{p_r} < p_r(1 + 1/n)$.

Let ϵ' satisfy

$$p(1-\epsilon') \le \frac{\widehat{p_a}(1-\widehat{p_a})\widehat{p_r}}{(1-\widehat{p_a}^2)} \le p(1+\epsilon') \quad \text{and} \quad p(1-\epsilon') \le \frac{(1-\widehat{p_a})^2\widehat{p_r}^2}{(1-\widehat{p_a}^2)} \le p(1+\epsilon'). \tag{1}$$

Then the proof of Lemma 13 gives that for any \$-terminal strings x and y, and any transformation T of length t,

$$(p(1-\epsilon'))^t - 1/n^2 \le \Pr_{\rho}[T \text{ is a prefix of } \mathcal{T}(x, y, \rho)] \le (p(1+\epsilon'))^t.$$

So long as $(1 \pm \epsilon')^t = \Theta(1)$ we are done. Clearly this is the case for $\epsilon' = O(1/n)$ since $t \le 2d = O(n)$. We prove each bound in Equation (1) one term at a time for $\epsilon' = O(1/n)$. First inequality (recall that $p_a \le 1/2$):

$$\frac{\widehat{p_a}(1-\widehat{p_a})\widehat{p_r}}{(1-\widehat{p_a}^2)} > \frac{p_a(1-p_a(1+1/n))p_r}{1-p_a^2} = p - \frac{p_a p_r}{n(1-p_a^2)} = p - \frac{p}{n(1-p_a)} = p(1-O(n))$$

Second inequality:

$$\begin{aligned}\frac{\widehat{p_a}(1-\widehat{p_a})\widehat{p_r}}{(1-\widehat{p_a}^2)} &< \frac{p_a(1+1/n)^2(1-p_a)p_r}{1-(p_a(1+1/n))^2} = \frac{p_a(1-p_a)p_r}{1/(1+1/n)^2 - p_a^2} \\ &= \frac{p_a(1-p_a)p_r}{1-O(1/n^2)-p_a^2} < \frac{p_a(1-p_a)p_r}{(1-p_a^2)(1-O(1/n^2))} \\ &= p(1+O(1/n^2))\end{aligned}$$

Third inequality (since $p_a \le 1/2$, $2p_a \le 4(1-p_a)^2$):

$$\begin{aligned}\frac{(1-\widehat{p_a})^2\widehat{p_r}^2}{(1-\widehat{p_a}^2)} &\ge \frac{(1-p_a(1+1/n))^2 p_r^2}{(1-p_a{}^2)} \\ &= \frac{(1-2p_a(1+1/n)+p_a^2(1+1/n)^2)p_r^2}{(1-p_a{}^2)} \\ &> \frac{((1-p_a)^2 - 2p_a/n)p_r^2}{(1-p_a{}^2)} = p - \frac{2p_a p_r^2}{n(1-p_a{}^2)} \\ &\ge p(1-O(1/n))\end{aligned}$$

Fourth inequality (this is largely the same as the second inequality):

$$\frac{(1-\widehat{p_a})^2\widehat{p_r}^2}{(1-\widehat{p_a}^2)} \le \frac{(1-p_a)^2 p_r(1+1/n)^2}{1-(p_a(1+1/n))^2} = p(1+O(1/n^2))$$ ◀

References

1 Thomas Dybdahl Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proc. 28th Symposium on Discrete Algorithms (SODA)*, pages 239–256. SIAM, 2017.

2 Josh Alman and Ryan Williams. Probabilistic polynomials and Hamming nearest neighbors. In *Proc. 56th Symposium on Foundations of Computer Science (FOCS)*, pages 136–150. IEEE, 2015.

3 Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. Approximate nearest neighbor search in high dimensions. In *Proc. International Congress of Mathematicians (ICM)*, pages 3271–3302, 2018.

4 Alexandr Andoni, Thijs Laarhoven, Ilya Razenshteyn, and Erik Waingarten. Optimal hashing-based time-space trade-offs for approximate near neighbors. In *Proc. 28th Symposium on Discrete Algorithms (SODA)*, pages 47–66. ACM-SIAM, 2017.

5 Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 2019.

6 Leonid Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *Journal of Experimental Algorithmics (JEA)*, 16:1–1, 2011.

7 Eric Brill and Robert C Moore. An improved error model for noisy channel spelling correction. In *Proc. 38th Meeting on Association for Computational Linguistics*, pages 286–293. Association for Computational Linguistics, 2000.

8 Diptarka Chakraborty, Elazar Goldenberg, and Michal Koucký. Streaming algorithms for embedding and computing edit distance in the low distance regime. In *Proc. 48th Annual Symposium on Theory of Computing (STOC)*, pages 712–725. ACM, 2016.

9 Ho-Leung Chan, Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. A linear size index for approximate pattern matching. In *Proc. 17th Symposium on Combinatorial Pattern Matching (CPM)*, pages 49–59. Springer, 2006.

10 Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. 34th Symposium on Theory of Computing (STOC)*, pages 380–388. ACM, 2002.

11 Flavio Chierichetti, Ravi Kumar, and Mohammad Mahdian. The complexity of LSH feasibility. *Theoretical Computer Science*, 530:89–101, 2014.

12 Tobias Christiani. A framework for similarity search with space-time tradeoffs using locality-sensitive filtering. In *Proc. 28th Symposium on Discrete Algorithms (SODA)*, pages 31–46. Society for Industrial and Applied Mathematics, 2017.

13 Tobias Christiani and Rasmus Pagh. Set similarity search beyond minhash. In *Proc. 49th Symposium on Theory of Computing (STOC)*, pages 1094–1107. ACM, 2017.

14 Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. Lower bounds for text indexing with mismatches and differences. In *Proc. 30th Symposium on Discrete Algorithms (SODA)*, pages 1146–1164. ACM-SIAM, 2019.

15 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. 36th ACM Symposium on Theory of Computing (STOC)*, pages 91–100. ACM, 2004.

16 Benjamin Coleman, Richard Baraniuk, and Anshumali Shrivastava. Sub-linear memory sketches for near neighbor search on streaming data. In *Proc. 27th International Conference on Machine Learning (ICML)*, pages 2089–2099. PMLR, 2020.

17 Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proc. 20th Conference on the World Wide Web (WWW)*, pages 577–586. ACM, 2011.

18 Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

19 Piotr Indyk. Approximate nearest neighbor under edit distance via product metrics. In *Proc. 15th Symposium on Discrete Algorithms (SODA)*, pages 646–650. ACM-SIAM, 2004.

20 Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. 30th Symposium on Theory of Computing (STOC)*, pages 604–613. ACM, 1998.

21 Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

22 Tamer Kahveci, Vebjorn Ljosa, and Ambuj K Singh. Speeding up whole-genome alignment by indexing frequency vectors. *Bioinformatics*, 20(13):2122–2134, 2004.

23 Subhash Khot and Assaf Naor. Nonembeddability theorems via fourier analysis. *Mathematische Annalen*, 334(4):821–852, 2006.

24 Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, Chi-Kwong Wong, and Siu-Ming Yiu. Compressed indexing and local alignment of dna. *Bioinformatics*, 24(6):791–797, 2008.

25 Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *Transactions on Knowledge and Data Engineering (TKDE)*, 32(8):1475–1488, 2019.

26 Moritz G Maaß and Johannes Nowak. Text indexing with errors. In *Proc. 16th Symposium on Combinatorial Pattern Matching (CPM)*, pages 21–32. Springer, 2005.

27 Yury A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2020.

28 Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 1994.

29 Guillaume Marçais, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. Locality sensitive hashing for the edit distance. *Bioinformatics*, 35(14):i127–i135, 2019.

30 Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis.* Cambridge university press, 2017.

31 Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

32 Rafail Ostrovsky and Yuval Rabani. Low distortion embeddings for edit distance. *Journal of the ACM (JACM)*, 54(5):23, 2007.

33 Ozgur Ozturk and Hakan Ferhatosmanoglu. Effective indexing and filtering for similarity search in large biosequence databases. In *Proc. 3rd Symposium on Bioinformatics and Bioengineering*, pages 359–366. IEEE, 2003.

34 Rasmus Pagh, Ninh Pham, Francesco Silvestri, and Morten Stöckel. I/O-efficient similarity join. *Algorithmica*, 78(4):1263–1283, 2017.

35 Aviad Rubinstein. Hardness of approximate nearest neighbor search. In *Proc. 50th Symposium on Theory of Computing (STOC)*, pages 1260–1268. ACM, 2018.

36 Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1-3):100–118, 1985.

37 Yiqiu Wang, Anshumali Shrivastava, Jonathan Wang, and Junghee Ryu. Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 889–903. ACM, 2018.

38 W John Wilbur, Won Kim, and Natalie Xie. Spelling correction in the pubmed search engine. *Information retrieval*, 9(5):543–564, 2006.

39 Haoyu Zhang and Qin Zhang. Embedjoin: Efficient edit similarity joins via embeddings. In *Proc. 23rd International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 585–594. ACM, 2017.

40 Haoyu Zhang and Qin Zhang. Minjoin: Efficient edit similarity joins via local hash minima. In *Proc. 25th International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1093–1103. ACM, 2019.