MinSearch: An Efficient Algorithm for Similarity Search under Edit Distance*

Haoyu Zhang
hz30@umail.iu.edu
Indiana University Bloomington
Bloomington, IN, USA

Qin Zhang†
qzhangcs@indiana.edu
Indiana University Bloomington
Bloomington, IN, USA

ABSTRACT

We study a fundamental problem in data analytics: similarity search under edit distance (or, *edit similarity search* for short). In this problem we try to build an index on a set of n strings $S = \{s_1, \ldots, s_n\}$, with the goal of answering the following two types of queries: (1) the *threshold* query: given a query string t and a threshold K, output all $s_i \in S$ such that the edit distance between s_i and t is at most K; (2) the *top-k* query: given a query string t, output the k strings in S that are closest to t in terms of edit distance. Edit similarity search has numerous applications in bioinformatics, databases, data mining, information retrieval, etc., and has been studied extensively in the literature.

In this paper we propose a novel algorithm for edit similarity search named `MinSearch`. The algorithm is randomized, and we can show mathematically that it outputs the correct answer with high probability for both types of queries. We have conducted an extensive set of experiments on `MinSearch`, and compared it with the best existing algorithms for edit similarity search. Our experiments show that `MinSearch` has a clear advantage (often in orders of magnitudes) against the best previous algorithms in query time, and `MinSearch` is always one of the best among all competitors in the indexing time and space usage. Finally, `MinSearch` achieves perfect accuracy for both types of queries on all datasets that we have tested.

ACM Reference Format:
Haoyu Zhang and Qin Zhang. 2020. MinSearch: An Efficient Algorithm for Similarity Search under Edit Distance. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403099

1 INTRODUCTION

In this paper we study the following problem: given a set of n strings $S = \{s_1, \ldots, s_n\}$, and a query string t, output all strings in S satisfying certain query criteria. We consider two types of queries on string similarity under edit distance (ED): (1) the *threshold* query: given a threshold K, output all $s_i \in S$ such that $ED(s_i, t) \leq K$, and (2) the *top-k* query: output a set $O \subseteq S$ of k strings such that for any $s_i \in O$ and $s_j \in O \backslash S$, we have $ED(s_i, t) \leq ED(s_j, t)$. Recall that $ED(s, t)$ is defined to be the minimum number of insertions, deletions and substitutions needed to transform s to t.

The problem of similarity search under edit distance (or, *edit similarity search* for short) has numerous applications in bioinformatics, databases, data mining and information retrieval. For example, given a database of papers and a new paper, we want to find out whether the new paper has any preliminary versions, or whether it is similar to some previous papers for the purpose of plagiarism detection. In healthcare, we want to find all DNA sequences in a database that are similar to a given patient's DNA sequence; those similar sequences may provide useful information for the treatment. Edit similarity search has been studied extensively for almost two decades [3–5, 8, 10, 11, 14, 15, 20, 23, 24, 27]. We leave a discussion on the related work to Section 6.

In this paper, we propose a novel algorithm for edit similarity search named `MinSearch`. `MinSearch` is built upon on a string partition scheme recently proposed for a related problem called *edit similarity joins* [9], where we are given a set of strings $S = \{s_1, \ldots, s_n\}$ and a threshold value K, the goal is to output all pairs (s_i, s_j) such that $ED(s_i, s_j) \leq K$. However, due to the inherently different nature of the two problems, we need to make significant modifications and augmentations in order to adapt the partition technique to the setting of edit similarity search. One major difference is that in edit similarity joins, the threshold K is given as an input, and the string partition algorithm inherently depends on the value K. While in edit similarity search, in the threshold version K is given at the time of query, and in the top-k version there is *no* threshold even at the query time. To handle these situations we need to design a hierarchical partition scheme. See Section 2 for the details.

Another issue with the partition scheme in [9] is that it may not work well on short strings compared with long strings. We found that this is largely caused by small repeats in the short strings, and propose a method to fix this issue. We also use a number of filters to reduce the workload of the verification step so as to improve the overall performance of `MinSearch`.

We have conducted an extensive set of experiments; our results are presented in Section 5. Though `MinSearch` is randomized and may have false negatives (theoretically with a very small probability), we found that `MinSearch` achieves 100% accuracy on all datasets that we have tested. We observe that for *both* short and long strings, `MinSearch` significantly outperforms all existing algorithms in terms of query time (often by several orders of magnitudes), and is among one of the best in terms of indexing time

*Authors are supported in part by NSF IIS-1633215, CCF-1844234, and CCF-2006591.
†Corresponding author.

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403099

Notation	Definition
$\mathcal{S}$	set of input strings $\mathcal{S} = \{s_1, \ldots, s_n\}$
$s_{i..j}$	substring of s, from the i-th letter to the j-th letter
$\|s\|$	length of string s
$[x]$	$[x] = \{1, \ldots, x\}$
Σ	string alphabet
n	number of input strings
N	maximum length of strings in $\mathcal{S}$
q	size of q-grams
K	edit distance threshold for threshold query
k	number of returned strings in top-k query
F	$F : \Sigma^* \rightarrow (0, 1)$ a random hash function

Table 1: Table of Notations

and space usage. We view this practical performance as one of the main contributions of this paper.

2 THE INDEXING ALGORITHM

Overall, our indexing algorithm for each input string works as follows. We first compute a rank for each letter of the string, and then use the ranks of the letters to partition the string to a hierarchy of substrings, each of which is associated with a level. We then insert all substrings at the same level into the corresponding hash table for the level to the future query lookup.

In the following, we start by presenting the algorithms, and then give the time and space analysis. We list a set of notations in Table 1.

2.1 The Algorithm

In this section we present the indexing algorithm which will be used for both threshold query and top-k query.

The Ranks of Letters. The first step of the indexing is to generate ranks for letters of the input string. Given an input string s, Algorithm 1 returns an array of pairs $\mathcal{R} = \{(p_i, R_i)\}$, where p_i denotes the index of the letter in s corresponding to the i-th pair, and R_i stands for its *rank*.

Intuitively speaking, we assign each letter α in s a value which is a random hash value of the q-gram starting from α. The rank of α is the size of the largest neighborhood in which the value of α is the local minimum. Formally:

Definition 1 (Ranks of letters). *Given a string s, let $A[i] = F(s_{i..i+q-1})$ for $i = 2, \ldots, |s| - q + 1$, where $F : \Sigma^q \rightarrow (0, 1)$ is a random hash function. The* rank *of its i-th ($2 \le i \le |s| - q$) letter R_i is defined as the* maximum *integer $d > 0$ such that*

$$\forall j \in [i - d, i + d] \setminus \{i\}, j \in [1, |s| - q + 1] \text{ and } h[i] < h[j].$$

If such a d does not exist, then we set $R_i = 0$. Furthermore, we set the ranks of the 1-st and $(|s| + 1)$-th letters of s to be ∞. The ranks of letters in $s[|s| - q + 2.. |s|]$ are undefined (and will not be used in our algorithms).

Algorithm 1 first hashes all q-grams of s using hash function F to build the array A, and then computes the ranks of letters of s according to Definition 1. Finally, it includes letters whose ranks are at least r to the output set $\mathcal{R}$. Note that all letters included in $\mathcal{R}$ are sorted according to their indices in the increasing order. We add the first and $(|s| + 1)$-th letters with rank infinite for the convenience of the subsequent steps of the indexing algorithm.

Algorithm 1 Rank(s, r)

Input: input string s; minimum rank r
Output: rank array $\mathcal{R} = \{(p_i, R_i)\}$: p_i denotes the index of the letter in s corresponding to the i-th pair, and R_i is its rank

```
build an array A with |s| − q + 1 elements, where the i-th element A[i] = F(s_{i..i+q−1}), where F : Σ^q → (0, 1) is a random hash function
R ← {(1, ∞)}
for each i ∈ [|A|] do
    x ← 1
    while i − x ≥ 1 and i + x ≤ |A| do
        if A[i] < A[i + x] and A[i] < A[i − x] then x ← x + 1
        else
            exit the while loop
        end if
    end while
    if x > r then
        R ← R ∪ {(i, x − 1)}
    end if
end for
R ← R ∪ {(|s| + 1, ∞)}
```

Algorithm 2 Partition($s, \mathcal{R}, start, end$)

Input: input string s; rank array $\mathcal{R} = \{(p_i, R_i)\}$ of s; two indices $start$ and end
Output: set of partitions $\mathcal{P} = \{(s_{sub}, l)\}$, where (s_{sub}, l) denotes a substring s_{sub} with level l

```
if end = start + 1 then
    return ∅
end if
maxR = −∞, M ← ∅
i ← start + 1
while i < end do
    if R_i > maxR then
        maxR = R_i, M ← {i}
    end if
    if R_i = maxR then
        M ← M ∪ {i}
    end if
    i ← i + 1
end while
P ← ∅
M ← {start} ∪ M ∪ {end} sort elements in M in the increasing order; denote the j-th element of M by M[j].
for each M[j] ∈ M and M[j] ≠ end do
    u ← M[j], v ← M[j + 1]
    P ← P ∪ {(s_{p_u..p_v−1}, min(R_{p_u}, R_{p_v}))}
    P ← P ∪ Partition(s, R, u, v)
end for
```

The String Partition Process. The next step is to partition each input string. Given an input string s and the rank array $\mathcal{R}$ of s computed by Algorithm 1, Algorithm 2 returns a set $\mathcal{P}$ of partitions of s. We first introduce a concept called *valid partition*.

Definition 2 (Valid partition). *A substring $s_{p_i..p_j-1}$ ($j \ge i+1$) corresponding to $((p_i, R_i), (p_j, R_j))$ is a valid partition if and only if $j = i + 1$, or $\forall k \in \{i + 1, \ldots, j - 1\}, min(R_i, R_j) > R_k$.*

We record for each valid partition a *level* $l = \min(R_i, R_j)$, which indicates that the partition will be stored in the hash table $\mathcal{H}_l$. A partition with a high level is often long, and is thus hard to get

Algorithm 3 BuildIndex($\mathcal{S}$)

Input: set of input strings $\mathcal{S} = \{s_1, \ldots, s_n\}$
Output: set of hash tables $\{(\mathcal{H}_i, f_i)\}$, where for the i-th hash table, each string s is hashed into the $f_i(s)$-th bucket of $\mathcal{H}_i$

1: **for each** $s_i \in \mathcal{S}$ **do**
2: $\mathcal{R}_i \leftarrow$ Rank(s_i, 1)
3: $\mathcal{P}_i \leftarrow$ Partition(s_i, $\mathcal{R}_i$, 1, $|\mathcal{R}_i|$)
4: **for each** $(s_{sub}, l) \in \mathcal{P}_i$ **do**
5: **if** $\mathcal{H}_l$ does not exist **then**
6: create an empty hash table $\mathcal{H}_l$ and initialize a random hash function $f_l : \Sigma^* \rightarrow \mathbb{N}$
7: **end if**
8: store i in the $f_l(s_{sub})$-th bucket of $\mathcal{H}_l$
9: **end for**
10: **end for**

matched. Consequently, if two strings have a common substring with high level, then very likely they have small edit distance. With the level parameter we can organize the hash tables (to be described in Algorithm 4 and Algorithm 5) in a top-down manner to facilitate efficient queries.

Let us explain Algorithm 2 briefly in words. The algorithm first finds the maximum rank (denoted by *maxR*), and the corresponding set of letters whose indices are stored in the array M in the increasing order. We also include into M the starting and ending indices of the letters in s that we are going to consider (denoted by *start* and *end*). Next, we record the substrings between any two adjacent indices u, v in M, and call the algorithm recursively with parameters $start = p_u$ and $end = p_v$. It is easy to see that Algorithm 2 correctly computes all valid partitions satisfying Definition 2.

The Index. We now describe our main indexing algorithm, which is shown in Algorithm 3.

Given a set of input strings $\mathcal{S} = \{s_1, \ldots, s_n\}$, Algorithm 3 returns a set of hash tables $\{(\mathcal{H}_i, f_i)\}$, where $\mathcal{H}_i$ refers to the i-th hash table with strings as keys, and f_i is its hash function for calculating bucket ID for a given key. The algorithm is pretty straightforward. It first computes the set of partitions $\mathcal{P}_i$ for each string s_i, and then for each partition (s_{sub}, l) it adds the string ID into the $f_l(s_{sub})$-th bucket of $\mathcal{H}_l$.

At Line 2 we set $r = 1$ as the parameter for Algorithm 1. This is because we want to keep as much information as possible at the indexing stage. In the query algorithms (Algorithm 4 and 5), we will choose different r in order to make the query more efficient.

Finally we would like to mention that although the string partition algorithms (Algorithm 1 and 2) are inspired by the one in [9] (for edit similarity joins), we have made several important and non-trivial modifications to make it suitable for edit similarity search. Particularly,

- In edit similarity joins the query distance threshold K is given, while in the edit similarity search problem K is either not known in advance (for threshold query) or does not exist (for top-k query). This is why we need to introduce a hierarchical partition scheme based on the ranks of individual letters of the string.
- For edit similarity search we store partitions in different hash tables according to their levels, which enables us to design efficient algorithms for threshold and top-k queries. While in [9] all partitions are stored in the same hash table.

ID	Strings
s_1	ACGTTCGACTGGTTAG
s_2	CCGTTCGAACTGGTTAG
s_3	ACATTCGACTGGTTGAG
s_4	TCGAACGTTCGAACGT

Table 2: A collection of input strings

x	$F(x)$	x	$F(x)$	x	$F(x)$	x	$F(x)$
CTG	0.01	ACT	0.05	CGA	0.03	GAG	0.06
GTT	0.11	ATT	0.16	TTA	0.19	TAG	0.21
GGT	0.22	TGG	0.26	GAC	0.29	CGT	0.35
TTC	0.38	TCG	0.40	ACG	0.43	TCG	0.45
CCG	0.48	AAC	0.50	GAG	0.52	GAA	0.56
TGA	0.58	TTG	0.62	ACA	0.64	CAT	0.66

Table 3: Hash values $F(x)$ for 3-gram x

Level	Hash Tables
6	ACATTCGA $\rightarrow \{s_3\}$, CTGGTTGAG $\rightarrow \{s_3\}$
5	ACGTTCGA $\rightarrow \{s_1\}$, CTGGTTAG $\rightarrow \{s_1, s_2\}$ CCGTTCGAA $\rightarrow \{s_2\}$
4	TCGAACGTT $\rightarrow \{s_4\}$, CGAACGT $\rightarrow \{s_4\}$
2	AC $\rightarrow \{s_1, s_3\}$, CC $\rightarrow \{s_2\}$, CTG $\rightarrow \{s_1, s_2, s_3\}$ GTT $\rightarrow \{s_2, s_4\}$, CGAA $\rightarrow \{s_2\}$, GTTAG $\rightarrow \{s_1, s_2\}$ GTTCGA $\rightarrow \{s_1\}$, GTTGAG $\rightarrow \{s_3\}$ ATTCGA $\rightarrow \{s_3\}$, TCGAAC $\rightarrow \{s_4\}$
1	T $\rightarrow \{s_4\}$ GTT $\rightarrow \{s_1\}$, CGA $\rightarrow \{s_1, s_3\}$ ATT $\rightarrow \{s_3\}$, CGAAAC $\rightarrow \{s_4\}$

Table 4: Hash tables built with input strings

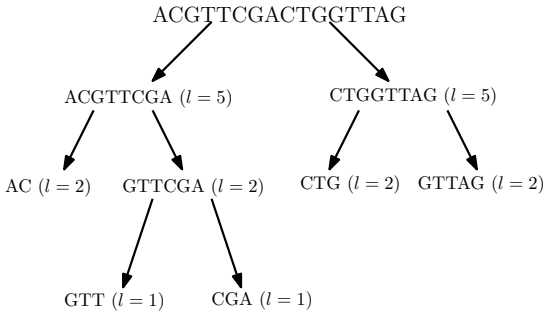


Figure 1: Example of string partition process for s_1

2.2 A Running Example

Before analyzing the time and space complexity of the indexing algorithm, we first give a running example. Table 2 shows a collection of input strings s_1, s_2, s_3, s_4, and Table 3 shows hash values for 3-grams returned by function F which is used by Algorithm 1. Figure 1 visualizes the partitions of s_1 as a rooted tree where each node corresponds to a call of Algorithm 2. Table 4 presents the hash tables built with input strings returned by Algorithm 3, where the key for a hash table is a string and the value is a list of IDs.

Let us describe the partition of s_1 in more detail. Algorithm 1 returns the rank set $\mathcal{R} = \{(1, \infty), (3, 2), (6, 2), (9, 5), (12, 2), (17, \infty)\}$. Then, we partition s_1 recursively. Algorithm 3 calls Algorithm 2 with parameters $i_s = 1, i_e = 6$. Algorithm 2 finds that $(9, 5)$ is the one with the largest rank, and produces partitions ACGTTCGA and CTGGTTAG with $l = 5$ (min$(\infty, 5)$). It then calls itself recursively with parameters $i_s = 1, i_e = 4$ and $i_s = 4, i_e = 6$. The recursion

stops when $i_e = i_s + 1$. We have 6 indices in rank set $\mathcal{R}$, and thus have 5 leaf nodes in the partition tree.

2.3 The Complexity Analysis

Clearly, the running time of Algorithm 3 is dominated by the sum of the running time of Algorithm 1 and that of Algorithm 2. We bound two parts separately.

For the convenience of the analysis we assume that each q-gram is unique and the hash function F has no collision. The latter is easy to achieve by keeping sufficient precision of the hash values. We will discuss the validity of the former assumption and how to further modify our partition algorithm to handle potential issues in Section 4.1.

Lemma 2.1. *The expected running time of Algorithm 1 is bounded by $O(N \ln N)$.*

Proof: First, the array A can be constructed in $O(N)$ time by a rolling hash (e.g., the one in [12]). For each $i \in [|A|]$, the expected time of computing the rank of $s[i]$ can be upper bounded as follows (x is the variable initialized at Line 4 of Algorithm 1)

$$\begin{aligned} & \sum_{i=1}^{|s|} \mathbf{Pr}[x = i] \cdot 2x \\ = \quad & 2 \cdot \sum_{i=1}^{|s|} (\mathbf{Pr}[x \geq i] - \mathbf{Pr}[x \geq i+1]) \cdot x \\ = \quad & O(1) \cdot \sum_{i=1}^{|s|} \left(\frac{1}{2x+1} - \frac{1}{2x+3} \right) \cdot x \\ = \quad & O(\ln |s|) = O(\ln N). \end{aligned}$$

The lemma follows since $|A| = O(N)$. □

Lemma 2.2. *The expected running time of Algorithm 2 is bounded by $O(N \ln N)$.*

Proof: First, if we view that the partition process on each string produces a tree as that in Figure 1, then it is easy to see that the total running time at each level of the tree is bounded by $O(N)$.

The next step is to bound the number of levels of the partition tree. Let us consider any root-to-leaf path $v_1..v_t$, where node v_1 corresponds to the root (the original string), and node v_t corresponds to the leaf substring. Let $|v_i|$ denote the length of the corresponding substring of v_i. Since the ranks of the letters in the string are assigned randomly, we have $\mathbf{E}[|v_{i+1}|] \leq \frac{|v_i|}{2}$ for $i = 1, \ldots, t-1$.

Define indicator variable $X_i = 0$ if $|v_{i+1}| \geq \frac{3|v_i|}{4}$, and $X_i = 1$ otherwise. By a Markov inequality, we have $\mathbf{Pr}[X_i = 0] \leq \frac{2}{3}$. Let $X = \sum_{i=1}^{t-1} X_i$. By a Chernoff bound, we have that with probability at least $1 - 2^{-\Omega(t)}$, we have $X \geq \frac{3}{4}t$, which means that $(1 \leq) X_t \leq \left(\frac{3}{4}\right)^{\frac{3}{4}t} \cdot N$, We thus have $t = O(\ln N)$.

Thus the expected running time is bounded by

$$O(N \ln N) + 2^{-100 \ln N} \cdot O(N \cdot N) = O(N \ln N),$$

where we have used the fact that in the worst case, we still have $t \leq N$. □

Combining Lemma 2.1 and Lemma 2.2, and using the linearity of expectation, we have the following.

Algorithm 4 MinSearch-Threshold $(\mathcal{S}, t, K)$

Input: set of strings $\mathcal{S} = \{s_1, \ldots, s_n\}$; query string t; distance threshold K
Output: $O \leftarrow \{i \mid s_i \in \mathcal{S}; ED(s_i, t) \leq K\}$

```
1:  {(H_i, f_i)} ← BuildIndex(S)
2:  O ← ∅, C ← ∅, α ← 120
3:  R ← Rank(t, r(t, α, K))                    ▷ r(t, α, K) is defined in Eq. (1)
4:  P ← Partition(t, R, 1, |R|)
5:  for each (t_sub, l) ∈ P do
6:      for each i in the f_l(t_sub)-th bucket of H_l do
7:          if ||s_i| − |t|| ≤ K then
8:              C ← C ∪ {s_i}
9:          end if
10:     end for
11: end for
12: remove duplicates in C
13: for i ∈ C do
14:     if ED(s_i, t) ≤ K then
15:         O ← O ∪ {i}
16:     end if
17: end for
```

Lemma 2.3. *The expected running time of Algorithm 3 is bounded by $O(nN \ln N)$.*

Finally, it is clear that the space usage of the algorithm is bounded by the running time. We arrive at the following theorem.

Theorem 2.4. *Under the assumption that for any input string, all of its q-grams are unique, the indexing algorithm (Algorithm 3) uses $O(nN \ln N)$ time and space in expectation.*

3 THE QUERY ALGORITHMS

In this section we describe algorithms for threshold query and top-k query. They share the same indexing step (Algorithm 3). Define

$$r(s, \alpha, K) = \max\left\{1, \left\lfloor \frac{|s| - q + 1 - \alpha K}{2\alpha K + 2} \right\rfloor\right\}. \tag{1}$$

3.1 Threshold Query

The algorithm for threshold query is described in Algorithm 4, which, given a set of input strings $\mathcal{S} = \{s_1, \ldots, s_n\}$, a query string t and a distance threshold K, returns all strings $s_i \in \mathcal{S}$ such that $ED(s_i, t) \leq K$. We note that our indexing can be used for different thresholds K at query time.

In Algorithm 4 we first build the index (i.e., sets of hash tables) for the set of input strings $\mathcal{S}$ using Algorithm 3. We then generate the partitions for the query string t using Algorithm 1 and 2. Next, for each partition (t_{sub}, l) of t, we add strings in the $f_l(t_{sub})$-th bucket of $\mathcal{H}_l$ into the candidate set C (with a simple filtering step at Line 7). Finally, we verify all candidates with an exact edit distance computation algorithm, and add all output strings to O. We have the following theorem. Due to space constraints, we leave the proof to Appendix C.

Theorem 3.1. *Given a query string t, Algorithm 4 outputs each $s_i \in \mathcal{S}$ with $ED(s_i, t) \leq K$ with probability at least 0.99. The expected running time of Algorithm 4 is $O(N \log N + \frac{nK^2}{|\mathcal{H}|} + |C|\, NK)$, where $|\mathcal{H}|$ the size of the hash tables and $|C|$ is the number of candidates for the verification step.*

Algorithm 5 MinSearch-Topk $(\mathcal{S}, t, k)$

Input: set of strings $\mathcal{S} = \{s_1, \ldots, s_n\}$, query string t, number of returned results k
Output: $O \leftarrow$ top-k closest strings in $\mathcal{S}$ to query string t in terms of edit distance

1: $\{(\mathcal{H}_i, f_i)\} \leftarrow$ BuildIndex($\mathcal{S}$)
2: $O \leftarrow \emptyset$, $\alpha \leftarrow 120$
3: $V \leftarrow \emptyset$ ▷ set of IDs of strings whose edit distances to t have been verified
4: $\mathcal{R} \leftarrow$ Rank(t, 1)
5: $\mathcal{P} \leftarrow$ Partition(t, $\mathcal{R}$, 1, $|\mathcal{R}|$)
6: sort all the partitions $\mathcal{P} = \{(t_{sub}, l)\}$ according to l in the increasing order
7: $Minl = \infty$ ▷ the minimum level to be considered
8: **for each** $(t_{sub}, l) \in \mathcal{P}$ **do**
9: **if** $|O| = k$ and $l < Minl$ **then**
10: **return** O
11: **end if**
12: **for each** $i \notin V$ and i in the $f_l(t_{sub})$-th bucket of $\mathcal{H}_l$ **do**
13: **if** $|O| < k$ **then**
14: $V \leftarrow V \cup \{i\}$
15: $dist \leftarrow ED(t, s_i)$
16: insert $(i, dist)$ into O
17: **end if**
18: **if** $||s_i| - |t|| \leq O.top.dist$ **then**
19: $V \leftarrow V \cup \{i\}$
20: $dist \leftarrow ED(t, s_i)$
21: **if** $dist < O.top.dist$ **then**
22: insert $(i, dist)$ into O
23: pop the top element of O
24: $Minl \leftarrow min(Minl, r(t, \alpha, O.top.dist))$
25: **end if**
26: **end if**
27: **end for**
28: **end for**

Remark 1. *We note that the* 0.99 *success probability can be amplified to* $(1 - 1/n^{100})$ *by repeating the partition process (Line 3 and 4) for* $O(\log n)$ *times and then taking the union of the partitions for the subsequent search. Though this is often not necessary in practice, as a single partition is already good enough.*

3.2 Top-k Query

The algorithm for top-k query is a bit more involved, and is described in Algorithm 5. Given a set of input strings $\mathcal{S} = \{s_1, \ldots, s_n\}$ and a query string t, the algorithm outputs k strings in $\mathcal{S}$ that are closest to t in terms of edit distance. The high level idea of the algorithm is to traverse the hash tables in the order from higher levels to lower levels. During the visit, it maintains a priority queue, which includes the current k results with smallest edit distances we find. At the same time, it keeps track of the lowest level it needs to visit and terminates when it reaches that level. Finally all strings in the priority queue are the final outputs.

Let us describe Algorithm 5 in a bit more detail. We first build the index on the input strings $\mathcal{S}$. Line 2 to 7 are the initialization steps, where we create an empty priority queue O which will contain a set of tuples $\{(s_i, dist_i)\}$ ranked according to $dist_i$. The element with the largest $dist_i$ is on the top of O (represented as $O.top$); we can thus quickly determine whether we should add a new string to O by comparing its edit distance to t with that of $O.top$. We maintain a set V to record the IDs of strings for which we have verified their edit distances to the query string t, and a variable $Minl$ which denotes the lowest level hash table that we need to search from.

We sort the partitions of the query string t according to their levels, and perform the search from the higher level to the lower level hash tables. If the algorithm reaches a partition with level smaller than $Minl$, it terminates and outputs at Line 10. Otherwise, for each unverified string in the $f_l(t_{sub})$-th bucket of $\mathcal{H}_l$, we add it to O if O has less than k elements (Line 16), or if it has a smaller edit distance to t than the top one of O. In the latter case the algorithm also pops the top element and updates $Minl$ to be $r(t, 120, O.top.dist)$, which is the lowest level at which any string whose edit distance to t is smaller than $O.top.dist$ will have substring being hashed to the hash table at that level. We have the following theorem. For a string t, let Top$k(t)$ denote the set of k strings in $\mathcal{S}$ that have the smallest edit distances to t.

Theorem 3.2. *Given a query string* t*, Algorithm 5 outputs each* $s_i \in$ *Topk*(t) *with probability at least 0.99. The expected running time is* $O(N \log k \log N + |V| NK_{\max})$*, where* V *is the set of candidates for the verification step and* $K_{\max} = \max_{i \in V} ED(s_i, t)$.

Proof: For convenience, let us assume that if we set $r = r(t, 120, K)$, then for any s_i such that $ED(s_i, t) \leq K$, s_i must have a substring colliding with t in some buckets in $\cup_{i \geq r} \mathcal{H}_i$ with high probability using the analysis of [9].

Let O^* be the exact top-k solution, and O be the output of Algorithm 5. We have $|O| = |O^*| = k$. We prove $O^* = O$ by contradiction. Suppose that there exists a pair $(s_i, d_i) \in O \backslash O^*$, then there must be another pair $(s_j, d_j) \in O^* \backslash O$ with $d_j \leq d_i$. Suppose Algorithm 5 first processes s_i, then we must have $s_j \in \cup_{z \geq d_i} \mathcal{H}_z$, and thus the algorithm will also process s_j, and consequently $s_j \in O$ if $s_i \in O$. A contradiction. Suppose Algorithm 5 first processes s_j, then since $d_j \leq d_i$ we also have that $s_j \in O$ if $s_i \in O$. Again a contradiction.

For the running time, similar as before, the expected time for computing the partition is $O(N \log N)$. The time for maintaining the priority queue is $O(|\mathcal{P}| \log k) = O(N \log k \log N)$. We have $|V|$ pairs of strings for which we need to compute the exact edit distance, which costs $O(|V| NK_{\max})$ where $K_{\max} = \max_{i \in V} ED(s_i, t)$. (In the worst case we have $K_{\max} = N$, but in practice $K_{\max}$ is typically much smaller.) The total running time is $O(N \log k \log N + |V| NK_{\max})$. □

We note that the same amplification (Remark 1) can be applied to Algorithm 5 to boost the success probability to $1 - 1/n^{100}$.

4 FURTHER IMPROVEMENTS

4.1 Improvements on Indexing Algorithms

In our analysis of the partition procedure (Algorithm 2 in Section 2) we have made an overoptimistic assumption that all q-grams are unique. Indeed, if the string is totally random, then setting $q = O(\log_{|\Sigma|} N)$ will satisfy this assumption with high probability.

However, real world datasets are never totally random. For example, in the DNA datasets that we use for our experiments, we have observed periodic substrings such as "CATACATACATA", "ACACACACAC", "GGGGGG" and "AAAAAAAAAAAAAA", and in

Algorithm 6 BuildArray(s)

Input: Input string s
Output: Array A containing hash values of q-grams

```
1:  T ← ∅                                                    ▷ An hash table
2:  A[1] ← F(s_{1..q})
3:  for i = 2, 3, ..., |s| − q + 1 do
4:      if A[i − 1] ≠ F(s_{i..i−q+1}) ∧ F(s_{i..i−q+1}) ∉ T then
5:          A[i] ← F(s_{i..i−q+1})
6:          T ← ∅
7:      else
8:          end ← i + q − 1
9:          while (end ≤ |s|) ∧ (A[i−1] = F(s_{i..end}) ∨ F(s_{i..i−q+1}) ∈ T)
    do
10:             end ← end + 1
11:         end while
12:         A[i] ← F(s_{i..end})
13:         add F(s_{i..end}) into T
14:     end if
15: end for
```

text datasets, we observe "is is", "the the", "barbar" and "inging" etc. Such periodic substrings are inevitable in real world datasets: they may come from systematic errors from the biological sequencing, typos in documents, or the biological sequences themselves. Consider an extreme case of a substring with 15 consecutive "A"s, if $q < 15$, then there will be continuous identical q-grams.

Having identical q-grams in a small neighborhood could be problematic: they will produce the same hash values and cause nearby letters having the same hash values. Recall that in Algorithm 1 we only record the rank of a letter if it has the strictly smallest hash value in a neighborhood of radius r. Thus, repeated q-grams will decrease the rank of letters and possibly result in zero index selection (to be included in $\mathcal{R}$) in a long substring, and the job of the query algorithm will be difficult (or even impossible) when two similar strings share a substring with repeated q-grams. This is particularly true for the top-k query since the search can go down to the very bottom level of the hash tables. We observe that in practice, for example, this issue *prevents* Algorithm 5 to achieve 100% accuracy on the READS dataset.

We propose a new procedure *BuildArray*$(\cdot)$ (Algorithm 6) to replace Line 1 of Algorithm 1 ("$A \leftarrow$ *BuildArray*(s)") to form a new partition algorithm for the indexing. Given a string s, Algorithm 6 builds an array A, where $A[i]$ refers to a hash value for the i-th letter of s. The algorithm maintains a table T to record hash values in the region where q-grams have repeated hash values. When the hash value of the current q-gram is different from that of the previous one and all values in T, we add it to A and empty T. Otherwise, we increment the ending index (i.e., *end*) of the substring until we find a distinct hash value (Line 9).

We note that our algorithm only deals with *consecutive* identical q-grams since we only check the previous hash value $A[i-1]$ at step i in the *for* loop. In other words, Algorithm 6 cannot handle substrings such as "ACACACACAC" and "barbar". It is actually very easy to extend this "duplication" detection step by comparing the current hash value with the previous $t > 1$ hash values in the array A. For example, using $t = 2$ we can already detect "ACACACACAC" and make all hash values generated from this region distinct. However, this comes with an increased time cost (for the indexing). On the other hand, avoiding consecutive repeats already resolves the accuracy issue mentioned above for top-k queries (the one for which the algorithm with the original partition scheme cannot achieve perfect accuracy), since now two adjacent letters already have different hash values.

4.2 Improvements on Query Algorithms

In this section we propose several tricks to further improve the query algorithms. The first two may be folklore. The third is new to the best of our knowledge.

String Ordering. The number of strings may be large in a hash bucket, and checking all of them (i.e., consider them as candidate output strings in C in Algorithm 4) can be time expensive. It is obvious that for two strings s and t with $ED(s,t) \le K$, if they share a common substring $x = s_{i..i+p-1} = t_{j..j+p-1}$ on their optimal alignment, then we must have $i \in [j-K, j+K]$. We can use this property to update our indexing and query algorithms to avoid visiting all the elements inside each hash bucket.

When computing the partitions for each string in Algorithm 2, we not only record the substring s_{sub} and its level l, but also the position of the first letter of s_{sub}, denoted by *pos*. Similarly, in hash tables built in Algorithm 3, we store not only the string index i but also the position of the substring *pos*, as a pair (i, pos). We then sort all the elements inside each bucket according to its *pos* in the increasing order. At the time of query, given the distance threshold K in the threshold query (or $O.top.dist$ in the top-k query), for a query (s_{sub}, l) with starting position *pos*, we only need to check the substrings in bucket $f_l(s_{sub})$ with starting positions in $[pos - K, pos + K]$ using a binary search.

Count Filtering. Exact computation of edit distance in Algorithm 4 (Line 14) and Algorithm 5 (Line 15, 20) can be expensive: Ukkonen's algorithm takes time $O(NK)$ where N is the string length and K is the distance threshold. This is often the bottleneck of the query algorithms (in particular, for top-k query). We can make use of the count filtering to exclude candidate pairs whose edit distance cannot be at most K, or at most $O.top.dist$ in the top-k query. The count filtering is based on the ℓ_1 distance of the q-gram vectors. We first convert each string s to a bag of q-grams, and represent it as a vector $v(s)$ with dimension $|\Sigma|^q$. It is easy to see that if $ED(s,t) \le K$, then we must have $\ell_1(v(s), v(t)) \le 2qK$. We thus can discard candidate pairs whose ℓ_1 distance is larger than $2qK$.

We comment that the q value here is different from that of the q-grams used in our indexing and query algorithms, and in practice we fix this q to be 2.

Neighborhood Filtering. We propose neighborhood filtering to further improve the efficiency for the top-k query. This filtering step is based on the triangle inequality of edit distance: for three strings s_1, s_2, s_3 with $ED(s_1, s_2) = d_1$ and $ED(s_2, s_3) = d_2$, then we must have $ED(s_1, s_3) \in [|d_1 - d_2|, d_1 + d_2]$. When building the index we store all *near neighbors* of each string s within edit distance γ where γ is a small constant parameter (we choose $\gamma = 1$ in our experiments). This can be done using the edit similarity join algorithm in [9] with threshold $K = \gamma$.

We can make use of the near neighbors of strings in the search in two ways. First, for each element s_i we add into the priority queue O

with $dist = ED(s_i, t) \leq O.top.dist$ (Line 16, 22 of Algorithm 5), we check all near neighbors s_j of s_i whether $dist_{ij} + dist < O.top.dist$, where $dist_{ij}$ is the edit distance between s_i and s_j which we have recorded in the indexing stage. If this is the case then we compute $ED(s_j, t)$ exactly, pop the top element of O if O is full, and insert $(j, dist_j)$ into O. This will reduce the value of $O.top.dist$ and speed up the "converge" of the search process. Second, for each string s_i that cannot be added into O due to $dist > O.top.dist$, we check all near neighbors s_j of s_i whether $|dist_{ij} - dist| \geq O.top.dist$, and add s_j to V if this is the case. This early pruning can avoid the computation of the exact edit distance between s_j and t.

5 EXPERIMENTS

5.1 The Setup

Datasets. We will make use of the following datasets, all of which are publicly available. The detailed statistics of these datasets are presented in Table 5.

- DBLP: A dataset of DBLP publication records (including the authors, title and key words of papers) obtained from the DBLP website.[1] We use the pre-processed dataset from [24], and convert all letters to uppercase and all special characters other than letters and numbers to space.
- TREC: A dataset of publication information of papers in 270 medical journals.[2] We concatenate the author, title and abstract of papers into strings, and convert all letters to uppercase and all special characters other than letters and numbers to space.
- READS: A dataset contains short DNA sequencing reads, which was used in the edit similarity joins and search competition [18]. We download the dataset from the competition website.[3]
- UNIREF: A dataset of protein sequences obtained from the website of UniProt project.[4] We remove all the strings with length shorter than 200.
- GENOME: A dataset of genome sequences obtained by randomly sampling substrings from Chromosome 20 of 50 people. We obtained the dataset from the code release of [26].[5]

Algorithms. We select the following state-of-the-art algorithms for edit similarity search as the competitors, according to the recommendation of [24]. We download the source codes of these algorithms from their websites.[6] Among them, HS-tree [24] and Bed-tree [27] are designed for both types of query, Pivotal [4] is only for threshold query, and Range [5] is only for top-k query. We will discuss these algorithms in more detail in Appendix B.

Metrics and Parameters. We use the following measurements in our experiments: indexing time, indexing memory usage, and query time. For each plot on the query time we perform 100 queries and then compute the average.

[1] https://dblp.uni-trier.de/db/
[2] http://trec.nist.gov/data/t9_filtering.html
[3] https://www2.informatik.hu-berlin.de/~leser/searchjoincompetition2013/
[4] http://www.uniprot.org/
[5] https://github.com/kedayuge/Embedjoin
[6] HS-tree: https://github.com/TsinghuaDatabaseGroup/Similarity-Search-and-Join.
Bed-tree: https://github.com/ZhangZhenjie/bed-tree.
Pivotal: http://people.csail.mit.edu/dongdeng/projects/pivotal/index.html.
Range: http://people.csail.mit.edu/dongdeng/projects/topksearch/index.html.

Datasets	n	Avg Len	Min Len	Max Len	$\|\Sigma\|$	Size(MB)
DBLP	863053	104	21	632	37	87
TREC	233435	1217	80	3947	37	270
READS	1500000	139	86	177	5	199
UNIREF	400000	435	201	5093	25	166
GENOME	50000	5000	4829	5152	4	238

Table 5: Statistics of datasets

For algorithms with input parameters and/or different subroutines (e.g., filtering methods), we always choose the best combination of parameters and subroutines for the best query time. In particular, Range and HS-tree have no input parameter, Pivotal has one parameter q (i.e., q-gram size), and Bed-tree has a number of parameters to choose, including *order type, gram length, bucket number, maximum bits, gram number, page size, buffer size, maximum node size.* MinSearch has one parameter α which we will discuss in Appendix A. We fix $q = \lceil \log_{|\Sigma|} N \rceil$ which is enough to avoid repeats in most parts of the strings.

Since MinSearch may have false negatives, we define accuracy for two type of queries as follows: for threshold query, the accuracy is the number of results found by MinSearch divided by the number of ground truths; note that there is no false positive due to the exact verification step. For top-k query, given the ground truth $O^* = \{o_1, o_2, \ldots, o_k\}$, and the output of MinSearch $O' = \{o'_1, o'_2, \ldots, o'_{k'}\}$ ($k' \leq k$), we create two sets $D = \{ED(o_1, t), ED(o_2, t), \ldots, ED(o_k, t)\}$ and $D' = \{ED(o'_1, t), ED(o'_2, t), \ldots, ED(o'_{k'}, t)\}$. We define the accuracy of the top-k query to be $|D \cap D'| / |D|$, which is a more robust accuracy measurements than directly comparing the two output sets O^* and O' (e.g., $|O^* \cap O'| / |O^*|$).

To be fair, we always choose the parameter α in MinSearch such that MinSearch achieves a 100% accuracy. As we note (see more details in Appendix A) that a small α value (e.g., $\alpha = 3$) can already achieve this goal, compared with $\alpha = 120$ (Line 2, Algorithm 4) which we use for the convenience of the theoretical analysis.

5.2 Performance Comparison

In this subsection we report the performance of MinSearch and compare it with previous algorithms. Due to space constraints we leave some empirical study of MinSearch to Appendix A.

Time and Space for Indexing. We show the memory usage and running time for indexing in Table 6.

For threshold query, the memory usage of Pivotal and MinSearch are the smallest among all algorithms, followed by Bed-tree. HS-tree uses a much larger amount of memory compared with others. With respect to the running time, Bed-tree uses the smallest amount of time. On most datasets MinSearch uses the second smallest amount of time, and HS-tree is the slowest. We note that although Bed-tree has the advantage on the indexing time for both types of queries, its query performance is not as good as the others, as we shall present shortly.

For top-k query, the memory usage of MinSearch is the best, followed by Bed-tree. Range and HS-tree use a significantly larger amount of memory compared with others. With respect to the running time, Bed-tree and MinSearch are still the best. Range spends a much longer time than others.

Time for Threshold Query. Figure 2 presents the average query time when varying edit threshold K. MinSearch is clearly the best

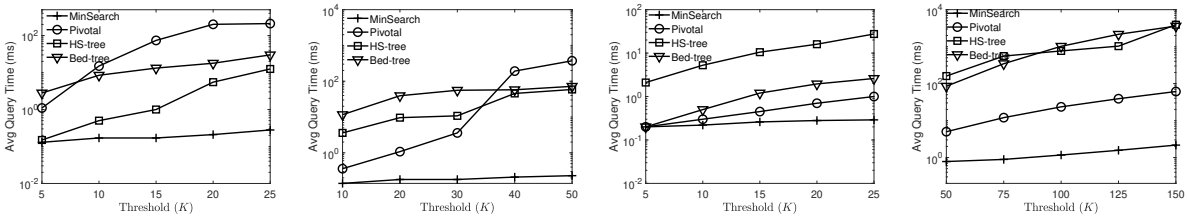


Figure 2: Running time for threshold query, varying threshold K

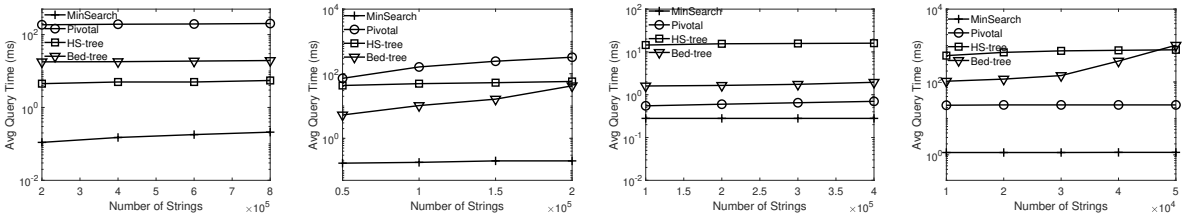


Figure 3: Running time for threshold query, varying number of strings n

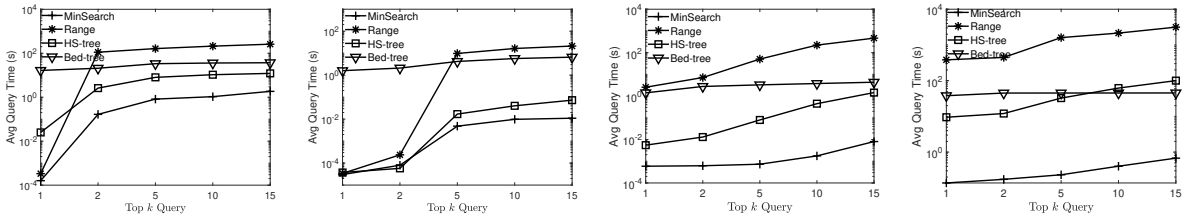


Figure 4: Running time for top-k query, varying k

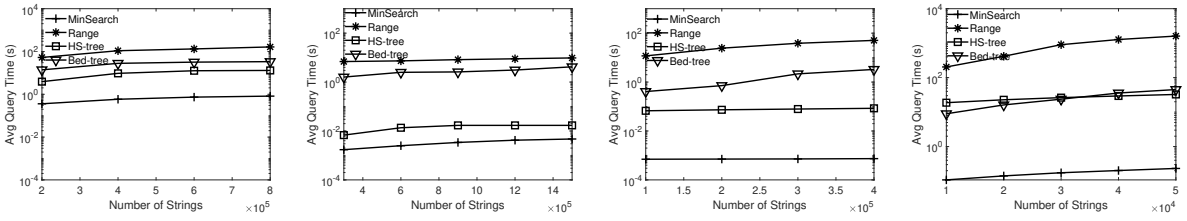


Figure 5: Running time for top-k query, varying number of strings n

on all datasets, with 31.1, 235.2, 3.5, 27.6 times speedup over the best competitors on DBLP($K = 25$), TREC($K = 50$), UNIREF($K = 25$), GENOME($K = 150$) respectively. Among the competitors, HS-tree is the second on DBLP and TREC datasets, and Pivotal is the second on UNIREF and GENOME datasets. There is abrupt time increase of Pivotal on TREC dataset when $K > 30$. The reason is that when K is large Pivotal has to choose a smaller q to guarantee $qK + 1 \leq N$, which results in a much larger number of candidate pairs for the verification step. The performance of HS-tree is much worse on UNIREF compared with GENOME, which is due to the fact that HS-tree builds a different tree for each string length, and the variance of string lengths is greater on UNIREF than GENOME. Bed-tree has a relatively stable performance and stays in the third place in most cases.

In Figure 3, we show the average query time when varying the number of strings n. MinSearch always performs the best, with 26.2, 210.6, 2.5, 20.1 times speedup over the best competitors on DBLP($n = 8 \times 10^5$), TREC($n = 2 \times 10^5$), UNIREF($n = 4 \times 10^5$), GENOME($n = 5 \times 10^4$) datasets respectively. The trend of all the algorithms is similar; the only exception is that the query time of Bed-tree increases suddenly on GENOME when n is large.

Time for Top-k Query. Figure 4 shows the average query time when varying the number of outputs k. We find that the top-k query is harder compared with the threshold query for all algorithms. MinSearch is again the best on all datasets, with 6.6, 7.2, 180.1, 66.8

Dataset	Algorithm	Memory usage(GB)	Time(s)
DBLP	MinSearch	1.2	15.5
	Range	26.8	73.4
	Pivotal	0.52	10.6
	HS-tree	59.4	17.6
	Bed-tree	10.0	6.0
READS	MinSearch	4.2	74.1
	Range	40.5	276.5
	HS-tree	100.4	201.2
	Bed-tree	10.1	14.0
TREC	MinSearch	3.7	33.1
	Pivotal	0.49	49.1
	HS-tree	55.0	92.1
	Bed-tree	10.1	10.0
UNIREF	MinSearch	2.7	39.3
	Range	39.9	160.2
	Pivotal	0.4	58.5
	HS-tree	98.3	49.7
	Bed-tree	10.1	8.0
GENOME	MinSearch	2.3	55.0
	Range	56.2	268.8
	Pivotal	0.63	66.8
	HS-tree	94.2	123.8
	Bed-tree	10.1	9.0

Table 6: Time and space usages for indexing. Pivotal is only for threshold query, and Range is only for top-k query.

times speedup over the best competitors on DBLP($k = 15$), READS($k = 15$), UNIREF($k = 15$), GENOME($k = 15$) datasets respectively. Among the competitors, HS-tree has the second best performance in most cases.

In Figure 5, we show the average query time when varying the number of strings n. MinSearch always performs the best, with 6.3, 3.6, 140.2, 114.2 times speedup over the best competitors on DBLP($n = 8 \times 10^5$), READS($n = 1.5 \times 10^6$), UNIREF($n = 4 \times 10^5$), GENOME($n = 5 \times 10^4$) datasets respectively. The trends of all the algorithms are similar.

6 RELATED WORK AND DISCUSSIONS

In this section we will review the literature on edit similarity search. We leave more discussion on related work in Appendix D.

Edit similarity search has been studied extensively in the literature [3–5, 8, 10, 11, 14, 15, 20, 23, 24, 27]. Most algorithms in these works use the so-called *filter-and-verify* framework: Given the query string t, we generate a set of *signatures* (e.g., substrings) of t and query the index for each signature. Each sub-query will give us a set of candidate strings. We may apply some additional filtering steps on the candidate strings, and then verify the distances between the remaining strings and t to generate the final output. In this framework two factors play important roles on the efficiency of the query: the *number of sub-queries* we generate, and the *quality of signatures*. We observe in our experiments that to some extent, the quality of signatures is the key factor top-k queries, while the number of sub-queries is more critical for threshold queries.

We observe that most existing algorithms do not perform well on relatively long strings. Naturally, in a dataset of long strings similar strings may share long substrings. If signatures can capture these long substrings as potential matches, then we may save a significant amount of time at the verification step. This is because long substring matches likely lead to true matches. Many existing algorithms make use of q-gram based signatures (e.g., [4, 10, 15, 20, 23]), where the value q is typically very small. As a result they cannot capture long substring matches.

REFERENCES

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *PVLDB*, pages 918–929, 2006.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[3] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

[4] Dong Deng, Guoliang Li, and Jianhua Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*, pages 673–684, 2014.

[5] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[6] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *PVLDB*, pages 518–529, 1999.

[7] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *PVLDB*, pages 491–500, 2001.

[8] Marios Hadjieleftheriou, Nick Koudas, and Divesh Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD*, pages 429–440, 2009.

[9] Zhang Haoyu and Zhang Qin. Minjoin: Efficient edit similarity joins via local hash minima. *KDD*, pages 1093–1103, 2019.

[10] Wang Jiannan, Li Guoliang, and Feng Jianhua. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. *SIGMOD*, pages 85–96, 2012.

[11] Tamer Kahveci and Ambuj K. Singh. Efficient index structures for string databases. In *PVLDB*, pages 351–360, 2001.

[12] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[13] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[14] Chen Li, Bin Wang, and Xiaochun Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In *PVLDB*, pages 303–314, 2007.

[15] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD*, pages 1033–1044, 2011.

[16] Venu Satuluri and Srinivasan Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[17] Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1-3):100–118, 1985.

[18] Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, and Ulf Leser. State-of-the-art in string similarity search and join. *SIGMOD Record*, 43(1):64–76, 2014.

[19] Jiannan Wang, Guoliang Li, and Jianhua Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[20] Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, and Zhenjie Zhang. Efficient and effective KNN sequence search with approximate n-grams. *PVLDB*, 7(1):1–12, 2013.

[21] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[22] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[23] Zhenglu Yang, Jianjun Yu, and Masaru Kitsuregawa. Fast algorithms for top-k approximate string matching. In *AAAI*, 2010.

[24] Minghe Yu, Jin Wang, Guoliang Li, Yong Zhang, Dong Deng, and Jianhua Feng. A unified framework for string similarity search with edit-distance constraint. *The VLDB Journal*, 26(2):249–274, 2017.

[25] Jiaqi Zhai, Yin Lou, and Johannes Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *SIGMOD*, pages 997–1008, 2011.

[26] Haoyu Zhang and Qin Zhang. Embedjoin: Efficient edit similarity joins via embeddings. *KDD*, pages 585–594, 2017.

[27] Zhenjie Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, and Divesh Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, pages 915–926, 2010.

A EXPERIMENTAL STUDY OF MINSEARCH

In this section we present some empirical study of MinSearch when varying parameter α and the effects of the improved partition scheme.

Influence of Parameter α for Threshold Query. We plot the influence of α on accuracy and running time for threshold query in Figure 6.

We observe that with $\alpha = 1$ and $\alpha = 3$, we can already achieve 100% accuracy for threshold query on GENOME and UNIREF respectively. This is much smaller than the theoretical result 120 obtained in the analysis of [9]. We note again that this phenomenon is consistent with the experiments (for similarity joins) in [9], and is due to the fact that the theoretical analysis in [9] considers the worst case and does not optimize on constants. Real world datasets are much better than the worse case, where similar pairs may share much longer substrings than the worst case.

The UNIREF dataset requires a larger α compared with GENOME. This may be because there are both short strings and long strings in UNIREF, and the threshold may be too large for short strings. While in GENOME the string length is almost uniform. We also observe that smaller thresholds give slightly worse accuracy. This is because the cardinality of pairs of strings with small edit distance is smaller. Thus, false negatives will have more impacts on the accuracy.

Since α is proportional to the number of partitions, larger α will lead to (slightly) larger query time in any dataset.

Influence of Parameter α for Top-k Query. We plot the influence of α on accuracy and running time for top-k query in Figure 7.

The observation for top-k query is similar to that for threshold query. On UNIREF dataset, when α is small, smaller k is more challenging for accuracy. This is because our measures allow algorithm to get "partial credits", which favors larger k. As expected, when α is larger, MinSearch is more accurate. We observe that with $\alpha = 0.5$ and $\alpha = 2$, we can achieve 100% accuracy for top-k query on GENOME and UNIREF datasets respectively. Again, as expected, larger α will lead to slightly larger query time.

Effects of the Improved Partition Scheme. We plot the effects of the improved partition scheme (Algorithm 6) on accuracy and running time on READS dataset in Figure 8.

We note that strings in READS are relatively short, and common substrings between some similar pairs have repeats due to systematic sequencing errors. Without the improved partition scheme we may miss some similar strings and achieve a relatively low accuracy (80% when $k = 15$). While with the improved partition scheme we can achieve 100% accuracy for all k values. Though we do not include the experimental results here, we notice that repeated substrings is less of an issue for other datasets; in most cases the accuracy of the original partitions scheme is already 100%.

The improved partitions scheme slightly increases the running time of the query algorithm, typically by no more than 50%.

B A DISCUSSION ON COMPETITOR ALGORITHMS

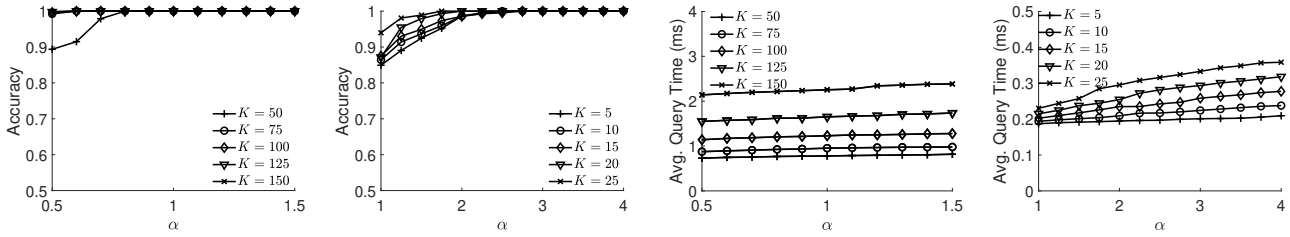
In our experiments we have compared MinSearch with four state-of-the-art algorithms for edit similarity search, among which HS-tree, Bed-tree, and Pivotal are signature based. We would like to add a bit more discussion on these algorithms, in particular, on the number of sub-queries they generate and the quality of their signatures.

- HS-tree [24] builds a tree based index for both threshold and top-k queries. For each string length it builds a different tree for that string length. The algorithm recursively partitions strings into substrings, generating substrings of length $N, N/2, \ldots, 1$. Substrings of the same length are stored in the nodes on the same level of the tree; the i-th node on level ℓ of the tree stores the i-th segment of length 2^ℓ of each string. An inverted list is built inside each node to enable efficient search at the time of query. Though HS-tree is strong in terms of signature quality, the number of sub-queries it needs to make to the index at query time is fairly large – for the threshold query it is $O(NK^2)$ where K is the distance threshold, which is much larger than the $O(N \log N)$ bound of MinSearch for large K (e.g., $K = 20\% \cdot N$).
- Bed-tree [27] is also a tree based algorithm, and can be used for both threshold query and top-k query. It uses several word ordering strategies together with the B^+-tree structure to perform the search; ordering strategies include *string orders*, *gram counting order* and *gram location order*. The weakness of Bed-tree is that these orders are often not very informative for the purpose of pruning candidates.
- Pivotal [4] is a prefix filter based algorithm for threshold query. It tries to improve the original prefix filtering [21] by selecting signatures with high pruning power and reducing the number of signatures needed for the query. More precisely, it gives all q-grams of a string a global order based on their frequency, and selects the first $(Kq + 1)$ q-grams as the "prefix" of the string. It then computes *pivots* of the string as $(K + 1)$ disjoint q-grams in the prefix of the string using a dynamic programming procedure. The issue with Pivotal is that we have to tune the parameter q for different query threshold K, and the index time/memory usage and query time are sensitive to q.

We have also compared MinSearch with Range [5], which uses a very different approach to support top-k query. Intuitively speaking, Range uses a trie data structure to fill multiple dynamic programming tables (one for each query string and an input string) simultaneously. The idea is to use shared common prefixes between strings can help to avoid filling unnecessary entries of the dynamic programming tables, and only consider pivotal entries in those tables. The main issue of Range is that its pruning power is limited if strings in the dataset do not share long prefixes (e.g., beyond short string datasets such as names, words, email titles, etc.).

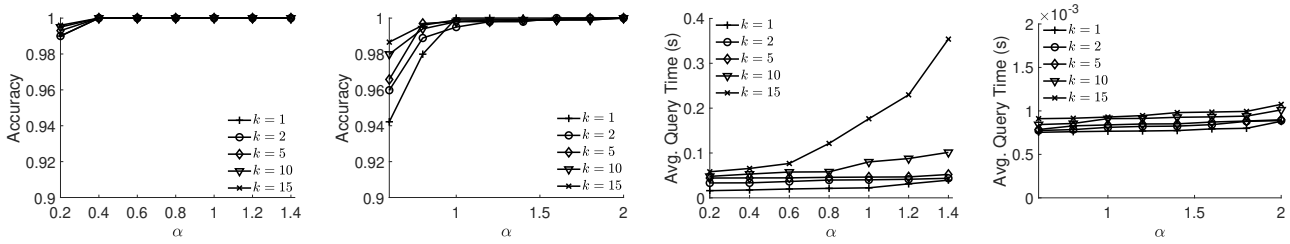
C PROOF OF THEOREM 3.1

The proof for the correctness of the algorithm is similar to the one in [9], where it was shown that for any pair of strings (s, t) with $ED(s, t) \le K$, if we partition them at letters with ranks larger $r = r(t, \alpha, K)$ for $\alpha = 120$, then with probability 0.99, s and t will share at least one common partition. The only difference is that in Algorithm 3 the partitions of each string are stored in different hash tables, while in the algorithm in [9] all partitions are stored in the same hash tables. But this is not an issue for the correctness since in Algorithm 4 we *effectively* search $\cup_{i \ge r} \mathcal{H}_i$ for each partition of



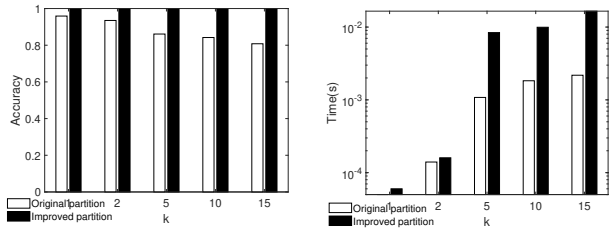
GENOME(Accuracy) UNIREF(Acuracy) GENOME(Time) UNIREF(Time)

Figure 6: Influence of α on accuracy and running time for threshold query



GENOME(Accuracy) UNIREF(Acuracy) GENOME(Time) UNIREF(Time)

Figure 7: Influence of α on accuracy and running time for top-k query



Accuracy Time

Figure 8: Influence of the improved partition scheme on top-k query on READS

the query string. More precisely, for each partition of the query string t, we read its level ℓ (computed in the string partition step) and search in the corresponding hash table $\mathcal{H}_\ell$.

For the running time, it was shown in [9] that with parameter $r = r(t, 120, K)$ where t is the query string, with probability $(1 - e^{-\Omega(K)})$, we have $|\mathcal{R}| = \Theta(K)$. By Lemma 2.3 we have that the rank computation and partition at Line 3 and 4 take $O(N \log N)$ time in expectation. Assuming that all hash tables have the same size, and partitions of all strings are evenly distributed into $|\mathcal{H}|$ buckets of any random hash table $\mathcal{H}$, the running time of Line 5 and 10 can be bounded by $O\left(K \cdot \frac{nK}{|\mathcal{H}|} \log N\right)$. The time cost for the verification step is $O(|C|\, NK)$ where $|C|$ is the number of candidate strings after deduplication, and $O(NK)$ is the running time for exactly testing whether $ED(s_i, t) \leq K$ using, e.g., Ukkonen's algorithm [17]. Therefore the expected total running time is $O\left(N \log N + \frac{nK^2}{|\mathcal{H}|} \log N + |C|\, NK\right)$.

D MORE RELATED WORK

As mentioned, edit similarity joins is a closely related problem, and has been studied extensively in the literature as well [1, 7, 9, 10, 13, 15, 19, 21, 26]. In fact, some algorithms for similarity joins can also be used for the threshold query in edit similarity search, such as the AdaptJoin [10] and the QChunk [15]. Note that in the search problem, we are allowed to spend more time on building the index in order to speed up the query. While in joins, we target a good balance of the time usage on the two components. Therefore algorithms that are good for joins may not fit search well without non-trivial modifications.

There are also many works studying similarity search on other distances and similarity measurements [1, 2, 6, 7, 10, 13, 16, 22, 25]. [22] designs algorithms for vector/set-based distances including the Cosine, Jaccard and Overlap distances; algorithms in [1, 2, 7, 10, 13] support both vector/set-based distances and edit distance. [6, 16, 25] give approximation algorithms for the similarity search problem, where [6] focuses on Hamming distance and [16, 25] study Cosine and Jaccard distances. It is well known that edit distance is harder to work with than set/vector-based distances in the sense that there is no efficient Locality Sensitive Hashing (LSH) for edit distance, while there are often good ones for vector/set-based distances. For example, [25] uses LSH-based ideas for the similarity search problem for Cosine and Jaccard distances.