The VLDB Journal (2024) 33:281–299
https://doi.org/10.1007/s00778-023-00806-z

REGULAR PAPER

MinJoin++: a fast algorithm for string similarity joins under edit distance

Nikolai Karpov[1] · Haoyu Zhang[2] · Qin Zhang[1]

Received: 20 July 2022 / Revised: 23 January 2023 / Accepted: 13 July 2023 / Published online: 21 August 2023

Abstract
We study the problem of computing similarity joins under edit distance on a set of strings. Edit similarity joins is a fundamental problem in databases, data mining and bioinformatics. It finds many applications in data cleaning and integration, collaborative filtering, genome sequence assembly, etc. This problem has attracted a lot of attention in the past two decades. However, all previous algorithms either cannot scale to long strings and large similarity thresholds, or suffer from imperfect accuracy. In this paper, we propose a new algorithm for edit similarity joins using a novel string partition-based approach. We show that, theoretically, our algorithm finds all similar pairs with high probability and runs in linear time (plus a data-dependent verification step). The algorithm can also be easily parallelized. Experiments on real-world datasets show that our algorithm outperforms the state-of-the-art algorithms for edit similarity joins by orders of magnitudes in running time and achieves perfect accuracy on most datasets that we have tested.

Keywords String similarity joins · Edit distance · Local hash minima

1 Introduction

Edit similarity joins is a fundamental problem in the database and data mining literature, and finds numerous applications in data cleaning and integration, collaborative filtering, genome sequence assembly, etc. In this problem, we are given a set of strings $\{s_1, \ldots, s_n\}$ and a distance threshold K, and asked to output all pairs of strings (s_i, s_j) such that $\mathrm{ED}(s_i, s_j) \leq K$, where $\mathrm{ED}(\cdot, \cdot)$ is the edit distance function, which is defined to be the minimum number of insertions, deletions and substitutions to transfer one string to another. There is a long line of research on edit similarity joins [1–3, 6, 8, 9, 11, 17–20].

A major challenge for most existing algorithms, as pointed out by the recent work [21], is that they do not scale well to long strings and large edit thresholds. Long strings and large thresholds are critical for applications involving long sequence data such as big documents and DNA sequences, where a small threshold K may just give zero output. For example, in the genome sequence assembly, in which the first step is to find all pairs of similar reads under edit distance, the third generation sequencing technology such as single molecule real time sequencing (SMRT) [12] generates reads of 1000–100,000 bps long with 12–18% sequencing errors (i.e., percentage of insertions, deletions and substitutions). Large threshold is also identified as the main challenge in a recent string similarity search/join competition [16].

Different from previous algorithms which are deterministic and return the exact answers, in [21] the authors proposed a randomized algorithm named `EmbedJoin`, which is more efficient on long strings and large thresholds. However, the accuracy (more precisely, the *recall*, i.e., the number of pairs found by the algorithm divided by the total number of similar pairs) of `EmbedJoin` is only 95–99% on real-world datasets tested in [21]. As we shall explain shortly, the imperfect accuracy is inherent to `EmbedJoin`. The main question we are going to address in this paper is:

Authors are ordered alphabetically.

✉ Qin Zhang
qzhangcs@indiana.edu

Nikolai Karpov
nkarpov@iu.edu

Haoyu Zhang
kedayuge@gmail.com

1 Indiana University, Bloomington, USA
2 Meta Inc., Menlo Park, CA, USA

Can we solve edit similarity joins efficiently on long string and large edit threshold while achieving perfect accuracy?

Our contribution We propose a novel algorithm named MinJoin++ to address the above question. In the high level, MinJoin++ first partitions each string into a set of substrings, and then uses hash join on these substrings to find all pairs of strings that share at least τ common substrings for a fixed threshold parameter τ. At the end, a verification step is used to remove all false positives.

We design a string partition scheme using *local hash minima*. The idea of our partition schemes is as follows: We first assign each letter α in the string s a rank, which is a random hash value of the q-gram starting from α. We then determine the *anchors* of string s using the following strategy: a letter α is an anchor if and only if its rank is the smallest among all letters in a certain neighborhood of α. At the end, we partition s at all of its anchors.

Via a rigorous mathematical analysis, we show that under our partition scheme, with properly chosen parameters, any pair of strings with edit distance at most K will share at least a constant fraction of partitions with high probability.

We also show that MinJoin++ can be easily implemented in the parallel computation environment.

We have verified the effectiveness of MinJoin++ by an extensive set of experiments. The experimental results show that MinJoin++ is able to achieve perfect accuracy on all tested datasets when there is an exact deterministic algorithm that can finish within 24 h (so that we get the ground truth). Moreover, MinJoin++ is faster than all previous algorithms by orders of magnitudes on datasets of long strings and large edit thresholds.

More on the motivation and applications As mentioned, this paper targets long strings, particularly biological sequences such as genome and protein sequenc es, whose lengths are from hundreds to tens of thousands. Edit distance has been widely used in the bioinformatics literature for comparing the similarity of strings [10, 13].

We would like to comment that MinJoin++ can also be generalized to handle sequence alignment scores, in which each insertion, deletion, or substitution carries a weight, while in edit distance, each operation has a unit cost. Due to the different weights of the edit operations, we have to use a more conservative bound of the number of edit operations. Fortunately, the absolute values of the weights in the score matrices are typically less than 10. For example, in the PAM250 matrix for amino acid, only 3 out of 576 entries have absolute values larger than 8. While we are *not* aware of any similarity join algorithm designed for the general sequence alignment scores, we believe that MinJoin++ will still significantly outperform the existing algorithms for edit similarity joins in the setting that we properly generalize all these algorithms for alignment scores.

Another scenario in which MinJoin++ excels is finding *near-duplicated* pairs of text strings, which arises frequently in the tasks such as *entity resolution* and *plagiarism detection*, where we are interested in finding pairs of strings such that one is edited from the other [14, 23]. For example, one dataset that we have used in our experiments contains the titles and abstracts of research papers. Nowadays it is common for a paper to have multiple online versions, including workshop, conference, journal, and (potentially multiple) arXiv versions. Identifying near-duplicates would be very useful for data cleaning and integration. In these applications, edit distance is naturally more effective measurement than token-based distance functions such as Jaccard similarity.

1.1 Related work

Many of the existing algorithms on edit similarity joins also follow the string partition framework. The performance of the algorithm is largely determined by the number of partitions generated for each string, and the number of queries made to the indices (e.g., hash tables) to search for similar strings.

We discuss several state-of-the-art algorithms according to the experimental studies in [7].

QChunk [11] is an algorithm based on string partition. QChunk first obtains a global order σ of q-grams. It then partitions each string into a set of chunks with starting positions $1, q+1, 2q+1, \ldots$, and stores the first $K+1$ chunks (according to the order σ) in a hash table. Next, for each string the algorithm queries the hash table with the string's first $(N - (\lceil (N-K)/q \rceil - K) + 1)$ q-grams according to σ to check if there is any match, where N is the string length.[1]

PassJoin [9] is another algorithm based on string partition. The algorithm partitions each string into $(K+1)$ equal-length segments, and records each segment into an inverted index. Next, for each string the algorithm queries some of the inverted indices to find similar strings; the number of queries made for each string is $\Theta(K^3)$, which is $\Theta(N^3)$ when K is proportional to N.

VChunk [19] is the algorithm that is closest to MinJoin++ among all algorithms that we are aware of. In VChunk, each string is partitioned into at least $(2K+1)$ chunks of possibly different lengths, determined by a *chunk boundary dictionary (CBD)*. More precisely, each string is cut at positions of appearances of each word in CBD to obtain its chunks. The CBD is data dependent and the optimal one is NP-hard to compute. In [19], the authors proposed a greedy algorithm for computing a CBD in time $O(n^2 N^2 / K)$, where

[1] Alternatively, for each string we can store the first $N - (\lceil (N-K)/q \rceil - K) + 1$ q-grams in the hash table, and make queries with the first $K+1$ chunks.

n is the number of input strings, and N is the maximum string length.

The algorithm EmbedJoin [21] uses a very different approach. EmbedJoin first embeds each string from the edit distance metric space to the Hamming space, translating the original problem to finding all pairs of strings that are close under Hamming distance. It then uses Locality Sensitive Hashing to compute (approximate) similarity joins in the Hamming space. However, the embedding algorithm employed by EmbedJoin has a worst case distance distortion K, which can be very large. Although in practice the distortion is much smaller, it still contributes a non-negligible percentage of false negatives which prevent a perfect accuracy.

Compared with existing algorithms, MinJoin++ has the following advantages:

– For each string, MinJoin++ only generates $O(K)$ partitions and effectively makes the same number of queries in the join stage, which are significantly smaller than QChunk and PassJoin.
– MinJoin++ can compute partitions of all strings in time $O(nN)$, i.e., linear in the input size, which is even faster than the computation of CBD in VChunk.
– MinJoin++ is able to reach perfect accuracy (100%) on datasets when we know the ground truth, compared with 95–99% of EmbedJoin. For large datasets on which we do not know the ground truth (when exact algorithms are unable to finish within 24 h), MinJoin++ also significantly outperforms EmbedJoin in terms of accuracy.

A comparison with MinHash We would like to note that MinJoin++ is very different from the following folklore algorithm using MinHash: For each string, we collect all of its q-grams and hash them to numbers, and then pick the one with the smallest hash value as the signature for the subsequent hash join. To improve the accuracy, we can pick multiple signatures using different hash functions for each string.

To see the difference, in MinJoin++ the hash values of the q-grams are used to partition a string to substrings/signatures, while in the MinHash-based algorithm the q-grams are the signatures themselves. In MinJoin++ we set q to be a small number (more precisely, $q = \Theta(\log_{|\Sigma|}(N/K))$ where Σ is the alphabet of the string) in order to make all q-grams distinct in every small neighborhood of the string. And one partition will give us all the signatures of the string. While in the MinHash-based approach, it is not clear how to find the best combination of the value q and the number of signatures (or, hash functions) for the purpose of achieving a good accuracy under a small running time. We are not aware of any theory for guiding the choices of q and the number of signatures in the MinHash-based algorithm. In Sect. 4.6, we will show experimentally that MinJoin++ significantly performs the MinHash-based algorithm in both accuracy and running time.

Other related work There is a large body of work on similarity joins under edit distance. A large number of the existing algorithms fall into the category called the *signature-based* approach, in which we compute for each string a set of signatures, and then apply various filtering methods to those signatures to select a set of candidate pairs for verification. All the string partition-based algorithms that we have already discussed belong to this category. Other algorithms in this category include GramCount [6], AllPair [2], FastSS [3], ListMerger [8], EDJoin [20], and AdaptJoin [18].

There are a few algorithms that use different approaches, including the embedding-based algorithm EmbedJoin discussed previously, the tree-based algorithm M-Tree [4], the enumeration-based algorithm PartEnum [1], and the trie-based algorithm TrieJoin [17]. However, except EmbedJoin, others' performance is not as good as the best partition-based approaches.

Dai et al. [5] proposed a convolutional embedding-based algorithm CNN-ED working on GPUs. However, CNN-ED is not suitable to run in our CPU computational environment.[2]

Compared with the preliminary version [22] The algorithm presented in this paper can be seen as an improvement of the MinJoin algorithm presented in the preliminary version [22]. This is also why we name our new algorithm MinJoin++. The key difference between MinJoin++ and MinJoin is that in MinJoin, we always set $\tau = 1$, that is, we will include all pairs of strings who share at least *one* common partition as candidate pairs. While in MinJoin++, we set $\tau = T/20$, where $T = \Theta(K)$ is the targeted number of partitions we would like to generate for each input strings. In other words, we require multi-match instead of single-match when collecting candidate pairs.

Multi-match allows us to filter out those dissimilar pairs more effectively, leading to a significant saving in the verification stage. We try to illustrate the advantage of multi-match for candidate pair filtering via the following example. Consider two strings:

s_1 = ABCDEFGHIJKLMNOP;
s_2 = QRSTUVWXYZABMNOP.

Let us say that we want to find pairs (u, v) such that u can be obtained by performing 3 substitutions on v. In MinJoin, we partition s_1 and s_2 into 4 substrings each. We then use the pigeonhole principle to argue that if s_1 and s_2 are similar, then

[2] If we run the algorithm in [5] in our CPU computational environment, then its embedding step (only) is already 10–100× slower than the entire running time of MinJoin++ on the datasets that we use in this paper.

they share at least 1 substring. For simplicity, let us consider equal-length partitions:

$p(s_1)$ = {ABCD, EFGH, IJKL, MNOP};
$p(s_2)$ = {QRST, UVWX, YZAB, MNOP}.

Thus s_1 and s_2 share a common substring and will be treated as a candidate pair for verification, however, (s_1, s_2) is *not* a valid output pair since their distance is larger than 3. In `MinJoin++`, we try to partition the two strings to more substrings (i.e., we increase the value T). In the above example, if we partition each string to 8 substrings, then any similar pair of strings will share at least 5 substrings. We again consider equal-length partitions:

$p(s_1)$ = {AB, CD, EF, GH, IJ, KL, MN, OP};
$p(s_2)$ = {QR, ST, UV, WX, YZ, AB, MN, OP}.

We observe that s_1 and s_2 only have 2 common substrings; therefore, they will not be considered as a candidate pair, reducing the amount of time needed for verification.

In the edit similarity join problem, we also need to consider insertions and deletions. Moreover, the partitions in `MinJoin`/`MinJoin++` are *not* equal length (recall that the partition is randomized), which makes the analysis more complicated. We managed to show the correctness of multi-match via a more delicate theoretical analysis (Sect. 2.2) and demonstrated its superiority through experiments (Sect. 4).

We have also added a parallel implementation of `MinJoin++` in Sect. 3.2. To this end, we replace the hash tables in the join stage of `MinJoin` with simple arrays for better parallelizing the algorithm.

Our experimental results in Sect. 4 shows that `MinJoin++` outperforms `MinJoin` by $2 \sim 6$ times on large datasets.

1.2 Preliminaries

In Table 1, we list a set of notations which will be frequently used in this paper.

We will make use of the following tools in probability.

Lemma 1 (Chebyshev's inequality) *For any random variable $X \geq 0$ and a constant $c > 0$, we have*

$$\mathbf{Pr}[|X - \mathbf{E}[X]| \geq c] \leq \frac{\mathbf{Var}[X]}{c^2}.$$

Lemma 2 (Chernoff bound) *Let $X_1, X_2, \ldots, X_k$ be independent random variables taking values in $[0, \Delta]$ and $X = \sum_{i \in [k]} X_i$. For any $a > 0$, we have*

$$\mathbf{Pr}[\mathbf{E}[X] - X \geq a] \leq e^{-\frac{a^2}{k \cdot \Delta^2}}.$$

Algorithm 1 Partition-String (s, T, Π)

```
Input: Input string $s$, number of targeted partitions $T$, random hash
    function $\Pi : \Sigma^q \to (0, 1)$
Output: Partitions of $s$: $\mathcal{P} = \{(pos, len)\}$, where $(pos, len)$ refers a
    substring of $s$ starting at the $pos$-th position with length $len$
1: $\mathcal{P} \leftarrow \emptyset$
2: $A = \{a_1, \ldots, a_p\} \leftarrow$ Find-Anchor$(s, T, \Pi)$
3: for each $i \in [1, p-1]$ do
4:     $\mathcal{P} \leftarrow \mathcal{P} \cup (a_i, a_{i+1} - a_i)$
5: end for
```

2 A string partition scheme using local hash minima

In this section, we present the string partition algorithm and analyze its properties.

2.1 The algorithm

We start by giving some high level ideas of our partition scheme. As mentioned, in `MinJoin++` we first partition each string into a set of substrings, and then find pairs of strings that share at least a pre-defined number (denoted by τ) of common partitions as candidates for verification. Consider a pair of strings x and y ($|x| = |y| = N$) with edit distance K. Let $\rho : [N] \to [N] \cup \{\perp\}$ be the *optimal* alignment between x and y, where $\rho(i) = j \in [N]$ means that either $x[i] = y[j]$ or $x[i]$ is substituted by $y[j]$ in the optimal transformation, and $\rho(i) = \perp$ means that $x[i]$ is deleted in the optimal transformation. If we pick any $T = K + \tau$ indices $1 < i_1 < \cdots < i_T = N$ such that $\rho(i_t) \neq \perp$ $(t \in [T-1])$, partition x at indices $i_1, \ldots, i_{T-1}$ to T substrings, and partition y at indices $\rho(i_1), \ldots, \rho(i_{T-1})$ to T substrings, then by the pigeonhole principle x and y must share at least $T - K = \tau$ common partition.

Of course, computing an optimal alignment between x and y before the partition is unrealistic. Our goal is to partition each string independently, while still guarantee that with a good probability, any pair of similar strings will share at least τ common partitions.

We present our partition algorithm in Algorithm 1 and Algorithm 2. Let us briefly describe them in words. Algorithm 1 first calls Algorithm 2 to obtain all *anchors* (to be defined shortly) of the input string s, and then cuts s at each anchor into a set of substrings.

To compute all anchors, Algorithm 2 first hashes all the substrings of s of length q (i.e., $s[1..q], s[2..q+1], \ldots$) into values in $(0, 1)$. Now we have effectively transferred s to an array $h[]$ of size $|s| - q + 1$, with each coordinate taking a value in $(0, 1)$. We call the coordinates within a distance r of $h[i]$ the *r-neighborhood* of i. We call r the *neighborhood radius*, which is determined by the length of the string $|s|$ and the number of targeted partition T. For $i = r + 1, \ldots, (|s| -$

Table 1 Summary of notations

Notation	Definition
$[n]$	$[n] = \{1, 2, \ldots, n\}$
K	Edit distance threshold
$\mathcal{S}$	Set of input strings
s_i	i-th string in $\mathcal{S}$
n	Number of input strings, i.e., $n = \lvert\mathcal{S}\rvert$
$\lvert s\rvert$	Length of string s
$s_{i..j}$	Substring of s starting from the i-th letter to the j-th letter
N	Maximum string length
Σ	Alphabet of strings in $\mathcal{S}$
q	Length of q-gram
Π	Random hash function $\Sigma^q \to (0, 1)$
r	Neighborhood radius for computing local minimum
T	Number of targeted partitions; $T = \Theta(K)$
τ	Similarity threshold, $\tau = T/20$

Algorithm 2 Find-Anchor(s, T, Π)

```
Input: Input string $s$, number of targeted partitions $T$, random hash
    function $\Pi : \Sigma^q \to (0, 1)$
Output: The set of anchors $A$ on $s$
1: $A \leftarrow \{1\}$
2: $r \leftarrow \lfloor \frac{|s|-q+1-T}{2T+2} \rfloor$
3: Initialize an empty array $h$ with $|s| - q + 1$ elements
4: for each $i \in [|s| - q + 1]$ do
5:     $h[i] \leftarrow \Pi(s_{i..i+q-1})$
6: end for
7: for each $i \in [1 + r, |s| - q + 1 - r]$ do
8:     $Label \leftarrow 1$
9:     for each $j \in [i - r, i + r]$ and $j \neq i$ do
10:        if $h[i] \geq h[j]$ then
11:            $Label \leftarrow 0$
12:            Exit the for loop
13:        end if
14:    end for
15:    if $Label = 1$ then
16:        $A \leftarrow A \cup \{i\}$
17:    end if
18: end for
19: $A \leftarrow A \cup \{|s|\}$
20: return $A$
```

$q+1)-r$, we say $h[i]$ a *local minimum* if its value is strictly smaller than all other coordinates in the r-neighborhood of i. Algorithm 2 outputs the corresponding i-th letter in string s as an anchor. For convenience, we also call a local minimum coordinate in $h[]$ an anchor.

Intuitively, anchors are the "cutting points". We use anchors to conduct an *oblivious* partition of the strings (i.e., we perform the partition on each string independently). If we choose anchors to be the letters with the smallest hash rank in a neighborhood of a specific radius, we can ensure that every pair of strings with at least one long common substring will be selected as a candidate. More precisely, we will show that for a pair of strings x, y, if they share a common substring

Table 2 Hash values of 3-grams

3-gram	Value	3-gram	Value	3-gram	Value
CTA	0.01	ACG	0.39	GAA	0.69
GCT	0.05	AAA	0.42	AAT	0.74
TGC	0.12	AAC	0.46	ATC	0.77
TAA	0.21	CCT	0.53	GTC	0.83
ACC	0.25	TCG	0.58	TGG	0.89
CGT	0.31	ATC	0.62	GGA	0.91
GTG	0.33	CGA	0.64	GCG	0.97

σ that is long enough, then there must be at least two letters u, v in σ such that u and v are two adjacent anchors in both x and y, which means that if we use anchors to partition x and y, then they must share at least one common partition. On the other hand, we know that for two strings of length N and edit distance $K \ll N$, they must share many long substrings. Thus by properly choosing the neighborhood radius r, we can guarantee that two similar strings will share at least $\tau = T/20$ common partitions.

A running example Before analyzing Algorithm 1, we first give a running example. Table 2 presents the hash values of all 3-grams in $\mathcal{S}$ under the hash function Π. Table 3 presents a collection of input strings $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5\}$ and their lengths. We want to find all pairs of strings with edit distance less than or equal to $K = 4$. We set neighborhood size $r = 2$, and try to find all pairs of strings that share at least $\tau = 1$ partition.

Table 4 presents the partitions of strings obtained by Algorithm 2. Considering string s_1 as an example, its 6-th 3-gram "CTA" has a smaller hash value than all its neighbors within distance $r = 2$ (i.e., "TGC", "GCT", "TAA", "AAC"). Thus "CTA" is selected as an anchor of s_1. Same to the 14-th 3-

Table 3 Input strings

ID	String	Length
s_1	ACGTGCTAACGTGCTAACGTG	21
s_2	AAACGTGCTAACGTGCTAACCT	22
s_3	TCGAATCGTCGAATCGTCGAA	21
s_4	TCGAATCGTCGAATCGTGGAA	21
s_5	GTGCGAATCGTCGAATCGTCG	21

Table 4 Partitions of strings by Algorithm 1 ($T = 3$)

ID	Partitions of string	r
s_1	ACGTG, CTAACGTG, CTAACGTA	2
s_2	AAACGTG, CTAACGTG, CTAACCT	2
s_3	TCGAAT, CGTCGAAT, CGTCGAA	2
s_4	TCGAAT, CGTCGAAT, CGTGGAA	2
s_5	GTGCGAAT, CGTCGAAT, CGTCG	2

gram "CTA". We then partition s_1 to {ACGTG, CTAACGTG, CTAACGTA}.

We find that strings s_1, s_2 share a common partition "CTAACGTG", s_3, s_4 share a common partition "TCGAAT", and s_3, s_4, s_5 share a common partition "CGTCGAAT", which give the following candidate pairs: (s_1, s_2), (s_3, s_4), (s_3, s_5), (s_4, s_5). After computing the exact edit distance of each pair, we output (s_1, s_2), (s_3, s_4), (s_3, s_5) as the final answer (i.e., those whose edit distances are no more than $K = 4$).

Discussion We would like to discuss a few things regarding our local hash minima-based partition scheme. First, we require the value of an anchor in the hash array $h[]$ to be *strictly* smaller than its $2r$ neighbors. The purpose of this is to reduce the number of false positives generated by periodic substrings with short periods; false positives will increase the running time of the verification step of the `MinJoin++` algorithm. In real-world datasets, periodic substrings are often caused by systematic errors, and may be shared among different strings. For example, consider the following periodic substring on genome data "…AAAAAAAA …" produced by sequencing errors, if we allow the value of an anchor to be equal to its neighbors, then we may have many anchors 'A' in this substring if the hash rank of 'A' is the smallest among that of all letters in a specific neighborhood radius. Such singleton partitions will easily cause hash collisions and form candidate pairs, even for pairs of strings with large edit distances.

Second, we use different neighborhood radius r for strings of different lengths. More precisely, we set $r = \lfloor \frac{|s|-q+1-T}{2T+2} \rfloor$ where $|s|$ is the length of the string. The purpose of doing this, instead of choosing a fixed r for all strings, is again to reduce false positives. Indeed, if we choose the same r for all strings, then long strings will generate many partitions, since in order to achieve perfect accuracy we cannot set r to be too large at the presence of short strings. Consequently, the large number of partitions generated by long strings will contribute to many false positives. This is in contrast to `VChunk`, who cuts the string whenever it finds a word in CBD appearing on the string. Consequently, two strings of very different length but sharing a relatively long substring are likely to be considered as a candidate pair, producing a false positive for the verification.

Third, in the discussion above, we treat the number of substrings after partitioning each input string as a fixed value T. However, since our partition scheme is a randomized algorithm, the number of partitions it generates for each input string is a random variable T'. Fortunately, we can show that T' is tightly concentrated around T. This is also why we call the parameter T the *targeted* number of partitions. We will give the concentration result in Lemma 3 in Sect. 2.2.

2.2 The analysis

In this section, we examine the properties of Algorithm 1. We will analyze (1) how many partitions Algorithm 1 will generate? And (2) what is the probability for two similar strings to share at least τ common partitions?

To keep the analysis clean, we assume that in any r-neighborhood of the array $h[]$, all the coordinates are distinct, which is true if (1) we assume that all corresponding q-grams are different, and (2) the hash function $\Pi : \Sigma^q \rightarrow (0, 1)$ does not produce a collision when applying to q-grams. The later is easily satisfied if we maintain an $O(\log N)$-bit precision (N is the maximum input string length) in the range of Π, in which case there will not be a hash collision with probability $1 - 1/N^{\Omega(1)}$. For the former, we set $q = 3\log_{|\Sigma|}(N/T)$. Note that by our choice of r we have $r \approx N/(2T)$. If all letters in a substring of size r are random, then the probability that two q-grams in this substring are the same is $1/|\Sigma|^q = \left(\frac{T}{N}\right)^3$. By a union bound, with probability $1 - o(1)$, all q-grams in a substring of size $2r$ are different. We emphasize that this assumption is only used for the convenience of the concentration analysis; Algorithm 1 works regardless of whether this assumption is true or not.

For simplicity, in the rest of the paper, we often ignore the floor/ceiling operations whose effect is negligible to the analysis.

The following lemma states that the number of anchors produced by Algorithm 2 is tightly concentrated around the number of targeted partitions T.

Lemma 3 *Given an input string and a parameter T, for any $c > 0$, the number of anchors generated by Algorithm 2, denoted by X, satisfies* $\mathbf{Pr}[|X - T| \geq \sqrt{cT}] < 1/c$.

Proof Consider the array $h[1..|s|-q+1]$ constructed in Algorithm 2; $h[i]$ is the hash value of the i-th q-gram of s. Let $w = |s| - q + 1 - 2r$. For $i = 1, \ldots, w$, define a random variable X_i whose value is 1 if $h[i+r]$ is the smallest coordinate in the *window* $h[i..i+2r]$, and 0 otherwise. Let $X = \sum_{i\in[w]} X_i$, which is the total number of anchors generated by Algorithm 2. We next analyze the mean and variance of the random variable X.

We start by computing its expectation. Recall that we have set r to be $\frac{|s|-q+1-T}{2T+2}$ at Line 2 of Algorithm 2.

$$\begin{aligned}\mathbf{E}[X] &= \sum_{i\in[w]} \mathbf{E}[X_i] = \sum_{i\in[w]} \mathbf{Pr}[X_i = 1] \\ &= \frac{w}{2r+1} = T. \end{aligned} \tag{1}$$

We next compute the variance.

$$\begin{aligned}\mathbf{Var}[X] &= \sum_{i\in[w]} \mathbf{Var}[X_i] + \sum_{i\neq j} \mathbf{Cov}[X_i, X_j] \\ &= \sum_{i\in[w]} \mathbf{Var}[X_i] + \frac{1}{2}\sum_{i}\sum_{j\neq i} \mathbf{Cov}[X_i, X_j]. \end{aligned} \tag{2}$$

We compute the two terms of (2) separately. For the first term,

$$\begin{aligned}\sum_{i\in[w]} \mathbf{Var}[X_i] &= \sum_{i\in[w]} \left(\mathbf{E}[X_i^2] - (\mathbf{E}[X_i])^2\right) \\ &= w \times \left(\frac{1}{2r+1} - \frac{1}{(2r+1)^2}\right) \\ &\le \frac{w}{2r+1}. \end{aligned} \tag{3}$$

For the second term of (2), by the definition of the covariance,

$$\begin{aligned}\mathbf{Cov}[X_i, X_j] &= \mathbf{E}[X_i X_j] - \mathbf{E}[X_i]\mathbf{E}[X_j] \\ &= \mathbf{E}[X_i X_j] - \frac{1}{(2r+1)^2}.\end{aligned}$$

We analyze $\mathbf{E}[X_i X_j]$ in three cases.
Case I $|i-j| \ge 2r+1$. It is easy to see that in this case X_i and X_j are independent, since their corresponding windows $h[i..i+2r]$ and $h[j..j+2r]$ are disjoint. We thus have $\mathbf{E}[X_i X_j] = \mathbf{E}[X_i]\mathbf{E}[X_j]$, and consequently $\mathbf{Cov}[X_i, X_j] = 0$.
Case II $|i-j| \le r$. In this case, $h[i+r]$ is inside the window $h[j..j+2r]$, and symmetrically $h[j+r]$ is inside the window $h[i..i+2r]$. Thus if $X_i = 1$, then we must have $X_j = 0$, and if $X_j = 1$, then we must have $X_i = 0$. Therefore $\mathbf{E}[X_i X_j] = 0$, and consequently $\mathbf{Cov}[X_i, X_j] = -\frac{1}{(2r+1)^2}$.
Case III $r < |i-j| < 2r+1$. The analysis for this case is a bit more complicated. Consider two windows $W_i = h[i..i+2r]$ and $W_j = h[j..j+2r]$ which overlap. We divide their union into three areas; see Fig. 1 for an illustration. Area 2 denotes the intersection of the two windows, and Area 1 and Area 3 denote the coordinates that are only in W_i and W_j, respectively. It is easy to see that the number of coordinates in Area 1 and Area 3 are equal; let α ($r < \alpha < 2r+1$) denote this number.

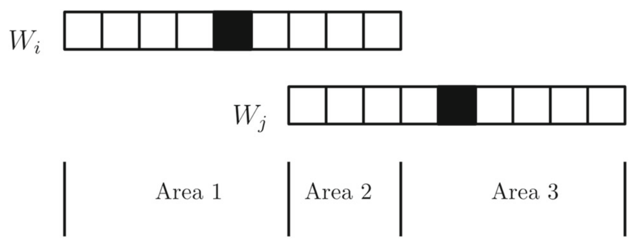

Fig. 1 Illustration of windows W_i, W_j when $r < |i-j| < 2r+1$. Black square represents the central coordinate of the window. The squares in same column correspond to same coordinate in the array $h[]$; we duplicate them for the illustration purpose

We write

$$\begin{aligned}\mathbf{E}[X_i X_j] &= \mathbf{Pr}[X_i = 1, X_j = 1] \\ &= \mathbf{Pr}[X_j = 1 \mid X_i = 1] \cdot \mathbf{Pr}[X_i = 1] \\ &= \mathbf{Pr}[X_j = 1 \mid X_i = 1] \cdot \frac{1}{2r+1}.\end{aligned}$$

We thus only need to analyze $\mathbf{Pr}[X_j = 1 \mid X_i = 1]$. Define a random variable Y such that $Y = 1$ if the central coordinate of W_i (i.e., $h[i+r]$) is smaller than all coordinates in Area 3. We have

$$\begin{aligned}&\mathbf{Pr}[X_j = 1 \mid X_i = 1] \\ &\quad = \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 1] \\ &\qquad \times \mathbf{Pr}[Y = 1 \mid X_i = 1] \\ &\qquad + \mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 0] \\ &\qquad \times \mathbf{Pr}[Y = 0 \mid X_i = 1]. \end{aligned} \tag{4}$$

Note that $(X_i = 1) \wedge (Y = 1)$ implies that the central coordinate of W_i is smaller than all coordinates in W_j, which, however, does not give any information about the relationship between all coordinates in W_j. We thus have

$$\begin{aligned}\mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 1] &= \mathbf{Pr}[X_j = 1] \\ &= \frac{1}{2r+1}. \end{aligned} \tag{5}$$

On the other hand, $(X_i = 1) \wedge (Y = 0)$ implies that the central coordinate of W_i is smaller than all coordinates in Area 2, and is larger than some coordinate in Area 3. We thus know that the minimum coordinate of W_j must lie in Area 3. Therefore $X_j = 1$ if and only if the central coordinate of W_j is larger than all other coordinates in Area 3. We get

$$\mathbf{Pr}[X_j = 1 \mid X_i = 1, Y = 0] = 1/\alpha. \tag{6}$$

Plugging in (5) and (6) to (4), we have

$$\begin{aligned}\mathbf{Pr}[X_j = 1 \mid X_i = 1] &= \frac{1}{2r+1} \cdot \mathbf{Pr}[Y = 1 \mid X_i = 1] \\ &\quad + \frac{1}{\alpha} \cdot \mathbf{Pr}[Y = 0 \mid X_i = 1] \\ &\le \frac{1}{\alpha} \le \frac{1}{r+1}.\end{aligned}$$

Consequently we have

$$\begin{aligned}\mathbf{Cov}[X_i, X_j] &\le \frac{1}{2r+1} \cdot \frac{1}{r+1} - \frac{1}{(2r+1)^2} \\ &< \frac{1}{(2r+1)^2}.\end{aligned}$$

Summing up, we have

$$\mathbf{Cov}[X_i, X_j] \begin{cases} = -\frac{1}{(2r+1)^2}, & \text{if } |i-j| \le r \\ < \frac{1}{(2r+1)^2}, & \text{if } r < |i-j| < 2r+1 \\ = 0, & \text{if } |i-j| \ge 2r+1 \end{cases}$$

which, together with (3) and (2), give

$$\begin{aligned}\mathbf{Var}[X] &< \frac{w}{2r+1} + \frac{1}{2} \cdot w \cdot 2r \cdot \frac{1}{(2r+1)^2} \\ &\quad - \frac{1}{2} \cdot w \cdot 2r \cdot \frac{1}{(2r+1)^2} \\ &= \frac{w}{2r+1} = T.\end{aligned} \tag{7}$$

By (1), (7), and the Chebyshev's inequality, we have that for any constant $c > 0$,

$$\mathbf{Pr}[|X - T| \ge \sqrt{cT}] < 1/c.$$

□

We have empirically verified the concentration result in Lemma 3 on two real-world datasets (to be introduced in Sect. 4); see Fig. 2. It is clear that the number of partitions Algorithm 1 generates is tightly concentrated around the number of target partitions T.

The following lemma gives another key property of our local hash minima-based partition. For the convenience of the analysis, we will assume $|s| = \omega(Kq)$ and $K = \omega(\log n)$. That is, the string length is sufficiently long and the distance threshold is sufficiently large.

Lemma 4 *For two strings s, t with $ED(s, t) \le K$, let $\mathcal{P}_s$ and $\mathcal{P}_t$ be the partitions outputted by Algorithm 1 (setting $T = 200K$) on s and t respectively. The probability that $\mathcal{P}_s$ and $\mathcal{P}_t$ share at least $T/20$ common partitions is at least $1 - 1/n^4$.*

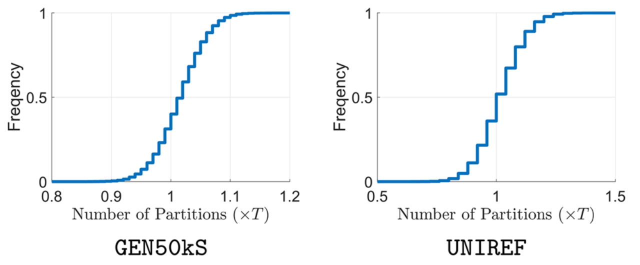

Fig. 2 The CDFs of numbers of partitions on each string returned by Algorithm 1 on GEN50kS and UNIREF datasets, with parameters $T = 100$ and $T = 25$ respectively

Proof Since $\mathrm{ED}(s, t) \le K$, we have $|t| \in [|s| - K, |s| + K]$. Moreover, s and t must share at least $T/12$ common disjoint substrings, each of which has length $L = \frac{|s|}{T/12+K}$. To see this, suppose string s has been partitioned into substrings of equal length L. The K edits can only "destroy" at most K of these L substrings. It follows that there are at least $|s|/L - K = T/12$ common disjoint substrings of length L in s and t.

We next show that for any common substring of s and t of length L, it must produce a common partition after running Algorithm 2.

Let γ be any common substring of length L. Let $r_s = \frac{|s|-q+1-T}{2T+2}$, and let $\eta = \frac{L-q+1-2r_s}{2r_s+1}$. When running Algorithm 2 on s, by an almost identical argument as that for the proof of Lemma 3 and setting $c = 4$, we have that the number of anchors X on γ satisfies

$$\mathbf{Pr}[|X - \eta| \ge \sqrt{3\eta}] < 1/3. \tag{8}$$

For $T = 200K$ and $|s| = \omega(Kq)$, we have

$$\begin{aligned}\eta &= \frac{L - q + 1 - 2r_s}{2r_s + 1} \\ &\ge \left(\frac{|s|}{T/12 + K} - q + 1 - 2r_s\right) \cdot \frac{T+1}{|s| - q + 2} \\ &\ge \left(\frac{|s|}{200K/12 + K} - q + 1 - 2r_s\right) \cdot \frac{200K+1}{|s| - q + 2} \\ &\ge 10.\end{aligned} \tag{9}$$

Plugging (9) to (8), we have with probability at least $(1 - 1/3) = 2/3$ that

$$X \ge \eta - \sqrt{3\eta} > 4, \tag{10}$$

which means that with probability 2/3 there are at least four anchors on γ.

Let a_1, a_2, a_3, a_4 be four anchors on γ when processing s using Algorithm 2. Let $r_t = \frac{|t|-q+1-T}{2T+2}$. Since $\text{ED}(s,t) \le K$ and $T = 200K$, it holds that $|r_t - r_s| \le 1$. In the case that $r_t = r_s = r$, a_2 and a_3 must also be anchors when processing t using Algorithm 2, since an anchor is fully determined by a neighborhood of radius r.

For the case when $|r_t - r_s| = 1$, w.l.o.g., assume that $r_s = r$ and $r_t = r+1$. Now the probability that a_2 is still an anchor when processing t, given the fact that a_2 is an anchor when processing s, is at least $1 - 1/(r+1)$. Same argument holds for a_3. Thus with probability $2/3 - 2/(r+1) \ge 19/30$ (note that $r = r_s = \frac{|s|-q+1-T}{2T+2} = \omega(1)$ given $|s| = \omega(qK)$ and $T = 200K$), a_2 and a_3 are also anchors when processing t.

Finally, observe that once s and t share two adjacent anchors a_2 and a_3, they must share at least one common partition.

Up to this point, we have shown that for each of the at least $\kappa = T/12$ common disjoint substrings, with probability at least $19/30$, it produces a common partition. Define $X_i \in \{0,1\}$ to be the indicator random variable of the event that the i-th common substring produces a common partition. We have that $\mathbf{E}[X_i] \ge 19\kappa/30$. Let $X = \sum_{i=1}^{\kappa} X_i$. The number of common partitions between the two input strings is at least X.

By Chernoff bound (Lemma 2), setting $a = \sqrt{4\kappa \log n}$ and noting that $\kappa = T/12 = \omega(\log n)$, with probability at least $1 - e^{-\frac{4\kappa \log n}{\kappa}} \ge 1 - \frac{1}{n^4}$, we have

$$X \ge \mathbf{E}[X] - \sqrt{4\kappa \log n} \ge \frac{19\kappa}{30} - \sqrt{4\kappa \log n} \ge \frac{T}{20}. \qquad \square$$

The following theorem summarizes the property of Algorithm 1.

Theorem 1 *If we apply Algorithm 1 on all the n input strings with $T = 200K$, then with probability at last $1 - 1/n^2$, all pair of strings with edit distance at most K will share at least $T/20$ common partitions. The expected running time and the space of the algorithm are both linear in terms of the input size.*

Proof Since there are at most n^2 pairs of input strings, and by Lemma 4 we have for each pair of strings, they share at least $T/20$ common partitions with probability at least $1 - 1/n^4$. The correctness follows from a union bound.

We next analyze the time and space. We can just show that the time and space for partitioning one string s (by Algorithm 1) is linear in terms of the string length $|s|$.

The running time of Algorithm 1 is dominated by that of its subroutine Algorithm 2. The hash values of all q-grams of s can be computed by, for example, the rolling hash[3] in $O(|s|)$ time. For Line 7–18 of Algorithm 2, since each number in $h[]$ is a random hash value, the inner for-loop (Line 9–14) runs in $O(1)$ time in expectation. Therefore, the total running time of Algorithm 1 is $O(|s|)$ in expectation.

Clearly, the space usage of Algorithm 1 is also $O(|s|)$. $\square$

[3] https://en.wikipedia.org/wiki/Rolling_hash.

Algorithm 3 MinJoin++($\mathcal{S}, K, T$)

```
Input: Set of input strings $\mathcal{S} = \{s_1, \dots, s_n\}$, distance threshold $K$, number of targeted partitions $T$
Output: $\mathcal{O} \leftarrow \{(s_i, s_j) \mid s_i, s_j \in \mathcal{S}; i \ne j; \text{ED}(s_i, s_j) \le K\}$
1:  $\mathcal{O} \leftarrow \emptyset, \mathcal{C} \leftarrow \emptyset$        ▷ $\mathcal{C}$ : collection of candidate pairs
2:  Pick a hash function $f : \Sigma^* \to \mathbb{N}$
3:  Generate a random hash function $\Pi : \Sigma^q \to (0,1)$
4:  Initialize an empty array $A$
5:  for each $s_i \in \mathcal{S}$ do
6:      $\mathcal{P} \leftarrow$ PartitionString($s_i, T, \Pi$)
7:      for each $(pos, len) \in \mathcal{P}$ do
8:          Add $(f((s_i)_{pos..pos+len-1}), pos, len, i)$ to $A$
9:      end for
10: end for
11: Sort elements in $A$ by the first component (i.e., the hash value $f(\cdot)$), and partition elements in $A$ into groups $A_1, A_2, \dots, A_\zeta$ according to $f(\cdot)$
12: for each $G \in \{A_1, A_2, \dots, A_\zeta\}$ do
13:     for each pair of elements $(h, pos_i, len_i, i), (h, pos_j, len_j, j)$ in $G$ do
14:         if $i \ne j$ and $||s_i| - |s_j|| \le K$ then
15:             if $|pos_i - pos_j| + |(|s_i| - pos_i) - (|s_j| - pos_j)| \le K$ then
16:                 $\mathcal{C} \leftarrow \mathcal{C} \cup (i, j)$
17:             end if
18:         end if
19:     end for
20: end for
21: Remove all pairs in $\mathcal{C}$ that appear less than $T/20$ times
22: for each $(i, j) \in \mathcal{C}$ do
23:     if $\text{ED}(s_i, s_j) \le K$ then
24:         $\mathcal{O} \leftarrow \mathcal{O} \cup (i, j)$
25:     end if
26: end for
```

3 The MinJoin++ algorithm

We now present our main algorithm MinJoin++, described in Algorithm 3. We briefly explain it in words below.

The MinJoin++ algorithm has four stages: initialization (Line 1–4), partition (Line 5–10), join (Line 11–21) and verification (Line 22–26). In the first stage, we initialize an empty set $\mathcal{C}$ for candidate pairs and an empty array A. We also generate a hash function Π for random ranking in the string partition procedure, and another hash function f for generating the keys of elements in A. In our experiments, we used MurmurHash3 to implement Π and SeaHash to implement f.[4]

[4] See https://en.wikipedia.org/wiki/MurmurHash for MurmurHash3, and https://docs.rs/seahash/latest/seahash/ for SeaHash.

In the second stage, we compute the partitions for each input string using Algorithm 1.

In the third stage, we first partition the elements in A into ζ groups using the first components of elements (i.e., ζ is the number of distinct values of $f(\cdot)$ in elements of A). Next, for each pair (h, pos_i, len_i, i) and (h, pos_j, len_j, j) in each group, we perform two quick tests. The first test (Line 14) says that if the lengths of s_i and s_j differ by more than K, then it is impossible to have $\mathrm{ED}(s_i, s_j) \leq K$. The second test (Line 15) concerns the following scenario: if s_i and s_j match at indices pos_i and pos_j, which divide s_i and s_j into substrings $\nu_1 = (s_i)_{1..pos_i - 1}$, $\nu_2 = (s_i)_{pos_i..|s_i|}$ and $\mu_1 = (s_j)_{1..pos_j - 1}$, $\mu_2 = (s_j)_{pos_j..|s_j|}$ respectively, and if pos_i and pos_j are indeed matched in the optimal alignment, then we must have

$$\mathrm{ED}(\nu_1, \mu_1) + \mathrm{ED}(\nu_2, \mu_2) \leq K,$$

in which case we have

$$\left|(|s_i| - pos) - (\left|s_j\right| - pos_j)\right| + \left|pos - pos_j\right| \leq K.$$

We add all pairs that pass both tests into the candidate set $\mathcal{C}$. Finally, we identify all pairs in $\mathcal{C}$ that appear at least $T/20$ times in the same groups as candidate pairs for the verification step. It should be noted that matches that fail either of the two tests (Line 14 and Line 15) are not counted towards the $T/20$ threshold.

In the fourth stage, we verify whether each pair of strings in $\mathcal{C}$ indeed have edit distance at most K, using the standard dynamic programming algorithm by Ukkonen [15]. Due to this verification step, our algorithm will never output any false positive. Therefore, by Theorem 1, MinJoin++ correctly output all similar pairs of strings with probability $1 - 1/n^2$.

Time and space analysis Let N be the maximum string length in the set of input strings $\mathcal{S}$, and $n = |\mathcal{S}|$. By Theorem 1, the running time of the partition is bounded by $O(nN)$.

The total number of pairs fed into the two tests (Line 14 and 15) inherently depends on the dataset. Suppose that the partitions of all strings are evenly distributed into the ζ groups $A_1, \ldots A_\zeta$, we can upper bound the expected number of pairs for the tests by $O\left(\frac{nT}{\zeta}\right)^2 = O\left(\frac{nK}{\zeta}\right)^2$.

The verification step can be done in $O(|\mathcal{C}|\, NK)$ where $\mathcal{C}$ is the set of the candidate pairs.

The space usage is clearly bounded by $O(nN + |\mathcal{C}|)$, that is, the size of the input plus the number of candidate pairs.

Theorem 2 *Setting $T = 200K$, the* MinJoin++ *algorithm has the following theoretical properties.*

- *It outputs all pairs of similar strings with probability* $1 - 1/n^2$.
- *Assuming that the partitions of all strings are evenly distributed into the groups $A_1, \ldots, A_\zeta$, the expected running time of* MinJoin++ *is at most*

$$O\left(nN + \left(\frac{nK}{\zeta}\right)^2 + |\mathcal{C}|\, NK\right),$$

where $\mathcal{C}$ is the set of the candidate pairs produced before the verification step.
- *The space usage of* MinJoin++ *is* $O(nN + |\mathcal{C}|)$.

3.1 Remarks on the choice of *T*

As mentioned, MinJoin++ is a randomized algorithm. Theorem 2 states that MinJoin++ will output the exact answer with a very high probability (that is, $1 - 1/n^2$). To achieve this accuracy, we have set $T = 200K$, but the value of T can be much lower in practice for three reasons.

First, setting $T = 200K$ in the theoretical analysis is largely for the simplicity of the calculation. A more careful analysis can bring T to $c_T K$ for a much smaller constant c_T.

Second, on real data, the edits often occur in clusters. If we partition a string into T substrings, then we would expect that the (at most) K edits may only occur in K/c_k partitions for a constant $c_k > 1$. Consequently, similar pairs of strings are likely to share more common partitions.

Third, in practice, we might not need to aim for an error probability as small as $1/n^2$; we can use a smaller T at the cost of a weaker error probability (or, a very small number of false negatives).

The above discussion also holds for the MinJoin algorithm in the preliminary version of this paper [22].

In our experiments, we have set $T = 20 + K/15$ for MinJoin, and $T = 20 + K/8$ for MinJoin++. The leading constant 20 is used to make sure that the concentration inequalities (Chebyshev's inequality and Chernoff bound) give good success probabilities even when K is very small. The coefficient of K for MinJoin is smaller than that for MinJoin++; this is mainly because MinJoin only requires a single matching partition between similar pairs while MinJoin++ requires multiple matching partitions. For MinJoin++, the constant 8 in $K/8$ already gives very good accuracy for datasets tested in this paper. And we believe that 5–8 are *all* the constants that need to be tested for other datasets

3.2 Parallel implementation

MinJoin++ enjoys a straightforward parallel implementation, and is thus fully scalable in the multi-thread computation environment.

First, it is clear that the partition stage (Line 5–10) can be fully parallelized, since the partition on each string is independent on other strings. We split the set of input strings evenly to the M machines/threads. Each thread runs Partition-String on a subset of input strings.

In the join stage (Line 12–20), we merge the M local arrays into the global array A using merge-sort across multiple threads, getting the groups $A_1, \ldots, A_\zeta$. We then split groups evenly into the M threads for the tests, and then merge the local candidate sets into the global candidate set $\mathcal{C}$.

The verification stage can also be fully parallelized by splitting $\mathcal{C}$ evenly between the threads.

Table 5 Statistics of tested datasets (from [21])

Datasets	n	Avg Len	Min Len	Max Len	$\|\Sigma\|$
UNIREF	400,000	445	200	35,213	25
TREC	233,435	1217	80	3947	37
GEN50kS	50,000	5000	4829	5152	4
GEN20kS	20,000	5000	4829	5109	4
GEN20kM	20,000	10,000	9843	10,154	4
GEN20kL	20,000	20,000	19,821	20,109	4
GEN80kS	80,000	5000	4814	5109	4
GEN320kS	320,000	5000	4811	5154	4

4 Experiments

In this section, we present our experimental studies. We start by describing the datasets and algorithms used in our experiments. We then give a study of several aspects of MinJoin++. Next, we compare MinJoin++ with the state-of-the-art algorithms for edit similarity joins. After that, we test the parallel implementation of MinJoin++. At the end, we compare MinJoin++ with the folklore MinHash-based algorithm.

4.1 The setup

We implemented our algorithms in C++ and performed experiments on a Dell PowerEdge T630 server with 2 Intel Xeon E5-2667 v4 3.2GHz CPU with 8 cores each, and 256GB memory.

Datasets We use the datasets in [21] which are publicly available.[5] Table 5 describes the statistics of tested datasets.

UNIREF: A dataset consists of UniRef90 protein sequence data obtained from UniProt Project.[6] The sequences whose lengths are smaller than 200 are removed, and the first 400,000 protein sequences are extracted.

TREC: A dataset consists of titles and abstracts from 270 medical journals. The title, author, and abstract fields are extracted and concatenated. Punctuation marks are converted into white space and all letters are in uppercase.

GEN-X-Y's: Datasets contain 50 human genomes obtained from the Personal Genomes Project,[7] where X denotes the number of strings (range from 20k to 320k), and Y denotes the string length (S $\approx$ 5k, M $\approx$ 10k, L $\approx$ 20k). Each string is a substring randomly sampled from the Chromosome 20 of human genome.

Algorithms We compare MinJoin++ with the state-of-the-art algorithms for string similarity joins under edit distance, including PassJoin[9], QChunk[11], VChunk [19], and EmbedJoin [21]. We also compare MinJoin++ with the original MinJoin algorithm in the preliminary version of this paper [22]. Except in Sect. 4.5, MinJoin++ always uses a single thread of execution.

All codes are downloaded from the corresponding project websites.

We choose the parameter T for MinJoin++ and MinJoin following the discussion in Sect. 3.1. We always choose the *best* parameters of other tested algorithms. QChunk has two parameters: q (the size of q-gram) and indexing method. We found that the *indexchunk* always performs better than *indexgram* on all datasets, and we always choose the best q for each experiment. VChunk has a parameter *scale* to tune. PassJoin has no parameter. EmbedJoin has three parameters m, r, z. We choose the parameters based on the recommendation of [21]: We select the best combinations of parameters to achieve at least 95% accuracy on UNIREF and TREC datasets, and at least 99% accuracy on GEN50kS dataset; and we select $r = z = 7, m = 15 - \lfloor \log_2 x \rfloor$ on the rest of datasets, where $x\%$ is the edit threshold.

Measurements We use three metrics to measure the performance of tested algorithms: time, space, and accuracy.

We note that except MinJoin++, MinJoin and EmbedJoin, which are randomized and may have false negatives, all other tested algorithms are deterministic and output the exact number of similar pairs. For parameters on which there is at least one deterministic algorithm that can finish within 24 h, we use this output size as the ground truth. For parameters on which there is *no* deterministic algorithm that can finish within 24 h, we use the output of the maximum size among MinJoin++, MinJoin and EmbedJoin as the "ground truth". The accuracy of an algorithm is defined to be the ratio between its output size and the ground truth. We will mark the accuracy on points where it is not 100%.

Each result for the randomized algorithms is an median of 11 independent runs with different randomness.

[5] See the documentation from the project website of [21]: https://github.com/kedayuge/Embedjoin.

[6] http://www.uniprot.org/.

[7] https://www.personalgenomes.org/us.

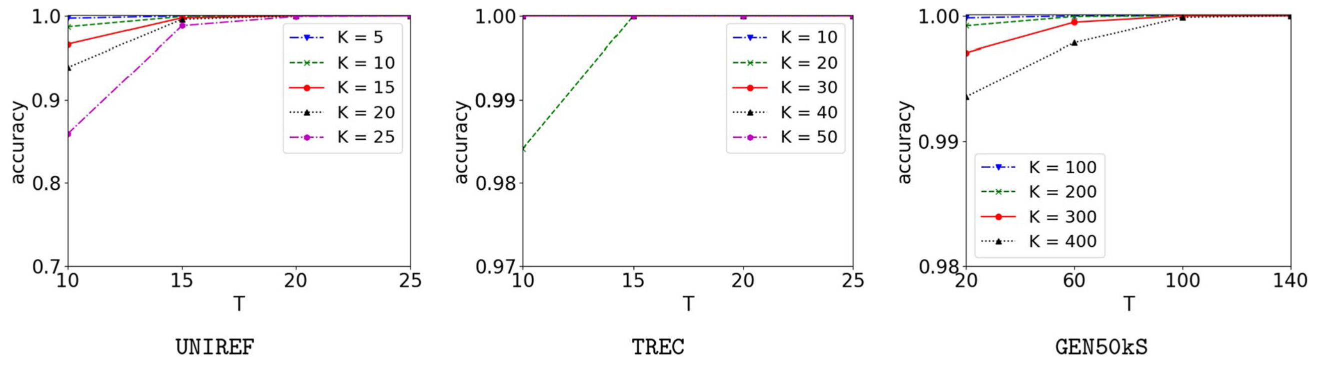

Fig. 3 Influence of T on accuracy

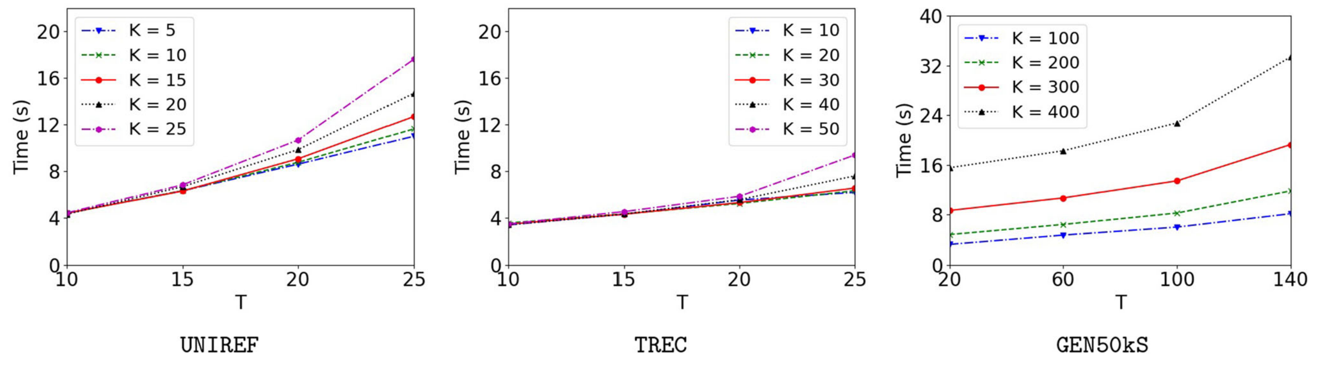

Fig. 4 Influence of T on running time

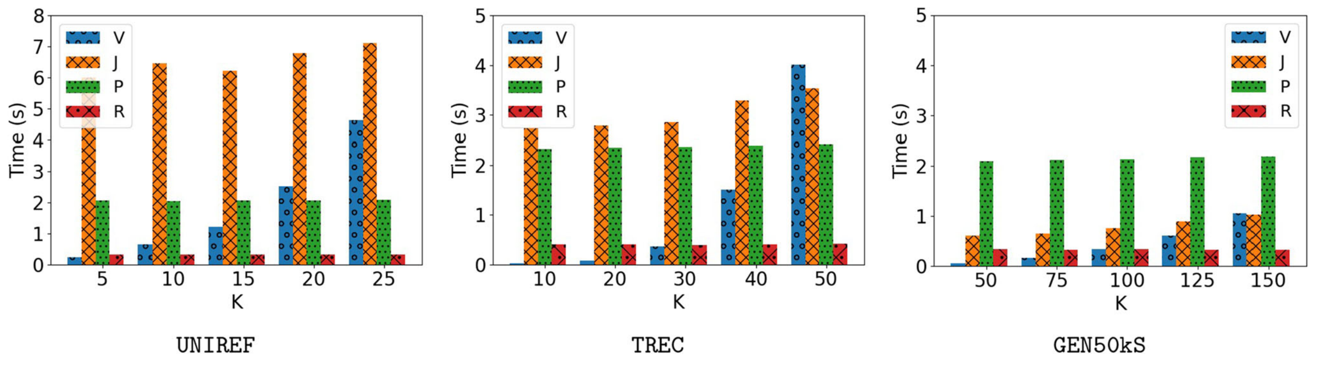

Fig. 5 Running time distribution of input reading (R), string partition (P), join (J), and verification (V) of `MinJoin++`, varying K

4.2 Experiments for `MinJoin++`

We first investigate the influence of parameter T on the running time and accuracy, and then present the running time distribution of the four stages of `MinJoin++`.

Influence of parameter T Figures 3 and 4 show how parameter T influences the accuracy and the running time of `MinJoin++`. As predicted by theory, both time and accuracy increase when T increase. We also tested different distance thresholds K. We observe that when K becomes larger, we need a larger T to achieve the perfect accuracy, which is also consistent with the theory where we need to pick $T = \Theta(K)$.

Running time distribution We present in Fig. 5 the running time of `MinJoin++` on (1) input reading, (2) string partition, (3) join, and (4) verification when varying the distance threshold K. It is clear that the time on input reading will not change when varying K. We observe that the time on join increases slightly when K increases, that on partition is stable, and that on verification increases considerably when K increases.

4.3 A comparison with the state-of-the-art

We now compare `MinJoin++` with the state-of-the-art algorithms for edit similarity joins, including `QChunk`, PassJoin, VChunk, EmbedJoin, and the preliminary version MinJoin. We will first make use of datasets UNIREF, TREC and GEN50kS for a basic comparison. These datasets are of modest size so that all algorithms can finish within 24 h. We then use larger genomics datasets to test the scalability of all algorithms.

Influence of the distance threshold K Fig. 6 presents the running time of different algorithms on UNIREF, TREC and GEN50kS when varying K. Compared with EmbedJoin, MinJoin and MinJoin++ have clear advantage on the accuracy. The running time of MinJoin++ is slightly better than MinJoin on most points. MinJoin++ and MinJoin performs slightly worse than EmbedJoin on UNIREF, similar on TREC, and slightly better on GEN50kS.

All three randomized algorithms (MinJoin++, MinJoin, EmbedJoin) have significant advantage over the deterministic algorithms. MinJoin++ outperforms the best deterministic algorithm by a factor of 2 in UNIREF ($K = 25$), 8 on TREC ($K = 50$), and 52 on GEN50kS ($K = 150$). The running time of PassJoin increases quickly when K becomes large; this is consistent with the theory, which predicts that the query time in PassJoin for each string is proportional to K^3. VChunk performs relatively well on UNIREF, but much worse on TREC and GEN50kS. This may be because the preprocessing time of VChunk has a quadratic dependence on string length N, which is larger in TREC and GEN50kS than UNIREF.

Influence of the input size n Figure 7 presents the running time of different algorithms on UNIREF, TREC and GEN50kS when varying the number of input strings n. The relative accuracy and running time between EmbedJoin, MinJoin, and MinJoin++ are similar as that in Fig. 6 (varying K).

The running time of MinJoin++ outperforms the best deterministic algorithm by a factor of 2 on UNIREF ($n = 400{,}000$), 10 on TREC ($n = 200{,}000$), and 25 on GEN50kS ($n = 50{,}000$). The trends of running time of all algorithms increase near linearly in terms of n, except VChunk whose performance deteriorates significantly when n increases on TREC and GEN50kS, which may again due to the expensive preprocessing step.

Scalability of the algorithms We next test all algorithms on larger datasets. Figure 8 presents the results of the running time when we scale string length up to 20,000 and the distance threshold K up to 16% of the string length. Figure 9 presents the results when we scale the number of strings up to 320,000, and K up to 16% of the string length. The first plot of Fig. 9 is just a repeat of that of Fig. 8. Note that the running time is in logarithmic scale.

We note that some algorithms cannot produce some of the points, which may be because they cannot finish within 24 h, or there are some implementation issues (e.g., memory overflow). As mentioned earlier, in cases when there is no deterministic algorithm that can finish in time, the accuracy of algorithms is computed using the best result returned by EmbedJoin, MinJoin++ and MinJoin as the “ground truth”.

We observe that MinJoin++ generally outperforms EmbedJoin by $6 \sim 12$ times, and outperforms MinJoin by $2 \sim 6$ times. The advantage of MinJoin++ over MinJoin increases when the number of strings n or the string length N becomes larger. This is because when n or N increases, the filtering procedure of MinJoin++ is more powerful than MinJoin. Recall that for MinJoin++ and MinJoin, the verification time increases much faster than other parts of the algorithms. Similar as that on small datasets, the accuracy of EmbedJoin is about 96–99%. On the majority of points, both MinJoin++ and MinJoin achieve perfect accuracy. On those who are not, the accuracy of MinJoin++ and MinJoin is at least 99.7%.

All the deterministic algorithms do not scale well on these large datasets. On the smallest dataset GEN20kS, PassJoin and QChunk can run up to the 8% distance threshold, while VChunk can only go up to the 4% distance threshold. Their running times increase significantly when K increases. Only PassJoin can produce some points on GEN20kL and GEN80kS. On GEN320kS, none of the deterministic algorithms can finish within 24 h.

Memory usage We have also compared the memory usage of all tested algorithms. Figures 10 and 11 present the memory usage of different algorithms on UNIREF, TREC and GEN50kS when varying distance threshold K and the number of input strings n. While the difference on the memory usage is not as large as running time, MinJoin++ still performs the best among all algorithms. The performance of PassJoin is sensitive to K, and is much worse than other algorithms when K is large.

4.4 Short strings and limitation of MinJoin++

As we have shown above, the advantage of MinJoin++ against the state-of-the-art algorithms increases when the lengths of the strings grow. On the other hand, the performance of MinJoin++ indeed degenerates when the strings lengths decrease. This does *not* come at a surprise since MinJoin++ is a partition-based *randomized* algorithm. As with almost all randomized algorithms, MinJoin++'s correctness depends on various concentration inequalities (or, the law of large numbers). We have to create a sufficient number of partitions for each string. If the string length is small, then the partition process will create many short strings which may lead to a significantly larger number of false positive pairs for the verification step and substantially increase the running time. On the other hand, when the string length increases, the matching of substrings will become more accurate, thereby decreasing the number of candidate pairs for

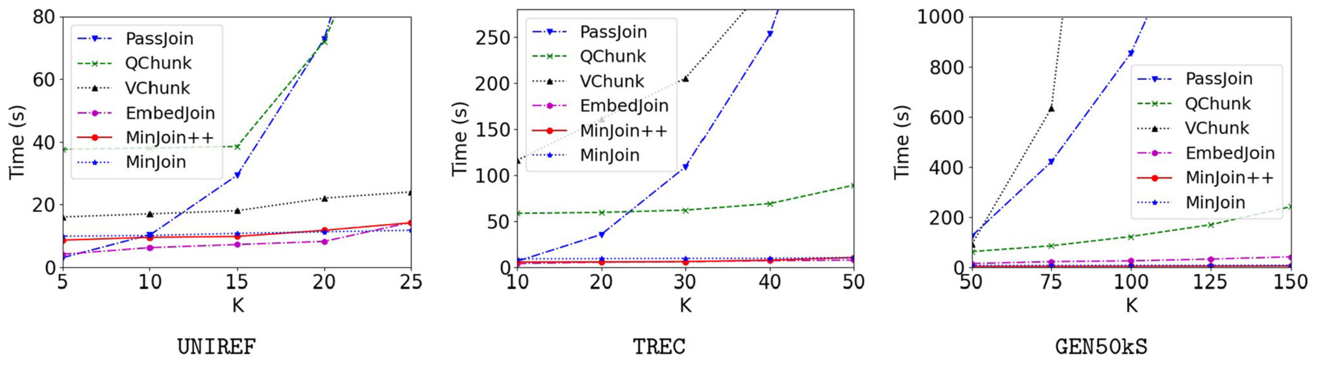

Fig. 6 A comparison on running time, varying K. MinJoin++ and MinJoin achieve perfect accuracy (i.e., 100%) on all points. The accuracy of EmbedJoin is in the range of (95.5–99.5%)

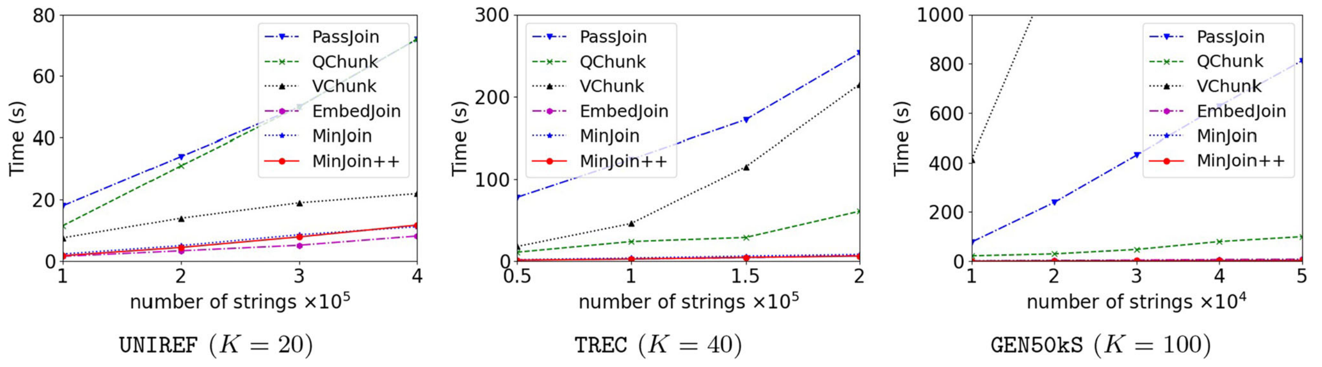

Fig. 7 A comparison on running time, varying n. MinJoin++ and MinJoin achieve perfect accuracy (i.e., 100%) on all points. The accuracy of EmbedJoin is in the range of (95.5–99.5%)

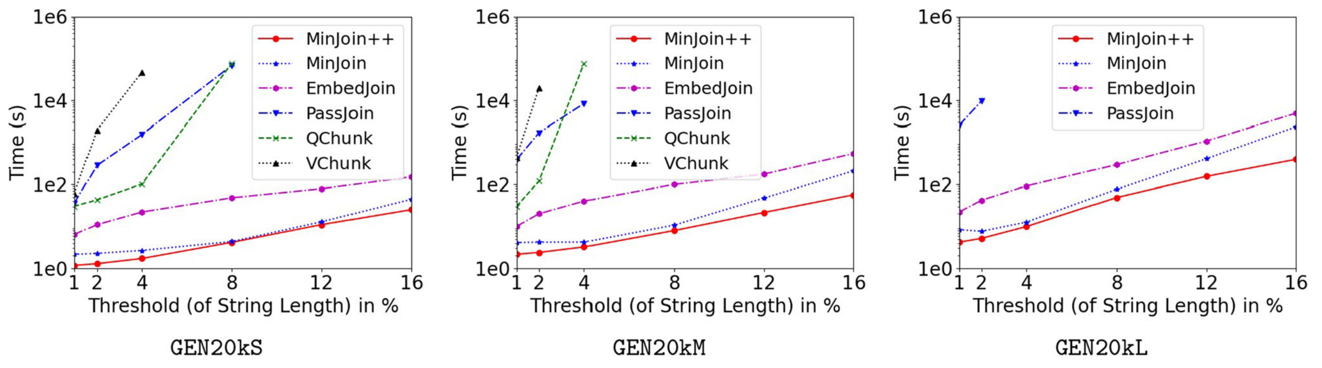

Fig. 8 Scalability of different algorithms, varying N. The accuracy of EmbedJoin is in the range of (95.5–99.9%); that of MinJoin is in the range of (99.9–100%); and that of MinJoin++ is in the range of (99.7–100%)

verification. This is a unique feature of MinJoin++ and is different from most previous algorithms for edit similarity joins.

We have conducted experiments for MinJoin++ on short strings. Figure 12 presents the running time of MinJoin++ on strings of different lengths. For each percentage $p\%$ in the x-axis, we create a new dataset by taking the prefix of length $\frac{p|s|}{100}$ of each string s in the original dataset. We observe on the GEN50kS dataset that when the length of the string goes below $5K$ where K is the distance threshold, the running time increases significantly. This is because substrings generated by the partition procedure in MinJoin++ are too short. Consequently, the join procedure returns a significantly larger number of candidate pairs for verification, since short substrings easily get matches (i.e., cause hash collisions). We also observe on the TREC dataset that if the majority of string lengths fall below 500, then the running time also increases significantly. This is a result of the "law of large numbers" effect, which asserts that regardless of the value of K, we must create a sufficient number of partitions to ensure

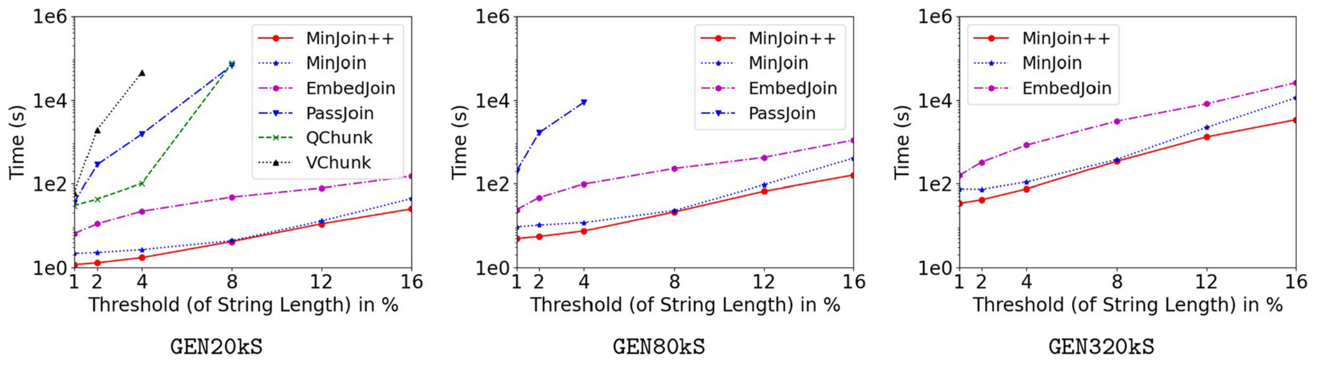

Fig. 9 Scalability of different algorithms, varying n. The accuracy of `EmbedJoin` is in the range of (96.5–99.6%); that of `MinJoin` is in the range of (99.8–100%); and that of `MinJoin++` is in the range of (99.7–100%)

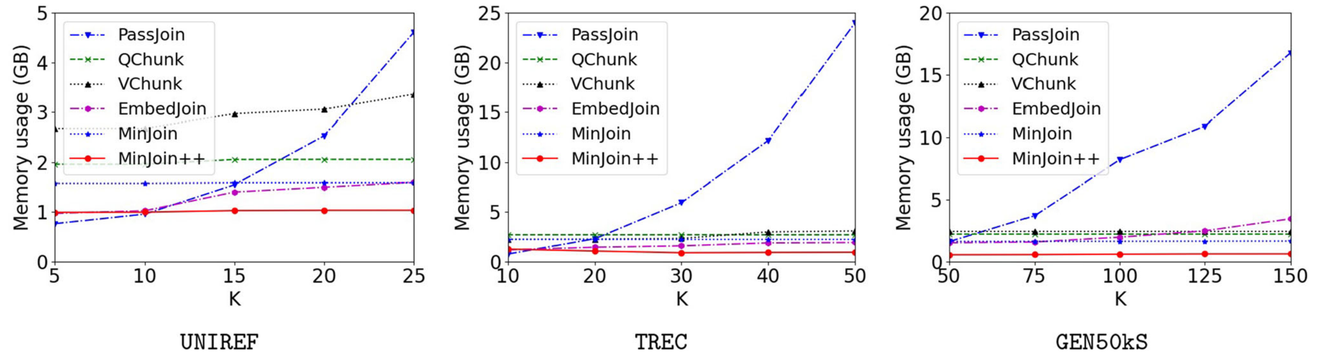

Fig. 10 A comparison on memory usage, varying K

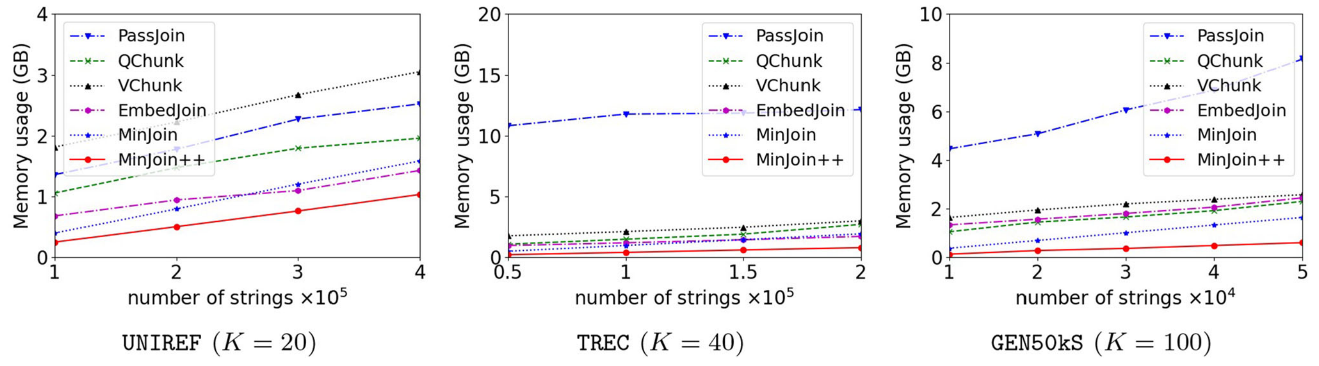

Fig. 11 A comparison on memory usage, varying n

the correctness. If the absolute value of string length is too small, then we will again generate many short substrings.

We have also compared `MinJoin++` with the state-of-the-art algorithm `EmbedJoin` on strings of different lengths on `GEN50kS`. Figure 13 presents the results. We observe that the trends of the two algorithms are very different: the running time of `EmbedJoin` increases when the string lengths increase, while that of `MinJoin++` decreases when the string lengths increase. When the string length is larger than $7K$, `MinJoin++` outperforms `EmbedJoin` in most experiments.

4.5 Parallel implementation of `MinJoin++`

Figures 14, 15 and 16 show the running time of the parallel implementation of `MinJoin++` on all datasets mentioned above. One can see that `MinJoin++` scales well in the multi-thread environment, particularly on large genomics datasets with large distance thresholds. All curves are convex, because the sequential part of the algorithm (input reading) and the partially parallelizable part (e.g., the merge of sorted lists at Line 11 of Algorithm 3) consumes a larger proportion of the overall running time as the number of threads increases.

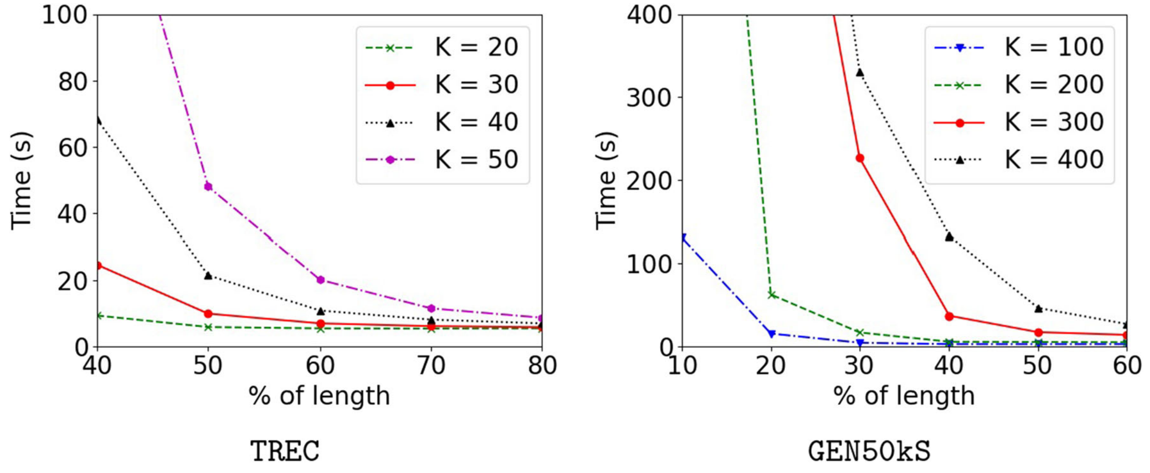

Fig. 12 Running time of `MinJoin++` on `TREC` and `GEN50kS`, varying lengths of strings

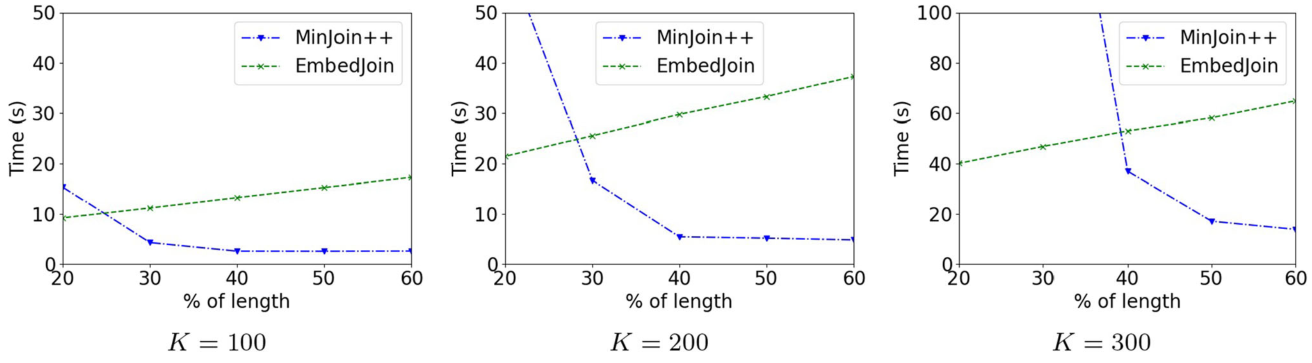

Fig. 13 Performance of `MinJoin++` and `EmbedJoin` on `GEN50kS`, varying lengths of strings

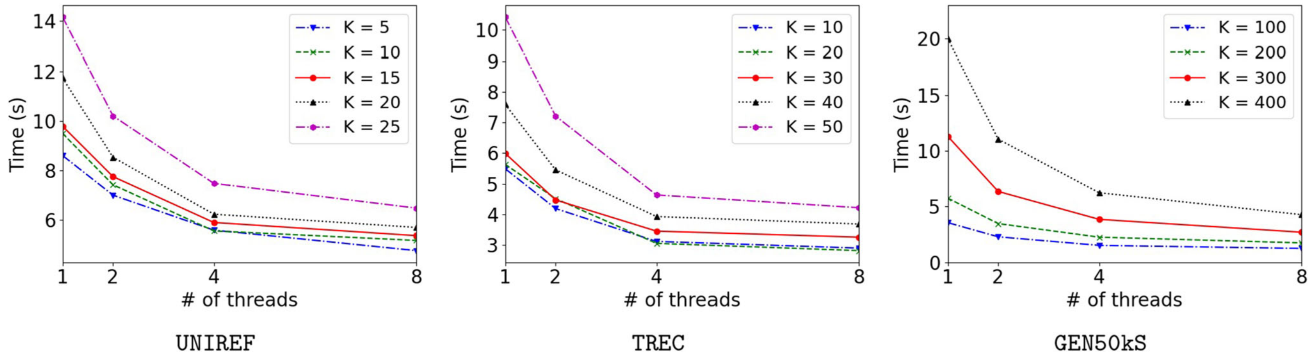

Fig. 14 The influence of the number of threads on running time of `MinJoin++`

Figure 17 gives the running time distribution of `MinJoin++` in the multi-thread setting on `UNIREF`, `TREC`, and `GEN50kS`. It is clear that input reading takes the same amount of time when the number of threads vary. The string partition and the verification stages scales perfectly. The join stage is partially parallelizable mainly due to the merging of sorted lists.

4.6 A comparison with MinHash

Finally, we would like to compare `MinJoin++` with a folklore MinHash-based algorithm. In the MinHash-based algorithm is straightforward: we convert each string into a set which consists of the hash values of all q-grams of the string, and then pick the smallest value as the signature of the string for the subsequent hash join. To boost the accuracy, we can use ℓ such MinHash functions, and get ℓ signatures for

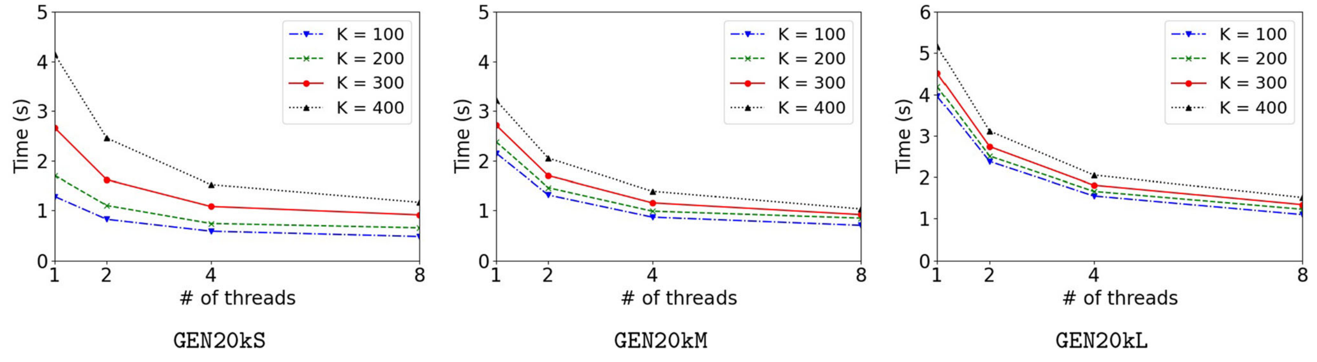

Fig. 15 The influence of the number of threads on running time of `MinJoin++`

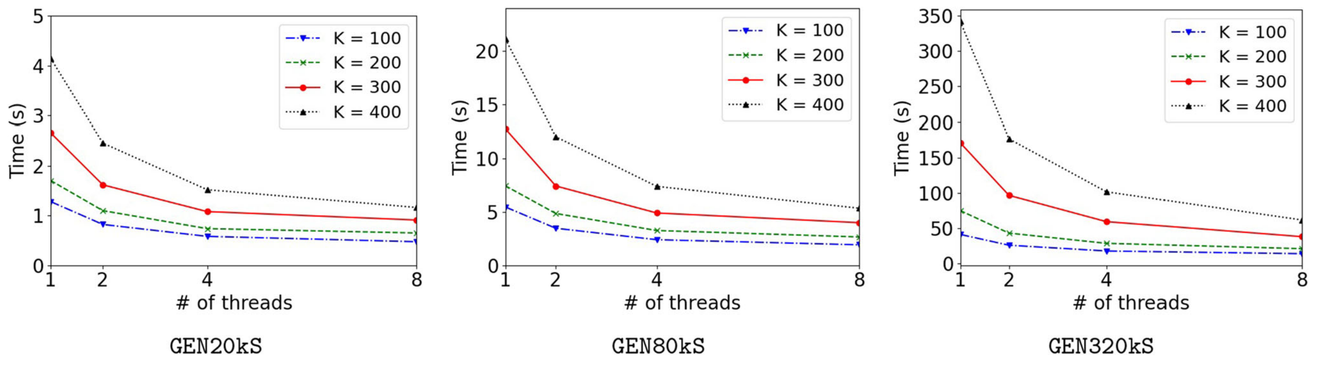

Fig. 16 The influence of the number of threads on running time of `MinJoin++`

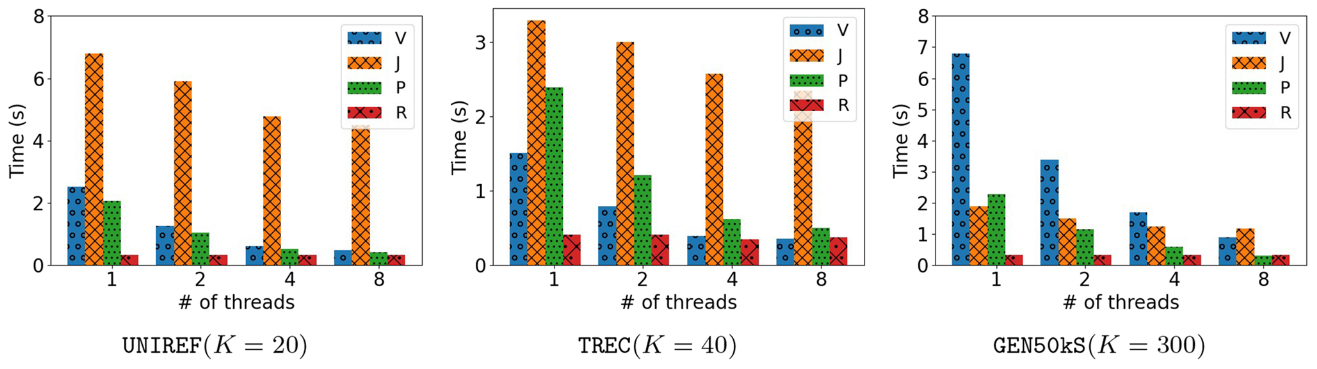

Fig. 17 Running time distribution of input reading (R), string partition (P), join (J), and verification (V) of `MinJoin++`, varying the number of threads

each string. Applying ℓ hash functions to get the signatures is expensive. A standard optimization method is to use only one hash function, and then select the top-ℓ smallest hash values as the signatures. This is what we use in our experiments.

The reason that we discuss it separately is that this folklore algorithm has two parameters, q and ℓ, for which we do not have any guideline for the tuning. We thus try to present its performance by testing different combinations of these parameters.

Figure 18 shows the running time and accuracy of the MinHash-based algorithm when varying the number of hash functions ℓ and the length of signature q. The running time is shown as a multiple of `MinJoin++` at 100% accuracy. We find that the running time and accuracy of the MinHash-based algorithm depend on the two parameters q and ℓ: When increasing parameter ℓ, both running time and accuracy increase; when increasing parameter q, the running time first decreases and then increases a little bit, and the accuracy decreases. We observe that the accuracy and running time are sensitive to parameters, and there is no principle yet on how to select them for edit similarity joins. This is in contrast to `MinJoin++` where the only parameter is T (the targeted number of partitions). And we have already discussed how to choose T theoretically and practically. Most importantly,

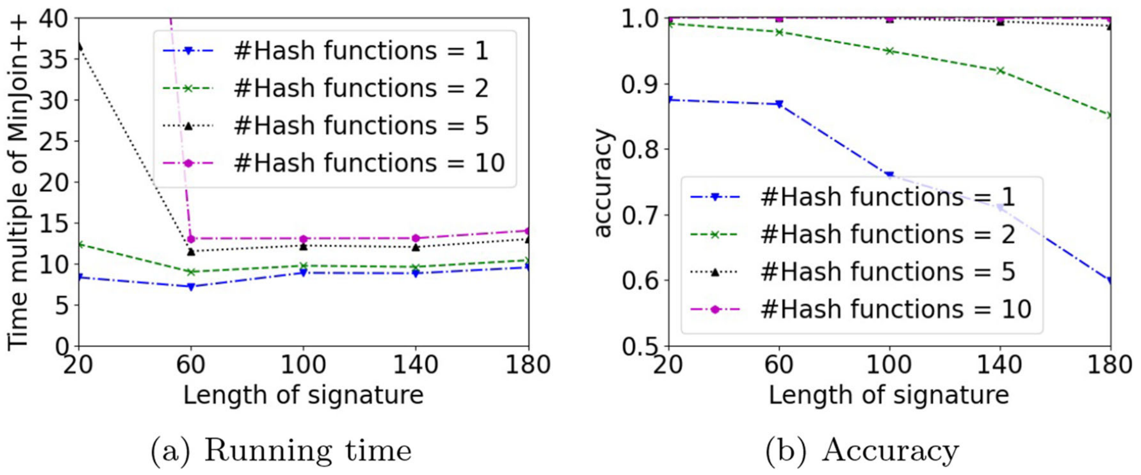

Fig. 18 Performance of the MinHash-based algorithm on `GEN50kS` dataset with $K = 100$. **a** The running time of the MinHash-based algorithm as a multiple of that of `MinJoin++` at 100% accuracy. **b** The accuracy of the MinHash-based algorithm

even we choose the best combination of ℓ and q, the running time of the MinHash-based algorithm is still more than 10 times of that of `MinJoin++` at 100% accuracy. We thus conclude that `MinJoin++` outperforms the MinHash-based algorithm in all aspects.

5 Conclusion

In this paper, we have presented `MinJoin++`, an algorithm for edit similarity joins based on string partition using local hash minima. `MinJoin++` has rigorous mathematical properties, and significantly outperforms previous methods on long strings with large edit thresholds. We feel that local hash minima-based string partition is a natural and elegant way for solving the edit similarity join problem: it can be applied to each string independently by a linear scan, without any synchronization between strings or global statistics of the datasets. Moreover, despite being a randomized algorithm, `MinJoin++` can achieve perfect accuracy on all of the datasets that we have tested when there is an exact deterministic algorithm that can finish within 24 h. We believe `MinJoin++` is the right choice for edit similarity joins in many applications.

Acknowledgements N. Karpov, H. Zhang, and Q. Zhang are supported in part by NSF CCF-1844234.

References

1. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB, pp. 918–929 (2006)
2. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW, pp. 131–140 (2007)
3. Bocek, T., Hunt, E., Stiller, B., Hecht, F.: Fast similarity search in large dictionaries. University (2007)
4. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: VLDB, pp. 426–435 (1997)
5. Dai, X., Yan, X., Zhou, K., Wang, Y., Yang, H., Cheng, J., Sigir. J., Huang, X., Chang, Y., Cheng, X., Kamps, J., Murdock, V., Wen, J., Liu, Y. (eds.) ACM, pp. 599–608
6. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: VLDB, pp. 491–500 (2001)
7. Jiang, Y., Li, G., Feng, J., Li, W.: String similarity joins: an experimental evaluation. PVLDB **7**(8), 625–636 (2014)
8. Li, C., Lu, J., Lu, Y.: Efficient merging and filtering algorithms for approximate string searches. In: ICDE, pp. 257–266 (2008)
9. Li, G., Deng, D., Wang, J., Feng, J.: PASS-JOIN: a partition-based method for similarity joins. PVLDB **5**(3), 253–264 (2011)
10. Myers, G.: Efficient local alignment discovery amongst noisy long reads. In: Brown, D.G., Morgenstern, B. (eds.), Algorithms in Bioinformatics—14th International Workshop, WABI 2014, Wroclaw, Poland, September 8–10, 2014. Proceedings, vol. 8701 of Lecture Notes in Computer Science, pp. 52–67. Springer (2014)
11. Qin, J., Wang, W., Lu, Y., Xiao, C., Lin, X.: Efficient exact edit similarity query processing with the asymmetric signature scheme. In: SIGMOD, pp. 1033–1044 (2011)
12. Roberts, R.J., Carneiro, M.O., Schatz, M.C.: The advantages of SMRT sequencing. Genome Biol. **14**(6), 405 (2013)
13. Song, Y., Tang, H., Zhang, H., Zhang, Q.: Overlap detection on long, error-prone sequencing reads via smooth q-gram. Bioinformatics **36**(19), 4838–4845 (2020)
14. Su, Z., Ahn, B.-R., Eom, K.-Y., Kang, M.-K., Kim, J.-P., Kim, M.-K.: Plagiarism detection using the levenshtein distance and smith-waterman algorithm. In: 2008 3rd International Conference on Innovative Computing Information and Control, pp. 569–569 (2008)
15. Ukkonen, E.: Algorithms for approximate string matching. Inf. Control **64**(1–3), 100–118 (1985)
16. Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., Leser, U.: State-of-the-art in string similarity search and join. SIGMOD Record **43**(1), 64–76 (2014)

17. Wang, J., Li, G., Feng, J.: Trie-join: efficient trie-based string similarity joins with edit-distance constraints. PVLDB **3**(1), 1219–1230 (2010)
18. Wang, J., Li, G., Feng, J.: Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In: SIGMOD, pp. 85–96 (2012)
19. Wang, W., Qin, J., Xiao, C., Lin, X., Shen, H.T.: Vchunkjoin: an efficient algorithm for edit similarity joins. IEEE Trans. Knowl. Data Eng. **25**(8), 1916–1929 (2013)
20. Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
21. Zhang, H., Zhang, Q.: Embedjoin: efficient edit similarity joins via embeddings. In: KDD, pp. 585–594 (2017)
22. Zhang, H., Zhang, Q.: Minjoin: efficient edit similarity joins via local hash minima. In: KDD, pp. 1093–1103. ACM (2019)
23. Zini, M., Fabbri, M., Moneglia, M., Panunzi, A.: Plagiarism detection through multilevel text comparison. In: Proceedings of the Second International Conference on Automated Production of Cross Media Content for Multi-Channel Distribution, AXMEDIS 2006, Leeds, UK, December 13–15, 2006, pp. 181–185. IEEE Computer Society (2006)