Allign: Aligning All-Pair Near-Duplicate Passages in Long Texts

Weiqi Feng*
Shanghai Jiao Tong University
fengweiqi@sjtu.edu.cn

Dong Deng†
Rutgers University
dong.deng@rutgers.edu

ABSTRACT

In this paper, we study the problem of aligning all-pair near-duplicate passages in two long texts. A passage is a sequence of consecutive words in a text. It can begin and end with any word in the text, whether around a period or not. Due to the high computation cost of this problem, existing work all compromise to heuristic alignment methods, which can harm the recall of downstream applications, such as deduplication and plagiarism detection. To address this problem, in this paper, we propose a min-hash based method Allign to find all near-duplicate passage pairs in two long texts. Allign generates a few min-hash values for each passage in the texts and report all the passage pairs sharing enough common min-hash values. However, for a pair of texts with n and m words, there are in total $O(n^2m^2)$ passage pairs (each text contains $O(n^2)$ and $O(m^2)$ passages respectively). Thus it is prohibitively expensive to enumerate all passage pairs in two texts and count their common min-hash values. To address this issue, Allign packs a large number of nearby and overlapping passages with the same min-hash to a "compact window". In total, Allign generates $O(n)$ compact windows to represent all the $O(n^2)$ passages in a text with n words. Next, a pair of compact windows in two texts are matched if they have the same min-hash. The rest of unmatched compact windows are removed. Finally, Allign reports all the passage pairs contained by enough number of matched compact window pairs, which must share the same enough number of min-hash values. In this way, Allign avoids enumerating the enormous number of passage pairs. Last but not least, to make the reported near-duplicate passages more relevant and Allign more efficient, we show how to support a few practical constraints efficiently, including reporting only longest near-duplicates and sentence-level near-duplicates. Experimental results on real-world datasets show that Allign significantly outperforms the state-of-the-art text alignment methods.

ACM Reference Format:
Weiqi Feng and Dong Deng. 2021. Allign: Aligning All-Pair Near-Duplicate Passages in Long Texts. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457548

*Work done at Rutgers University during an internship.
†Corresponding author.

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457548

1 INTRODUCTION

Given two long texts, all-pair near-duplicate passage detection finds all pairs of similar passages in them. For example, in Figure 1, a pair of near-duplicate passages in the 2008 DNC speech and the 2016 RNC speech delivered by the two first ladies are highlighted [20]. It plays an important role in many applications, e.g., deduplication [44] and plagiarism detection [28, 32]. All-pair near-duplicate passage detection is different from the traditional near-duplicate text detection [10, 49], which tells whether two entire texts are similar. For example, in academic papers, reusing a small piece of text from another paper without quotation is deemed as plagiarism. Checking whether the overall papers are near-duplicate cannot identify this kind of academic plagiarism, while the all-pair near-duplicate passage detection can come to rescue.

" … Barack and I were raised with so many of the same values, like you work hard for what you want in life. That your word is your bond; that you do what you say you're going to do. That you treat people with dignity and respect, even if you don't know them and even if you don't agree with them …" --- Michelle Obama Aug 25, 2008	*" … my parents impressed on me the values: that you work hard for what you want in life, that your word is your bond and you do what you say and keep your promise. That you treat people with respect. They taught and showed me values and morals in their daily life. That is the lesson that I continue to pass along to our son …"* --- Melania Trump July 18, 2016

Figure 1: A near-duplicate passage pair in two speeches [20].

One step (the detailed analysis step) in plagiarism detection is to identify all pairs of passages in a source document and a suspicious document that are similar enough [13, 30, 31]. The computation cost of this step is so high that all existing work compromise to some heuristic alignment methods [28, 29]. This is because a document with n words contains $O(n^2)$ passages and there are $O(n^2m^2)$ pairs for a source document with n words and a suspicious document with m words. For long documents (such as academic papers), n and m could easily be tens of thousands, which leads to quintillions (i.e., 10^{18}) of passage pairs, not to mention the cost of calculating the similarities of passage pairs.

Limitations of Existing Work. Existing work resort to seeding-extension-filtering [28] or the fixed-width windows [47]. The fixed-width window strategy only reports similar passage pairs of fixed length, which obviously leads to many false negatives. Many work use the sentences in the texts as the seeds and find similar sentence pairs first (e.g., their jaccard similarity is above a pre-defined threshold). Then nearby similar sentences (say, gap at most x sentences) are merged to form similar passage pairs. However, there are many cases the above strategy cannot capture. For example, one long sentence in a text might be split to multiple short sentences in another text. Then the jaccard similarity between the long sentence

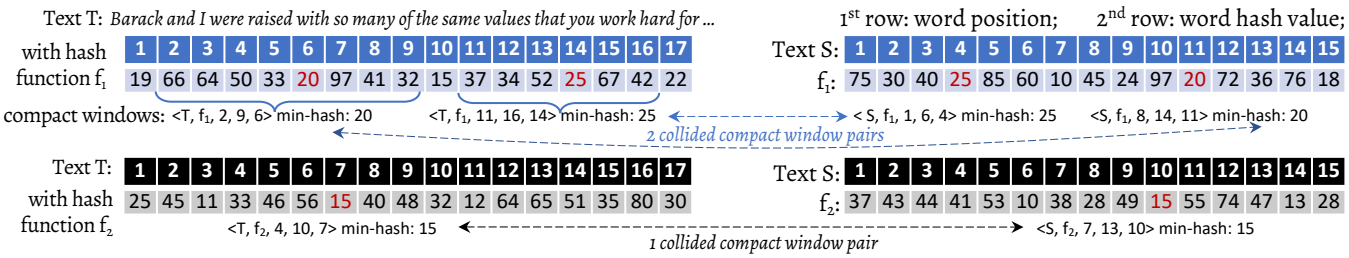


Figure 2: Two texts and their three pairs of example collided compact window pairs under two hash functions.

and any of the short sentences would be very low. Figure 1 shows such an example, where each *that clause* on the left is similar to a sentence on the right. Moreover, such strategy may also generate many false positives for short sentences, whose similarities could be high even if they only share a few common words. In addition, there are languages having no punctuations and sentence breaking is not possible or very difficult. Methods using fingerprints (e.g., n-grams [28]) as seeds have similar problems as discussed above.

In this paper, we study how to find all-pair near-duplicate passages in two long texts whose jaccard similarities are above a given threshold. We propose an efficient min-hash based [5] method Allign. Informally speaking, the min-hash of a passage is the minimum hash value of its words when evaluating against a universal hash function [24]. With k random universal hash functions, k min-hash values can be generated for each passage. Then the jaccard similarity of two passages can be estimated as the percentage of common (i.e., collided) min-hash values in their k pairs of min-hash when k is large enough [24]. Allign reports all passage pairs sharing adequate amount of common min-hash values based on a given jaccard similarity threshold. Though the jaccard similarities between passage pairs can be efficiently estimated by their min-hash, the number of passage pairs in two long texts remains enormous. Next we show how Allign addresses this problem.

Proposed Solution. Let $T[i, j]$ be the window (i.e., passage) from the i-th word to the j-th word in the text T and $T[i]$ be its i-th word. To avoid enumerating all the quadratic number of windows in a text, Allign packs a collection of nearby and overlapping windows with the same min-hash to a *compact window*. For example, Figure 2 on the top-left shows a text T and the hash values of its words obtained with the universal hash function f_1. We observe that the nearby and overlapping windows $T[4, 7], T[4, 8], T[5, 7]$, and $T[5, 8]$ all have the same min-hash, which is the hash value of the word $T[6]$. Based on this observation, we define a compact window as a tuple $\langle l, r, c\rangle$ of three integers, where $T[c]$ has the minimum hash value among all the words in $T[l, r]$. Then, by definition, the hash value of $T[c]$ is the min-hash for all the sub-windows in $T[l, r]$ containing the word $T[c]$. Thus Allign uses the compact window $\langle l, r, c\rangle$ to concisely represent all the windows $T[i, j]$, where $l \leq i \leq c \leq j \leq r$. A challenge here is how to efficiently generate all the compact windows from a text. We show that a text with n non-duplicate words has $O(n)$ compact windows in total, which represent exactly all the $O(n^2)$ windows in the text. Furthermore, we develop an efficient divide-and-conquer algorithm to generate all the $O(n)$ compact windows in $O(n \log n)$ time.

Another challenge is how to avoid enumerating the enormous number of window pairs in two texts. The naive method still needs to enumerate all the window pairs in the matched compact window pairs whose min-hash are the same to count their common min-hash. To address this issue, we propose to look for the collections of matched compact window pairs that share common window pairs and whose sizes are large enough. For each of such collections, the common window pairs in it must share enough min-hash and are near-duplicate passages. Our approach can largely reduce the computation cost as in practice such collections are few in number.

The third challenge is how to deal with the word multiplicity, i.e., each appearance of the same word in a passage leads to a different hash value. That is to say the hash value of a word in a passage is no longer static. We slightly adapt the definition of compact windows to seamlessly handle the word multiplicity in our approach.

Last but not least, we show how to seamlessly enforce a few constraints on the results in Allign, which not only makes the results more relevant but also significantly reduces computation cost. The first one only reports the longest near-duplicate passage pair in the same place, i.e., omitting a near-duplicate pair if they are fully covered by another longer pair. The second one enforces a passage length threshold so that only passages no shorter than the threshold is reported. The third one reports only sentence-level near-duplicate passages. That is the pair of reported passages must begin and end with periods. The longest near-duplicate passage constraint can avoid many redundant results (e.g., the overlapping near-duplicate pairs). The length-threshold constraint can avoid very short near-duplicates, which are usually noise and undesired [13]. The sentence-level constraint can reduce the computation cost. In practice, many near-duplicate passages start and end with periods.

Contributions. In summary, we make the following contributions.

- We propose to use a compact window to represent a large number of windows in a text and develop a divide-and-conquer algorithm to generate all the compact windows in a text in $O(n \log n)$ time.
- We avoid enumerating the enormous number of window pairs in two texts by designing an efficient algorithm to generate all the window pairs sharing enough min-hash directly from their matched compact window pairs.
- We show how to seamlessly deal with word multiplicity and efficiently enforce three practical constraints and their combinations on the near-duplicate passages in Allign.
- We conduct extensive experiments on real-world datasets, which show that Allign significantly outperforms the state-of-the-art text alignment algorithms.

2 PRELIMINARIES

Problem Definition. We first introduce some notations and then define our problem. A text T is a sequence of $|T|$ words. $T[i]$ is its i-th word. We denote the passage from the i-th word to the j-th word

as $T[i, j]$, where $1 \leq i \leq j \leq |T|$. For ease of presentation, we use "passage" and "window" interchangeably. Let $|T[i_1, j_1] \cap S[i_2, j_2]|$ be the number of common words in the two passages $T[i_1, j_1]$ and $S[i_2, j_2]$ and $|T[i_1, j_1] \cup S[i_2, j_2]|$ be the total number of their distinct words. The jaccard similarity of these two passages is defined as

$$jaccard(T[i_1, j_1], S[i_2, j_2]) = \frac{|T[i_1, j_1] \cap S[i_2, j_2]|}{|T[i_1, j_1] \cup S[i_2, j_2]|}.$$

We first consider the general version of the all-pair near-duplicate passage alignment problem in two long texts as defined below.

Definition 2.1 (all-pair near-duplicate passage alignment). Given two texts T and S, a similarity threshold θ, it finds all the passage pairs $T[i_1, j_1]$ and $S[i_2, j_2]$ s.t. $jaccard(T[i_1, j_1], S[i_2, j_2]) \geq \theta$.

For example, consider the two texts T on the left and S on the right in Figure 1. Let the similarity threshold $\theta = 0.7$. Then $T[13, 47]$ and $S[7, 39]$ (the highlighted ones) are a pair of near-duplicate passages as $jaccard(T[13, 47], S[7, 39]) = \frac{21}{27} \geq \theta$.

For ease of understanding, we first consider the simple case where all words in a text are distinct, i.e., no word appears multiple times in a text. We show how to adapt our techniques to lift the above no-duplicate-word assumption in Section 6. In addition, we show how to address a few more practical variants of the all-pair near-duplicate passage alignment problem in Sections 5.4 and 5.5, including (1) generating only deduplicated, longest near-duplicate passages; (2) generating sentence-level near-duplicate passages rather than word-level ones; and (iii) generating near-duplicate passages no shorter than a given threshold.

Min-Hash. Let π be a random permutation $\pi : U \rightarrow U$, where U is the universe. $\pi(S)$ is the sequence of elements in the set S sorted in the order of π. The min-hash of a set S is the first element in $\pi(S)$, denoted as $min(\pi(S))$. The probability at which the min-hash of two sets are equal is equivalent to their jaccard similarity, i.e.,

$$\mathbf{Pr}(min(\pi(S_1)) = min(\pi(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = jaccard(S_1, S_2).$$

Then one can estimate the jaccard similarity between two sets S_1 and S_2 from k independent permutations as

$$\frac{1}{k} \sum_{i=1}^{k} 1\{min(\pi_i(S_1)) = min(\pi_i(S_2))\}.$$

However, it is expensive to generate and store k permutations. In practice, the universal hashing is usually employed and entails good enough performance [24]. A universal hashing is a family of hash functions. In our implementation, we use $hash(x) = (ax+b) \mod p$ where a and b are randomly chosen integers modulo p with $a \neq 0$ and p is a large prime and x is an integer id of a word. Each distinct word is mapped to an auto-increment integer id. Since the min-hash is an accurate estimate of the jaccard similarity, hereinafter, we focus on finding all window pairs whose min-hash collide at least $\tau = \lceil \theta k \rceil$ times out of k random universal hash functions.

Example 2.2. As shown in Figure 2, consider the two texts T and S on the left and right sides. The random universal hash function f_1 hashes each word in the two texts to an integer as shown in the two blue tables on top, where the first row shows word positions and the second row shows the word hash value. Similarly, another random universal hash function f_2 hashes the words in the two texts and generates the two black tables on the bottom. For the first hash function, the min-hash of T is $f_1(T[10]) = 15$ as the word $T[10]$ has the minimum hash value. Similarly the min-hash of S is $f_1(S[7]) = 10$. For the second hash function, the min-hash of T is $f_2(T[3]) = 10$, which collides with the min-hash $f_2(S[6]) = 10$ of S. Thus we estimate the jaccard similarity of T and S as $1/2$ as their min-hash collide once out of two hash functions (i.e., permutations).

Algorithm 1: FRAMEWORK(T, S, k, θ)

Input: T and S: two texts; k: the number of hash functions; θ: the similarity threshold;
Output: All the near-duplicate passage pairs in T and S.

```
initialize an empty hash table H;
foreach i = 1 to k do
    randomly choose a universal hash function hash;
    calculate the min-hash for all the windows in T;
    calculate the min-hash for all the windows in S;
    for each pair of windows from T and S with the same
    min-hash, increase its occurrence in H by 1;
return all window pairs in H with at least ⌈θk⌉ occurrences;
```

3 THE FRAMEWORK

Our framework enumerates every window of the two texts and calculate their min-hash. A pair of windows in the two documents collide if they share the same min-hash value. This process is repeated k times and all window pairs collide at least $\tau = \lceil \theta k \rceil$ times are returned. The pseudo-code is shown in Algorithm 1.

Let n and m be the numbers of words in the two texts. There are $O(n^2)$ and $O(m^2)$ windows in the two texts and in the worst case all windows share the same min-hash. Thus the time complexity of this approach is $O(kn^2m^2)$ and the space complexity is $O(n^2m^2)$.

Clearly, the framework approach is prohibitively expensive as a large number of windows and window pairs are enumerated. We observe that the nearby or overlapping windows often share the same min-hash. Thus we propose to pack the nearby and overlapping windows with the same min-hash to a compact representation and generate results directly from the compact representations.

4 COMPACT WINDOW

To avoid enumerating the enormous number of windows in a text, in this section, we first propose to use a few "compact windows" to represent all the windows in a text and then show how to efficiently generate all the compact windows in a text.

4.1 Compact Window

A compact window concisely represents a large number of nearby and overlapping windows that share the same min-hash in a text. As shown in Figure 2 on the top-left, we observe that any window in $T[2, 9]$ containing $T[6]$ would have the same min-hash $hash(T[6]) = 20$. This is because the word $T[6]$ has the minimum hash value among all the words in $T[2, 9]$. By the definition of min-hash, its hash value is the min-hash for any window in $T[2, 9]$ containing $T[6]$. Based on this observation, we define a compact window as a tuple $\langle l, r, c \rangle$ of three integers, where $T[c]$ has the minimum hash value among all the words in $T[l, r]$. Moreover, to

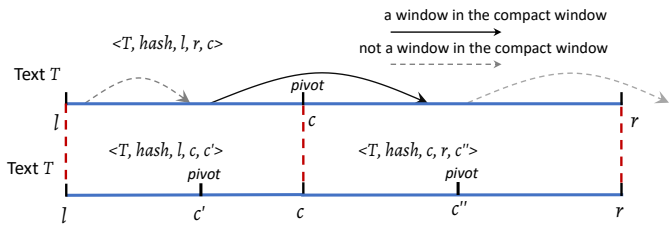


Figure 3: Generating compact windows: divide-and-conquer.

reduce redundancy, the range $[l, r]$ must be maximal, i.e., extending the range $[l, r]$ on either side (if possible) would introduce a word with a smaller hash value than $T[c]$. We formally define a compact window as below. Note here we assume there is no duplicate word and hash value in a text. We lift the assumptions in Section 6.

Definition 4.1 (Compact Window). Given a hash function *hash*, a compact window in a text T is a tuple $\langle T, hash, l, r, c\rangle$ where

- $1 \le l \le c \le r \le |T|$;
- $hash(T[c]) < hash(T[i])\ \forall\ l \le i \ne c \le r$;
- $hash(T[c]) > hash(T[l-1])$ if $l \ne 1$;
- $hash(T[c]) > hash(T[r+1])$ if $r \ne |T|$.

For example, $\langle T, f_1, 2, 9, 6\rangle$ is a compact window as shown in Figure 2 on the top-left side. For ease of presentation, we abbreviate a compact window as $\langle l, r, c\rangle$ when the hash function *hash* and the text T are clear from the context. As shown in Figure 3 on the top, we say $T[i, j]$ is a window of the compact window $\langle l, r, c\rangle$ iff. $l \le i \le c \le j \le r$. Obviously, a compact window $\langle l, r, c\rangle$ has $(c - l + 1)(r - c + 1)$ windows, which are all the sub-windows in $T[l, r]$ containing the word $T[c]$. All the windows of a compact window share the same min-hash, as stated below.

Lemma 4.2. *For any compact window $\langle T, hash, l, r, c\rangle$, the min-hash values of all its windows are the same, which are $hash(T[c])$.*

The proof is straight-forward following Definition 4.1. The formal proof is omitted here due to the space limit. For any compact window $\langle T, hash, l, r, c\rangle$, we define its *pivot* as c and pivot word as $T[c]$. Based on the maximality property of compact windows, it is easy to see that, there is one and only one compact window with pivot c in a text T for each $1 \le c \le |T|$, as formalized below.

Lemma 4.3. *There is one and only one compact window with pivot c in a text T for each $1 \le c \le |T|$.*

Proof. We prove it by contradiction. Suppose there are two compact windows $\langle l, r, c\rangle$ and $\langle l', r', c\rangle$ where $l \ne l'$ or $r \ne r'$ for some $1 \le c \le |T|$. Without loss of generality, suppose $l \ne l'$ and $l < l'$. On one hand, by Definition 4.1, as $\langle l', r', c\rangle$ is a compact window, $hash(T[l'-1]) < hash(T[c])$. On the other hand, since $l \le l' - 1 < c \le r$, we have $l' - 1 \in [l, r]$. In addition, based on Definition 4.1, as $\langle l, r, c\rangle$ is a compact window, $hash(T[l'-1]) > hash(T[c])$, which contradicts with $hash(T[l'-1]) < hash(T[c])$. □

Clearly, a text T with n words has exactly n compact windows, one for each position. We define the min-hash of a compact window $\langle T, hash, l, r, c\rangle$ as $hash(T[c])$, i.e., the hash value of its pivot word. Clearly, none of any two compact windows in a text have the same min-hash under the "no-duplicate-word" assumption. Thus any window in T can be in at most one compact window of T (otherwise the window has two different min-hash values, which contradicts with the definition of min-hash), as stated below.

Lemma 4.4. *Any window in T is in one and only one compact window of T.*

Proof. For any window $T[i, j]$, let its min-hash be the hash value of $T[c]$. Clearly $i \le c \le j$. Consider the compact window $\langle l, r, c\rangle$ with pivot c. By Definition 4.1 and the definition of min-hash, $l \le i$ and $r \ge j$ must hold. Thus the window $T[i, j]$ must be in this compact window. On the other hand, none of any two compact windows in the text T have the same min-hash value. If there is a window exists in two compact windows, the window must have two different min-hash values based on Lemma 4.2, which contradicts with the definition of min-hash. □

Based on Lemma 4.4, all the $O(n^2)$ windows in a text T are in the compact windows of T, which there are only $O(n)$ of them. We show in Section 5 how to produce all near-duplicate passages directly from the compact windows. Next we first develop two algorithms to generate all the compact windows in a given text.

4.2 Generating Compact Windows

Based on Lemma 4.3, we develop Algorithm 2 to generate all the compact windows in a text T. It visits each position c in the input text T and generates the compact window with pivot c. Specifically, it starts with $l = r = c$ and decreases l (or increases r) by 1 (if possible) as long as $hash(T[c]) < hash(T[l])$ (or $hash(T[c]) < hash(T[r])$) holds. Once l and r are fixed, $\langle l, r, c\rangle$ must be a compact window based on Definition 4.1 and is added to the result set. Finally, all the n compact windows in the result set are returned. The time complexity of this algorithm is $O(n^2)$. This is because in the worst case the whole text is scanned for each position c. The space complexity is $O(n)$ for storing the n compact windows.

Example 4.5. As shown in Figure 2, consider the text T on the left and the hash function f_1. It first generates a compact window with pivot $c = 1$. At the beginning, $l = r = c = 1$. As l cannot be smaller, we have $l = 1$. As $f_1(T[c = 1]) = 18 < f_1(T[r + 1 = 2]) = 66$, we increase r by 1 and r is 2. Next, as $f_1(T[1]) < f_1(T[3]) = 64$, r is increased by 1 again and becomes 3. This is repeated until r is increased to 9. As $f_1(T[1]) = 18 > f_1(T[10]) = 15$, the while loop stops. A compact window $\langle l = 1, r = 9, c = 1\rangle$ is generated. Next it finds the compact window for the pivot $c = 2$ and gets $\langle 2, 2, 2\rangle$. This is repeated until $c = 17$ when all compact windows are generated.

A Divide-and-Conquer Algorithm. Next we design a more efficient divide-and-conquer algorithm to accommodate long texts. As shown in Figure 3, instead of finding compact windows from the beginning of the text, we propose to generate the compact window for the word with the smallest hash value first. More specifically, let $T[c]$ be the word in T with the smallest hash value. Based on Definition 4.1, $\langle 1, n, c\rangle$ must be a compact window. Moreover, no other compact window in T can go across (i.e., contain) $T[c]$; otherwise, suppose $\langle l', r', c'\rangle$ is a compact window where $l' \le c' \ne c \le r'$. Based on Definition 4.1, the hash value of $T[c']$ must be smaller than that of $T[c]$, which contradicts with that the hash value of $T[c]$ is minimum among all words in T. Thus it is sufficient to recursively find compact windows in the two parts $T[1, c-1]$ and $T[c+1, n]$

Algorithm 2: ScanCompactWindow(T, *hash*)

Input: T: a text; *hash*: a hash function.
Output: All compact windows in T.

```
foreach 1 ≤ c ≤ |T| do
    l = r = c;
    while l ≥ 1 and hash(T[l−1])>hash(T[c]) do  l=l−1
    while r≤|T| and hash(T[r+1])>hash(T[c]) do  r=r+1
    add ⟨l, r, c⟩ to A;
return A;
```

Algorithm 3: DivCompactWindow(T, *hash*)

Input: T: a text; *hash*: a hash function.
Output: All compact windows in T.

```
build a segment tree ST with all the hash values in T;
DivideConquer(ST, 1, |T|, A);
return A;
```

Algorithm 4: DivideConquer(ST, l, r, A)

Input: ST: a segment tree; $[l, r]$: a range; A: compact windows.

```
if r ≥ l then
    find T[c] with the min. hash value by ST where l ≤ c ≤ r;
    add ⟨l, r, c⟩ to A;
    DivideConquer(ST, l, c − 1, A);
    DivideConquer(ST, c + 1, r, A);
```

independently. This procedure is repeated until all parts are empty. Since each position generates one and only one compact window, the above process can obtain all the n compact windows in the text. The pseudo-code is shown in Algorithms 3 and 4.

Example 4.6. Consider the text T and the hash function f_1 in Figure 2. Starting with the range $l = 1$ and $r = |T| = 17$, the minimum hash value in $T[1, 17]$ is $f_1(T[10]) = 15$. Thus $c = 10$ and a compact window $\langle 1, 17, 10\rangle$ is generated. Then, it divides the range into two $[l, c-1] = [1, 9]$ and $[c+1, r] = [11, 17]$ and finds compact windows in them. For the former one, the minimum hash value in it (i.e., $T[1, 9]$) is $f_1(T[1]) = 18$. Therefore $c = 1$ and a compact window $\langle 1, 9, 1\rangle$ is generated. Next the range $[1, 9]$ is divided into two $[1, 0]$ and $[2, 9]$. For the first one, as $r = 0 < l = 1$, no compact window is generated and the range is not further divided.

Theorem 4.7. *Algorithms 2 and 3 are sound (i.e., all compact windows in the input text are generated) and correct (i.e., all tuples generated are compact windows of the input text).*

Complexity Analysis. Note that to efficiently find the minimum hash value in a range, we build a segment tree, which is a balanced binary tree. Each leaf node corresponds to a position in the text. In addition, each node in the segment tree is associated with a range, the minimum hash value in the range, and its position. For each leaf node of position p, its range, minimum hash value and word position are $[p, p]$, $hash(T[p])$, and p, respectively. The range of an internal node is the union of the ranges in its two children. The minimum hash value and position are copied from the child node with smaller minimum hash value. Using the segment tree, the minimum hash value and its position in any range can be retrieved in $O(\log n)$ time [8], where n is the text length.

The time and space complexity of the divide-and-conquer algorithm are respectively $O(n \log n)$ and $O(n)$. This is because the function DivideConquer is invoked once for each position in the text. Thus in total it is invoked $O(n)$ times. By leveraging the segment tree, each invocation of the function takes $O(\log n)$ time to find the minimum hash value. The segment tree takes $O(n)$ time and space to construct. Thus the total time complexity is $O(n \log n)$.

5 FINDING ALL-PAIR NEAR-DUPLICATES

In this section, we discuss how to generate all-pair near-duplicate passages directly from the compact windows of two texts and how to enforce a few practical constraints in the results.

5.1 Collided Compact Window Pairs

Once all the compact windows of two texts T and S are generated, we pair their compact windows with the same min-hash, i.e., $\langle T, h_1, l_1, r_1, c_1\rangle$ from T and $\langle S, h_2, l_2, r_2, c_2\rangle$ from S where $h_1 = h_2$ and $h_1(T[c_1]) = h_2(S[c_2])$, which we call a *collided compact window pair*. For example, Figure 2 shows three collided compact window pairs. The compact window $\langle T, f_1, 2, 9, 6\rangle$ collides with another compact window $\langle S, f_1, 8, 14, 11\rangle$ as $f_1(T[6]) = f_1(S[11]) = 20$.

Based on Lemma 4.2, all window pairs in a collided compact window pair share the same min-hash. A naive approach would enumerate all the window pairs in every collided compact window pair and count the occurrence of the window pairs. All window pairs with at least $\tau = \lceil k\theta \rceil$ occurrence are near-duplicate passages. Clearly, this naive approach is prohibitive expensive as the number of window pairs is huge as discussed earlier.

To avoid the enumeration, we observe that the number of results, i.e., the window pairs sharing at least τ min-hash, is not too many in practice. Thus, in contrast, we propose to find the set of at least τ collided compact window pairs whose "intersection" is non-empty, i.e., at least one window pair appear in every collided compact window pair in the set and thus share all their min-hash. Obviously, all the window pairs in the "intersection" of such a set are results. Moreover, we can prove every result is in the intersection of one such set. Thus it is sufficient for us to find all such sets.

5.2 Generating Near-Duplicate Passage Pairs

For ease of presentation, for a compact window $\langle l, r, c\rangle$, we name $[l, c]$ and $[c, r]$ as its *left* and *right* intervals, respectively. Let W_x be the set of all compact windows in T. We first generate all the subsets of W_x i) whose size is at least $\tau = \lceil k\theta \rceil$, ii) the intersection of all the left intervals of whose compact windows is non-empty, and iii) the intersection of all the right intervals of whose compact windows is non-empty. For each of such subset W'_x, we have $|W'_x| \geq \tau$, the intersection of its left intervals $[l^l_x, l^r_x] \neq \phi$, and the intersection of its right intervals $[r^l_x, r^r_x] \neq \phi$. Obviously, for any $l_x \in [l^l_x, l^r_x]$ and $r_x \in [r^l_x, r^r_x]$, the window $T[l_x, r_x]$ must appear in every compact window in the subset W'_x. Next, we process S similarly as follows.

Let W_y be the set of compact windows in S collided with at least one compact window in W'_x. Then, again, we generate all the subsets of W_y i) whose size is at least τ, ii) the intersection of whose left intervals is non-empty, and iii) the intersection of whose right intervals is non-empty. For each of such subset W'_y, we have $|W'_y| \geq \tau$ and any window $S[l_y, r_y]$ starting from $l_y \in [l^l_y, l^r_y]$,

the intersection of all the left intervals in W'_y, and ending with $r_y \in [r^l_y, r^r_y]$, the intersection of all the right intervals in W'_y, must be in every compact window in W'_y. As every compact window in W'_y collides with at least one compact window in W'_x, the window pair $T[l_x, r_x]$ and $S[l_y, r_y]$ must appear in at least τ collided compact window pairs for any $l_x \in [l^l_x, l^r_x]$, $r_x \in [r^l_x, r^r_x]$, $l_y \in [l^l_y, l^r_y]$, and $r_y \in [r^l_y, r^r_y]$. That is to say the window pair must share at least τ min-hash values and are a near-duplicate passage pair.

Moreover, every near-duplicate passage pair in T and S (i.e., a window pair sharing at least τ common min-hash) corresponds to a pair of such subsets W'_x and W'_y, as formalized below.

Lemma 5.1. *A pair of windows $T[l_x, r_x]$ and $S[l_y, r_y]$ share τ common min-hash iff. there are τ collided compact window pairs $\langle T, h_i, l^i_x, r^i_x, c^i_x \rangle$ and $\langle S, h_i, l^i_y, r^i_y, c^i_y \rangle$, where $1 \le i \le \tau$, such that $l_x \in \cap_i [l^i_x, c^i_x]$, $r_x \in \cap_i [c^i_x, r^i_x]$, $l_y \in \cap_i [l^i_y, c^i_y]$, and $r_y \in \cap_i [c^i_y, r^i_y]$.*

5.3 The Allign Algorithm

Algorithm 5 shows the pseudo-code for aligning all-pair near-duplicate passages in two texts T and S. It first generates all the compact windows W_x and W_y in the two texts with k different hash functions (Lines 1 to 4). Those compact windows in W_x do not collided with any compact window in W_y are removed from W_x as they lead no collided compact window pair (Line 5). Then, the procedure FindSubsets finds all qualified subsets of W_x, i.e., subsets of size at least $\tau = \lceil k\theta \rceil$ and whose left and right interval intersections are both non-empty (Line 7). Next, for each of such subset W'_x, the algorithm first gets its corresponding collided subset W'_y in W_y, i.e., the set of compact windows in W_y collided with those in W'_x. Then it invokes FindSubsets again to find all the qualified subsets of W'_y (Lines 8 to 10). Finally, for each of such subset W''_y, the algorithm gets its corresponding collided subset W''_x in W'_x. The compact windows in W''_x and W''_y form collided compact window pairs and there are at least τ of them. Every window pair in the "intersection" of all these collided compact window pairs are a near-duplicate passage pair (Lines 11 to 13). Specifically, let the i-th compact window in W''_x and W''_y be $\langle l^i_x, r^i_x, c^i_x \rangle$ and $\langle l^i_y, r^i_y, c^i_y \rangle$, respectively. Then the window pair $T[l_x, r_x]$ and $S[l_y, r_y]$ are a near-duplicate pair for every $(l_x, r_x, l_y, r_y) \in \cap_i [l^i_x, c^i_x] \times \cap_i [c^i_x, r^i_x] \times \cap_i [l^i_y, c^i_y] \times \cap_i [c^i_y, r^i_y]$.

Algorithms 6, 7, and 8 show the pseudo-code for finding all qualified subsets of an input compact window set W. FindSubsets is the entry point. It repeatedly invokes PushDown to insert all the left (or right) intervals of W to a segment tree ST. The segment tree (a.k.a., balanced binary tree) ST records the widest disjoint ranges/segments each interval covers. It can be implemented using an array of size $2n$, where n is the text length. Each tree node has a range field. For the leaf node ST[n+i].range = [i+1, i+1], where $0 \le i < n$. The range of an internal node is the union of the ranges of its two child nodes, i.e., ST[i].range = ST[2i].range $\cup$ ST[2i+1].range for $1 \le i < n$. In addition, we maintain a global timestamp for ST and a local timestamp for each node ST[i] to reuse the segment tree.

As illustrated in Algorithm 7, each interval is inserted to the first tree nodes whose ranges are covered by the interval in a top-down manner (Lines 1 to 7), which there are at most $\log n$ of them [8]. The field "list" of a node keeps the *id*s of compact windows whose intervals are inserted to this node. All tree nodes whose ranges fully covered by any interval are added to C as candidates (Line 2).

Algorithms 8 is invoked by FindSubsets to refine the candidates. For each node in C, it merges the compact windows on this node

Algorithm 5: Allign(T, S, k, θ)

Input: T: a text; S: another text; k: the number of hash functions; θ: similarity threshold;
Output: All-pair near-duplicate passages in T and S.

1 **foreach** $i = 1$ *to* k **do**
2 Generate a random universal hash function h_i;
3 $W_x = W_x \cup$ DivCompactWindow(T, h_i);
4 $W_y = W_y \cup$ DivCompactWindow(S, h_i);
5 Remove from W_x compact windows have no collision in W_y;
6 Build a segment tree ST with the range $[1, max(|T|, |S|)]$;
7 A_x = FindSubsets(ST, $\lceil k\theta \rceil$, W_x);
8 **foreach** $W'_x \in A_x$ **do**
9 Let W'_y be the "collided" subset of compact windows in W_y collided with those in W'_x;
10 A_y = FindSubsets(ST, $\lceil k\theta \rceil$, W'_y);
11 **foreach** $W''_y \in A_y$ *and its collided subset* W''_x *of* W'_x **do**
12 Add the pair $T[l_x, r_x]$ and $S[l_y, r_y]$ to A for every
$l_x \in \cap_{\langle l^i_x, c^i_x, r^i_x \rangle \in W''_x} [l^i_x, c^i_x] = [max_i(l^i_x), min_i(c^i_x)]$,
$r_x \in \cap_{\langle l^i_x, c^i_x, r^i_x \rangle \in W''_x} [c^i_x, r^i_x] = [max_i(c^i_x), min_i(r^i_x)]$,
$l_y \in \cap_{\langle l^i_y, c^i_y, r^i_y \rangle \in W''_y} [l^i_y, c^i_y] = [max_i(l^i_y), min_i(c^i_y)]$,
$r_y \in \cap_{\langle l^i_y, c^i_y, r^i_y \rangle \in W''_y} [c^i_y, r^i_y] = [max_i(c^i_y), min_i(r^i_y)]$;
13 **return** A;

Algorithm 6: FindSubsets(ST, τ, W)

Input: ST: a segment tree; τ: an integer; W: compact windows.
Output: A: all subsets of W of size at least τ whose left and right interval intersections are both non-empty.

1 **foreach** *the i-th compact window* $\langle l, r, c \rangle \in W$ **do**
2 PushDown(ST$'$, ST$'$.root = 1, l, c, i, C');
3 Refine(ST$'$, τ, C', A');
4 **foreach** $W' \in A'$ **do**
5 **foreach** $\langle l, r, c \rangle \in W'$, *which is the i-th one in* W **do**
6 PushDown(ST, ST.root = 1, c, r, i, C);
7 Refine(ST, τ, C, A);
8 **return** A;

Algorithm 7: PushDown(ST, *node*, l, r, i, C)

Input: ST: a segment tree; *node*: node index; $[l, r]$: an interval; i: a compact window id; C: candidates.

1 **if** ST[*node*].*range is fully covered by* $[l, r]$ **then**
2 add i to ST[*node*].list and *node* to C;
3 **else**
4 **if** $[l, r]$ *intersects with* ST[$2 \times$ *node*].*range* **then**
5 PushDown(ST, $2 \times$ *node*, l, r, i, C)
6 **if** $[l, r]$ *intersects with* ST[$2 \times$ *node* + 1].*range* **then**
7 PushDown(ST, $2 \times$ *node* + 1, l, r, i, C)

Algorithm 8: REFINE(ST, τ, C, A)

Input: ST: segment tree; τ: threshold; C: candidates; A: results.

1 **foreach** *node* $\in C$ **do**

2 let W be the union of list in ST[*node*] and all its ancestors;

3 add W to A if it is generated by $\geq \tau$ unique hash functions;

4 deduplicate A and **return** A;

and all its ancestors (Lines 1 to 2). These compact windows are exactly all the compact windows whose left (or right) intervals covering the range of the node. Only if those compact window are generated by at least $\tau = \lceil k\theta \rceil$ distinct hash functions out of all the k ones, the set W of all these compact windows is added to the result A as a qualified subset (Line 3). This is because the range of this node is in the intersection of all the intervals of W, which is non-empty and the size of W is no smaller than τ. The requirement for distinct hash functions is to guarantee the collisions are from at least τ different universal hash functions. Finally, the deduplicated results A are returned (Line 4).

Example 5.2. Consider the example in Figure 3 and suppose $k = 2$, $\theta = 0.8$, and $\tau = \lceil 1.6 \rceil = 2$. ALIGN first generates all the compact windows in T and S using the two universal hash functions f_1 and f_2. After removing the compact windows with no collision, W_x contains 6 compact windows $w_1 = \langle T, f_1, 2, 9, 6 \rangle$, $w_2 = \langle T, f_1, 7, 7, 7 \rangle$, $w_3 = \langle T, f_1, 11, 16, 14 \rangle$, $w_4 = \langle T, f_2, 4, 10, 7 \rangle$, $w_5 = \langle T, f_1, 6, 6, 6 \rangle$, $w_6 = \langle T, f_2, 12, 14, 14 \rangle$. Then ALIGN generates the collection A_x of qualified subsets, where there are four of them $\{w_1, w_4\}$, $\{w_1, w_5\}$, $\{w_2, w_4\}$, and $\{w_3, w_6\}$, whose left and right interval intersections are respectively $[4, 6]$ and $[7, 9]$, $[6, 6]$ and $[6, 6]$, $[7, 7]$ and $[7, 7]$, $[12, 14]$ and $[14, 14]$. Next for each of them, ALIGN checks the collided compact windows in W_y. For the subset $W'_x = \{w_1, w_4\}$, its collided subset W'_y contains $w_7 = \langle S, f_1, 8, 14, 11 \rangle$ and $w_8 = \langle S, f_2, 7, 13, 10 \rangle$. Then ALIGN finds qualified subsets of W'_y, where there is only one $\{w_7, w_8\}$ whose left and right interval intersections are respectively $[8, 10]$ and $[11, 13]$. Finally, ALIGN finds 81 near-duplicate passage pairs $T[i, j]$ and $S[p, q]$ where $(i, j, p, q) \in [4, 6] \times [7, 9] \times [8, 10] \times [11, 13]$ from the two subsets.

Complexity Analysis. It takes $O(k(n \log n + m \log m))$ to generate the compact windows for the two texts and $O(max(n \log n, m \log m))$ to build the segment tree. There are $O(kn)$ and $O(km)$ compact windows in W_x and W_y respectively. It takes $O(|W| \log n)$ invoking PUSHDOWN to insert all compact windows in W to the segment tree. REFINE takes $O(|C| \log n)$ time to verify the candidate node set C. In total, FINDSUBSET takes $O(|W| \log n) + O(|C| \log n) + |C|(O(|W| \log n) + O(|C| \log n))$ where $|C| = O(|W| \log n)$. Thus the time complexity of FINDSUBSET is $O(|W|^2 \log^3 n)$. In addition, the result size $|A|$ is the smaller one of $O(|C|^2) = O(|W|^2 \log^2 n)$ and $O(n^2)$. In total, the time complexity of ALIGN is $O(k^2 n^2 \log^3 n + k^2 n^2 m^2 \log^3 m)$. Note that this is the worst case time complexity where W'_x, W''_x, W'_y, and W''_y all have the largest possible size $O(kn)$ (or $O(km)$). However, in practice, the sizes of these subsets decrease exponentially. Moreover, for two texts with few or no near-duplicate passages, the size of W_x becomes very small after removing the non-collided compact windows (Line 5 in Algorithm 5).

5.4 Finding The Longest Near-Duplicate Pairs

In practice, there are many redundant near-duplicate passages where one contains another. For example, in Figure 1, all excerpts of the highlighted passages are near-duplicates, e.g., the pair "*treat people with dignity*" and "treat people with dignity and respect" and the identical pair "*word is your bond*" and "word is your bond".

Problem Definition. To avoid the redundant near-duplicate passages, we propose to generate only the longest near-duplicate passages at the same place, i.e., a near-duplicate window pair will not be reported if they are sub-windows of (i.e., fully covered by) another pair of longer near-duplicate windows, as formalized below.

Definition 5.3 (all-pair longest near-duplicate passage alignment). Given two texts T and S, a similarity threshold θ, it finds all the passage pairs $T[i_1, j_1]$ and $S[i_2, j_2]$ s.t. i) $jaccard(T[i_1, j_1], S[i_2, j_2]) \geq \theta$ and ii) $\forall\ T[i'_1, j'_1] \supseteq T[i_1, j_1]$ and $S[i'_2, j'_2] \supseteq S[i_2, j_2]$, except for $T[i'_1, j'_1]$=$T[i_1, j_1]$ and $S[i'_2, j'_2]$=$S[i_2, j_2]$, $jaccard(T[i'_1, j'_1], S[i'_2, j'_2])$<$\theta$.

Example 5.4. Consider the two texts T and S in Figure 1. Let $\theta = 0.78$. Though the jaccard similarity of the pair $T[14, 47]$ and $S[8, 39]$ is $\frac{29}{37} \geq \theta$, they will not be reported. This is because there are another longer pair of near-duplicate passages $T[13, 47]$ and $S[7, 39]$ (the highlighted ones) whose jaccard similarity $\frac{30}{38} \geq \theta$. As one can verify, this pair is the longest pair and it will be reported.

Refine Results. We still use the min-hash to estimate the jaccard similarity. We observe that among all the window pairs $T[l_x, r_x]$ and $S[l_y, r_y]$ where $l_x \in \cap_i [l_x^i, c_x^i]$, $r_x \in \cap_i [c_x^i, r_x^i]$, $l_y \in \cap_i [l_y^i, c_y^i]$, and $r_y \in \cap_i [c_y^i, r_y^i]$, the pair $T[l_x^{max}, r_x^{min}]$ and $S[l_y^{max}, r_y^{min}]$ are the longest one and dominate all the others, where $l_x^{max} = max_i(l_x^i)$, i.e., the largest one among all l_x^i, and r_x^{min}, l_y^{max}, r_y^{min} are similarly defined. Thus we only generate one longest window pair for each pair of subsets W''_x and W''_y in ALIGN (Line 13 in Algorithm 5).

Moreover, when enforcing the longest near-duplicate passage constraint in ALIGN, the computation cost can be largely reduced. This is because a large number of candidate subsets can be removed from C in Algorithm 8, REFINE, as they are dominate by the others.

Removing Dominated Candidate Subsets. Specifically, consider a node u in the segment tree ST and any of its leftmost descendant v, i.e., the left boundaries of u.range and v.range are identical. We observe that u.range fully covers v.range. Thus $W_u \subseteq W_v$, where W_u and W_v are the subsets of compact windows whose left intervals cover u.range and v.range, respectively. We show we can skip W_u even if it is a qualified subset w.r.t., the left intervals, i.e., the intersection of its left intervals are non-empty and its size is at least $\tau = \lceil k\theta \rceil$.

More specifically, if W_u is a qualified subset (w.r.t., left intervals), its superset W_v must also be a qualified subset (w.r.t., left intervals) as $|W_v| \geq |W_u| \geq \tau$ and the intersection of the left intervals in W_v covers v.range by definition, which is non-empty. In addition, as discussed earlier, only the longest pair $T[l_x^{max}, r_x^{min}]$ and $S[l_y^{max}, r_y^{min}]$ are required to report, where l_x^{max} is the maximum left boundary in the qualified subset. As u.range and v.range have the same left boundary, the maximum one among all left boundaries in W_v and W_u are also identical. The same applies to l_y^{max}. Thus any longest near-duplicate passage pair result from W_u can also be generated by W_v and it is safe to skip the subset W_u.

This can be implemented by removing the node from C if one of its leftmost descendant is also in C when processing the left intervals in Refine (Line 3 in Algorithm 6). Similarly, when processing the right intervals (Line 7 in Algorithm 6), we remove the node from C if one of its rightmost descendant is also in C.

Complexity Analysis. By removing the dominated qualified subsets, $|C| = O(n)$. Thus the worst-case time complexity of Allign becomes $O(kn^2 \log n + kn^2m^2 \log m)$.

5.5 Avoiding Short Near-Duplicate Passages

Passage Length Threshold. Short near-duplicate passages are usually meaningless. For example, any pair of common words in two texts are near-duplicate passages as their jaccard similarity is 1. To avoid short near-duplicate passages, a passage length threshold β can be enforced, where near-duplicate passage pair $T[i_1, j_1]$ and $S[i_2, j_2]$ are reported only if both $j_1 - i_1 + 1 \geq \beta$ and $j_2 - i_2 + 1 \geq \beta$. Allign can seamlessly support a passage length threshold while largely reduce the computation cost. For this purpose, when generating compact windows, divide-and-conquer stops whenever $r - l + 1 < \beta$, instead of $r < l$ in Algorithm 4.

Sentence-level Near-duplicate Passages. Alternatively, we can find the sentence-level near-duplicate passages to avoid short near-duplicates. Specifically, near-duplicate passage pair $T[i_1, j_1]$ and $S[i_2, j_2]$ are reported only if both words $T[i_1]$ and $S[i_2]$ are right after periods and both words $T[j_1]$ and $T[j_2]$ are right before periods.

To support this constraint in Allign, for each generated compact window $\langle l, r, c\rangle$, we map it to a smaller one $\langle l', r', c\rangle$, where $l' \geq l$ is the smallest position such that $T[l']$ is a word right after a period and $r' \leq r$ is the largest position such that $T[r']$ is a word right before a period. Note that the compact window is dropped if $l' > c$ or $r' < c$. The number of converted compact windows equals the number of sentences in the text. Thus the computation cost is largely reduced. A verification step can be added to remove the remaining short or non-sentence-level near-duplicate passage pairs.

The two constraints, along with the longest constraint in Section 5.4, can be combined. That is a sentence-level near-duplicate passage pair are reported only if each passage contains at least β sentences. In addition, a near-duplicate passage pair is omitted if they are fully covered by another pair of longer near-duplicate passages. Any combination of the three constraints can be seamlessly supported by Allign.

Note that seeding-and-merging with sentences as seeds [13, 30, 31] cannot address our sentence-level all-pair near-duplicate passage alignment problem. This is because they cannot detect near-duplicate passages of break-up sentences. For example, as shown in Figure 1, the one sentence "*that you work* $\cdots$ *and respect*" has low jaccard similarity with each of the three break-up sentences "That you work $\cdots$", "That your word $\cdots$", and "That you treat $\cdots$". However, the three consecutive sentences as a whole is near-duplicate to the one sentence, which Allign can detect them.

6 WORD MULTIPLICITY

In the case where there are multiple appearances of a word in a window, we need to assign a different hash value to each of its appearances. For this purpose, we associate an integer x with each word $T[c] = w$ in a window $T[i, j]$, where x is the number of occurrences of w in $T[i, c]$, and denote it as a multiplicity word w_x. The hash function $hash(T[c], T[i, j]) = hash(w_x)$ either takes a multiplicity word w_x as input or takes a window and a word in it as input and outputs the hash value of the word in the window. Note that for any $x \neq y$, the two multiplicity words $w_x \neq w_y$ and $hash(w_x) \neq hash(w_y)$ [1]. For example, the passage (A, A, B, B, B) become $(A_1, A_2, B_1, B_2, B_3)$ and the passage (A, A, A, B, B) become $(A_1, A_2, A_3, B_1, B_2)$. Their intersection is (A_1, A_2, B_1, B_2) and their union is $(A_1, A_2, A_3, B_1, B_2, B_3)$. Thus their jaccard similarity is $\frac{4}{6}$.

6.1 Multiplicity Compact Window

The hash value of the word $T[c] = w$ depends on the window $T[i, j]$ containing it, for any $1 \leq i \leq c \leq j \leq |T|$. For each possible hash value $hash(w_x)$, similar as before, we use the compact window to represent all the windows $T[i, j]$ whose min-hash is $hash(T[c], T[i, j]) = hash(w_x)$, i.e., $T[c]$ is the x-th occurrence of the word w in $T[i, j]$ and has the minimum hash value among all the words in $T[i, j]$.

For this purpose, we define a compact window as a tuple $\langle l, l_e, c, r\rangle$ of four integers. It represents/contains all the windows $T[i, j]$ where $l \leq i \leq l_e$ and $c \leq j \leq r$. Note that the second field l_e guarantees that the word $T[c] = w$ has the same multiplicity x in all the represented windows. In addition, the word $T[c]$ must also have the minimum hash value $hash(T[c], T[i, j]) = hash(w_x)$ among all the words in any window $T[i, j]$ of the compact window.

Definition 6.1 (Multiplicity Compact Window). A multiplicity compact window is a tuple $\langle T, hash, l, l_e, c, r\rangle$ where

- $1 \leq l \leq l_e \leq c \leq r \leq |T|$;
- Let x be the number of occurrence of the word $T[c] = w$ in $T[l_e, c]$. The min-hash of the window $T[i, j]$ for any $l \leq i \leq l_e$ and $c \leq j \leq r$ is the same, which is $hash(T[c], T[i, j]) = hash(w_x)$.

Example 6.2. In Figure 5, row x shows the hash values of the words with multiplicity x (if there is any). For the hash function f_1, the hash value of the first appearances of you is $f_1(T[1], T[1, 13]) = f_1(\mathsf{you}_1) = 84$. The tuple $\langle T, f_1, 1, 1, 3, 13\rangle$ is a multiplicity compact window and contains 11 windows (e.g., $T[1, 3]$, $T[1, 4]$, $T[1, 5]$) whose min-hash are all $f_1(T[3], T[1, 13]) = f_1(\mathsf{you}_2) = 10$. Note the min-hash of T is $f_1(\mathsf{you}_2) = 10$ as the word $T[3]$ appears $x = 2$ times in $T[1, 3]$ and has the minimum hash value.

6.2 Generating Multiplicity Compact Windows

Similar to the no-duplicate-word case, we develop a divide-and-conquer algorithm to generate a collection of multiplicity compact windows to represent all the windows in a text. The divide-and-conquer method first generates multiplicity compact windows for pivot words with the minimum hash value.

Specifically, as shown in Figure 4, given a window $T[l, r]$ and a hash function $hash$, let w_x be the word in $T[l, r]$ with the minimum hash value. Suppose $T[c_1], T[c_2], \cdots, T[c_q]$ are all the occurrences of the word w in the window $T[l, r]$, where $c_i < c_j$ for any $i < j$. Clearly, the min-hash of the window $T[l, r]$ is $hash(T[c_x], T[l, r]) =$

[1] We assume there is no duplicate hash values, i.e., the hash function is conflict-free. This can be implemented by replacing hashing with permutation of the words. Fisher-Yates shuffle takes $O(n)$ time and space for one permutation, where n is the total number of distinct multiplicity words. The cost is very small compared to other steps.

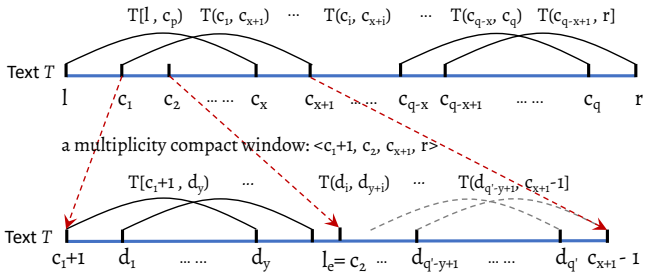


Figure 4: Divide & conquer multiplicity compact windows

$hash(w_x)$. Next, for every $i \in [0, q-x]$ (let $c_0 = l-1$), we show the tuple $\langle c_i+1, c_{i+1}, c_{i+x}, r\rangle$ is a multiplicity compact window.

Consider any window $T[u, v]$ where $u \in [c_i+1, c_{i+1}]$ and $v \in [c_{i+x}, r]$. On the one hand, as one can easily verify, the min-hash of a window must be no larger than the min-hash of any of its sub-windows. Thus the min-hash of $T[u, v] \subseteq T[l, r]$ must be no smaller than $hash(w_x)$. On the other hand, $T[c_{i+x}]$ must be the x-th occurrence of the word w in $T[u, v]$. Thus $hash(T[c_{i+x}], T[u, v]) = hash(w_x)$. Therefore, based on the definition, the min-hash of $T[u, v]$ must be $hash(T[c_{i+x}], T[u, v]) = hash(w_x)$. Based on Definition 6.1, the tuple $\langle c_i+1, c_{i+1}, c_{i+x}, r\rangle$ is a multiplicity compact window.

Divide-and-Conquer. The above multiplicity compact windows represent all the windows containing at least x times the word w. Next, consider the rest of unrepresented windows containing less than x times the word w. Clearly, for any $0 \le i \le q$, all the sub-windows in $T[c_i+1, c_{i+x}-1]$ contain less than x times the word w. We can recursively use the above process to generate multiplicity compact windows to represent all those sub-windows. However, we observe that two adjacent windows $T[c_i+1, c_{i+x}-1]$ and $T[c_{i+1}+1, c_{i+x+1}-1]$ may share common sub-windows. To avoid representing the same window multiple times, we recursively generate multiplicity compact windows to represent only those sub-window in $T[c_i+1, c_{i+x}-1]$ whose starting positions are within $[c_i+1, c_{i+1}]$. In this way, none of any sub-window is represented twice. To this end, we add another parameter to indicate the range of valid starting positions in the divide-and-conquer algorithm.

Algorithms 9 and 10 show the pseudo-code for generating multiplicity compact windows in a text. In each iteration, it takes a range $[l, r]$ and a maximum starting position l_e as input and generates multiplicity compact windows to represent all the windows $T[i, j]$ where $l \le i \le l_e$ and $i \le j \le r$. For this purpose, it first finds the word w_x with the minimum hash value in $T[l, r]$ and all of its appearances $T[c_1], \cdots, T[c_q]$ in $T[l, r]$. Clearly $q \ge x$. Then any window containing at least x times the word w must have $hash(w_x)$ as its min-hash, which is the minimum for all words in $T[l, r]$. To represent these windows, it generates a compact window $\langle c_i+1, c_{i+1}, c_{i+x}, r\rangle$ for each $i \in [0, q-x]$ where $c_0 = l-1$. Furthermore, it recursively invokes itself to represent windows in the range of $[c_i+1, c_{i+x}-1]$ and whose maximum starting position be c_{i+1}. It stops when $l_e < c_i$ at which time all the windows are represented. Initially, the range and the maximum start position are $[1, |T|]$ and $|T|$ respectively to represent all the windows in T.

Example 6.3. As shown in Figure 5. ALLIGN first finds multiplicity compact windows in the range of $[l = 1, r = 13]$ and with the maximum starting position $l_e = 13$. As the word you$_2$ has the minimum hash value, we have $x = 2$, $q = 3$, $c_0 = 0$, $c_1 = 1$, $c_2 = 3$, and $c_3 = 11$. For $i = 0$, a multiplicity compact window $\langle 1, 1, 3, 13\rangle$ is generated and the procedure is recursively invoked for the range $[1, 2]$ and the maximum position 1. For $i = 1$, a multiplicity compact window $\langle 2, 3, 11, 13\rangle$ is generated and the procedure is recursively invoked for the range $[2, 10]$ and the maximum position 3. Finally, the procedure is invoked again for the range $[4, 13]$ and the maximum position 13.

Algorithm 9: MULTIDIVCOMPACTWINDOWS(T, $hash$)

Input: T: a text; $hash$: a hash function.
Output: A: multiplicity compact windows in T.

```
build a segment tree ST with all the hash values in T;
MultiDivideConquer(ST, 1, |T|, |T|, A);
return A;
```

Algorithm 10: MULTIDIVIDECONQUER(ST, l, l_e, r, A)

Input: ST: a segment tree; $[l, r]$: a range; l_e: maximum starting position; A: multiplicity compact windows

```
if r < l then return find the minimum hash value hash(w_x)
  in T[l, r] and let T[c_1] = ··· = T[c_q] = w be all occurrences of
  the word w in T[l,r];
Let c_0 = l − 1;
foreach i = 0 to q − x do
    if l_e ≥ c_{i+1} then
        add ⟨c_i + 1, c_{i+1}, c_{i+x}, r⟩ to A;
        if x = 1 then
        MultiDivideConquer(ST, c_i + 1, c_{i+1} − 1, c_{i+x} − 1, A)
        else
        MultiDivideConquer(ST, c_i + 1, c_{i+1}, c_{i+x} − 1, A)
    else
        add ⟨c_i + 1, l_e, c_{i+x}, r⟩ to A;
        MultiDivideConquer(ST, c_i + 1, l_e, c_{i+x} − 1, A);
        return;
MultiDivideConquer(ST, c_{q−x+1} + 1, l_e, r, A);
return;
```

Text T: *you think you are the tallest in your class but you are not*

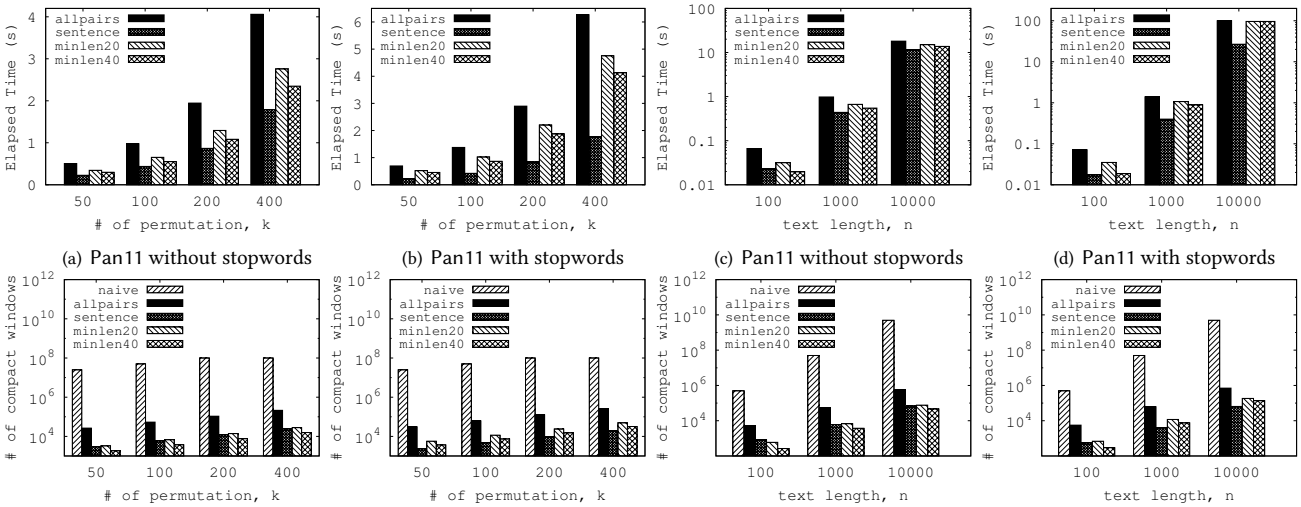
f_1	1	2	3	4	5	6	7	8	9	10	11	12	13
x = 1	84	37	84	55	40	33	20	69	45	36	84	55	56
x = 2	10		10	30							10	30	
x = 3	39		39								39		

<T, f_1, 1, 1, 3, 13> min-hash: 10

Figure 5: An example of word multiplicity

6.3 Near-Duplicate Passage Generation

For each multiplicity compact window $\langle l, l_e, c, r\rangle$, we use $[l, l_e]$ as its left interval and $[c, r]$ as its right interval. The rest remains all the same as the no-duplicate-word case for generating all near-duplicate passages. The three constraints, i.e., the longest passage constraint, the passage length constraint, and the sentence-level constraint, can be implemented in the same way as discussed earlier.



(a) Pan11 without stopwords (b) Pan11 with stopwords (c) Pan11 without stopwords (d) Pan11 with stopwords

(e) Pan11 without stopwords (f) Pan11 with stopwords (g) Pan11 without stopwords (h) Pan11 with stopwords

Figure 6: Evaluating the compact window generation.

Table 1: The settings of our parameters.

	Pan11	News
k	[50, 100, 200, 400]; default 100	[50, 100, 200, 400]; default 100
θ	[0.7, 0.8, 0.9]; default 0.8	[0.6, 0.7, 0.8]; default 0.7
β	[20, 40, 60, 80, 100]; default 40	[10, 20, 30,40,50]; default 20
n	[100, 1000, 10000]; default 1000	[2000, 4000, 6000]; default 4000

7 EXPERIMENT

In this section, we empirically evaluate Allign. We implemented all the proposed techniques using C++ and compiled the code using g++ 7.5 with -O3 optimization. All the experiments were conducted on a machine with Intel(R) Xeon(R) Gold 6212U CPU @ 2.40GHz and 62GB memory running Ubuntu 18.04LTS.

Datasets. We used two datasets Pan11[2] and News[3]. Pan11 is a plagiarism detection benchmark dataset. Each text in it is a book. There are manually created plagiarism passages in the texts. News is a news article dataset. Each text is an article on the web. We used a stopword list consisting of 130 most frequent words. The stopwords are removed from the texts if not mention otherwise.

Parameter Settings. We evaluated the impact of four parameters. The jaccard similarity threshold θ, the passage length threshold β, the number of random universal hash functions (i.e., permutations) k, and the text length n in words. The ranges and default values of the four parameters are shown in Table 1.

7.1 Evaluating Compact Window Generation

In this section, we evaluate the compact window generation by varying the number of permutations k and the text length n. Note the similarity threshold θ does not have any impact to the compact window generation. Two datasets, Pan11 with and without stopwords, were tested. We randomly drew 100 texts longer than the maximum n from the datasets and truncated the first n words to run the experiments. The average compact window generation time and the average number of compact windows generated are reported. We compared four methods. i) allpairs is the vanilla Allign method without any of the three constraints; ii) sentence is Allign with the sentence-level near-duplicate passage constraint as discussed in Section 5.5. iii) minlen20 and minlen40 are Allign with the passage length threshold constraint, where the length threshold are respectively 20 and 40. Note the longest near-duplicate passage constraint does not affect the compact window generation. Figure 6 shows the results. For reference, we show the total number of windows $\frac{n(n-1)}{2}k$ in the text using the legend naive.

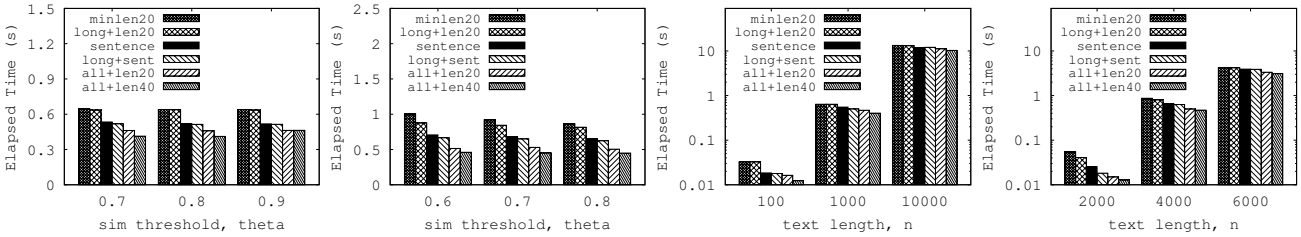
As we can see from the figure, the numbers of compact windows in all the methods were 2 to 4 orders of magnitude smaller than the total number of windows in naive. This is because the compact window is a concise representation of a large number of nearby windows. Moreover, sentence almost always achieved the best performance because the number of its compact windows is proportional to the number of sentences instead of the number of words in the text. allpairs consistently took the longest time and generated the maximum number of compact windows among all methods. This is because the passage length threshold constraint adds an early termination condition in the divid-and-conquer compact window generation algorithm. minlen40 always outperformed minlen20 as it terminates earlier with a longer length threshold.

Furthermore, we observe that the compact window generation time and the number of compact window generated both grew linearly with the number of permutations k. This is because the permutations are independent to each other in compact window generation. In addition, we observe the number of compact window generated grew linearly with the text length, while the compact window generation time grew super-linearly, especially when the stopwords were removed. This is consistent with our complexity analysis where there are $O(n)$ compact windows and the divide-and-conquer algorithm takes $O(n \log n)$ time in the no-duplicate-word case (when the stopwords were removed, it is more close to the no-duplicate-word case). It took more time and generated more compact windows when the stopwords were untouched as the divide-and-conquer algorithm were invoked more frequently.

7.2 Evaluating Near-Duplicate Generation

In this section, we evaluate the near-duplicate passage generation by varying the jaccard similarity threshold θ and the text length

[2]https://pan.webis.de/data.html
[3]https://components.one/datasets/all-the-news-2-news-articles-dataset/



(a) Pan11 dataset (b) News dataset (c) Pan11 dataset (d) News dataset

Figure 7: Evaluating near-duplicate passage generation.

Table 2: Comparison with existing text alignment methods.

methods	recall	precision	F1	parameter settings
PKWISE	0.27	0.046	0.078	$\theta = 0.15$, width = 40 words
SEF	0.425	**0.994**	0.595	same as Table 1 in [36]
ALLIGN	0.484	0.957	0.643	$\theta = 0.15$, $k = 100$, $\beta = 10$
ALLIGN + SEF	**0.514**	0.968	**0.672**	same as ALLIGN and SEF

Table 3: Comparing ALLIGN ($k = 100$, index size 798 KB) with naive enumeration and PKWISE (window width = 40 words).

θ	ALLIGN precision / recall / time	ENUM # of results / time	PKWISE index / time
0.15	0.953 / 0.984 / 3.30s	743 / 103.6s	169 KB / 0.75s
0.30	0.924 / 1.000 / 2.41s	123 / 102.4s	798 KB / 3.89s
0.45	0.923 / 1.000 / 2.38s	27 / 102.6s	2058 KB / 5.32s
0.60	0.929 / 1.000 / 2.37s	10 / 103.2s	51054 KB / 128.85s

n. Two datasets Pan11 and News were tested. For a pair of texts, the text length n was varied as described in Tabel 1 for one of them and the text length for the other one was fixed as 1000. We report the near-duplicate passage generation time (excluding the compact window generation time). Six methods were compared. i) sentence; ii) minlen20; iii) longest+sent is ALLIGN with the sentence-level and the longest near-duplicate passage constraints; iv) longest+len20 is ALLIGN with the length threshold and the longest near-duplicate passage constraints where the length threshold is 20; v) all+len20 and vi) all+len40 are ALLIGN with all the three constraints where the length thresholds are 20 and 40, respectively.

Figure 7 shows the experiment results. As we can see from the figure, longest+sent and longest+len20 slightly outperformed sentence and minlen20, respectively. The reason is because with the longest constraint, a candidate node n is removed if one of its left-most descendant is also a candidate node in the procedure REFINE. Furthermore, all+len20 and all+len40 achieved the best performance as it entails all the three constraints. In addition, with the increase of the threshold, the performance of all the methods slightly improved. This is because the number of qualified subsets decreases with the increase of the similarity threshold. The same can be observed when varying the text length n. In addition, with the increase of the text length n, the performance of all the methods decreased super-linearly. This is because the number of compact windows increases with n. Moreover, with a large n, there are more multiplicity words.

7.3 Comparison with State-of-the-art Methods

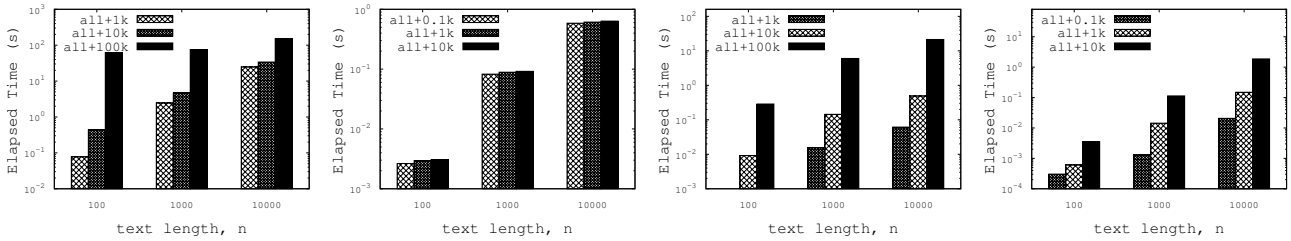
We compared ALLIGN with the best approach SEF [36] to the text alignment subtask at the plagiarism detection competition of PAN 2014 [31]. In the text alignment subtask[4], given a pair of long texts, it is required to find all pairs of contiguous maximal length reused passages. There are four kinds of reused passages. Among them, the summary obfuscation is the most difficult one to detect. A summary obfuscation in a text is a human-written summary of a passage in another text. More details about how the summary obfuscation is generated can be found in [32]. SEF is a sophisticated seeding-extension-filtering approach with ten knobs (i.e., parameters) as detailed in Table 1 in [36]. It enumerates every sentence pair in two texts and uses the sentence pair as a seed match if their similarity is above the given threshold, which is a tunable parameter. We used the open source implementation of SEF[5]. In addition, we also evaluated a fixed-width window approach PKWISE. It finds all the fixed-width window pair whose jaccard similarity is above a given threshold. To evaluate PKWISE for the text alignment task, we replace the seeding step in SEF with PKWISE, i.e., instead of using the similar sentence pairs as the seeds, we use the similar fixed-width window pairs found by PKWISE as the seeds. The extension and filtering steps stay the same as SEF. We report the precision, recall and F1-score of the three approaches on the text alignment task using the summary obfuscation dataset in the PAN 2014 benchmark. The dataset contains 238 pairs of source and suspicious texts. The average lengths of the source texts and the suspicious texts are respectively 5108 and 6715 words. The precision and recall capture the percentage of common characters between the detected plagiarism passage pairs and the ground truth plagiarism passage pairs. The formal definition is detailed in [33].

The results are shown in Table 2. As we can see, our approach ALLIGN can significantly improve the recall of the state-of-the-art approach SEF while achieving the similar precision. Overall, we largely improved the f1-score of SEF from 0.595 to 0.643. The reason is because there are few similar sentences in the summary obfuscation while the entire summary and the original passage share many common words and their jaccard similarity is not extremely low. PKWISE had the worst precision and recall. This is because for a large window width, PKWISE will miss many seed matches and lead a low recall while for a small window width, it will introduce many noise matches and lead to a low precision. The last row in Table 2 presents the performance of uniting the results of SEF and ALLIGN. It shows how much boost ALLIGN can bring to the state-of-the-art approach SEF. As we can see, ALLIGN can significantly boost the recall from 0.425 to 0.514 and the f1-score from 0.595 to 0.672.

Next, we compare ALLIGN with existing approaches against our all-pair near-duplicate passage detection problem. For this purpose, we implemented a naive enumeration method ENUM to address the sentence-level all-pair near-duplicate passage detection problem as defined in Section 5.5. It enumerates all the $\frac{n(n+1)}{2} \times \frac{m(m+1)}{2}$ pairs of contiguous sentences in two texts with m and n sentences and reports those whose jaccard similarities are above the given

[4]https://pan.webis.de/clef14/pan14-web/text-alignment.html

[5]https://github.com/CubasMike/plagiarism_detection_pan2015



(a) Pan11 (window generation time) (b) News (window generation time) (c) Pan11 (result generation time) (d) News (result generation time)

Figure 8: Evaluating scalability.

threshold θ. We used a pair of texts in the PAN 2014 benchmark with 261 words in 14 sentences and 1771 words in 130 sentences, respectively. Table 3 shows the results. We used the results (i.e., all near-duplicate passage pairs) of the enumeration method ENUM as the ground truth and report the precision and recall of ALLIGN. Note the precision of PKWISE is 1 and the recall is close to 0. In addition, when $\theta = 0.15$, the precision and recall of ALLIGN for $k = 300$ were respectively 0.947 and 1.00 while for $k = 1000$ they were respectively 0.943 and 1.00. We can see from the table, ALLIGN significantly outperformed the simple enumeration and the local similarity search method PKWISE even for short text pairs with only a few sentences with regard to total runtime. Moreover, ALLIGN found almost all the near-duplicate passage pairs even for a relatively small $k = 100$. The index size of ALLIGN is small compared with that of PKWISE. This is because ALLIGN uses compact windows to represent a large number of nearby and overlapping windows.

7.4 Scalability

In this section, we evaluate the scalability by varying the two text lengths n in [100, 1000, 10000] and m in [1k, 10k, 100k] for Pan11 and n in [100, 1000, 10000] and m in [0.1k, 1k, 10k] for News. We report the average compact window generation time and the average near-duplicate passage generation time. We evaluated our best method all+len20. Figure 8 shows the results. The numbers in the suffix of the legends denote the text lengths m while the x-axises are the other text lengths n. As we can see from the figure, with the growth of the text lengths n and m, the compact window generation time and the near-duplicate passage generation time grew super-linearly. This is consistent with our complexity analysis where the compact window generation time and the number of compact windows were respectively $O(n \log n)$ and $O(n)$ and with larger n and m there are more multiplicity words and more compact windows are generated.

8 RELATED WORK

Plagiarism Detecting. There are many work in plagiarism detecting due to its importance in many applications, such as machine translation [12], author profiling for marketing applications [35], spam detection [45], and law enforcement [35]. There are two kinds of plagiarism detection, the extrinsic and intrinsic plagiarism detections. The former aims to find suspicious plagiarism documents to a collection of reference documents. The latter aims at finding plagiarism within an input document. It exclusively analyzes the document itself and does not perform comparisons to documents in a reference collection. Plagiarism detection methods typically first locate the parts of documents with high enough similarities as potential plagiarism and then substantiate the suspicion through more in-depth analysis [18, 37, 43]. A popular method for external plagiarism is based on hashing or fingerprinting [6, 25]. These methods produce a few fingerprints from the documents. Near-duplicate passages are assumed to have similar fingerprints. Based on whether the fingerprints overlap with each other, these work can be classified as overlapping and non-overlapping methods. 0 mod p [25], super-shingles [6], Winnowing [39], Hailstorm [15] belong to the former class, while hash breaking [4], DCT fingerprinting [40], and qSign [19] are within the latter class. Textual information retrieval can also be used to detect external plagiarisms [2, 17]. They can be used to find highly similar "seeds" in the documents. Then one can either extend the seeds on the two ends or merging nearby seeds to generate the candidate plagiarism for in-depth analysis [28, 29].

Similarity Search. Similarity query processing have been extensively studied. Popular methods include the partition-based methods [10, 23, 34], the prefix-filtering methods [3, 7, 9, 46, 49], and the heap-based methods [11, 21]. Specifically, Xiao et al. [49] proposed to use set similarity join for near-duplicate text deduplication. Another line of work is approximate algorithms for similarity search and join [1, 26, 27, 38, 41, 42]. Among them, locality sensitive hashing (LSH) [14, 16] is the most widely used ones. Min-hash is one of the LSH signature for the jaccard similarity. However, these methods can only find out whether two entire documents are similar and do not produce near-duplicate passages. Approximate entity extraction [22, 48] can find similar entities in a document. However, it still does not produce near-duplicate passages. The most relevant work to us is local similarity search [47]. It uses a fixed-width sliding window to find all similar passages in another document. However, it only produces fixed-length near-duplicate passages.

9 CONCLUSION

In this paper, we study the all-pair near-duplicate passage detection problem in a pair of long texts. Due to the high computation cost of this problem, all existing work resort to heuristics such as seeding and merging and fixed-width window. This paper proposes a min-hash based algorithm ALLIGN to find all the passage pairs sharing enough common min-hash. To avoid the large number of passages and in the texts, we propose to use $O(n)$ compact windows to concisely represent all the $O(n^2)$ passages in a text with n words. An efficient $O(n \log n)$ divide-and-conquer algorithm is developed to generate the compact windows from a text. Furthermore, ALLIGN avoids enumerating the enormous number of passage pairs by generating all the near-duplicate passage pairs directly from the matched compact windows of two texts. Three constraints that make the results more relevant and ALLIGN more efficient can be supported by ALLIGN. They produce only the longest near-duplicate passage at the same place, near-duplicate passages that are long enough, and sentence-level near-duplicate passages, respectively. We also discuss how to deal with word multiplicity with ALLIGN. Experimental results show ALLIGN outperforms existing methods.

REFERENCES

[1] T. D. Ahle, R. Pagh, I. P. Razenshteyn, and F. Silvestri. On the complexity of inner product similarity join. In *PODS*, pages 151–164, 2016.

[2] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[4] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *SIGMOD*, pages 398–409, 1995.

[5] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES*, pages 21–29. IEEE, 1997.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Networks*, 29(8-13):1157–1166, 1997.

[7] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

[8] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

[9] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*, pages 673–684, 2014.

[10] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *Proc. VLDB Endow.*, 9(4):360–371, 2015.

[11] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, pages 905–920, 2018.

[12] A. Eyecioglu and B. Keller. Twitter paraphrase identification with simple overlap features and SVMs. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 64–69. Association for Computational Linguistics, June 2015.

[13] T. Foltýnek, N. Meuschke, and B. Gipp. Academic plagiarism detection: A systematic literature review. *ACM Comput. Surv.*, 52(6):112:1–112:42, 2020.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[15] O. A. Hamid, B. Behzadi, S. Christoph, and M. R. Henzinger. Detecting the origin of text segments efficiently. In *WWW*, pages 61–70. ACM, 2009.

[16] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of Computing*, 8(1):321–350, 2012.

[17] T. C. Hoad and J. Zobel. Methods for identifying versioned and plagiarized documents. *J. Assoc. Inf. Sci. Technol.*, 54(3):203–215, 2003.

[18] V. K and D. Gupta. Detection of idea plagiarism using syntaxâĂŞsemantic concept extractions with genetic algorithm. volume 73, pages 11 – 26, 2017.

[19] J. W. Kim, K. S. Candan, and J. Tatemura. Efficient overlap and content reuse detection in blogs and online news articles. In *WWW*, pages 81–90. ACM, 2009.

[20] S. Kliff. 2016 (accessed August 23, 2020). https://www.vox.com/2016/7/19/12227658/melania-plagarism-trump-excuses.

[21] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[22] G. Li, D. Deng, and J. Feng. Faerie: efficient filtering algorithms for approximate dictionary-based entity extraction. In *SIGMOD*, pages 529–540. ACM, 2011.

[23] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[24] P. Li, A. B. Owen, and C. Zhang. One permutation hashing. In P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *NIPS*, pages 3122–3130, 2012.

[25] U. Manber. Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference*, pages 1–10. USENIX Association, 1994.

[26] B. Neyshabur and N. Srebro. On symmetric and asymmetric lshs for inner product search. In *ICML*, pages 1926–1934, 2015.

[27] R. Pagh. Locality-sensitive hashing without false negatives. In *SODA*, pages 1–9, 2016.

[28] M. Potthast, A. Barrón-Cedeño, A. Eiselt, B. Stein, and P. Rosso. Overview of the 2nd international competition on plagiarism detection. In *CLEF 2010 LABs and Workshops, Notebook Papers*, volume 1176 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[29] M. Potthast, A. Eiselt, A. Barrón-Cedeño, B. Stein, and P. Rosso. Overview of the 3rd international competition on plagiarism detection. In *CLEF 2011 Labs and Workshop, Notebook Papers*, volume 1177 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[30] M. Potthast, T. Gollub, M. Hagen, J. Kiesel, M. Michel, A. Oberländer, M. Tippmann, A. Barrón-Cedeño, P. Gupta, P. Rosso, and B. Stein. Overview of the 4th international competition on plagiarism detection. In *CLEF 2012 Evaluation Labs and Workshop, Online Working Notes*, volume 1178 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[31] M. Potthast, M. Hagen, A. Beyer, M. Busse, M. Tippmann, P. Rosso, and B. Stein. Overview of the 6th international competition on plagiarism detection. In *Working Notes for CLEF 2014 Conference*, volume 1180 of *CEUR Workshop Proceedings*, pages 845–876. CEUR-WS.org, 2014.

[32] M. Potthast, M. Hagen, T. Gollub, M. Tippmann, J. Kiesel, P. Rosso, E. Stamatatos, and B. Stein. Overview of the 5th international competition on plagiarism detection. In *Working Notes for CLEF 2013 Conference*, volume 1179 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[33] M. Potthast, B. Stein, A. Barrón-Cedeño, and P. Rosso. An evaluation framework for plagiarism detection. In *COLING*, pages 997–1005. Chinese Information Processing Society of China, 2010.

[34] J. Qin and C. Xiao. Pigeonring: A principle for faster thresholded similarity search. *Proc. VLDB Endow.*, 12(1):28–42, 2018.

[35] P. Rosso, F. Rangel, M. Potthast, E. Stamatatos, and B. Stein. Overview of pan'16. In *International Conference of the Cross-Language Evaluation Forum for European Languages*, pages 332–350, 2016.

[36] M. A. Sanchez-Perez, A. F. Gelbukh, and G. Sidorov. Adaptive algorithm for plagiarism detection: The best-performing approach at PAN 2014 text alignment competition. In *6th International Conference of the CLEF Association*, volume 9283 of *Lecture Notes in Computer Science*, pages 402–413. Springer, 2015.

[37] F. Sánchez-Vega, E. Villatoro-Tello, M. Montes-y-Gómez, L. V. Pineda, and P. Rosso. Determining and characterizing the reused text for plagiarism detection. *Expert Syst. Appl.*, 40(5):1804–1813, 2013.

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

[39] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *SIGMOD*, pages 76–85. ACM, 2003.

[40] J. Seo and W. B. Croft. Local text reuse detection. In *SIGIR*, pages 571–578. ACM, 2008.

[41] A. Shrivastava and P. Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *NIPS*, pages 2321–2329, 2014.

[42] A. Shrivastava and P. Li. Asymmetric minwise hashing for indexing binary inner products and set containment. In *WWW*, pages 981–991, 2015.

[43] B. Stein, S. M. zu Eissen, and M. Potthast. Strategies for retrieving plagiarized documents. In *SIGIR*, pages 825–826. ACM, 2007.

[44] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *SIGIR*, pages 563–570. ACM, 2008.

[45] A. K. Uysal and S. Gunal. Text classification using genetic algorithm oriented latent semantic features. volume 41, pages 5938 – 5947, 2014.

[46] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.

[47] P. Wang, C. Xiao, J. Qin, W. Wang, X. Zhang, and Y. Ishikawa. Local similarity search for unstructured text. In *SIGMOD*, pages 1991–2005. ACM, 2016.

[48] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *SIGMOD*, pages 759–770. ACM, 2009.

[49] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140. ACM, 2008.