Algorithm Engineering for All-Pairs Suffix-Prefix Matching*

Jihyuk Lim[1] and Kunsoo Park†[2]

1 **Department of Computer Science and Engineering, Seoul National University, Seoul, Korea**
jhlim@theory.snu.ac.kr

2 **Department of Computer Science and Engineering, Seoul National University, Seoul, Korea**
kpark@theory.snu.ac.kr

Abstract

All-pairs suffix-prefix matching is an important part of DNA sequence assembly where it is the most time-consuming part of the whole assembly. Although there are algorithms for all-pairs suffix-prefix matching which are optimal in the asymptotic time complexity, they are slower than SOF and Readjoiner which are state-of-the-art algorithms used in practice. In this paper we present an algorithm for all-pairs suffix-prefix matching that uses a simple data structure for storing input strings and advanced algorithmic techniques for matching, which together lead to fast running time in practice. Our algorithm is 14 times faster than SOF and 18 times faster than Readjoiner on average in real datasets and random datasets.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases all-pairs suffix-prefix matching, algorithm engineering, DNA sequence assembly

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.14

1 Introduction

The problem of all-pairs suffix-prefix (APSP) matching is defined as follows: Given a collection of k strings $S_1, S_2, \ldots, S_k$, find the longest suffix of S_i which is a prefix of S_j for all pairs S_i and S_j. Let N be the sum of lengths of the input strings $S_1, S_2, \ldots, S_k$. All-pairs suffix-prefix matching is an important part of DNA sequence assembly where it is the most time-consuming part of the whole assembly process. In DNA sequence assembly, a parameter om (for overlap minimum) is given for APSP matching where we want to find the longest overlap of S_i and S_j whose length is at least om. The output of APSP matching can be stored in a $k \times k$ matrix Ov, where $Ov[i, j]$ is the length of the longest suffix of S_i that is a prefix of S_j. Alternatively, the output can be a list of three integers $(i, j, Ov[i, j])$ such that $Ov[i, j] \geq om$ as a compact representation.

All-pairs suffix-prefix matching has been studied in the fields of stringology and bioinformatics. In general, a solution for APSP matching consists of two phases: the first phase is to build a data structure which represents all prefixes of the input strings, and the second

* This research was supported by Collaborative Genome Program for Fostering New Post-Genome industry through the National Research Foundation of Korea(NRF) funded by the Ministry of Science ICT and Future Planning (No. NRF-2014M3C9A3063541).

† Corresponding author.

matching phase is to search the data structure to find occurrences of suffixes of each input string (or equivalently, one can build a data structure representing all suffixes of the input strings, and search it for prefixes of each string). Gusfield et al. [8] proposed a novel algorithm of optimal $O(N + k^2)$ time for APSP matching by building a generalized suffix tree for the input strings. Ohlebusch and Gog [16] gave another $O(N + k^2)$-time algorithm by building an enhanced suffix array [1, 14] for the input strings, which improves upon Gusfield et al.'s in running time and space. Tustumi et al. [19] further improved the running time and space of Ohlebusch and Gog's algorithm. Louza et al. [13] presented a parallel algorithm for APSP matching which is based on Tustumi et al.'s. In bioinformatics too, many algorithms have been proposed for APSP matching [5, 6, 11, 15, 9, 10, 18, 17], and Readjoiner [6] and SOF [9] are state-of-the-art algorithms which show best performances in practice. Although the algorithms in [8, 16, 19] are optimal in the asymptotic worst-case time complexity, Readjoiner and SOF are faster than these algorithms in practice. Hence there is a mismatch between theoretical results and practice. A main reason for the mismatch is that the generalized suffix tree [8, 7] and even the enhanced suffix array [1, 14] are heavy machineries (though they provide powerful functionalities) and so the constants hidden in the asymptotic notations are quite big. On the other hand, SOF uses a simple but effective data structure called the compact prefix tree (also known as compact trie) for the input strings and its matching phase uses quite naive algorithmic techniques. As another approach, we can build the Aho-Corasick automaton [2] for the input strings, and solve APSP matching by searching the automaton for each input string. We implemented this approach, but the Aho-Corasick automaton is another piece of heavy machinery and just building it (without the matching phase) takes more time than the whole SOF.

In this paper we propose a fast algorithm for APSP matching. We first build a compact prefix tree for the input strings, but in the matching phase we need more advanced techniques because the only functionality that the compact prefix tree provides is to check whether a given string is a prefix of the input strings or not. We divide the matching phase into three cases depending on the lengths of suffixes of an input string, and in each case we use an appropriate algorithmic technique which finds efficiently occurrences of the suffixes of an input string corresponding to the case. We did experiments to compare our algorithm against SOF and Readjoiner with real datasets and random datasets. In the experiments our algorithm is 14 times faster than SOF and 18 times faster than Readjoiner on average. We also obtain reasonable scalability with a parallel implementation of our algorithm.

2 New algorithm for APSP matching

Let S be a string of length n over an alphabet Σ. We denote the length of S by $|S|$. The i-th character of S is denoted by $S[i]$ $(1 \leq i \leq |S|)$, and a substring $S[i]S[i+1]...S[j]$ by $S[i..j]$. A substring $S[1..i]$ for $1 \leq i \leq n$ is called a prefix of S and a substring $S[i..n]$ for $1 \leq i \leq n$ is called a suffix of S. For strings A and B, we use $A \prec B$ to denote that A is lexicographically smaller than B.

We describe the compact prefix tree CT of k input strings defined in [9]. The compact prefix tree is basically a compact trie, i.e., there is one leaf corresponding to each input string and every internal node has at least two children. An array *sorted* stores the lexicographic ordering of the input strings such that $S_{sorted[1]} \preceq S_{sorted[2]} \preceq ... \preceq S_{sorted[k]}$. The leaves of CT are in the lexicographic order, and each node v of CT has an interval $[a, b]$ such that $S_{sorted[a]}, \ldots, S_{sorted[b]}$ are the leaves in the subtree rooted at v. The compact prefix tree provides a function Find(s), which returns the node v nearest to the root such that s is a

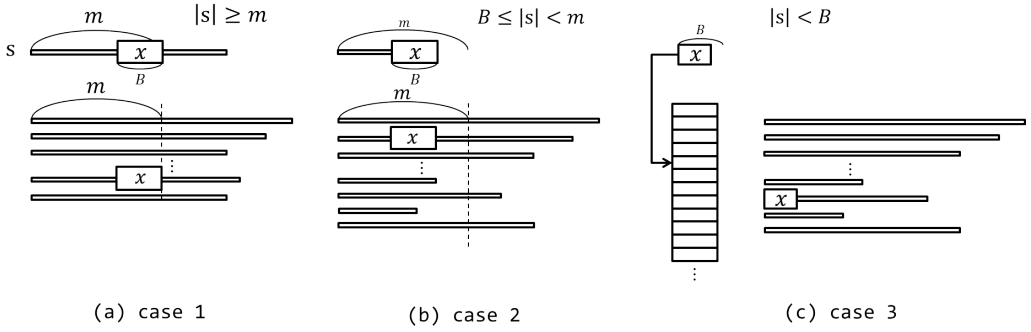

Figure 1 Three cases of the matching step.

prefix of the string on the path from the root to v. If such a node does not exist, Find(s) returns NULL. For notational convenience, let *isort* be the inverse function of *sorted*, i.e., $isort[j] = c$ if $j = sorted[c]$.

We first describe an overview of our algorithm for APSP matching. Our algorithm consists of three steps: preprocessing, matching, and output steps. Our algorithm uses two integers m and B as parameters ($m \geq B$).

- In the preprocessing step we construct a compact prefix tree CT for k input strings as in [9]. In addition, we build auxiliary data structures which will be used in the matching step.
- In the matching step we consider each input string S_i separately and do the following. For each suffix s of S_i, we find the interval $[a, b]$ such that $S_{sorted[a]}, \ldots, S_{sorted[b]}$ have s as their prefixes. If the interval $[a, b]$ is not empty, we insert $(a, b, |s|)$ into $oList[i]$. But if we find the interval $[a, b]$ by calling Find(s) for every suffix s of S_i, it will take too much time. We reduce the number of calls to Find by dividing the suffixes of S_i into three cases and using different techniques for the cases. Figure 1 illustrates three cases of the matching step.
  - In case 1, we consider each suffix s of S_i such that $|s| \geq m$. If $s[m - B + 1..m]$ appears in $S_j[m - B + 1..m]$ for some j (see Figure 1 (a)), we call Find(s); otherwise, it is guaranteed that s does not appear as a prefix of input strings and so we don't need to call Find(s).
  - In case 2, we consider each suffix s such that $B \leq |s| < m$. If $s[|s| - B + 1..|s|]$ appears in $S_j[|s| - B + 1..|s|]$ for some j (see Figure 1 (b)), we call Find(s); otherwise, s does not appear as a prefix of input strings.
  - In case 3, we consider suffixes s such that $|s| < B$. For this case we precompute Find(s') for every string s' of length less than B which appears as a prefix of input strings, and we store them in a table *Bprefix*. Hence there are no calls to Find during the matching step.
- In the output step we find the longest overlap of S_i and S_j for every j, which is the largest l such that (a, b, l) is in $oList[i]$ and interval $[a, b]$ contains $isort[j]$.

2.1 Preprocessing step

In the preprocessing step, we build data structures: a compact prefix tree CT, *sorted*, and auxiliary data structures *qrm*, *qList*, and *Bprefix*.

We construct a compact prefix tree CT by inserting input strings one by one into the tree while maintaining the lexicographic order of characters for children of each internal node. At the end of the insertions, then, the input strings appear in the leaves of CT in the lexicographic order. Hence *sorted* and the interval $[a, b]$ of each node can be obtained by traversing the tree.

The values of two parameters m and B are set as follows. Let m_s be the length of a shortest string among k input strings. We define m as follows.

$$m = \begin{cases} m_s & \text{if } \frac{N}{c_1 k} \leq m_s \leq \frac{N}{c_2 k} \\ \frac{N}{c_1 k} & \text{if } m_s < \frac{N}{c_1 k} \\ \frac{N}{c_2 k} & \text{if } m_s > \frac{N}{c_2 k}, \end{cases} \tag{1}$$

for some constants c_1 and c_2. We define $B = \min(\log_{|\Sigma|} 2mk, m)$, and a string of length B will be called a *block*.

We will use an auxiliary data structure qrm for case 1 in Figure 1. Let $m' = \max(m, om)$, and let k' be the number of input strings whose length is at least m'. Let $\mathcal{S}' = \{S'_1, S'_2, \ldots, S'_{k'}\}$ be the collection of the length-m prefixes of input strings whose length is at least m'. Hence $\mathcal{S}'$ looks like a $k' \times m$ matrix as in Figure 1 (a). For every string x of length B, $qrm[f(x)]$ is the rightmost occurrence of x in $\mathcal{S}'$, i.e.,

$$qrm[f(x)] = \begin{cases} \max\{q \mid x = S'_i[q - B + 1..q] \text{ for } 1 \leq i \leq k'\} & \text{if } x \text{ appear in } \mathcal{S}' \\ B - 1 & \text{otherwise,} \end{cases} \tag{2}$$

where $f(x)$ is a function mapping a string x to an integer used as an index of the qrm table, i.e.,

$$f(x) = \sum_{i=1}^{|x|} rank(x[i])|\Sigma|^{i-1}, \tag{3}$$

where $rank(c)$ is a function mapping a character c to a lexicographic order of c within the range $[0, \Sigma - 1]$. To compute the values of qrm, we scan all blocks (i.e., all substrings of length B) in $\mathcal{S}'$. The table qrm is initially set to $B - 1$. In each position $q = B, B + 1, \ldots, m$, we consider k' blocks $x_i = S'_i[q - B + 1..q]$ for $1 \leq i \leq k'$ and set $qrm[f(x_i)]$ to q.

We will use $qList$ for case 2 in Figure 1. Let $B' = \max(B, om)$ and let k'' be the number of input strings whose length is at least B'. Let $\mathcal{S}'' = \{S''_1, S''_2, \ldots, S''_{k''}\}$ be the collection of input strings whose length is at least B'. For the last block x of S''_j for every $1 \leq j \leq k''$, $qList[f(x)]$ is defined as a list of all distinct positions q such that $S''_i[q - B + 1..q] = x$ for $1 \leq i \leq k''$ and $B' \leq q \leq m$.

If $om \geq m$, we do not compute $qList$ because case 2 finds overlaps whose length is less than m. If $om < m$ we compute $qList$ by scanning all blocks in $\mathcal{S}''$ as follows. We define a temporary table T that indicates whether a block x appears at the end of some input string. That is, if $|S_j| \geq B$ and $x = S_j[|S_j| - B + 1..|S_j|]$, $T[f(x)] = 1$; otherwise, $T[f(x)] = 0$. In each position q, we consider k'' blocks $x_i = S''_i[q - B + 1..q]$ for $1 \leq i \leq k''$. If $T[f(x_i)] = 1$ and q is not in $qList[f(x_i)]$ (i.e., q is not at the front of $qList[f(x_i)]$), we insert q into $qList[f(x_i)]$.

We will use *Bprefix* for case 3 in Figure 1. Consider a string s such that $|s| < B$. If s is a prefix of some input string (i.e., s appears as a prefix in CT), $Bprefix[f'(s)]$ is a pointer to the node v nearest to the root of CT such that s is a prefix of the string on the path from

the root to v, where $f'(s)$ is a function mapping a string s to an integer:

$$f'(s) = \sum_{i=1}^{|s|-1} |\Sigma|^i + f(s). \tag{4}$$

Note that $f'(s)$ maps a string s into an integer within the range $[0, \sum_{i=1}^{B-1} |\Sigma|^i - 1]$. If s is not a prefix of the input strings, $Bprefix[f'(x)]$ is set to NULL.

If $om \geq B$, we do not compute *Bprefix* because case 3 finds overlaps whose length is less than B. If $om < B$, we compute *Bprefix* as follows. Initially, all entries of *Bprefix* are set to NULL. We traverse CT for all character depths (i.e., number of characters on the path from the root) less than B, and set $Bprefix[f'(s)]$ for string s that appears as a prefix in CT.

The space complexity of our algorithm is bounded by the memory space used by k input strings and the data structures CT, *sorted*, *qrm*, *qList*, and *Bprefix*. The input uses $N \log |\Sigma|$ bits (i.e. $O(N)$ space), and the data structures use $O(km)$ space, which is $O(N)$ because $km = \Theta(N)$.

2.2 Matching step

In the matching step we consider each input string S_i separately and we need to find $v = \text{Find}(s)$ for every suffix s of S_i. If v is not NULL, we insert $(a, b, |s|)$ into $oList[i]$, where $[a, b]$ is the interval of v. To reduce the number of calls to Find, we have the three cases in Figure 1.

In case 1, we consider suffixes s of S_i such that $|s| \geq m'$ from longest to shortest. Let p be the start position of a current suffix s to be considered. Initially, $p = 1$.

- If $qrm[f(x)]$ is not m, the current suffix s cannot appear in CT as a prefix and so we don't need to call Find(s). Moreover, suffixes of S_i starting at positions $p+1, p+2, \ldots, m - qrm[f(x)] - 1$ cannot appear in CT as prefixes. Hence p is updated to $p + m - qrm[f(x)]$.
- If $qrm[f(x)]$ is m, we make a call Find(s). If Find returns a node v, we insert $(a, b, |s|)$ into $oList[i]$, where $[a, b]$ is the interval of v. If Find returns NULL, we do nothing. Finally, we increase p by 1.

The preprocessing and matching steps of case 1 are essentially the same as those in Wu and Manber's algorithm [20], which is a Boyer-Moore type algorithm [3, 4, 12], but cases 2 and 3 are different from Wu and Manber's.

In case 2, we consider suffixes s of S_i such that $B' \leq |s| < m'$. If $om \geq m$ (i.e., $m' = \max(m, om) = om$), we skip case 2. In case 2, therefore, $m' = m$. Note that the last blocks of the suffixes considered in this case are $x = S_i[|S_i| - B + 1..|S_i|]$. Since a position q in $qList[f(x)]$ means that x appears at (ending) position q in one of the strings in $\mathcal{S}''$, the length-q suffix s of S_i (i.e. $s = S_i[|S_i| - q + 1..|S_i|]$) may appear in CT. Hence, we make a call Find(s) for every position q in $qList[f(x)]$.

In case 3, we consider suffixes s of S_i such that $|s| < B'$. If $om \geq B$ (i.e., $B' = \max(B, om) = om$), we skip case 3. In case 3, therefore, $B' = B$. For every suffix s of S_i such that $om \leq |s| < B$, we look up $Bprefix[f'(s)]$, which already contains the result of Find(s).

The pseudocode of the matching step is shown in Algorithm 1.

2.3 Output step

In the output step we find the longest suffix of S_i that is a prefix of S_j for every j, whose length is the largest l such that tuple (a, b, l) is in $oList[i]$ and interval $[a, b]$ contains $isort[j]$. (We assume that the output of APSP matching is a list of three integers (i, j, l) because it

Algorithm 1 Fast algorithm for APSP matching

```
1: procedure FastAPSP({S_1, S_2, ..., S_k}, om)
2:     Precompute m, B, m', B', CT, sorted, qrm, qList and Bprefix
3:     for i ← 1 to k do                                          ▷ Matching step
4:         p ← 1                                                  ▷ Case 1
5:         while p ≤ |S_i| − m' + 1 do
6:             x ← S_i[p + m − B..p + m − 1]
7:             if qrm[f(x)] = m then
8:                 v ← Find(S_i[p..|S_i|])
9:                 if v ≠ NULL then
10:                    oList[i].insert(v.interval, |S_i| − p + 1)
11:                p ← p + 1
12:            p ← p + m − qrm[f(x)]
13:        if |S_i| ≥ B' then                                     ▷ Case 2
14:            x ← S_i[|S_i| − B + 1..|S_i|]
15:            for each q in qList[f(x)] do
16:                v ← Find(S_i[|S_i| − q + 1..|S_i|])
17:                if v ≠ NULL then
18:                    oList[i].insert(v.interval, q)
19:        for p ← max(|S_i| − B + 2, 1) to |S_i| − om + 1 do     ▷ Case 3
20:            x ← S_i[p..|S_i|]
21:            v ← Prefix[f'(x)]
22:            if v ≠ NULL then
23:                oList[i].insert(v.interval, |S_i| − p + 1)
24:        Perform output step for oList[i]
```

is a more compact representation in DNA sequence assembly.) In other words, for every $1 \leq c \leq k$ we want to find the largest l such that (a, b, l) is in $oList[i]$ and interval $[a, b]$ contains c. Then l is the length of the largest overlap of S_i and $S_{sorted[c]}$ and thus we output $(i, sorted[c], l)$.

Imagine that intervals $[a, b]$ in $oList[i]$ are on the x-axis. We scan the intervals from $c = 1$ to k, and maintain the values of l in tuples (a, b, l) such that interval $[a, b]$ contains the current c in a max-heap. Then for every current c, we output $(i, sorted[c],$ max value in max-heap). This process can be implemented as follows. We first make an array tA of (a, l)'s and an array tB of (b, l)'s from tuples (a, b, l) in $oList[i]$. We sort tA in non-decreasing order of a's and tB in non-decreasing order of b's. Finally we increase c from 1 to k, and if c hits a of (a, l), we insert l into the max-heap, and if c hits b of (b, l'), we delete l' from the max-heap. Then $(i, sorted[c],$ max value in max-heap) for every current c is a correct output.

Since the number of tuples in $oList[i]$ is at most $|S_i|$, the space usage of $oList[i]$, tA, and tB is $O(|S_i|)$ (thus $O(N)$), and this space for $oList$ can be reused for every $1 \leq i \leq k$.

2.4 Implementation options

The implementation of our algorithm provides several options: *overlap minimum*, *output*, and *parallel* options. The *overlap minimum* option is given by $-om\ i$, where i is the value of overlap minimum om. The *output* option receives 1, 2, or 3 as a parameter. In the case of 1, the program provides the matrix Ov as output. In the case of 2, the program gives the list of

Table 1 Real datasets used in experiments.

	clementina	sinensis	trifoliata	C. elegans	Atta
N	104640576	154995828	46648250	167035020	315387616
k	118365	208909	62344	334465	2835
avg length	884.05	741.93	748.24	499.41	111247.84
m_s	18	13	89	7	1929

three integers $(i, j, Ov[i, j])$. In the case of 3, it gives the list of all overlaps for each pair (not only the longest overlap) like Readjoiner [6] and SOF [9] with $-o$ 2. Our program with option 3 presents all the overlaps by outputting $(i, sorted[a], l), (i, sorted[a+1], l), \ldots, (i, sorted[b], l)$ from each tuple (a, b, l) in $oList[i]$ instead of running the output step. Given the number p of threads as a parameter for the *parallel* option, our program is executed in parallel. In the preprocessing step, qrm and $qList$ are computed in parallel. The matching step is also executed in parallel by p threads.

3 Experiments

Tustumi et al. [19] and Louza et al. [13] compared only the matching phases of their optimal algorithms for APSP matching against SOF and Readjoiner, not accounting for the time to build the data structures to store input strings, and SOF and Readjoiner are in general faster than their algorithms. (If the time to build the data structures is included, the gap would be greater.) Among practical algorithms [5, 6, 11, 15, 9, 10, 18, 17] for APSP matching, SOF and Readjoiner show best performances. Therefore, we compared our algorithm[1] with SOF and Readjoiner. Our algorithm and SOF were compiled with g++ (v. 4.9.2) with the `-O3` optimization flag. Readjoiner (version 1.2) was compiled using the provided Makefile with "64bit=yes assert=no amalgamation=yes threads=yes". For parallel experiments, we used the OpenMP library. All experiments were conducted on a computer with Intel Xeon X5672 CPU, which has 8 cores, 32 GB RAM, and the Linux debian 3.2.0-4-amd64 operating system.

We used two types of datasets which are real and random. The five real datasets are the complete EST databases of *Citrus clementina*[2], *Citrus sinensis*[2], *Citrus trifoliata*[2], C. *elegans*[3], and *Atta cephalotes*[4], which were used as the datasets in the SOF paper [9]. The alphabet of the datasets is $\{A, C, G, T\}$. Table 1 shows specific information of the datasets. Whereas Readjoiner discards low-quality reads before APSP matching, we made three algorithms take all reads as input strings for a fair comparison. The random datasets are generated by a program[1] that gives k strings such that the lengths of the strings follow a normal distribution with mean μ and standard deviation σ and the characters of the strings follow a uniform distribution over the alphabet, where k, μ and σ are parameters given by the user. The alphabet is again $\{A, C, G, T\}$. We generated two datasets $rnd1$ and $rnd2$, where $rnd1$ has 300000, 1000, and 150 as k, μ, and σ, respectively, and $rnd2$ has 1000000, 500, and 100.

We compare the performances of SOF, Readjoiner, and our algorithm for the whole process of APSP matching, i.e., including the time to build their own data structures, the

[1] `http://theory.snu.ac.kr/?p=814`
[2] `http://www.citrusgenomedb.org`
[3] `http://www.uni-ulm.de/in/theo/research/seqana`
[4] `http://antgenomes.org`

Table 2 Running time (in second) of algorithms for real datasets.

om	algorithm	*clementina*	*sinensis*	*trifoliata*	*C. elegans*	*Atta*
10	SOF	22.68	44.18	8.07	62.44	23.91
	Readjoiner	114.13	450.48	57.52	231.66	-
	Our1	3.44	6.10	1.50	14.83	0.70
	Our2	5.07	11.21	1.94	14.52	0.86
15	SOF	21.98	40.60	7.70	61.94	24.58
	Readjoiner	50.47	137.54	33.87	84.25	-
	Our1	2.90	4.90	1.37	14.19	0.64
	Our2	3.36	6.44	1.42	13.84	0.73
20	SOF	20.82	36.60	7.35	60.48	23.79
	Readjoiner	11.53	38.42	3.36	44.15	-
	Our1	2.78	4.33	1.30	14.02	0.64
	Our2	2.94	4.75	1.29	13.67	0.69
25	SOF	20.35	35.89	7.33	59.62	23.86
	Readjoiner	7.26	18.80	3.07	27.37	-
	Our1	2.72	4.29	1.29	13.89	0.64
	Our2	2.77	4.42	1.28	13.54	0.66

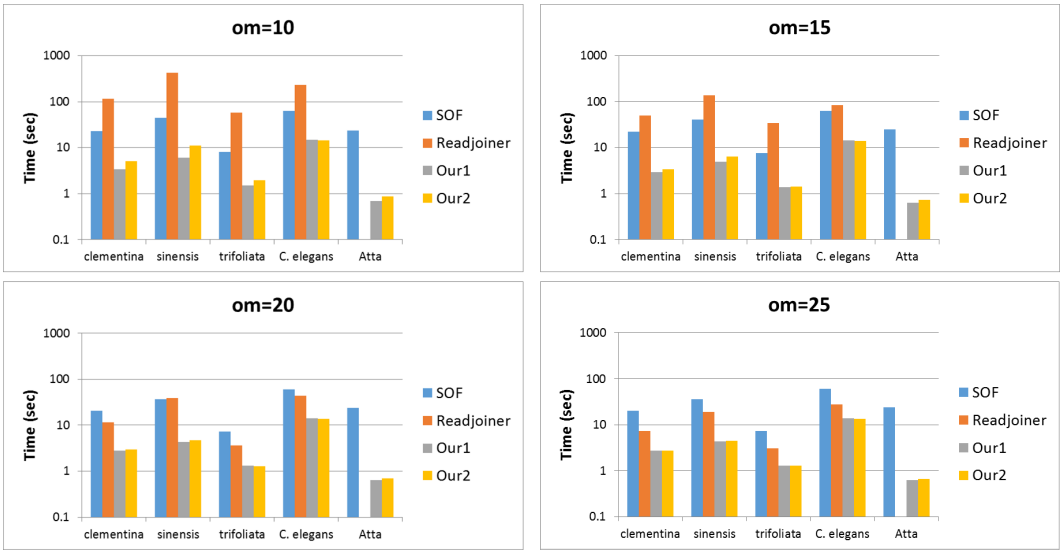

Figure 2 Running time (in second) of algorithms for real datasets (y-axis in log scale).

time for the matching phase, and the time to write the output (For Readjoiner, it is the overlap phase of the whole sequence assembly). The labels of Our1 and Our2 mean our algorithm with option $output = 2$ and $output = 3$, respectively. For SOF, option $-o$ 2 is used. Our2, SOF with $-o$ 2 and Readjoiner return the same output, which includes all overlaps for each pair of input strings. Our1 solves the problem of APSP matching as it is defined (i.e., it returns the longest overlap for each pair of input strings). Readjoiner does not have an option to return the longest overlap for each pair of input strings. SOF finds the longest overlap only when it returns the matrix Ov (with option $-o$ 1), but when it returns a list of $(i, j, Ov[i, j])$ as output (with option $-o$ 2), it returns all overlaps for each pair of input strings and there is no way to return the longest overlap for each pair. In our algorithm we set c_1 to 16 and c_2 to 8 in all experiments.

Table 3 Running time (in second) of algorithms for random datasets.

om	algorithm	rnd1	rnd2	om	algorithm	rnd1	rnd2
10	SOF	109.16	258.82	20	SOF	108.40	252.59
	Readjoiner	481.10	1322.40		Readjoiner	31.67	149.51
	Our1	5.26	13.44		Our1	4.85	12.67
	Our2	5.14	14.01		Our2	4.74	12.48
15	SOF	108.92	256.03	25	SOF	107.3	250.25
	Readjoiner	123.60	214.52		Readjoiner	7.34	14.70
	Our1	4.86	13.03		Our1	4.77	12.38
	Our2	4.81	12.84		Our2	4.83	12.16

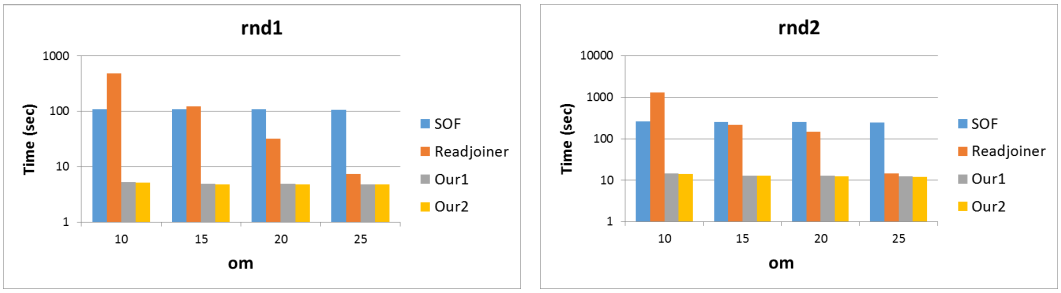

Figure 3 Running time (in second) of algorithms for random datasets (y-axis in log scale).

Table 2 and Figure 2 show the running time (in second) of each algorithm with the real datasets. We carried out experiments when om is 10, 15, 20, and 25. The y-axis (i.e. running time) of Figure 2 is in log scale. Our algorithm outperforms SOF and Readjoiner in all cases. In the experiment with *Atta*, we obtained a huge speed-up compared with SOF because case 1 of the matching step of our algorithm is very effective when m is large. When one of the input strings is very large (its length is over 15 millions in Atta), SOF shows a poor performance. In the experiment with *Atta*, Readjoiner stopped and printed "cannot realloc() memory" for all values of om. Readjoiner is not efficient for small values of om (e.g., $om = 5$). Experimental results show that our algorithm performs well consistently, not depending on one large input string or values of om.

Table 3 and Figure 3 show the running time of each algorithm with random datasets. Again we did experiments when om is 10, 15, 20, and 25. The y-axis (i.e. running time) is in log scale. The performance of our algorithm is better than those of SOF and Readjoiner in all cases. The running time of Readjoiner decreases as om increases.

We computed the average speedup of our algorithm Our2 over SOF and Readjoiner for all experiments. The average speedup of Our2 over SOF for all 28 experiments (5 real datasets + 2 random datasets and 4 values of om) is about 14 and that of Our2 over Readjoiner for all 24 experiments (4 read datasets + 2 random datasets and 4 values of om) is about 18.

Table 4 and Figure 4 show the running time of each algorithm with *parallel* options for real datasets. We used different numbers of threads (1, 2, 4, and 8) and a fixed value 15 of om, which was the value of om in all experiments of the SOF paper [9]. Our algorithm and SOF show reasonable scalability in all experiments. However, SOF does not scale well in the experiment with *Atta* because SOF has poor scalability when one of the input strings is very large. Readjoiner does not show good scalability in all experiments.

The peak value of memory usage is measured by `/usr/bin/time -v`. Table 5 and Figure 5 show the peak memory for SOF, Readjoiner, and Our2 on the real datasets with $om = 15$.

Table 4 Running time (in second) of algorithms with *parallel* options 1, 2, 4, and 8.

algorithm	thread	*clementina*	*sinensis*	*trifoliata*	*C. elegans*	*Atta*
SOF	1	21.98	40.60	7.70	61.94	24.59
	2	11.63	21.41	4.32	34.98	23.23
	4	6.95	11.41	2.26	20.10	23.15
	8	4.61	8.56	1.59	14.22	23.06
Readjoiner	1	50.47	137.54	33.87	84.25	-
	2	42.92	118.02	19.67	80.88	-
	4	41.57	117.39	18.35	77.86	-
	8	43.22	113.64	18.14	71.71	-
Our1	1	2.90	4.90	1.37	14.19	0.64
	2	1.88	3.17	0.99	8.94	0.62
	4	1.28	2.04	0.53	5.79	0.43
	8	1.09	1.97	0.44	4.96	0.42
Our2	1	3.36	6.44	1.42	13.84	0.73
	2	2.34	4.81	0.93	8.44	0.72
	4	1.77	3.52	0.61	5.69	0.54
	8	1.48	3.43	0.50	4.84	0.53

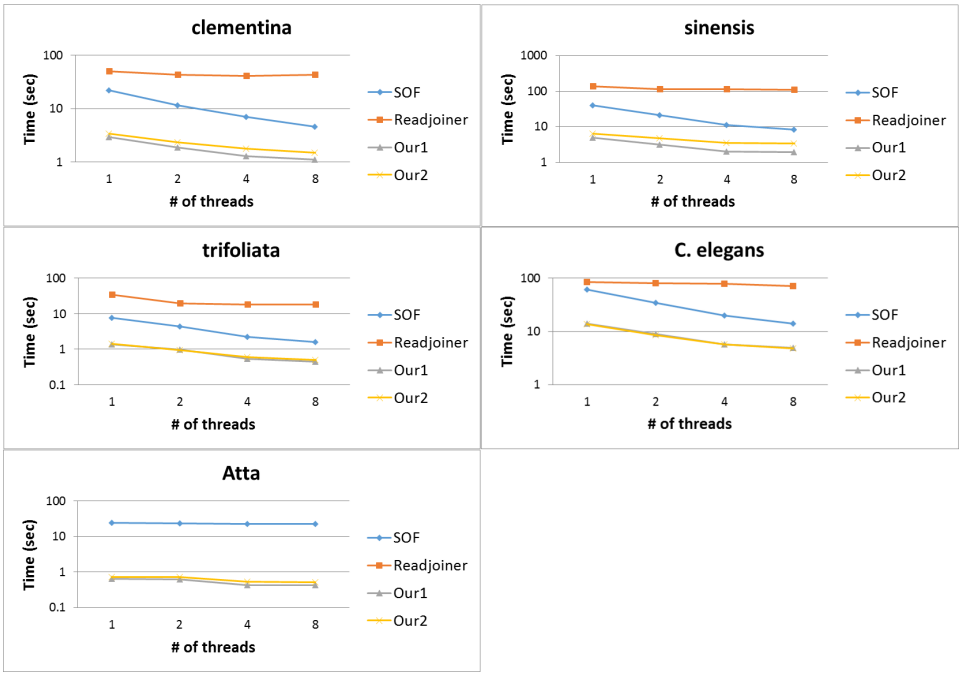

Figure 4 Running time (in second) of algorithms with *parallel* options 1, 2, 4 and 8 (y-axis in log scale).

Table 5 Peak memory usage (in MB) of algorithms.

	clementina	*sinensis*	*trifoliata*	*C. elegans*	*Atta*
SOF	53	81	26	101	111
Readjoiner	191	419	83	88	-
Our2	128	263	89	280	270

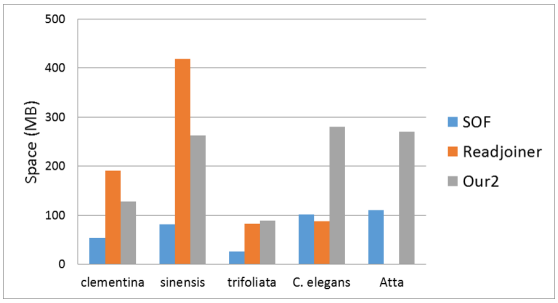

Figure 5 Peak memory usage of algorithms.

Our algorithm uses more memory than SOF in all cases because of additional auxiliary data structures, but the memory usage of our algorithm is still within the optimal bound of $O(N)$ as described in Sections 2.1 and 2.3. Our algorithm uses more memory or less memory than Readjoiner depending on datasets.

4 Conclusion

In this paper we have presented a fast algorithm for all-pairs suffix-prefix matching. The main idea of the algorithm is a combination of a simple but effective data structure for storing input strings and advanced algorithmic techniques for matching to achieve fast running time. Experimental results show that our algorithm runs much faster than previous state-of-the-art algorithms SOF and Readjoiner for APSP matching. Also we obtain reasonable scalability with a parallel implementation of our algorithm.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

2 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

3 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

4 Maxime Crochemore, Artur Czumaj, Leszek Gąsieniec, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. Fast practical multi-pattern matching. *Information Processing Letters*, 71(3-4):107–113, 1999.

5 Hieu Dinh and Sanguthevar Rajasekaran. A memory-efficient data structure representing exact-match overlap graphs with application for next-generation dna assembly. *Bioinformatics*, 27(14):1901–1907, 2011.

6 Giorgio Gonnella and Stefan Kurtz. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC bioinformatics*, 13(1):82, 2012.

7 Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

8 Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41(4):181–185, 1992.

9 Maan Haj Rachid and Qutaibah Malluhi. A practical and scalable tool to find overlaps between sequences. *BioMed research international*, 2015, 2015.

10 Maan Haj Rachid, Qutaibah Malluhi, and Mohamed Abouelhoda. Using the sadakane compressed suffix tree to solve the all-pairs suffix-prefix problem. *BioMed research international*, 2014, 2014.

11 David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome research*, 18(5):802–809, 2008.

12 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

13 Felipe A. Louza, Simon Gog, Leandro Zanotto, Guido Araujo, and Guilherme P. Telles. Parallel computation for the all-pairs suffix-prefix problem. In *International Symposium on String Processing and Information Retrieval*, pages 122–132. Springer, 2016.

14 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

15 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005.

16 Enno Ohlebusch and Simon Gog. Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Information Processing Letters*, 110(3):123–128, 2010.

17 Maan Haj Rachid, Qutaibah Malluhi, and Mohamed Abouelhoda. A space-efficient solution to find the maximum overlap using a compressed suffix array. In *Biomedical Engineering (MECBME), 2014 Middle East Conference on*, pages 329–333. IEEE, 2014.

18 Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

19 William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix–prefix problem. *Journal of Discrete Algorithms*, 37:34–43, 2016.

20 Sun Wu, Udi Manber, et al. A fast algorithm for multi-pattern searching. Technical report, University of Arizona. Department of Computer Science, 1994.