BIOINFORMATICS ORIGINAL PAPER Vol. 27 no. 4 2011, pages 464–470
doi:10.1093/bioinformatics/btq677

Sequence analysis Advance Access publication December 9, 2010

SlideSort: all pairs similarity search for short reads

Kana Shimizu[1,*] and Koji Tsuda[1,2]

[1]Computational Biology Research Center, National Institute of Advanced Industrial Science and Technology (AIST) and [2]ERATO Minato Project, Japan Science and Technology Agency, Japan

Associate Editor: Alex Bateman

ABSTRACT

Motivation: Recent progress in DNA sequencing technologies calls for fast and accurate algorithms that can evaluate sequence similarity for a huge amount of short reads. Searching similar pairs from a string pool is a fundamental process of *de novo* genome assembly, genome-wide alignment and other important analyses.

Results: In this study, we designed and implemented an exact algorithm *SlideSort* that finds all similar pairs from a string pool in terms of edit distance. Using an efficient pattern growth algorithm, *SlideSort* discovers chains of common k-mers to narrow down the search. Compared to existing methods based on single k-mers, our method is more effective in reducing the number of edit distance calculations. In comparison to backtracking methods such as BWA, our method is much faster in finding remote matches, scaling easily to tens of millions of sequences. Our software has an additional function of single link clustering, which is useful in summarizing short reads for further processing.

Availability: Executable binary files and C++ libraries are available at http://www.cbrc.jp/~shimizu/slidesort/ for Linux and Windows.

Contact: slidesort@m.aist.go.jp; shimizu-kana@aist.go.jp

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on August 19, 2010; revised on October 27, 2010; accepted on December 3, 2010

1 INTRODUCTION

Due to the dramatic improvement of DNA sequencing, it is required to evaluate sequence similarities among a huge amount of fragment sequences such as short reads. We address the problem of enumerating all neighbor pairs in a large string pool in terms of edit distance, where the cost of insertion, deletion and substitution is one. Namely, given a set of n sequences of equal length ℓ, $s_1,\dots,s_n$, the task is to find all pairs whose edit distance is at most d,

$$E=\left\{(i,j)\,|\,EditDist(s_i,s_j)\le d, i<j\right\}. \tag{1}$$

It is conventionally called *all pairs similarity search*.

All pairs search appears in important biological tasks. For example, it is required in finding seed matches in all pairs alignment necessary in sequence clustering (Abouelhoda *et al.*, 2004). Such alignments can then be used to detect and correct errors in short reads (Qu *et al.*, 2009). In the first step of *de novo* genome assembly (Simpson *et al.*, 2009; Zerbino and Birney, 2008), short reads are decomposed to k-mers, and suffix–prefix matches of length $k-1$ are detected. In most cases, exact matches are employed due to time constraint. Using approximate matches, the length of contigs can be extended, which leads to final assembly of better quality. This problem reduces to all pairs similarity search by collecting all $k-1$ prefixes and suffixes into a sequence pool. From the output, only prefix–suffix pairs are reported.

Basically, most popular methods solve the search problem by either of the following two approaches or a combination of them. (i) Finding a common k-mer and verify the match (Lipman and Pearson, 1985; Simpson *et al.*, 2009; Warren *et al.*, 2007; Weese *et al.*, 2009; Zerbino and Birney, 2008). (ii) Backtracking in an index structure (i.e. suffix array and FM-index) (Langmead *et al.*, 2009; Li and Durbin, 2009; Li *et al.*, 2009; Rajasekaran *et al.*, 2005; Sagot, 1998; Trapnell *et al.*, 2009). The first type finds common k-mers in strings (i.e. seed match) and verify if two strings sharing the k-mer are neighbors indeed by extending the match with dynamic programming. It works perfectly well when the string is long enough. However, when strings are short and the threshold d is large, the length of shared k-mers falls so short that too many candidate pairs have to be verified. The second type stores the strings into an index structure, most commonly a suffix array. Then, similar strings are found by traversing nodes of the corresponding suffix tree. This approach works fine if d is small, e.g. $d\le 2$, and employed in state-of-the-art short read mapping tools such as BWA (Li and Durbin, 2009), bowtie (Langmead *et al.*, 2009) and SOAP2 (Li *et al.*, 2009). However, it becomes rapidly infeasible as d grows larger, mainly because the complexity is exponential to d and no effective pruning is known. ELAND and SeqMap (Jiang and Wong, 2008) decompose sequences into blocks and use multiple indices to store all k-concatenations of blocks. Obviously, it requires much more memory compared with BWA, which would be problematic in many scenarios. Multisorting (Uno, 2008) uses multiple substring matches to narrow down the search effectively, but it can find neighbors in terms of Hamming distance only.

Our method termed *SlideSort* finds a chain of common substrings by an efficient pattern growth algorithm, which has been successfully applied in data mining tasks such as itemset mining (Han *et al.*, 2004). A pattern corresponds to a sequence of substrings. The space of all patterns is organized as a tree and systematically traversed. Our method does not rely on any index structure to avoid storage overhead. Instead, radix sort is employed to find equivalent strings during pattern growth. To demonstrate the correctness of our algorithm, the existence of a common substring chain in any neighbor pair is proved first. In addition, we deliberately avoid reporting the same pair multiple times by duplication checking. As a result, our method scales easily to 10 million sequences and is

*To whom correspondence should be addressed.

much faster than seed matching methods and suffix arrays for short sequences and large radius.

The rest of this article is organized as follows. Section 2 introduces our algorithm. In Section 3, results of computational experiments are presented. Section 4 concludes the article.

2 METHOD

Two similar strings share common substrings *in series*. Therefore, we can detect similar strings by detecting chains of common strings systematically. Before proceeding to the algorithm, let us describe fundamental properties first. Divide the interval $1,\dots,\ell$ into b blocks of arbitrary length $w_1,\dots,w_b$, $\sum_{i=1}^{b} w_i=\ell$. The starting position of each block is defined as $q_i=1+\sum_{j=1}^{i-1} w_j$. The alphabet is denoted as Σ. We assume that each string in the database $\{s_i\}_{i=1}^{n}$ consists of ℓ letters, $s_i\in|\Sigma|^{\ell}$. Given two strings s, t, $s=t$ holds if all letters are identical. The substring from positions i to j is described as $s[i,j]$.

A *pattern* of length k is defined as a sequence of strings and block indices,

$$X=[(\boldsymbol{x}_1,y_1),\dots,(\boldsymbol{x}_k,y_k)]$$

where $\boldsymbol{x}_i\in|\Sigma|^{w_{y_i}}$, $1\le y_1<y_2,\dots,<y_k\le b$. Pattern X matches to string s_i with offset $\boldsymbol{p}=(p_1,\dots,p_k)$, if

$$s_i[q_{y_j}+p_j,q_{y_j}+p_j+w_{y_j}-1]=\boldsymbol{x}_j,\ \text{for all}\ j=1,\dots,k.$$

All occurrences of X in the database are denoted as

$$C(X)=\{(i,\boldsymbol{p})\,|\,X\ \text{matches}\ s_i\ \text{with offset}\ \boldsymbol{p}\}.$$

For convenience, an index set $I(X)$ is defined as the set of sequences appearing in $C(X)$. The number of sequences in $I(X)$ is defined as $|I(X)|$.

The relationship between neighbor pairs and patterns are characterized by the following theorem.

THEOREM 1. *If s_i and s_j of equal length ℓ are neighbors, i.e. $EditDist(s_i,s_j)\le d, i<j$, there exists a pattern X of length $b-d$ such that X matches s_i with zero offset ($p_1=p_2=\dots=p_{b-d}=0$) and matches s_j with bounded offset $-\lfloor d/2\rfloor\le p_k\le\lfloor d/2\rfloor$, $k=1,\dots,b-d$.*

PROOF. There are multiple possible alignments of s_i and s_j. An alignment is characterized by the number of matches m, that of mismatches f, that of gaps g_i in s_i and that of gaps g_j in s_j.The length of s_i is equal to $m+f+g_j$ and that of s_j is $m+f+g_i$, because any letter in s_i is aligned to either a letter in s_j or gap symbol in s_j and vice versa. Thus, we obtain $g_i=g_j\le\lfloor d/2\rfloor$ by taking into account that the maximum number of gaps does not exceed d. Therefore, an aligned position of any letters is within a bound of $\lfloor d/2\rfloor$ letters from its original position.

Let us divide s_i into b blocks of length $w_1,\dots,w_b$. Since the number of mismatches are at most d, at least $b-d$ blocks match exactly with their counterparts in s_j in any alignment. Also, since aligned positions are bound within $\lfloor d/2\rfloor$ from their original positions, the matching counterpart of any block of s_i can be found in s_j within offset between $-\lfloor d/2\rfloor$ and $\lfloor d/2\rfloor$. ■

Figure 1 illustrates an example of patterns with $b=5, d=3$. This theorem implies that any neighbor pair has a chain of $b-d$ common blocks and the corresponding blocks lie close to each other. It serves as a foundation of our algorithm presented later on.

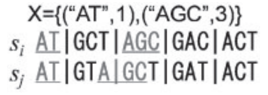

Fig. 1. An example pattern for block size 5 and edit-distance threshold 3. s_i matches to X with no offset in the first block and the third block. s_j matches to X with no offset in the first block but with -1 offset in the third block.

2.1 Pattern growth

In our algorithm, all patterns X of $|I(X)|\ge 2$ are enumerated by a recursive pattern growth algorithm. In a pattern growth algorithm, a pattern tree is constructed, where each node corresponds to a pattern (Fig. 2). Nodes at depth k contain patterns of length k.

At first, patterns of length 1 are generated as follows. For each block $y_1=1,\dots,d+1$, a string pool is made by collecting substring of $\{s_i\}_{i=1}^{n}$ starting at $q_{y_1}-\lfloor d/2\rfloor,\dots,q_{y_1}+\lfloor d/2\rfloor$. Applying radix sort to the string pool and scanning through the sorted result, repetition of equivalent strings can be detected (Fig. 3). Each pattern of length 1, denoted as X_1, is constructed as a combination of the repeated string $\boldsymbol{x}_1$ and y_1,

$$X_1\leftarrow\{(\boldsymbol{x}_1,y_1)\}.$$

At the same time, all occurrences $C(X_1)$ are recorded. If s_i matches the same pattern X_1 by several different offsets, only the smallest offset is recorded. They form the nodes corresponding to depth 1 of the pattern tree.

Given a pattern X_t of length t, its children in the pattern tree are generated similarly as follows. For each $y_{t+1}=y_t+1,\dots,d+t+1$, a string pool is made by collecting substrings of $I(X_t)$ starting at $q_{y_{t+1}}-\lfloor d/2\rfloor,\dots,q_{y_{t+1}}+\lfloor d/2\rfloor$. Because the string pool is made from the occurrence set only, the size of the pool decreases sharply as a pattern grows. By sorting and scanning, a next string x_{t+1} is identified and the pattern is extended as

$$X_{t+1}\leftarrow X_t+\{(\boldsymbol{x}_{t+1},y_{t+1})\},$$

and the occurrences $C(X_t)$ are updated to $C(X_{t+1})$ as well.

To avoid generation of useless patterns, the pattern tree is pruned as soon as the support falls below 2. Also, the tree is pruned if there is no string in $I(X)$ that matches X with zero offset. As pattern growth proceeds in a depth-first manner, memory is reserved as a pattern is extended, and then immediately released as the pattern is contracted to visit another branch. This dynamic memory management keeps peak memory usage rather small.

2.2 From patterns to pairs

As implied in Theorem 1, every neighbor pair (Fig. 1) appears in index set $I(X)$ of at least one pattern. Since one of the pair must have zero offset, the set of eligible pairs is described as

$$P_X=\{(i,j)|i<j,i,j\in I(X),s_i\ \text{matches}\ X\ \text{with zero offset}\}.$$

Since not all members of P_X correspond to neighbors, we have to verify if they are neighbors by actual edit-distance calculation.

A problem here is that the same pair (i,j) possibly appears in the index set of many different patterns. It is also possible that pair (i,j) in the same index set is derived from different offsets. In most applications, it is desirable to ensure that no pair is reported twice. The straightforward solution of the problem is to check if a new pair is previously reported by storing all pairs, which requires huge amount of memory. We propose an alternative solution that rejects *non-canonical* pairs without using any extra memory as follows.

A match of s_i and s_j can occur in various ways, each of which can be described as the tuple $(\boldsymbol{y},\boldsymbol{p})$, where $\boldsymbol{y}=y_1,\dots,y_{b-d}$ describe the blocks in the pattern and $\boldsymbol{p}$ is the offset with which the pattern matches s_j. We define the canonical match as the one with lexicographically smallest $\boldsymbol{y}$ and $\boldsymbol{p}$, where priority is given to $\boldsymbol{y}$. For example, consider the case $s_i=AATT$, $s_j=ATAT$, $d=2$ and all block widths set to 1. There are 10 different $(\boldsymbol{y},\boldsymbol{p})$ pairs as shown in Figure 4, where matching residues are shown in red squares. In this case, (1) is canonical. Among them, the matches with overlapping squares do not have correct alignment. We do not exclude such pairs to avoid an extra run of dynamic programming.

To judge if a given match $(\boldsymbol{y},\boldsymbol{p})$ is canonical or not, it is sufficient to check if there exists another match that is lexicographically smaller. More precisely, the match represented by $\boldsymbol{y},\boldsymbol{p}$ is not canonical, iff there exists a block $1\le z\le\max(\boldsymbol{y}),z\notin\boldsymbol{y}$ and an offset $-\lfloor d/2\rfloor\le r\le\lfloor d/2\rfloor$ such that

$$s_i[q_z,q_z+w_z-1]=s_j[q_z+r,q_z+r+w_z-1]. \quad (2)$$

This canonicity check can be done in $O(d\ell)$ time.

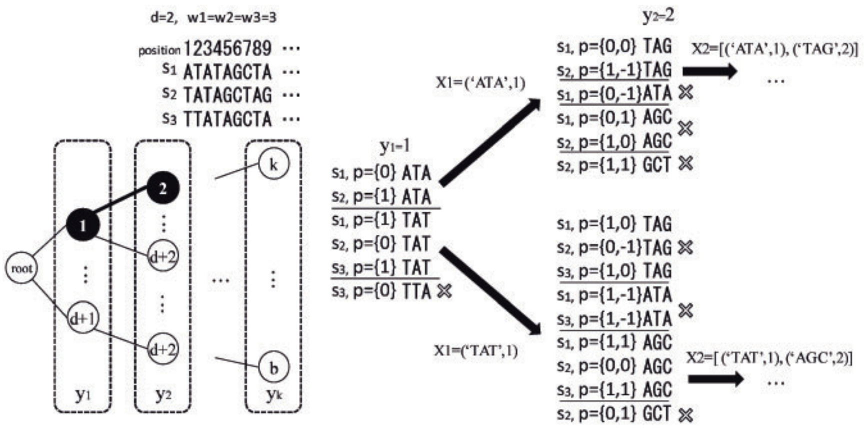

Fig. 2. Pattern growth and pruning process of the proposed method. Patterns are enumerated by traversing the tree in depth-first manner. In each node, new elements are generated by sorting substrings in sequence pool ('ATA', 'TAT', 'TTA' for $y_1 = 1$). Useless patterns ('TTA' in this case) are removed. Remaining elements are added to yield new patterns. This process is executed by recursive call until the pattern size reaches $b - d$.

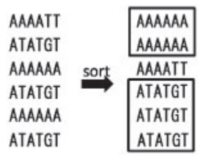

Fig. 3. Discovery of equivalent strings by radix sort.

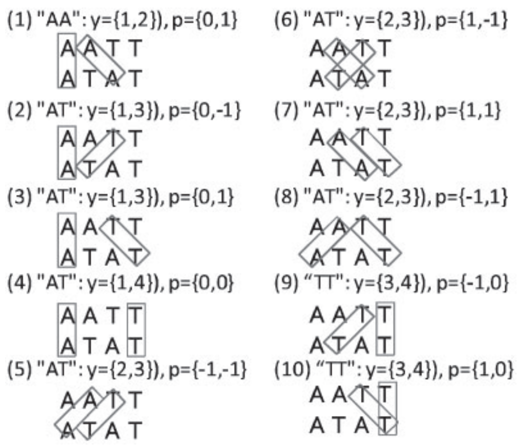
(1) "AA": y={1,2}), p={0,1}
(2) "AT": y={1,3}), p={0,-1}
(3) "AT": y={1,3}), p={0,1}
(4) "AT": y={1,4}), p={0,0}
(5) "AT": y={2,3}), p={-1,-1}
(6) "AT": y={2,3}), p={1,-1}
(7) "AT": y={2,3}), p={1,1}
(8) "AT": y={2,3}), p={-1,1}
(9) "TT": y={3,4}), p={-1,0}
(10) "TT": y={3,4}), p={1,0}
AATT
ATAT

Fig. 4. All $(\boldsymbol{y},\boldsymbol{p})$ of $s_i = AATT$, $s_j = ATAT$. Matching residues are shown in red squares. Since the red squares overlap in (6) and (10), they do not correspond to correct alignment.

Pseudocode of the overall algorithm is shown in Algorithm 1. In line 18, it suffices to compute diagonal stripe of width $2d+1$ of DP matrix. Thus, the distance calculation is performed in $O(d\ell)$ time.

2.3 Remarks

With small modification, SlideSort can deal with gap opening and gap extension penalties. Define the gap open and extension cost as γ_o and γ_e, respectively. Denote the number of mismatches, gap opens and gap extensions as f, g_o and g_e, respectively. Then our all pairs similarity search problem is reformulated as finding pairs such that $f + g_o\gamma_o + g_e\gamma_e \leq d$. Denote the number of gaps in each sequence as g_i and g_j. Then, $g_e = g_i + g_j$ and $g_o \geq 2$ (if $g_e \neq 0$), $g_o = 0$ (if $g_e = 0$) by definition. When $g_e \neq 0$, we have $(g_i + g_j)\gamma_e \leq d - 2\gamma_o$. Since the lengths of two sequences are equal, the number of gaps is also equal, $g_i = g_j$, leading to the following inequality,

$$g_i = g_j \leq \lfloor (d/2 - \gamma_o)/\gamma_e \rfloor.$$

Therefore, the offsets p_k, for $k = 1, \ldots, b-d$, are bounded by

$$-\lfloor (d/2 - \gamma_o)/\gamma_e \rfloor \leq p_k \leq \lfloor (d/2 - \gamma_o)/\gamma_e \rfloor. \tag{3}$$

When $g_e = 0$, we can find all pairs by zero offsets, hence the offset range (3) covers this case as well. Notice that the block size b must be larger than $\max(d, \lfloor d/(\gamma_o + \gamma_e) \rfloor)$. This modification is effective to reduce both computation time and memory space when γ_o and γ_e are larger than substitution cost.

It is worthwhile to notice that SlideSort can handle sequences of slightly different lengths without any essential modification. See a Supplementary Method in Supplementary file 1 for details.

2.4 Complexity

Denote by $\sigma = |\Sigma|$ the alphabet size. Space complexity of SlideSort is $O((b-d)dn\log n + n\ell\log\sigma)$, because it requires an pointer array to describe the pattern tree, and the original strings must be retained. Denote by m the number of all pairs included in the index set $I(X)$. Time complexity of SlideSort is $O(b^{d-1}d^{b-d}\ell n + md\ell)$, in which the first part is for sorting and the latter part is for edit-distance calculations. The time complexity depends on the effectiveness of pruning through m. The worst case of the latter part becomes $O(n^2 d\ell)$ when all the input short reads are identical. In most cases, however, short reads are quite diverse and m is expected to scale much better than $O(n^2)$.

The all pairs similarity search problem can be solved by finding approximate non-tandem repeats in the concatenated text of length $n\ell$. An enhanced suffix array can solve it with $O(\ell^{d+1}\sigma^d n)$ time and $O(n\log n + n\ell\log\sigma)$ space (Abouelhoda *et al.*, 2004). This time complexity is essentially achieved by producing all variants within distance d of all sequences and finding identical pairs. The difference is that the time complexity of the suffix array depends on the alphabet size and that not of SlideSort. Thus, SlideSort can be applied to large alphabets (i.e. proteins) as well.

3 EXPERIMENTS

From NCBI Sequence Read Archive (http://www.ncbi.nlm.nih.gov/sra/), we obtained two datasets: paired-end sequencing of

Algorithm 1 SlideSort

```
1.  function SLIDESORT
2.      SlideSortRecursive($\phi,\phi$)
3.  end function
4.  function SLIDESORTRECURSIVE($\boldsymbol{y}$ , $X$)
5.      if $y=\phi$ then
6.          $m\leftarrow 1$
7.          go to line 26
8.      end if
9.      if $|I(X)|<2$ then                        ▷ Pruning by frequency
10.         return
11.     end if
12.     if no strings in I(X)match $X$ with zero offset then
13.         return
14.     end if
15.     if $|\boldsymbol{y}|=b-d$ then
16.         for $(i,j)\in P_X$ do
17.             if $(i,j)$ is canonical then            ▷ See equation 2
18.                 if $EditDist(s_i,s_j)\leq d$ then
19.                     Report $(i,j)$
20.                 end if
21.             end if
22.         end for
23.         return
24.     end if
25.     $m=\max(\boldsymbol{y})+1$
26.     for $z=m,\cdots,d+|\boldsymbol{y}|$ do
27.         $R\leftarrow\phi$
28.         for $-\lfloor d/2\rfloor\leq r\leq\lfloor d/2\rfloor$ do   ▷ Generate a string pool
29.             $R\leftarrow R\cup\{s[q_z+r,q_z+r+w_z-1]\,|\,s\in I(X)\}$
30.         end for
31.         Sort and scan $R$ to find the set of new elements $\mathbf{X}$
32.         for all $(\boldsymbol{x}_{new},z)\in\mathbf{X}$ do
33.             SlideSortRecursive($\boldsymbol{y}+\{z\},X+\{(\boldsymbol{x}_{new},z)\}$)
34.         end for
35.     end for
36. end function
```

Human HapMap (ERR001081) and whole genome shotgun bisulfite sequencing of the IMR90 cell line (SRR020262). They will be referred to as dataset 1 and 2, respectively. Sequence length of dataset 1 is 51 and that of dataset 2 is 87. Both datasets were generated by Illumina Genome Analyzer II. Reads that do not include 'N' were selected from the top of the original fastq files. Our algorithm was implemented by C++ and compiled by g++. All the experiments were done on a Linux PC with Intel Xeon X5570 (2.93 GHz) and 32 GB RAM. Only a single core is used in all experiments.

As competitors, BWA (Li and Durbin, 2009) and SeqMap (Jiang and Wong, 2008) are selected among many alternatives, because the two methods represent two totally different methodologies, *backtracking* and *block combination*. BWA is among the best methods using index backtracking, while SeqMap takes an ELAND-like methodology of using multiple indexes for all block combinations. SlideSort is also compared to the naive approach that calculates edit distances of all pairs. BWA and SeqMap are applied to all pairs similarity search by creating an index from all short reads and querying it with the same set of reads.

Notice that both BWA and SeqMap are not originally designed for all pairs similarity search but for read mapping, which requires a larger search space. Although fair comparison is difficult between tools of different purposes, we used mapping tools as competitors, because no tool is widely available for all pairs similarity search, to our knowledge.

For our method, the number b of blocks has to be determined. In the following experiments, we set b relative to the distance threshold d as $b=d+k$. Here, k corresponds to the pattern size. In the following experiments, we tried $k=1,\ldots,5$ and reported the best result.

The number of neighbor pairs for both datasets are shown in Supplementary Figure S1. We confirmed that both SlideSort and the naive approach reported exactly the same number of neighbor pairs, which ensures correctness of our implementation of SlideSort.

3.1 Computation time and memory usage

Figure 5 plots computation time against the distance threshold d. SlideSort is consistently faster in all configurations. As the number of sequences grows and the distance threshold is increased, the difference from BWA and SeqMap becomes increasingly evident. Not all results are obtained, because of the 30 GB memory limit and 300 000 s time limit. Figure 6 compares peak memory usage of SlideSort, SeqMap and BWA. We separately measured the memory usage of the indexing step and searching step for BWA, because BWA is designed to execute those steps separately. The peak memory of BWA for the search step is the smallest in most of the configurations, while that of SlideSort is comparable or slightly better than BWA's peak indexing memory. Detailed results for 100 000 short reads are shown in Table 1.

BWA is most efficient in space complexity, because its index size does not depend on the distance threshold. Instead, BWA's time complexity rapidly deteriorates as the edit-distance threshold grows due to explosion of the number of traversed nodes in backtracking. In contrast, SeqMap indexes and hashes all the combination of key blocks, which leads to huge memory usage. SlideSort is similar to SeqMap in that it considers all block combinations, but is much more memory efficient. The difference is that SlideSort is an indexing free method, which dynamically generates the pattern tree by depth-first traversal. It allows us to maintain only necessary parts of tree in memory.

3.2 Effect of pattern size

Figure 7 investigates the influence of pattern size k on the efficiency. Except for $d=1$, the best setting was around $k=2$–4. Our method $k=1$ roughly corresponds to the single seed approach, so this result suggests the effectiveness of using chains. Overall, the computation time was not too sensitive to the choice of k.

3.3 Comparison to single seed

Our algorithm employs a chain of common substrings to narrow down the search. Compared with the single seed approach that uses a k-mer to derive candidate pairs, the total length of substrings can be much larger than the k-mer without losing any neighbors. It yields higher specificity leading to a smaller number of candidate pairs. Instead of a chain, one can detect multiple k-mers and verify those pairs containing multiple matches (Burkhardt and Kärkkäinen, 2002). However, this approach has lower specificity in comparison

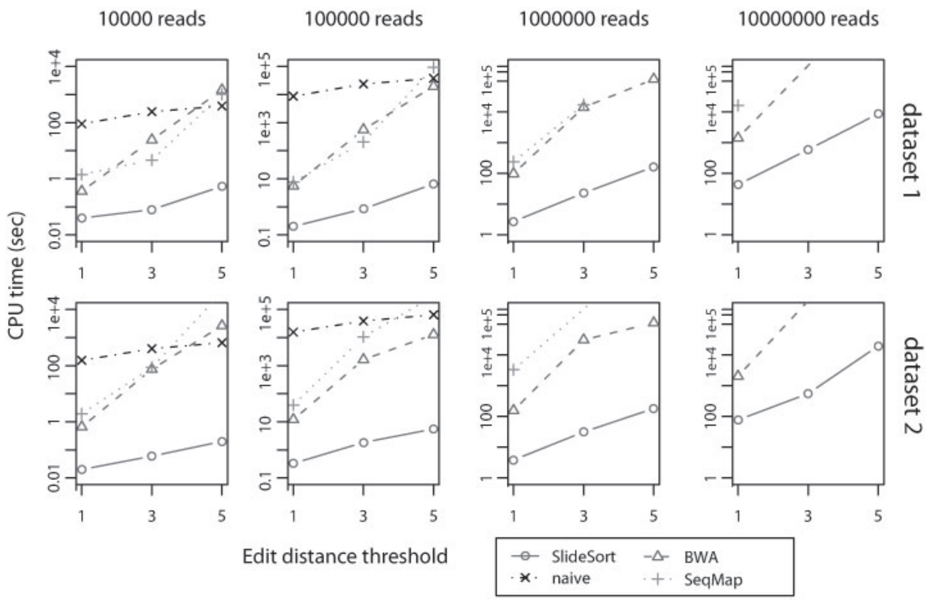

Fig. 5. Computation time on the two short read datasets. Among the four methods, 'naive' represents the exhaustive distance calculation.

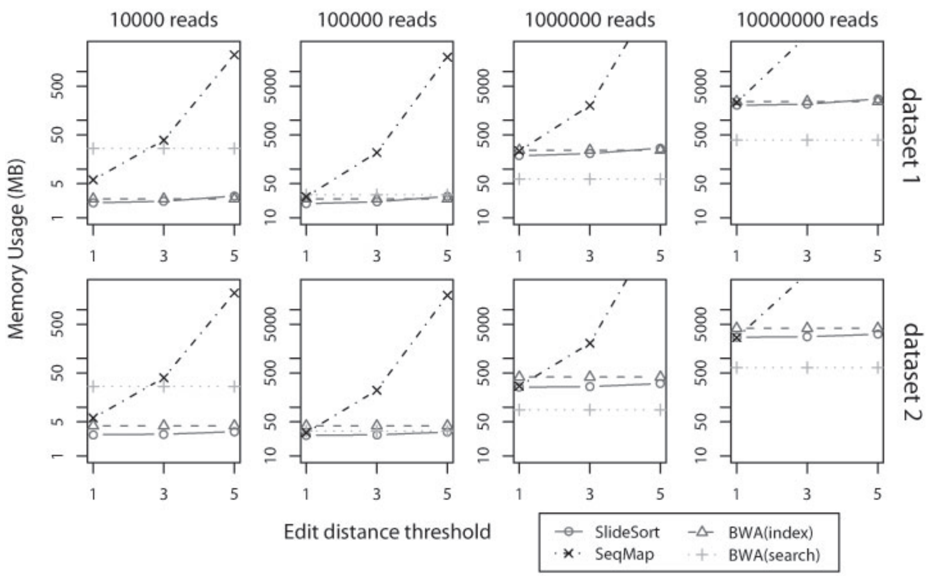

Fig. 6. Memory usage on the two short read datasets. BWA's memory usage is separately evaluated for the indexing step (index) and the search step (search).

Table 1. Computation time on 100 000 short reads

	Dataset 1					Dataset 2				
	SlideSort	BWA		SeqMap	Naive	SlideSort	BWA		SeqMap	Naive
		Index	Search				Index	Search		
d = 1	0.2	2.34	3.25	7.68	8743.46	0.33	5.07	6.91	39.59	15 678
d = 3	0.85	2.37	562.63	205.26	23 796.1	1.84	5.09	1647.16	10 698.6	39 046.3
d = 5	6.56	2.19	19 697.67	93 115.2	38 179.5	5.56	5.08	12 876.88	>300 000	65 244.6

to the chain of the same total length, because the matching positions of each elements of the chain are strictly localized due to Theorem 1.

Figure 8 compares the number of candidate pairs generated by our method and single seed (k-mer in plot). It corresponds to the number of edit-distance calculations. We have two variations of the single seed method: 'k-mer/nonredundant' stores previously reported pairs in memory, and does not include previously reported pairs in candidates. 'k-mer/redundant' does not use additional memory but counts the same pair multiple times. Here we set the length of the k-mer to ℓ/d so that no neighbors are lost. In the plot, one can see a significant reduction in the number of candidate pairs in our algorithm. Notice that the number of candidate pairs is shown in log scale. In our method, the number of candidates is minimum at the largest pattern size, because the total length of substrings is maximized and specificity becomes optimal. However, since the search space of patterns is expanded, the total computation time is not optimal in this case.

3.4 Clustering analysis of short reads

A main application of SlideSort is hierarchical sequence clustering, which would be used in correcting errors in short reads and preprocessing for metagenome mapping, for example. SlideSort provides an undirected graph G, where vertices represent short reads and weighted edges represent edit distances of neighbor pairs. Among hierarchical clustering algorithms, single link is most scalable (Manning *et al.*, 2008). Since the dendrogram of single link clustering is isomorphic to the minimum spanning tree (Gower and Ross, 1969), one can perform single link clustering via minimum spanning trees (MSTs) construction by the Kruskal or Prim algorithm (Kruskal, 1956; Prim, 1957).

Since storing all edges can require a prohibitive amount of memory, we used a well-known online algorithm for building MSTs (Tarjan, 1983). It creates MSTs from a stream of edges, discarding unnecessary edges along the way. It essentially maintains all cycle-free connected components and, if a cycle is made by a new edge, it removes the heaviest edge from the cycle. In our experiment, the additional computation time for finding MSTs was trivially small compared with that of SlideSort finding similar pairs (Table 2). Figure 9 visualizes largest MSTs found in 10 000 000 short reads of dataset 2 with edit-distance threshold 3 by the 3D visualization tool Walrus (http://www.caida.org/tools/visualization/walrus/).

4 CONCLUSION

In this study, we developed a novel method that enumerates all similar pairs from a string pool in terms of edit distance. The proposed method is based on a pattern growth algorithm that can effectively narrow down the search by finding chains of common k-mers. Using deliberate duplication checks, the number of edit distance calculations is reduced as much as possible. SlideSort was evaluated on large datasets of short reads. As a result, it was about 10–3000 times faster than other index-based methods. All these results demonstrate practical merits of SlideSort.

One naturally arising question is if SlideSort can be used for mapping. In fact, it is possible by storing the pattern tree (Fig. 2) in memory, and using it as an index structure. However, the index would cost too much memory for genome-scale data. What we learned from this study is that all pairs similarity search is essentially different from mapping in that one can employ pruning and dynamic memory management. Thus, all pairs similarity search is not a subproblem of mapping and deserves separate treatment.

In future work, we would like to implement SlideSort with parallel computation techniques. Recent progress in hardware technology enables end-users to use many types of parallel computing scheme such as SSE and GPGPU. SlideSort would be further improved by using these technologies.

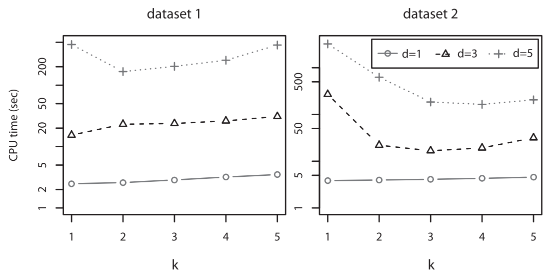

Fig. 7. Comparison of performance of the proposed method with different k evaluated on 1 000 000 short reads.

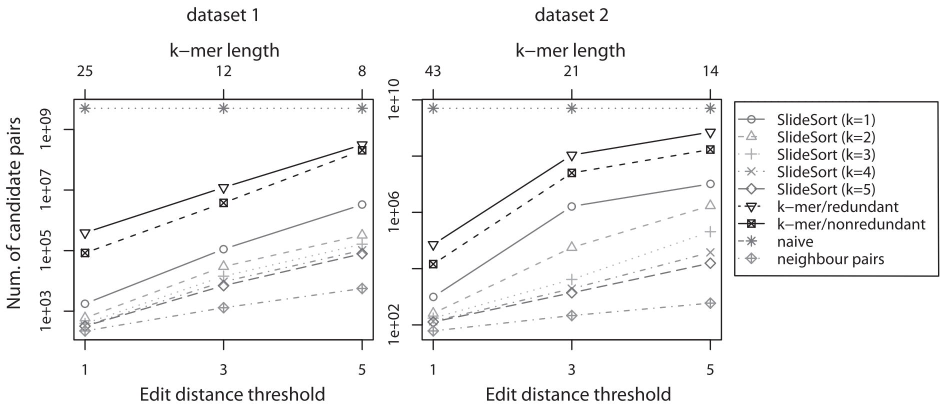

Fig. 8. Comparison of number of candidate pairs. The evaluations were done on 100 000 short reads. The proposed method was examined with $k=1,\dots,5$. 'Neighbor pairs' represent the actual number of neighbor pairs in data. 'k-mer/nonredundant' and 'k-mer/redundant' represent two variants of the single seed method (see text).

Table 2. Comparison of computation time of searching pairs and finding MSTs for two types of short read datasets with edit-distance threshold 3

Number of reads	Dataset 1		Dataset 2	
	Searching pairs only (s)	Finding MSTs (s)	Searching pairs only (s)	Finding MSTs (s)
10 000	0.08	0.00	0.06	0.00
100 000	0.85	0.08	1.84	0.01
1 000 000	23.08	1.08	31.34	2.06
10 000 000	495.15	17.69	554.09	5.61

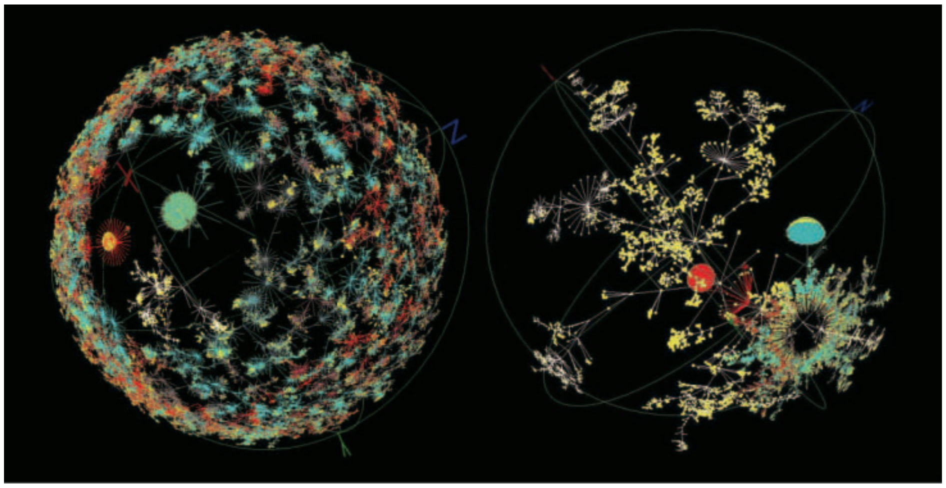

Fig. 9. Visualization of large MSTs from a neighbour graph of 10 000 000 short reads with edit-distance threshold 3. The left graph shows 360 MSTs of 112 995 nodes, each of which consists of more than 100 nodes. The right graph focuses on the largest MST that consists of 6990 nodes. It is straightforward to obtain the dendrograms of single link clustering from these MSTs.

ACKNOWLEDGEMENT

The authors thank Kiyoshi Asai, Hisanori Kiryu, Takeaki Uno, Tetsuo Shibuya, Yasuo Tabei and Martin Frith for their fruitful discussions.

Funding: Grant-in-Aid for Young Scientists (22700319, 21680025) by JSPS; FIRST program of the Japan Society for the Promotion of Science in part.

Conflict of Interest: none declared.

REFERENCES

Abouelhoda,M. *et al.* (2004) Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, **2**, 53–86.

Burkhardt,S. and Kärkkäinen,J. (2002) One-gapped q-gram filters for levenshtein distance. In *Proceedings of the 13th Symposium on Combinatorial Pattern Matching (CPM'f02)*. Vol. 2373 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 225–234.

Gower,J. and Ross,G. (1969) Minimum spanning trees and single-linkage cluster analysis. *Appl. Stat.*, **18**, 54–64.

Han,J. *et al.* (2004) Mining frequent patterns without candidate generation. *Data Min. Knowl. Discov.*, **8**, 53–87.

Jiang,H. and Wong,W.H. (2008) Seqmap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, **24**, 2395–2396.

Kruskal,J.B. (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Am. Math. Soc.*, **7**, 48–50.

Langmead,B. *et al.* (2009) Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol.*, **10**, R25.

Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, **25**, 1754–1760.

Li,R. *et al.* (2009) Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**, 1966–1967.

Lipman,D.J. and Pearson,W.R. (1985) Rapid and sensitive protein similarity searches. *Science*, **227**, 1435–1441.

Manning,C. *et al.* (2008). *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK.

Prim,R. (1957) Shortest connection networks and some generalizations. *Bell Syst. Tech. J.*, **26**, 1389–1401.

Qu,W. *et al.* (2009) Efficient frequency-based de novo short-read clustering for error trimming in next-generation sequencing. *Genome Res.*, **19**, 1309–1315.

Rajasekaran,S. *et al.* (2005) High-performance exact algorithms for motif search. *J. Clin. Monit. Comput.*, **19**, 319–328.

Sagot,M.-F. (1998) Spelling approximate repeated or common motifs using a suffix tree. In Lucchesi,C.L. and Moura,A.V. (eds), *LATIN '98: Theoretical Informatics, Third Latin American Symposium*, Vol. 1380 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 374–390.

Simpson,J.T. *et al.* (2009) Abyss: a parallel assembler for short read sequence data. *Genome Res.*, **19**, 1117–1123.

Tarjan,R. (1983) Data Structures and Network Algorithms. *Society for Industrial and Applied Mathematics (SIAM)*, Philadelphia, USA.

Trapnell,C. *et al.* (2009) Tophat: discovering splice junctions with rna-seq. *Bioinformatics*, **25**, 1105–1111.

Uno,T. (2008) An efficient algorithm for finding similar short substrings from large scale string data. *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining (PAKDD'08)*, Vol. 5012 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 345–356.

Warren,R.L. *et al.* (2007) Assembling millions of short dna sequences using ssake. *Bioinformatics*, **23**, 500–501.

Weese,D. *et al.* (2009) Razers-fast read mapping with sensitivity control. *Genome Res.*, **19**, 1646–1654.

Zerbino,D.R. and Birney,E. (2008) Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Res.*, **18**, 821–829.