A random-permutations-based algorithm is presented for alignment of long paired-end reads. An implementation of the algorithm is shown to process a million pairs of $100 - 500$bp-long reads in 3-10 minutes on a single CPU, correctly aligning more reads than popular fast aligners, 5-100 times faster.

A permutations-based algorithm for fast alignment of long paired-end reads

Roy Lederman[†]
Technical Report YALEU/DCS/TR-1474
April 10, 2013

[‡] Applied Mathematics Program, Yale University, New Haven CT 06511

Approved for public release: distribution is unlimited.
Keywords: *DNA, Sequencing, NGS, Alignment, Mapping, Random Permutations.*

1 Introduction

Read alignment is a common, computationally expensive, processing step in sequencing procedures. The rapid increase in sequencing throughput makes this step one of the bottlenecks in sequencing. The problem can be expected to become more significant because new sequencing machines produce longer reads while some existing fast alignment software packages process long reads much more slowly.

Most traditional read alignment algorithms belong to two classes of algorithms: prefix-tree-based algorithms and hash-tables-based algorithms. For surveys of algorithms and software, see [1, 2]. We proposed a different approach, based on string-permutations, in [8, 9].

In this manuscript, we present another permutations-based algorithm for read alignment, designed for paired-end long reads with mismatches. We present a prefix-tree structure that accelerates the search in indexes, and a generalized "hit-counter" filter which accelerates filtering. The algorithm can be extended to allow indels and to align long single-end reads.

An implementation of the algorithm is shown to align considerably more reads than existing fast alignment software packages at a considerably shorter time.

The organization of this manuscript: In section 2 we describe some properties of the problem, permutations-based search algorithms and prefix-trees. Subsection 2.2 is a short description of permutations-based algorithms that use lexicographically sorted libraries of strings. In subsection 2.3 we describe a simple prefix-tree data-structures used in the algorithm, and briefly explain why it is not to be confused with the prefix-tree-based approach to read alignment. The algorithm for fast alignment of long paired-end reads is presented in subsection 3.1 and then described more formally in subsection 3.2. Some extensions of the algorithm are mentioned in subsection 3.3. In section 4 we present an implementation of the algorithm and experimental results. Finally, our conclusions are presented in section 5.

2 Preliminaries

2.1 The read alignment problem

In the analysis of reads in DNA sequencing procedures, one is often confronted with the following problem: one is presented with a read, which is a query string composed of characters from the alphabet $\{A, C, G, T\}$. One also has a reference, which is a library of reference strings. One's goal is to find the string in the library which is similar to the query string, with the smallest number of substitutions (smallest Hamming distance). Typically, the library-strings are all the overlapping substrings of a long reference string,

so each string has an associated locus in the long reference string.

In the paired-end case, one is presented with two reads, $Y1$ and $Y2$, and is required to propose good alignments for both, under the constraint that the loci of proposed alignments for $Y1$ and $Y2$ must be within some distance from each other.

There are several extensions to the problem, where one uses different measures of string similarity. In some cases, one considers an "edit distance", allowing insertion and deletions (indels). In this manuscript we will discuss the case of Hamming distance and explain how the algorithm can be extended to the case of edit distance.

We use a two step framework for read alignment. The first step is a search step, in which a search algorithm proposes a list of likely alignments according to some unrefined specification. The second step is a refinement step, where the proposed alignments are checked and scored according to a more detailed model which is appropriate for the particular application. This framework is suitable for permutations-based aligners that rapidly produce lists of candidates, but may not be a natural framework for other fast search algorithms, some of which rely on scoring heuristics that allow them to stop searching when high-score alignments are found. In this manuscript, we discuss a search step which produces a small number of possible alignments using fast search and fast filters. We do not discuss the next refinement step, which can use any standard scoring approach to calculate scores for the small number of search results. In practice, little refinement is necessary.

2.2 Permutations-based search

The permutations-based read-alignment search algorithm presented in [8, 9] uses lexicographically sorted libraries of permuted substrings of the reference. To create these libraries, we first create a library of all possible substrings of the reference. We then generate a random permutation, which is a recipe for reordering the characters in each of these strings. We apply this permutation to all the strings in the library, thereby creating a library of permuted strings. We repeat this procedure several times, creating multiple libraries of permuted strings. All the strings in each of the libraries are created using the same permutation, but different permutations are used for different libraries.

It has been demonstrated that binary searches in lexicographically sorted lists of randomly permuted strings can provide lists of likely alignments. This property is a result of the high probability that at least some of the permutations "pushed" the mismatches in the read to the end of the permuted read, leaving a long "error-free" prefix.

2.3 Prefix trees

A prefix-tree is a data structure that groups strings hierarchically, according to their prefixes. The root of the tree, at level 0 contains all the strings. At level 1, the strings

divided into 4 sets: the set of strings that begin with A, the set of strings that begin with C, the set of strings that begin with G and the set of strings that begin with T. At level 2, there are 16 sets: the level 1 set of strings that begin with A is divided into 4 sets that begin with AA, AC, AG and AT, etc.

The key property which we use is that prefix-trees provide a fast and simple way of finding all strings that share the same prefix. To find a set that shares the first 3 characters of the string $ACGT$, we begin at the root, go to the level 1 set A, then to the its subset AC and then to its subset ACG, which contains all the strings in the library which begin with ACG.

Traditionally, the prefix-tree approach to read alignment considers the original reference, uses more advanced structures, such FM-index, and takes advantage of additional properties of prefix trees. The permutations-based approach uses permuted strings, rather than the original reference, and does not depend on the prefix-tree data structure or properties.

3 Algorithm

3.1 An informal description of the algorithm

Permutations-based read search algorithms create a library of all possible contiguous substrings of length M of the reference string. These algorithms then use a permutation to reorder the characters in each of the strings, creating a library of permuted strings. All the strings in the library are permuted in the same way. This procedure in repeated several times, each time with a different permutation scheme, producing several different libraries. Ideally, there are several permutations, and the algorithms chooses a subset of these permutations and their corresponding libraries for each search. The permutations-based search algorithm proposed here, uses a prefix-tree data structure to organize each library of strings.

In the case we discuss in this manuscript, we are given the paired-end reads $Y1$ and $Y2$, and we would like to find appropriate mappings of these pairs to loci in the reference that are no more than G characters apart. We assume that the reads are of length M (the algorithm can be extended beyond uniform length).

When we are given the read $Y1$, we use the first permutation to create a permuted version of $Y1$. This is the same permutation used to create the first library of permuted strings, so we can now use the prefix-tree data-structure to find a small bin that contains a small list of candidate loci in the reference. If we are "lucky", the permuted version of $Y1$ does not contain a mismatch in the prefix, so the correct alignment is found in the set we reached in the tree search. We repeat the procedure several times, to obtain a high probability of being "lucky" in at least some of the iterations.

At this point, we do not attempt to check the quality of the mapping using any refined score. Instead, we store a list of the locations that were proposed as candidates, and count the number of "hits" that each of these locations had.

We then repeat the same procedure for $Y2$, again, only storing a list of candidates and a "hit-count".

The idea behind storing lists of candidates and hit counts, is that the correct mapping for each string is very likely to have several hits, whereas many of the incorrect candidates are likely to have only a single hit. Storing the number of hits, allows us to filter out the unlikely candidates and focus our attention on a considerably smaller list of candidates for which the number of hits exceeded some threshold.

We now have a shortlist of candidate mappings for $Y1$ and candidate mappings for $Y2$. We look for candidates for $Y1$ and $Y2$ which are up to G apart in the reference. This produces a very short list of possible mappings for $Y1$ and $Y2$. We report the valid candidates.

3.2 A more formal description of the algorithm

3.2.1 Search and filter:

We randomly choose J permutations: $\{U_j\}$. For each permutation U_j and the corresponding library of permuted reference strings Lib_j, we create a prefix-tree $Tree_j$. This indexing procedure is done in advance. We denote the nodes of the tree by $Tree_j[node\_id]$, the list of strings in a node by $Tree_j[node\_id].candidates$, and 4 children of each node by $Tree_j[node\_id].child[A]$, $Tree_j[node\_id].child[C]$, $Tree_j[node\_id].child[G]$, $Tree_j[node\_id].child[T]$. There is no need to store all the libraries of strings, because we can generate them from the reference when we need them.

This indexing is done in advance. We then use the mapping algorithm, which generates multiple lists of candidates, one list for each permutation, and filters them to create a very short list of "good" candidates. The procedure for generating a list of candidates for each permutation using the trees created in advance, is described after the main algorithm.

In the following description of the main algorithm, $HitCount1$ counts the hits for string $Y1$ in each locus and $HitCount2$ counts the hits for string $Y2$ in each locus. $Hit1OverTH$ is used to indicate whether there have been more than $Hit1Threshold$ hits for $Y1$ within G reference positions. $BothOverTH$ is used to label areas where there are good candidates for mapping for both reads. The hit-counters and "flag" arrays may be implemented using modified array structures that simplify the initialization and queries.

Algorithm 1

```
Map (Y1,Y2):

    Initialize HitCount1 to 0
    Initialize HitCount2 to 0
    Initialize Hit1OverTH to FALSE
    Initialize BothOverTH to FALSE

    Randomly select J permutations {U_j} of the
    available indexed permutations.

    For each permutation
        Propose candidate mappings for Y1 in Tree_j
        For each candidate
            Increment HitCount1[cand]
            If HitCount1[cand] > Hit1Threshold
               Hit1OverTh[cand − G..cand + G] = TRUE
            End If
        End For
    End For

    Randomly select J permutations {U_j} of the
    available indexed permutations.

    For each permutation
        Propose candidate mappings for Y2 in Tree_j
        For each candidate
            Increment HitCount2[cand]
            If HitCount2[cand] > Hit2Threshold
               BothOverTH[cand − G..cand + G] = TRUE
            End If
        End For
    End For

    Make a short list of candidates for Y1 and Y2 in positions where
        BothOverTH is TRUE.

    Report valid candidates.
```

3.2.2 Creating lists of candidates:

The procedure for generating a list of candidates for the string Y using a permutation U_j and its corresponding $Tree_j$ begins with generating $Y^{(j)} = U_j(Y)$, the permuted version of Y. We then navigate $Tree_j$ from the root down. We stop traversing down the tree when we reach depth L_{Max} or find a node that contains fewer than K strings. The permuted strings in the node where we stop have the same prefix as $Y^{(j)}$. So all of these permuted strings have the first character $Y^{(j)}[0]$, the second character $Y^{(j)}[1]$ etc.

The lists created for each of the permutations are used in the main mapping function, described above.

Algorithm 2

`ProposeCandidates(`Y`,`U_j`,`$Tree_j$`):`

$Y^{(j)} = U_j(Y)$
`Set` $node = root$
`For` $l = 0$ `to` $L_{Max} - 1$
    `If` $|Tree_j[node].candidates| < K$
        `break`
    `End If`
    $node = Tree_j[node].child[Y^{(j)}[l]]$
`End For`

`Return (`$Tree_j[node].candidates$`)`

3.3 Extensions

The possible problem of storing a large number of indexes for different permutations is discussed in [8].

Basic permutations-based algorithms are designed to overcome mismatches. There are extensions that allow indels. One of the ways to extend this algorithm to allow a small number of indels in a single long read, or in pairs of reads, is to divide reads into several segments and permute each segment separately, using the pigeonhole principle to guarantee segments with no indels. Some of the other extensions have been demonstrated in [8, 9].

4 Implementation and results

We implemented the proposed algorithm in C. No SSE extensions are used. This version allows mismatches, but does not allow indels.

We used several software packages as benchmarks: BWA [4], BWASW [5], Bowtie2 [3] and CUSHAW2 [6]. Several sets of parameters were tested for each software package. Where possible, we used modes that penalize or do not allow indels. All experiments were done in single thread mode. BWASW does not support paired-end alignment, therefore, each read was aligned separately. CUSHAW2 was used in "have ssse3 = 0" mode, due to technical limitations of the servers. CUSHAW2 may be significantly faster when used on systems that support SSE4 extensions in addition to the SSE2 extensions that it used in our experiments. Our implementation does not use processor extensions.

We used wgsim [7] to generate paired-end reads of various lengths and mismatch rates.

The experiments were performed on nodes with (2) E5620 CPUs and $48GB$ RAM. This implementation has also been tested on a $\$500-600$ desktop computer with similar results (but slightly longer search time, as expected).

BWA requires about 2.8GB of RAM, BWASW requires 4.7GB, Bowtie2 requires 3.2GB and CUSHAW2 requires 3.7GB. Our implementation requires about 15GB of RAM for a human reference genome. It can also use more memory to increase the speed (not shown here). The algorithm can also be adapted to use less than 8GB RAM.

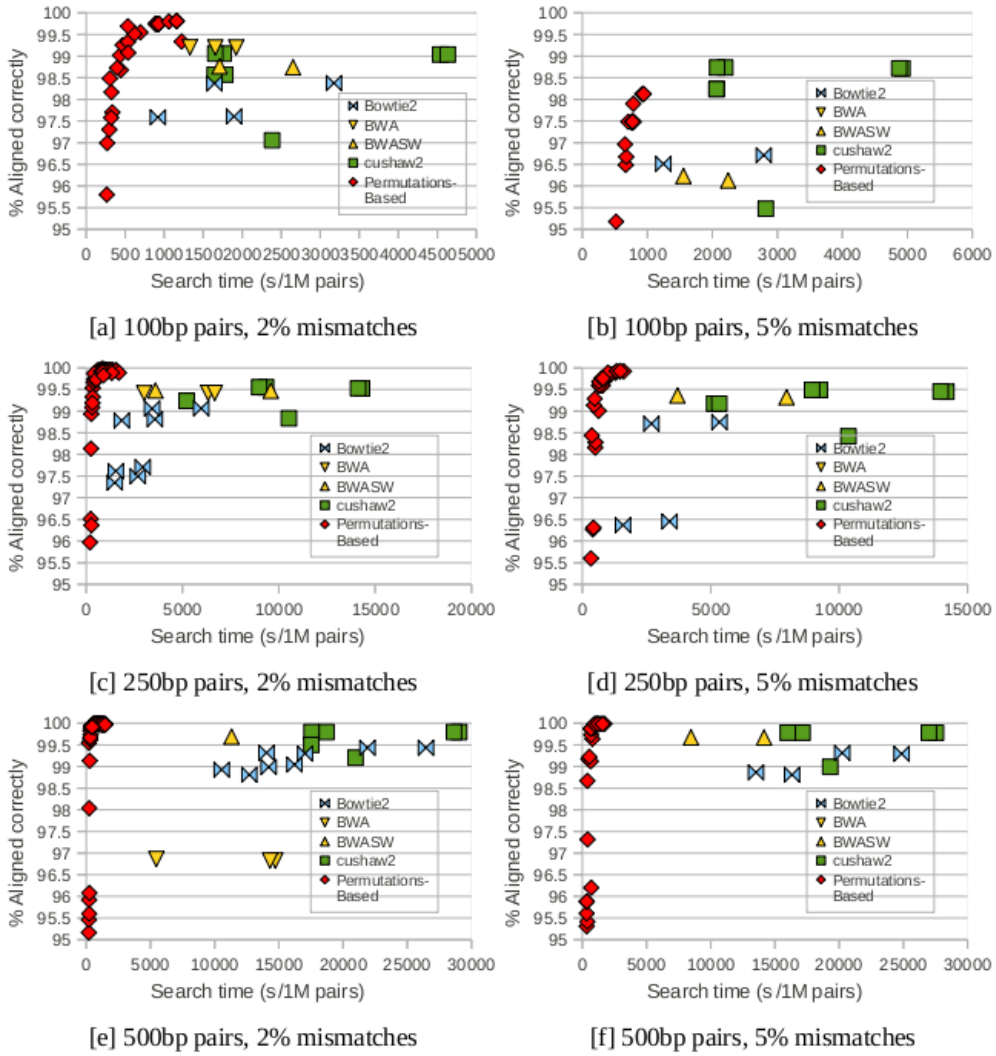
In order to have a single uniform definition that can be applied to all the software packages and their various modes, we considered a correct mapping of one of the reads to be sufficient and we also allowed some range around the correct locus.

The results are presented in figure 1.

The quality of the results of our implementation depends mostly on the number of search iterations (the parameter J). A very small number of iterations (as low as 5) is often enough to obtain the great majority of correct alignments with very high accuracy. A larger number of iterations (10-15) provides increased sensitivity at the expense of speed. A larger number of iterations ($20-30$ and up to 40) is required in order to obtain very high sensitivity in the presence of a large number of mismatches.

The fact that most reads are accurately mapped in very few iterations indicates that stopping heuristics can accelerate the implementation even more, while retaining a high level of accuracy.

Figure 1: Search times and number of reads mapped correctly



[a] 100bp pairs, 2% mismatches

[b] 100bp pairs, 5% mismatches

[c] 250bp pairs, 2% mismatches

[d] 250bp pairs, 5% mismatches

[e] 500bp pairs, 2% mismatches

[f] 500bp pairs, 5% mismatches

5 Conclusions

An algorithm for alignment of paired-end reads to a reference genome has been constructed. The algorithm is based on the permutations-based approach and a filter which reduces the number of incorrect candidates.

An implementation of the algorithm has been presented. Experiments conducted with this implementation suggest that it is faster, and aligns more reads than popular and recent fast alignment software packages. These properties of the implementation are maintained for higher mismatch rates and relatively long reads.

The underlying algorithm allows alignment of paired-end reads in the presence of multiple mismatches and can be extended to allow alignment of long single-end reads and paired-end reads in the presence of indels.

The "hit-counter" filters, and generalizations like the one described here, use manipulation of lists of candidates in order to reduce the number of direct examination of candidates. This manipulation of lists is much "cheaper" than manipulating the strings associated with each candidate. The same ideas can be used in other related algorithms, such as hash-tables based algorithms.

6 Acknowledgments

We would like to thank the Yale University Biomedical High Performance Computing Center (NIH grants RR19895 and RR029676-01) for providing some of the computation resources.

References

[1] Li, H. & Homer, N. *A survey of sequence alignment algorithms for next-generation sequencing.* Briefings in Bioinformatics 11, 473-483 (2010).

[2] Flicek, P. & Birney, E. *Sense from sequence reads: methods for alignment and assembly.* Nat Meth 6, S6-S12 (2009).

[3] Langmead, B. & Salzberg, S. L. *Fast gapped-read alignment with Bowtie 2.* Nat Meth 9, 357-359 (2012).

[4] Li, H. & Durbin, R. *Fast and accurate short read alignment with Burrows-Wheeler transform.* Bioinformatics 25, 1754 -1760 (2009). [PMID: 19451168]

[5] Li H. & Durbin R. *Fast and accurate long-read alignment with Burrows-Wheeler transform.* Bioinformatics. (2010) [PMID: 20080505]

[6] Liu, Y & Schmidt, B: *"Long read alignment based on maximal exact match seeds"*. Bioinformatics, 2012, 28(18): i318-324

[7] Li, H. *Wgsim* , https://github.com/lh3/wgsim (Accessed 05/2012)

[8] L, R. *A random-permutations-based approach to read alignment.* BMC Bioinformatics (2013), 14(Suppl 5) [doi:10.1186/1471-2105-14-S5-S8]

[9] L, R. *A nearest neighbors algorithm for strings.* Yale CS Technical Report 1453 (2012)