A randomized algorithm is presented for fast nearest neighbors search in libraries of strings. The algorithm is discussed in the context of one of the practical applications: aligning DNA reads to a reference genome.
An implementation of the algorithm is shown to align about 10^6 reads per CPU minute and about 10^8 base-pairs per CPU minute (human DNA reads). This implementation is compared to the popular software packages Bowtie and BWA, and is shown to be over $5-10$ times faster in some applications.

A Nearest Neighbors Algorithm for Strings

R. Lederman[†]
Technical Report YALEU/DCS/TR-1453
April 5, 2012

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

Approved for public release: distribution is unlimited.
Keywords: *Nearest Neighbors, DNA, alignment.*

1 Introduction

The use of genetic information for biological research has been rapidly growing due the growing needs in biological research, coupled with rapid technological advancements in acquiring such data. The classical process of producing raw genetic data, known as "sequencing", involves reading DNA sequences (and other types of sequences) by "sequencing machines". The output of current high-throughput "next-generation" sequencing machines is typically a very large number (tens of millions) of short strings of characters. Each of these strings, called "a read", represents a small part a of long DNA sequence. The DNA is composed of pairs of 4 nucleic acids, therefore the reads that represent parts of the DNA are strings composed of 4 possible characters/letters, A, C, G and T, each representing a different nucleic acid.

Different individual members of a species have different DNA sequences (moreover, each human has two versions of most of his or her chromosomes and, in fact, small variations may exist among the many cells in a human body). In the analysis of such differences, the following situation is often encountered: one has some "reference" strings of characters representing the "typical" genome of a species (a "typical genome") and one is interested in comparing the DNA of one individual member of the species to the reference. This is done by sequencing a sample of the DNA of that particular individual member and comparing the result to the reference.

Since the output of sequencing machines is given in the form of short strings, representing fragments of the DNA, the analytical process begins with deciding where each read "belongs". In other words, one estimates which interval in the reference strings corresponds to each of the reads. The process of estimating where each read "belongs" is known as "read alignment".

The typical approach to alignment is to take each of the reads and find a location in the reference where the the reference is very similar to the read. Many of the reads may be "perfectly aligned", meaning that they are identical to some substring of the in the reference. Because of the differences between individuals, it is clear that not all the reads

can be expected to be perfectly aligned to the reference. In addition to the differences caused by variations between individuals ("true variations"), there are often some errors in the sequencing process ("noise"), resulting in additional differences between reads and the intervals in the reference where these reads "belong".

In most of this paper, we restrict our attention to a specific type of difference and to a particular measure of similarity. When we consider two strings (a read and some substring of the reference), we say that the similarity between them is the number of positions where the two strings are identical or that the distance between two strings is the number of positions in which they differ ("Hamming distance"). The positions where the strings do not match are known as mismatches. Although considering mismatches is sufficient for many applications, there are other important variations ("indels": insertion or deletion of characters that change the relative positions of other characters). We will demonstrate some of the ways in which the Hamming distance alignment can be used to analyze other types of similarities in practice.

There are several software packages which are widely used to analyze the large volumes of sequenced DNA generated nowadays. These software packages use two major classes of algorithms: a) prefix/suffix-trees/tries and b) hashing [1, 2]. In this paper, we will introduce a new approach which may lead to faster algorithms and to software that can be customized for specialized research requirements.

2 Preliminaries

In this subsection we present a formulations of the problem and basic algorithms which we use in our algorithm. We also introduce notation and terminology which we will use throughout the reminder of the paper.

2.1 Basic notation

A string is defined as a "tuple"[1] of characters of the alphabet $\mathcal{A} = \{A, C, G, T\}$. We use uppercase letters (with indices, when needed) to denote these strings, for example:

[1] Here, a list/array of characters or numbers in a specific order.

Y or $Y^{(i)}$, when an index is required. When we wish to refer to a specific character in a specific position l in the the string, we use the lowercase version of the string's name and write y_l or $(y_l^{(i)})$.

When we wish to write a string of length M in a "tuple form", we write: $(y_1 y_2 ... y_M)$. This form is used in order to reference the order of characters. For example, if we have a string Y, which is composed of characters z_1, z_2, z_3 in reverse order, we write $Y = (y_1 y_2 y_3) = (z_3 z_2 z_1)$ which means that $y_1 = z_3$ etc.

Summing this up:

$$Y = (y_1 y_2 ... y_M) \;\; where \;\; y_j \in \mathcal{A} = \{A, C, G, T\}. \tag{1}$$

In examples, when we want to discuss a specific string, and express its content explicitly, we use the letters of the alphabet explicitly. For example: $Y = AGT$ is a string of length 3 with $y_1 = A,\ y_2 = G,\ y_3 = T$.

In some examples, when we write specific strings explicitly, we may use a lowercase version of letters from the alphabet. For example, we may write $Y = AgT$. This is done only for the purpose of highlighting that specific location in the string (for example, in order to emphasize that the string is different from some other string in that location). For any other purpose: $Y = AGT = AgT$.

Similar notation is used to describe "tuples"/ arrays/ lists of numbers. So, an array U, of M integer numbers in the range 1 *to* N is written as:

$$U = (u_1 u_2 ... u_M) \;\; where \;\; u_j \in \{1, 2, ..., N\} \ . \tag{2}$$

An array of numbers can also be written explicitly. For example: $U = (3, 4, 2, 7, 1)$.

2.2 DNA read alignment

We will investigate the following scenario: we are given a long reference string (the reference DNA) and a large number of short reads/query strings. For each of the reads, we would like to find an interval/substring of the reference that is the most similar to the query.

For simplicity, we will assume that all of our reads are strings of the same length. We denote the length of the reads by M (the actual values varies in application. $M \approx 100$ can serve as order of magnitude). We denote a read with an uppercase letter, such as Y. This string has M characters/letters, denoted by lowercase letters and indices: $y_1, y_2, ...y_M$. Each character is chosen from the alphabet $\mathcal{A} = \{A, C, G, T\}$. In other words:

$$Y = (y_1 y_2 ... y_M) \quad where \quad y_j \in \mathcal{A} = \{A, C, G, T\}. \tag{3}$$

We denote the long reference string, which represents the entire reference genome, by W. We denote the length of the long reference string (W) by N (for example, for the human genome we have $N \approx 6 \times 10^9$ when both strands are considered). Again, the characters of the string are denoted by indexed lowercase letters $w_1, w_2, ...w_N$, and we write

$$W = (w_1 w_2 ... w_N) \quad where \quad w_j \in \mathcal{A}. \tag{4}$$

We observe that the original long reference string contains $N - M + 1$ overlapping M-long intervals/substrings.

Definition 2.1 *We denote a substring of M consecutive characters, that begins at the jth position of the long reference DNA (W), by $X^{(i,M)}$ (or simply $X^{(i)}$ when the value of M is obvious from the context). Therefore, we have:*

$$X^{(i,M)} = (x_1^{(i,M)} x_2^{(i,M)} ... x_M^{(i,M)}) = (w_i w_{i+1} ... w_{i+M-1}). \tag{5}$$

This collection of substrings $\{X^{(i,M)}\}_{i=1}^{N-M+1}$ is referred to as the collection/library of reference strings.

The notation introduced in this section allows us to phrase our alignment problem as follows: given a read Y, find the most "similar" string $X^{(i)}$ in the reference collection. Below, we define the "Hamming distance", a concept of "similarity" that is often used to compare strings.

2.3 Hamming distance and the nearest neighbors problem

Definition 2.2 *The Hamming distance between two strings is defined as the number of positions where the strings differ. The distance between the string X and the string Y is denoted by $d_H(X, Y)$. Therefore, we have:*

$$d_H(Y, X) = |\{m : y_m \neq x_m\}| = \#Mismatches. \tag{6}$$

We will use this distance to measure the similarity between strings (smaller Hamming distance means higher similarity). So, our goal is to take each query string, and find the reference string that is closest to it in the Hamming sense.

For a query string Y of length M, we are looking for the "most similar" reference string $X^{(i,M)}$. So, $X^{(i,M)}$ is the reference string that has the smallest Hamming distance to our query Y. In other words, we are looking for i, the location in the long reference DNA (W), in which there is a substring ($X^{(i,M)}$), that is the most similar to the query string.

For simplicity, we assume that each query string has only one unique closest reference string (referred to as the "true nearest neighbor").

Summing this up using the notation that we introduced:

$$d_H(Y, X^{(i,M)}) = min_{i'}(d_H(Y, X^{(i',M)})), \tag{7}$$

i.e

$$i = argmin_{i'}(d_H(Y, X^{(i')})), \tag{8}$$

under the assumption that there are no two distinct locations $i \neq j$ such that $d_H(Y, X^{(j)}) = d_H(Y, X^{(i)}) = min_{i'}(d_H(Y, X^{(i')}))$.

This problem is a member of the class of "nearest neighbor search" (NN) problems. In the general NN problem, one has a collection of some arbitrary items and some measure of distance between them. Given a query item, one looks for the most similar item in the collection. In our case, we have a NN problem where the items are strings, and the distance between the items is the Hamming distance.

2.4 Operations on strings

In our discussion, we use several operations on strings. Here, we define the basic operations.

One operation which we often use in this discussion is extracting prefixes of strings.

Definition 2.3 *The l long prefix of a string Y of length $M > l$ is defined as a string of length l which is identical to the "beginning of" Y. We denote taking a prefix by: $Prefix(l, Y)$. Therefore, we have:*

$$Prefix(l, Y) = (y_1 y_2 ... y_l). \tag{9}$$

Another operation we use is "permutation" or "shuffling", in which we reorder the characters in the string.

Definition 2.4 *We define a "permutation pattern" $U^{(j)}$ of length M or a permutation code of length M as an array/ list of distinct numbers: $U^{(j)} = (u_1^{(j)} ..., u_M^{(j)})$, where $u_k^{(j)} \in \{1...M\}$ and $k' \neq k'' \Longrightarrow u_{k'}^{(j)} \neq u_{k''}^{(j)}$.*

A permutation pattern includes all the numbers from 1 to M, in some defined order.

We use the permutation pattern $U^{(j)}$ to define how the charecter in the string Y should be reordered when we produce a new, permuted, string.

Definition 2.5 *"Permuting"/ "shuffling" a string Y using the permutation code $U^{(j)}$ (both of length M), or "applying the code to the string", is defined as creating a new string, with the characters of Y moved ("permuted"/"shuffled") to new positions:*

$$T(U^{(j)}, Y) = (y_{u_1^{(j)}} y_{u_2^{(j)}} ... y_{u_M^{(j)}}). \tag{10}$$

2.5 Sorted arrays

Given an array Ar, we denote the item in location number i by $Ar[i]$[2]. The item may have several different "properties", we denote the the property "prop" of the item i in array Ar by $Ar[i].prop$.

[2]Indeed, we already defined notation for strings and arrays of numbers, but the notation introduced here is used in a different context, for more general arrays of "items" that are not simply letters of an alphabet or integer number in some range.

Definition 2.6 *Suppose we have a definition of order*[3] *between the "key" property of items. Sorted arrays are arrays in which items' locations are determined according to the ordering relation.*

If i, j are indices for items in the sorted array, and $i < j$, then the item in location i of the array is "smaller than, or equal to" the item in location j: $Ar[i].key \geq Ar[j].key$.

There are many algorithms that can be used to sort arrays based on the order of the keys. "Quicksort" and "Heapsort" are examples of fast sorting algorithms[3].

2.6 Lexicographically sorted arrays

In this discussion, we are interested in strings. We discuss arrays in which we store the reference strings (the library of substrings of the DNA, $\{X^{(i)}\}$, or some variations of these strings, as we explain later). The location in the DNA from which we extracted the string is stored along with the string. So, every item in our arrays has two properties: the "key" (which is the string, because our arrays are sorted according to the strings, as we describe below) and the "DNALocation". For brevity, we say that the items in the arrays are strings. So, we say that the item $Ar[i]$ in array Ar is some string, when, in fact, we mean that $Ar[i].key$ is some string.

Lexicographical[3] order of strings is defined as the strings' alphabetical order, or order in which the strings would appear in the dictionary. More formally:

Definition 2.7 *We define an order between letters of the alphabet: $A < C < G < T$.*

Definition 2.8 *Given two strings of length M, we say that string Z is "larger than" string "Y" if there is some $i \leq M$ such that $z_j = y_j$ for all $j < i$ and $z_i > y_i$. We denote "Z larger than Y" by $Z > Y$. We say that Z is smaller than Y (denoted by $Z < Y$) if $Y > Z$. And we say that Z is equal to Y (denoted by $Z = Y$) if the strings are identical.*

These rules for ordering strings are called "Lexicographical/ Lexical order".

[3]Whenever we say "order" in this paper, we refer to "total order". The "orders" that we will define can easily be shown to be "total orders". A definition of "total order" is available, for example, in [3].

Table 1: Part of a sorted array of strings

Location in table	Location in DNA	String/ the key									
		1	2	3	4	5	6	7	8	9	10
...											
9990	16103	C	T	T	A	T	G	A	G	T	G
9991	1516	C	T	T	A	T	G	G	A	A	A
9992	367	C	T	T	A	T	G	G	G	G	C
9993	2888	C	T	T	A	T	T	A	T	C	G
9994	7726	C	T	T	A	T	T	C	G	C	G
9995	12730	C	T	T	A	T	T	T	C	C	C
9996	13689	C	T	T	C	A	A	A	A	C	C
9997	5228	C	T	T	C	A	A	A	A	G	C
9998	12948	C	T	T	C	A	A	A	T	G	C
9999	8260	C	T	T	C	A	A	C	A	G	A
10000	17053	C	T	T	C	A	A	C	C	A	A
10001	6780	C	T	T	C	A	A	C	G	C	C
10002	254	C	T	T	C	A	A	C	T	C	G
10003	20145	C	T	T	C	A	A	G	C	C	G
10004	2549	C	T	T	C	A	A	G	G	C	G
10005	313	C	T	T	C	A	A	T	C	T	G
10006	7223	C	T	T	C	A	C	A	C	C	A
10007	11716	C	T	T	C	A	C	C	G	T	A
10008	19655	C	T	T	C	A	C	G	A	T	G
10009	1745	C	T	T	C	A	C	T	T	G	G
10010	5078	C	T	T	C	A	G	A	A	G	T
...											

A part of a sorted array of some strings of length $M = 10$. In this example, the library of strings was generated from a "simulated" string of randomly selected characters. Different items (strings) appear in different rows.

For example, item number 1000 is the string $CTTCAACCAA$, which is a substring of the reference DNA at location 17053.

2.7 Binary search

Definition 2.9 *Binary search is defined as an operation that takes a sorted array of strings Ar and a query string $Query$ and returns two integer numbers[4]: low and $high$ such that $Ar[high] \geq Query$, $Ar[low] \leq Query$ and $(high - low)$ is minimal. We denote this operation by:*

$$[low, high] = BinarySearch(Query, Ar). \tag{11}$$

A description of the Binary search algorithm can be found, for example, in [3]. Throughout the paper, we assume that all of the strings in our sorted lists are unique. Therefore, BinarySearch has a unique possible output. When there is a string in Ar which is identical to $Query$, we have that $low = high$ and $Ar[low].key = Query$. When no such string exists in Ar, we have $low + 1 = high$.

In our example (table 1): $[low, high] = BinarySearch(CTTCAACCAA, Ar)$ would give us $low = high = 1000$. However $[low, high] = BinarySearch(CTTCAACCAc, Ar)$ would give us $low = 1000,\ high = 1001$.

2.8 The "neighborhood" of a string in a sorted array

We now discuss relationships between strings and their "neighborhoods" (other strings which are located in close proximity in the sorted array of strings).

Definition 2.10 *The "unique prefix length"[5] of string Y in the collection reference of strings $\{X^{(i)}\}$ is the length of the shortest prefix of Y which is unique to "Y". We denote this by $UniquePrefixLength(Y, \{X^{(i)}\})$.*

If $l = UniquePrefixLength(Y, \{X^{(i)}\})$, and we build a collection of prefixes $\{Prefix(l, X^{(i)})\}$, for all the strings in the collection, there will be no more than one string $(Prefix(l, X^{(i)}))$

[4]Here, and in other places in the paper some care has to be taken at the ends of the array. We omit the trivial corrections required.

[5]The unique prefix length is sometimes referred to as the "Z" of a string.

in this collection of prefix strings that is equal to the prefix of Y ($Prefix(l, Y) = Prefix(l, X^{(i)})$).

In our example (table 1), the unique prefix length of the string $ATGCAGTTAC$ is 8.

We now extend the concept of "unique prefix length" by introducing the concept of "search resolution size".

Definition 2.11 *We say that a search for string Y in a library of strings $X^{(i)}$ is "resolved up to K 'suspects' in prefixes of length l", if there are no more than K strings in the library that share the same prefix of length l with Y, and l the shortest prefix length for which no more than K strings share the prefix.*

We denote this by $l = ResolutionLength(K, Y, \{X^{(i)}\})$. We refer to the value of K as the "desired resolution size".

In other words:

$$l = argmin_{l'}\{|\{i : Prefix(l', X^{(i)}) = Prefix(l', Y)\}| \leq K\} \ . \tag{12}$$

When the value of the "desired resolution size" (K) and the collection of strings are obvious from the context, we simply say that "the resolution length of Y is l". Note that when the desired resolution size is 1, the resolution length is the unique prefix length.

In our example (table 1), the search for the string $ATGCAGTTAC$ is resolved up to 10 in prefixes of length 6.

Definition 2.12 *We consider a string Y, a library of strings $\{X^{(i)}\}$ (all of the same length, M) and a "desired resolution size" of K. If Y has resolution length of l, the strings that have the same prefix of length l as Y are called the resolution set. We denote these strings by $ResolutionSet(K, Y, \{X^{(i)}\})$. When the collection of strings is stored in the sorted array Ar, we can also write $ResolutionSet(K, Y, Ar)$. So we have that:*

$$\begin{aligned} &X^{(i')} \in ResolutionSet(K, Y, \{X^{(i)}\}) \Leftrightarrow \\ &Prefix(ResolutionLength(K, Y, \{X^{(i)}\}), X^{(i')}) = \\ &Prefix(ResolutionLength(K, Y, \{X^{(i)}\}), Y). \end{aligned} \tag{13}$$

The resolution can be found by performing a binary search for Y (i.e $[low, high] = BinarySearch(Y, Ar)$) and checking the neighborhood around locations low and $high$ in the sorted array.

In our example (table 1), $ResolutionSet(10, CTTCAACCAA, Ar)$ gives the strings in rows 9996-1005.

$$\text{Set } Suspects(Y, j_\alpha) = \{Ar^{(j)}[low - K], ..., Ar^{(j)}[high + K]\}$$

2.9 Observation: searches for perfect matches and searches for imperfect matches

We observe that the search scheme described above is adequate for searches for "perfect matches". Given a sorted array of strings, and a query string that has a perfect match in the array, we can use the binary search method to find the perfect match.

In this discussion, we are interested in a search for the "true nearest neighbor", a string which is very similar, but not identical to our query string. We should, therefore, consider whether this scheme is also adequate in this case.

Suppose that the mismatches (the positions in the query string where it differs from its true nearest neighbor) are located very close to the end of the query string. Now suppose that instead of using the regular binary search, we look for some "resolution set" for the search. If we are "lucky", the resolution length of our query is short and the search is "resolved" in prefixes that do not include the mismatch. In this case, we can simply compare all the strings in the "resolution set" to our query and find the true nearest neighbor.

Clearly, we cannot expect to always be this "lucky". For example, the mismatches may occur in the first character of the string, making it impossible for us to use this simple type of search.

In our example (table 1), suppose that we are given the string $CTTCAACCAt$ (for which the string in DNA location number 17053 (and array location 10000), $CTTCAACCAA$ is the true nearest neighbor, with one mismatch in the last character). The string is

clearly in the resolution set (assuming desired resolution size is of 4, for example). However, if we are given the string $tTTCAACCAA$ (for which the string in DNA location 17053 is also the true nearest neighbor), we cannot use the resolution set approach to find the nearest neighbor.

3 The search algorithm

3.1 Informal description of the algorithm

Our proposed algorithm creates sorted arrays of permuted versions of the strings in the reference collection. In other words, we have several collections of permuted strings, each built by applying some permutation to all of the strings in our reference library and we build a sorted array for each of the new libraries of permuted reference strings.

We denote a collection of permuted strings, created by applying the permutation $U^{(j)}$ to the collection of reference strings $\{X^{(i)}\}$, by $\{T(U^{(j)}, X^{(i)})\}_{i=1}^{N-M+1}$. The corresponding sorted array is denoted $Ar^{(j)}$. this sorted array contains all of the strings in the collection $\{T(U^{(j)}, X^{(i)})\}_{i=1}^{N-M+1}$.

When we get a new query string, we first permute it, using the same permutation code we applied to the reference strings, so Y is replaced by the permuted version $T(U^{(j)}, Y)$. We note that both the strings in our reference library and the query string are permuted using the same code. Consequently, the Hamming distance between the permuted strings is the same as the Hamming distance between the original strings (which is also the number of mismatches):

$$\#Mismatches = d_H(Y, X^{(i)}) = d_H(T(U^{(j)}, Y), T(U^{(j)}, X^{(i)})) \tag{14}$$

After permuting the query string, we generate the "resolution set" for the permuted query string with respect to the collection of permuted reference strings. We call the strings in this resolution set "candidates" or "suspects".

We repeat this process several times with several different permutations (thus with several different libraries of permuted strings and several different sorted arrays), producing several lists od "suspects".

Finally, we can examine all the possible "suspects"/ "candidates" and report the best alignment among them.

Some (perhaps most) of the searches in the sorted array lead us only to incorrect reference strings. However, since we repeat the process several times, it is very likely that in one of our repetitions we will encounter a case where the mismatches are shuffled away from the beginning of the string, leading to a "suspects" list that includes the true nearest neighbor. (an analysis of the probability is presented below).

An example of a sorted array for the same collection of strings considered in table 1 is presented in table 2.

3.2 A more formal description of the algorithm

We now describe the algorithm more formally. First, we describe an indexing procedure (part 1). Then we describe the search for possible neighbors (part 2). Finally we describe an approach for evaluating the proposed neighbors (part 3).

3.2.1 Part 1: Index creation

Create a family of J random permutation codes $\{U^{(j)}\}_{j=1}^{J}$.

For each permutation code $U^{(j)}$:

Build a sorted array $Ar^{(j)}$ for the permuted strings $\{T(U^{(j)}, X^{(i)})\}_{i=1}^{N-M+1}$.

Store permutation code $U^{(j)}$ and "index" $Ar^{(j)}$ for use in part 2.

End For

3.2.2 Part 2: Listing "possible neighbors"/ "suspects"

For a query string Y:

Initialize $Suspects(Y, j) = \emptyset$ for all $j \in \{1..J\}$.

Table 2: Part of a sorted array of permuted strings

Location in table	Location in DNA	Permuted string / the key Position in permuted string (and original position)									
		1 (8)	2 (1)	3 (7)	4 (10)	5 (9)	6 (2)	7 (6)	8 (3)	9 (5)	10 (4)
...											
6658	11553	C	C	C	A	A	C	C	T	A	C
6659	18520	C	C	C	A	A	C	G	G	G	C
6660	14262	C	C	C	A	A	C	G	T	C	C
6661	12457	C	C	C	A	A	C	G	T	G	G
6662	17106	C	C	C	A	A	C	T	T	T	C
6663	9220	C	C	C	A	A	G	A	C	T	A
6664	13293	C	C	C	A	A	G	C	A	G	C
6665	17743	C	C	C	A	A	G	C	C	A	G
6666	9594	C	C	C	A	A	G	C	C	C	C
6667	11490	C	C	C	A	A	G	T	A	G	A
6668	17053	C	C	C	A	A	T	A	T	A	C
6669	11122	C	C	C	A	A	T	A	T	G	C
6670	18663	C	C	C	A	A	T	C	A	G	A
6671	12745	C	C	C	A	A	T	C	G	A	G
6672	6112	C	C	C	A	A	T	G	C	G	C
6673	20084	C	C	C	A	A	T	T	T	G	T
6674	240	C	C	C	A	C	A	A	A	G	G
6675	11987	C	C	C	A	C	A	G	T	C	A
6676	15131	C	C	C	A	C	A	T	C	A	G
6677	13914	C	C	C	A	C	A	T	C	A	T
6678	17742	C	C	C	A	C	C	A	G	G	C
...											

A part of a sorted array of permuted DNA strings of length $M = 10$.
We applied the permutation code $(8, 1, 7, 10, 9, 2, 6, 3, 5, 4)$ to the same library of strings used in the previous table.
The neighborhood of the permuted version of string number 17053 is presented again. The permuted string is located in a different place in the array, and has different neighbors.

Randomly choose $J_1 \leq J$ codes ($\{U^{j_\alpha}\}_{\alpha=1}^{J_1}$) from the J permutation codes.

For each permutation code U^{j_α}:

Set $Suspects(Y, j_\alpha) = ResolutionSet(K, Y(U^{(j_\alpha)}, Y), Ar^{(j_\alpha)})$

End For

End For

3.2.3 Part 3: Filter/ Analyzing suspects

For a query string Y:

For every possibility reported in some list

(i.e $\bigcup_j \{i : T(U^{(j)}, X^{(i)}) \in Suspects(Y, j)\}$)

Calculate the distance to the query string $d_H(Y, X^{(i)})$.

Keep track of the closest string.

End For

Report closest reference string as the alignment for Y.

End For

3.3 Analysis

In this section, we discuss the probability of obtaining the correct result ("true nearest neighbor") for a query read under some simplifying assumptions. We also discuss some of the properties of the read and the reference library that influence the probability of obtaining the correct result.

First, we present the problem and our assumptions. We then calculate the probability that the true nearest neighbor is in some specific list of "suspects" created in part 2 of the algorithm. Based on this calculation, we calculate the probability that the true nearest neighbor is in at least one of the lists of suspects. Finally, we argue that the condition

for correct output is that the true nearest neighbor is in at least one of the lists. We conclude that the probability which we calculated for this event is the probability of success.

Suppose that we have some query string, which has a true nearest neighbor in our reference library. We would like to calculate the probability of the algorithm returning the correct answer, which is the probability of finding the true nearest neighbor for the query string (i.e the probability of correct output).

We make the following assumptions (for all of the query strings):

- All the query strings are of length M.
- The true nearest neighbor is unique (there is no other reference string with the same distance to the wuery string).
- The true nearest neighbor is resolved up to K suspects in prefixes of identical length for any permutation code we use (the resolution length is constant, independent of the permutation code).
- We assume that we have "random permutations" in the following sense: There are $M!$ possible permutations for a list of M numbers. The permutations that we use are chosen randomly from among these $M!$ possibilities with equal probabilities.

In order to calculate the probability of success, we consider a particular query: The query sting is Y. The true nearest neighbor in the reference library, $X^{(i)}$ is resolved in prefixes of length L. The hamming distance between the query string and the true nearest neighbor (the number of mismatches) is p. We perform a search using J_1 random permutation codes.

Definition 3.1 *We denote the probability of success in this case by* $PrSuccess(p, L, M, J_1)$.

We now consider a list of suspects created in part 2, in the iteration that used the code $C^{(j_\alpha)}$ and the corresponding sorted list of permuted strings $Ar^{(j_\alpha)}$.

Definition 3.2 *The list of mismatches,* $MismatchPos(Y, X^{(i)})$*, is defined as the set of positions where the strings* Y *and* $X^{(i)}$ *differ:*

$$m \in MismatchPos(Y, X^{(i)}) \Leftrightarrow y_m \neq x_m^{(i)}. \tag{15}$$

As we mentioned, if the mismatches are permuted "far enough" from the beginning of the string, the nearest neighbor can be found in the resolution set. We now state this more formally:

Lemma 3.1 *For the permutation code* $U^{(j_\alpha)} = (u_1^{(j_\alpha)}..., u_M^{(j_\alpha)})$*, if the first* L *numbers in the list* $(u_1^{(j_\alpha)}..., u_L^{(j_\alpha)})$ *are not in the list of mismatches, then* $X^{(i)}$ *is in the* j_α *list of suspects.*

$$\{u_m^{(j_\alpha)}\}_{m=1}^{L} \cap MismatchPos(Y, X^{(i)}) = \emptyset \tag{16}$$

$$\Rightarrow T(C^{(j_\alpha)}, X^{(i)}) \in Suspects(Y, j_\alpha)$$

Proof. According to our assumption, the search is resolved in prefixes of length L, so, by definition, $ResolutionSet(K, T(C^{(j_\alpha)}, Y), Ar^{(j_\alpha)})$ includes all the strings that have the same L-long prefix as $T(C^{(j_\alpha)}, Y)$.

We consider the permuted versions of Y and $X^{(i)}$: $T(C^{(j_\alpha)}, Y)$ and $T(C^{(j_\alpha)}, X^{(i)})$. We observe that if $\{u_m^{(j_\alpha)}\}_{m=1}^{L} \cap MismatchPos(Y, X^{(i)}) = \emptyset$, then the first L characters of both permuted strings are identical:

$$Prefix(L, T(C^{(j_\alpha)}, Y)) = Prefix(L, T(C^{(j_\alpha)}, X^{(i)}))\ . \tag{17}$$

Therefore, the permuted string $T(C^{(j_\alpha)}, X^{(i)})$ is in the list.

□

So, the probability that $Suspects(Y, j_\alpha)$ includes the true nearest neighbor is the probability of not permuting any of the mismatches into one of the first L positions in the permuted string. We now calculate this probability.

Definition 3.3 *We denote the probability that none of the p mismatches in a string of length M is permuted into the first L position by* $PrLuckyPerm(p, L, M)$.

Lemma 3.2 *The probability that none of the p mismatches in a string of length M is permuted into the first L position is given by:*

$$PrLuckyPerm(p, L, M) = \frac{\frac{(M-L)!}{(M-L-p)!}(M-p)!}{M!} \tag{18}$$

Proof. We use a combinatorial argument: $\frac{(M-L)!}{(M-L-p)!}$ is the number of ways we can place p particular mismatches in the $(M-L)$ positions that are not in the prefix. Therefore, $\frac{(M-L)!}{(M-L-p)!}(M-p)!$ is the number of "lucky" permutations, that permute p given mismatches away from the prefix. Finally, we assumed that our permutations are chosen from among all $M!$ permutations with equal probabilities of $1/M!$, so the above expression gives the probability of choosing a "lucky" permutation. □

We combine these lemmas to calculate the probability that at least one of the lists contains the true nearest neighbor.

Theorem 3.3 *The probability that at least one of the suspects lists contains the true nearest neighbor is given by:*

$$1 - (1 - PrLuckyPerm(p, L, M))^{J_1}. \tag{19}$$

Proof. We have the probability for being "unlucky" in any one experiment: $1 - PrLuckyPerm(p, L, M)$.

We observe that our choices of permutations are independent, therefore the the event that we "get lucky" using a permutation j_α is independent from the event that we "get lucky" with another permutation $j_\beta \neq j_\alpha$ (In fact, independence requires us to allow ourselves to choose the same code more then once. The required corrections are small and increase the probability of success. We omit the these corrections).

Due to this independence, the probability of being "unlucky" in all of the experiments is $(1 - PrLuckyPerm(p, L, M))^{J_1}$. So the expression above gives the probability of "being lucky" at least once. □

Theorem 3.4 *If the reference string is in one of the lists of "candidates"/ "Suspects" calculated in part 2 of the algorithm, then the result of the algorithm is correct.*

Proof. If $X^{(i)}$ is in one of the lists of suspects, we calculate the distance to it in part 3 of the algorithm. The only way we do not report $X^{(i)}$ as the output of the calculation is if there is some other string, $X^{(i')}$ which is closer (or perhaps as close) to Y. However, there is no such reference string by the assumption. □

Therefore, it is enough to consider the probability that $X^{(i)}$ is in at least one of the lists of suspects. We already have an expression for this probability:

Corollary 3.5 *The probability of obtaining the correct output is given by the expression:*

$$PrSuccess(p, L, M, J_1) = 1 - (1 - PrLuckyPerm(p, L, M))^{J_1} \ . \qquad (20)$$

Note that according to this simplified analysis, the probability of success in finding the true nearest neighbor for some string of length M depend on the properties of its nearest neighbor with respect to the reference library (the nearest neighbor's resolution length, L), the number of permutation codes that we use (J_1), and the number of mismatches (p).

3.3.1 Example: alignment of multiple reads

We consider a reference collection of $N = 6 \times 10^9$ strings of length $M = 100$.

Suppose that most of the strings are resolved up to 10 "suspects" in prefixes of length 20. We refer to these strings as "group A".

Now, suppose that 5% of the strings have 999 other strings that share their 20-long prefix in any permutaion (resolved up to 1000 "suspects", but not up to 999 "suspects").

These strings are resolved up to 10 "suspects" in prefixes of length 50. We call these "group B".

We want to find the nearest neighbors for reads (query strings) for which the true nearest neighbor is in group A with a Hamming distance of up to 4 (up to 4 mismatches). In the "more difficult" case, where the nearest neighbor is in group B, we are satisfied with finding the correct nearest neighbor only for query strings that have no more than 1 mismatch (they have have a nearest neighbor of group B with hamming distance of no more than 1).

The algorithm described here produces the correct result with probability of more than 99%, using 9 permutation codes codes. In other words, under the assumptions, the algorithm reports the correct nearest neighbor for a query string if that nearest neighbor is in group A (and has a distance of 4 to the query string) or in group B (and has a distance of 1) with a probability of more than 99%. Higher probabilities can be obtained by using additional codes.

Note that when reads have more mismatches, the true nearest neighbor may also be found, but the probability of success is lower.

4 Implementation and results

4.1 A shuffling aligner

We implemented a version of the algorithm in order examine the performance with actual data.

We used the "Bowtie"[4] software package as a benchmark for performance evaluation. "Bowtie" is among the fastest popular alignment tools[4]. Similarly to our implementation, "Bowtie" does not allow indels. "Bowtie" was tested in the following modes: -v 3 -t -S (up to 3 mismatches) / -v 2 -t -S (up to 2 mismatches) / -n 3 -t -S (up to 3 mismatches in the "seed". Default seeds).

Very often, it is desirable to find multiple possible alignments for each read (equally good alignment and suboptimal ones) for alignment quality estimation etc. Our implementation reports multiple suboptimal alignment alignments (with the exception of

perfect matches) because these suboptimal alignments are generated in the process of finding the best alignment. Our implementation also has a "suboptimal match reporting version" where it reports suboptimal alignments with higher probability (up to 15 for each of the two read directions). The "Bowtie" software has several reporting modes, in the default mode it reports only one alignment. For each of the modes described above, we invoked "Bowtie" again in the -k 2 reporting mode (reports up to 2 alignments).

Both "Bowtie" and our implementation were used in single processor mode ("Bowtie" has a multithread mode. Our implementation can be parallelized).

We used collections of reads from the 1000 Genomes project[6] for the queries:

- ERR009392_1.filt.fastq (108 bp/ nucleotides / characters long reads)
- SRR023337_1.filt.fastq (78 bp/ nucleotides / characters long reads).

The paired reads (explained below) were ignored. We used the human genome GRCh37[7] (downloaded from the 1000 Genomes project website) as a the reference for both "Bowtie" and our implementation.

For timing a comparison, we took 10^7 reads from the original "fastq" files. For the quality comparison we used a subset of 10^5 reads.

The comparison was performed on a desktop computer with an AMD Athelon II X2 250 processor and 16GB RAM, running Ubuntu 10.04. Similar results were obtained on M610 servers with 48GB RAM .

The "Bowtie" software requires about 2.2GB RAM and our implementation requires about 16GB RAM.

Comments:

- "Bowtie" in -v x mode reports alignments with up to x mismatches, alignments with more mismatches are not reported.

Table 3: 108 nucleotides long reads

Software	Run time (sec)		% of reads aligned with $\leq n$ mismatches						
	Total	Per 1M reads	n=0	n=1	n=2	n=3	n=4	n=5	n=6
"Bowtie" -v 3	2903	290	52.7	68.5	75.0	78.8	78.8	78.8	78.8
"Bowtie" -v 2	1566	157	52.7	68.3	75.0	75.0	75.0	75.0	75.0
"Bowtie" -n 3	3961	196	52.7	68.0	74.6	78.2	80.4	82.0	83.2
"Bowtie" -v 3 -k 2	9088	909	52.7	68.5	75.0	78.8	78.8	78.8	78.8
"Bowtie" -v 2 -k 2	3375	338	52.7	68.4	75.0	75.0	75.0	75.0	75.0
"Bowtie" -n 3 -k 2	12558	1256	52.7	68.2	74.8	78.3	80.5	82.1	83.3
Our implementation	642	64	52.6	68.3	74.8	78.5	81.1	82.9	84.4
Our implementation (report subopt)	1156	116	52.6	68.3	74.8	78.6	81.1	83.0	84.4

Table 4: 78 nucleotides long reads

Software	Run time (sec)		% of reads aligned with $\leq n$ mismatches						
	Total	Per 1M reads	n=0	n=1	n=2	n=3	n=4	n=5	n=6
"Bowtie" -v 3	2942	249	58.2	73.0	77.6	80.2	80.2	80.2	80.2
"Bowtie" -v 2	1126	113	58.2	72.8	77.6	77.6	77.6	77.6	77.6
"Bowtie" -n 3	1754	175	58.2	72.3	76.8	78.9	80.2	81.0	81.6
"Bowtie" -v 3 -k 2	8159	816	58.2	73.0	77.6	80.2	80.2	80.2	80.2
"Bowtie" -v 2 -k 2	2736	274	58.2	72.9	77.6	77.6	77.6	77.6	77.6
"Bowtie" -n 3 -k 2	6730	670	58.2	72.5	77.0	79.0	80.3	81.1	81.7
Our implementation	770	77	58.2	72.8	77.3	79.9	81.9	83.4	84.5
Our implementation (report subopt)	1136	114	58.2	72.8	77.4	80.0	81.9	83.4	84.6

- "Bowtie" -n 3 and our implementation each uniquely reported some alignments when the best alignment had > 3 mismatches: There were reads for which only one of the two could find an alignment with > 3 and ≤ 6 mismatches. Our implementation had considerably more such uniquely aligned reads.
- By default, "Bowtie" reports a single alignment that is "valid" according to the parameters selected by the user (for example, a single alignment with up to 3 mismatches in the -v 3 mode). This may not be the best possible alignment. In -k 2 mode, "Bowtie" attempts to find up to 2 possible "valid" alignments.
- Our implementation allows an accuracy-time trade off. Less detailed reporting (fewer suboptimal or equally good matches) and slightly less accurate alignment have been performed in under one minute per 10^6 reads.
- We note that our implementation can be faster for longer reads (less time per read, not just per nucleotide).
- We note that in this implementation of the algorithm we did not use optimal, independent codes. This allowed us to simplify the implementation and save computer memory. We also note that some simple heuristics were added to the filter (part 3) to accelerate the execution. These modification reduce the accuracy of the algorithm. The result presented in the tables are of the implementation with this reduced accuracy.
- Due memory constraints, we indexed only one direction of the genome (only one of the two strands in the real DNA). Indexing the other direction (the "reverse complement") as well could produce a significant speed increase and may improve the accuracy.

4.2 A version for "paired-end" reads with "indels"

We note that in many cases, we are interested in strings which are close in "edit distance", allowing insertion and deletions from the string and not only changes in particular characters in the strings.

We also note that reads are often available in related pairs ("pair-end reads"). The model for this case requires us to find nearest neighbors for both strings $(Y^{(1)}, Y^{(2)})$, subject to some constraint on the distance between the locations in the reference genome. The problem is, therefore, choosing (i_1, i_2) that minimize $d_H(Y^{(1)}, X^{(i_1)}) + d_H(Y^{(2)}, X^{(i_2)})$ under the constraint $|i_1 - i_2| < \tilde{M}$ (when indels are present, we should use an "edit distance" rather than the "Hamming distance" in this definition).

We implemented a variation of the algorithm that simultaneously deals with both problems using some heuristics. This version of our implementation requires about $24GB$ of RAM for a human genome.

Since the "Bowtie" software package does not allow indels, we compared our implementation to the popular BWA[5] software package. We used simulated $108bp$-long reads from the human genome, with 3% mismatch probability, 0.04% indel probability and indel sizes of up to 5. We used a constant quality for all the nucleotides.

In the experiments, our implementation was over 10 times faster than the BWA software package. BWA mapped 98% of the read pairs to the correct location and our implementation mapped 96.5% of the reads to the correct location.

5 Conclusions

An algorithm has been constructed for the fast alignment of DNA reads to a reference genome. The algorithm inherently deals with mismatches, and it has been demonstrated that additional heuristics allow it to accommodate inserts and deletions as well.

The algorithm is based on a randomized approach. It allows trade-offs between error probability and run time. We note that in some applications, in particular when the reads are very short and no alignment errors are acceptable, other methods may prove to be more effective. However, as reads become longer, and taking into consideration the errors in the sequencing procedures themselves, the possible alignment errors that in other algorithms, and the various methods of verifying results, we propose that this algorithm is useful for many practical applications.

An implementation of the algorithm, is used to demonstrate aligning of about 10^6 reads per CPU minute. Future implementations of the algorithm are expected to be faster and more accurate.

6 Acknowledgements

The author would like to thank Prof. Vladimir Rokhlin, Prof. Ronald R. Coifman and Dr. Yaniv Erlich for their help.

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] Black, P E, ed. *Dictionary of Algorithms and Data Structures, U.S. National Institute of Standards and Technology.* http://xlinux.nist.gov/dads/, (Accessed 03/2012)

[4] Langmead, B., Trapnell, C., Pop, M. & Salzberg, S. L. *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.* Genome Biol. 10, R25 (2009).

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

[6] *A map of human genome variation from population-scale sequencing.* Nature 467, 1061-1073 (2010).

[7] *Human Genome Sequencing ConsortiumInternational Finishing the euchromatic sequence of the human genome.* Nature 431, 931-945 (2004).