ABSTRACT

Title of dissertation: SEARCHING, CLUSTERING AND EVALUATING BIOLOGICAL SEQUENCES

Mohammadreza Ghodsi, Doctor of Philosophy, 2012

Dissertation directed by: Professor Mihai Pop
Department of Computer Science

The latest generation of biological sequencing technologies have made it possible to generate sequence data faster and cheaper than ever before. The growth of sequence data has been exponential, and so far, has outpaced the rate of improvement of computer speed and capacity. This rate of growth, however, makes analysis of new datasets increasingly difficult, and highlights the need for efficient, scalable and modular software tools.

Fortunately most types of analysis of sequence data involve a few fundamental operations. Here we study three such problems, namely searching for local alignments between two sets of sequences, clustering sequences, and evaluating the assemblies made from sequence fragments. We present simple and efficient heuristic algorithms for these problems, as well as open source software tools which implement these algorithms.

First, we present approximate seeds; a new type of seed for local alignment search. Approximate seeds are a generalization of exact seeds and spaced seeds, in that they allow for insertions and deletions within the seed. We prove that

approximate seeds are completely sensitive. We also show how to efficiently find approximate seeds using a suffix array index of the sequences.

Next, we present DNACLUST; a tool for clustering millions of DNA sequence fragments. Although DNACLUST has been primarily made for clustering 16S ribosomal RNA sequences, it can be used for other tasks, such as removing duplicate or near duplicate sequences from a dataset.

Finally, we present a framework for comparing (two or more) assemblies built from the same set of reads. Our evaluation requires the set of reads and the assemblies only, and does not require the true genome sequence. Therefore our method can be used in de novo assembly projects, where the true genome is not known. Our score is based on probability theory, and the true genome is expected to obtain the maximum score.

CLUSTERING, SEARCHING AND EVALUATING BIOLOGICAL SEQUENCE DATA

by

Mohammadreza Ghodsi

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Advisory Committee:
Professor Mihai Pop, Chair/Advisor
Professor Najib El-Sayed, Dean's Representative
Professor Samir Khuller
Professor Hector Corrada Bravo
Professor Sridhar Hannenhalli

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computers are wonderful tools. They have numerous, seemingly different applications. Most of these applications, however, fall into a few broad categories. Data analysis is one of these broad categories.

Even though there are many techniques and recipes used in data analysis, most of them can be grouped into a few fundamental tasks. Looking for a specific pattern in the data (e.g. finding all patients on record with symptoms (exactly or approximately) similar to the current patient) or arranging the data in a particular way (e.g. ordering the chemical elements based on their atomic mass). The most basic forms of these tasks (sorting, clustering, and searching) are a classic part of computer science. It is not always possible, however, to use the classic algorithms directly on different types of data.

In fact all of these fundamental algorithms also have a single underlying theme: they involve comparison of some parts of the data with other parts.

The data for analysis may come from a variety sources. It may be, for the most part, gathered by humans, as in the census data. Or it may be mostly gathered by machines, such as weather measurements. Scientific data in particular, used to be mostly gathered by technicians. But nowadays, it is increasingly the case that it is generated by machines. This is particularly true in the field of molecular biology.

The fact that scientific measurement data can now be almost automatically generated by machines, means that the quantity (and sometimes, but not always, the quality) of the data increases every year, as these machines (which are mostly controlled by computers themselves) are improved. This poses a challenge, because the old methods for analysis of the same type of data may not scale well to such large quantities.

Our goal in this thesis is to look at a few well known analysis on biological sequence data, and try to improve them. (e.g. by improving sensitivity/specificity, by increasing the efficiency (speed), or by evaluating the quality of the results.)

The main type of data that we are concerned with is DNA sequence data. The DNA is a long molecule in all living cells that encodes most of the data which governs how the cell functions and reproduces. The DNA is a polymer, a long chain of four types of monomers. For our purposes, it is sufficient to consider the DNA to be a long string over a four letter alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. These are commonly referred to as DNA bases. The underlying theme of all of our work is comparing DNA sequences. This task is complicated by the fact that we usually are interested in discovering similar, but not exactly the same sequences.

The difference between DNA sequences can be due to actual difference between individuals organisms, or they can be due to measurement errors (commonly referred to as sequencing errors). Another complication due to the limitations of the sequencing machines is that they can only sequence relatively short pieces of DNA.

For an overview of classical algorithms on strings see Appendix A.

1.1 Sequence comparison

The goal of sequence comparison, in general, is to find how similar two sequences are. The exact interpretation of "similarity" can vary. A few examples are the Hamming distance (the number of substitutions necessary to transform one sequence into the other), the edit distance (the minimum number of insertions, deletions (indels), or substitutions that transform one string into the other), or more generally, methods based on probability (e.g. the probability of observing a sequence, assuming it is a modified version of another sequence based on an underlying model for change.)

Sequence similarity as defined above has a close connection with sequence alignment. A (pairwise) sequence alignment is obtained by shifting, and/or inserting gaps into a pair of sequences. Each letter from either sequence is aligned (i.e. appears in the same column) to a letter in the other sequence, or a gap. The score of an alignment is simply the sum (or the multiplication) of the scores corresponding to the columns in the alignment. The alignment between two sequences, usually implicitly refers to the optimal alignment corresponding to the best (maximum or minimum) score.

In general, we can have any score for aligning a pair of letters, or a letter and a gap. These scores can be represented by a matrix and are based on the underlying model that governs the change from one sequence to the other.

All similarity measures mentioned above can be computed using variants of the same algorithmic technique, called dynamic programming. Simply put, dynamic

programming allows us to find the best alignment (and the corresponding similarity measure), without the need to try all possible alignments. This is necessary, because the number of all possible alignments grows very quickly[1] as the lengths of the sequences grow.

Many popular variants of dynamic programming algorithm used for sequence comparison have $O(n_1 n_2)$ running time complexity, where n_1 and n_2 are the lengths of the two sequences to be compared. Unfortunately, this quadratic running time is too slow in practice on very large data. However for small data sets, the simple quadratic algorithm is practical.

1.2 Seeds for local alignment

We discussed alignments of two whole sequences before. This is known as global alignment. It is sometimes useful to find substrings of a pair of sequences that are similar to each other. This is commonly referred to as a local alignment. The two sequences within which we want to find local alignments may be very long. For example, if we want to find alignments between two sets of sequences, one possible approach is to concatenate the sequences in each set, and find local alignments between the two new sequences. The dynamic programming algorithm can be slightly modified to find all good local alignments.

The quadratic running time of the dynamic programming is too slow for large sequence data. Most of the programs that are used in practice for finding local

[1] There are $\binom{n+m}{n}$ possible alignments between two sequences of lengths n and m. This number grows exponentially as n and m increase.

alignments are based on seed-and-extend heuristic.

A seed is a relatively short sequence that occurs in both of the sequences to be compared. The seed can appear exactly in both sequences, or there could be small differences within the seed itself. In any case, the seeds for finding an alignment are designed to have higher similarity or a specific structure not found in the full alignment.

The dynamic programming algorithm (without using seeds) essentially compares every region of one sequence to every region of the other sequence. Therefore the running time is quadratic in terms of the lengths of the sequences. However, in most cases, we expect each region of the each sequence to match to zero, one or a few places in the other sequence. That is, we expect the number of matches to be roughly linear in terms of the sum of the lengths of the sequences.

If we can predetermine the (possible) positions of alignment and perform the more time consuming dynamic programming algorithm only on those regions, we can have a significant speedup over the quadratic approach. Therefore, a seed must have simultaneously good sensitivity and specificity, and must be possible to find without comparing every position of one sequence to every position of the other. For example, exact seeds can be found using a hash table, which removes the necessity of all-vs-all comparison.

There is an important sensitivity–specificity tradeoff regarding alignment seeds. For example, in the case of exact seeds, the sensitivity and specificity depend on the length of the seed. Longer exact seeds have fewer false positives, but they have more false negatives. Shorter exact seeds have more false negatives, but fewer false

positives. Depending on the alignments that we are looking for, we can pick a seed length that has zero false negatives. For example, if we are looking for an alignment of length greater than or equal to 104, with at most four mismatches, then it is guaranteed that exact seeds of length 20 (or shorter) will find all such alignments.

However if we are looking for less similar alignments (e.g. alignments of length 104 with up to 20 mismatches), we have to use shorter exact seeds to maintain perfect sensitivity, which results in many false positives.

To simultaneously keep low false positive, and zero (or provably low) false negative rate, we have to use approximate seeds.

One type of approximate seeds proposed in literature are spaced seeds. A spaced seed requires only certain letter within the seed to match, and does not care about the rest of the letters. For example, the seed `11*11`, requires that, out of the seed of length 5, every letter matches, except maybe the third letter.

Spaced seeds have some limitations however. They do not allow for insertions or deletions within the seed. Also, it is not straightforward to use spaced seeds if the alignment score is based on an arbitrary score matrix, rather than simply based on the number of matches and mismatches.

We propose a new type of approximate seeds which allow us to remove the above restrictions, and an algorithm based on suffix arrays and dynamic programming for finding these seeds efficiently.

1.3 Sequence clustering

Another fundamental analysis that depends on sequence similarity is clustering. Generally, given a set of items, we want to generate a set of clusters (subsets of items) such that each item belongs to exactly one cluster. A good clustering has two properties:

1. The items within any cluster are similar.

2. There are as few clusters as possible.

The two criteria above must be simultaneously optimized. Because if we only consider the first criteria, the trivial clustering that puts each item in a separate cluster is optimal. Similarly, if we only consider the second criteria, the trivial clustering that puts all items in one big cluster is optimal.

The notion of similarity is fundamental for clustering analysis. Conventionally, the distance between the items is either given as part of the input, or it can be calculated by a given function for any pair of items. We rely on the same definitions of sequence similarity discussed before. We use the global alignment in this case (not local alignments).

The main difficulty in our case is that the number of sequences is very large (several millions), and therefore it is computationally intractable to compute all pairwise distances between the input sequences. Because of this limitation, many of the classical clustering algorithms are inefficient for our data.

In order to avoid the all-vs-all comparison, we use a simple greedy clustering algorithm: We pick one of the sequences as the next cluster center, then we find all

of the sequences that are "close" to the center, create a new cluster, and remove the clustered sequences and repeat.

Note that the main component of this clustering algorithm is sequence similarity search. Unfortunately, since the sequences to be clustered are expected to be similar, we can not use any of the earlier sequence alignment search algorithms directly. For example, the seed and extend heuristic does not work because many of the sequences to be clustered may have large substrings in common. However, some of the ideas used for finding the seeds (e.g. building a lexicographically sorted index) can be used to speed up the search for cluster sequences. We also use additional heuristics, such as k-mer filtering, to speed up the search.

1.4 Assembly comparison based on likelihood

As we mentioned before, the machines for sequencing DNA can only read relatively short pieces of DNA continuously. These pieces are called "reads". The length of the reads is dependent on sequencing technology, but the newest generation of sequencing machines can read at most a few hundred bases. In contrast, the genome of a bacteria is millions of bases long, and the genome of more complex organisms are billions of bases long. Therefore currently, and in the near future, we are unable to read the whole genome of these organisms in one piece.

The dominant strategy to overcome this problem is to randomly shear multiple copies of the long DNA sequence, and to sequence the resulting pieces. This is commonly known as "whole genome shotgun sequencing".

Ideally, we want to reconstruct the original genome from these short read fragments. This problem is known as the genome assembly problem. To solve the assembly problem, several formulations have been proposed. Two of the most popular formulations: de Bruijn and Overlap-Layout-Consensus, are based on graphs. These formulations restrict the space of possible assemblies to the tours on the assembly graph. However, finding the "best" tour is still NP-hard and assembly programs (assemblers) use heuristic methods to find a near optimal solution.

The fundamental question as to which assembly is the best, however, remains unanswered. Suppose two different assemblers (or the same assembler run with different set of parameters) generate two assemblies from the same set of reads. We want to find out which of the assemblies is better.

The set of reads can be thought of as the set of our observations. We also need a model for the assembly process which generated the observed reads from the unknown original genome. Note the same set of reads may possibly be observed from different genomes. Therefore, instead of asking whether an assembly is correct, we ask how likely it is that an assembly is the unknown genome from which the observed reads have been generated.

This framework is very similar to the Hidden Markov Models. Similar to HMMs, we want to estimate the likelihood of observing a sequence of symbols from an underlying sequence of internal states. The same analogy can be used to describe the problem of building an assembly, i.e. discovering the most likely sequence of hidden states, given the sequence of observations from a model.

To calculate the probability of observing a set of reads given a genome or

assembly, we use a model of the sequencing process. This model depends on the sequencing technology. The parameters for the model are things like read length and error rate (probability of substitution, insertion, deletion, homo-polymer errors, etc.). To model paired-reads, we need a few more parameters. For example, assuming a normal distribution of insert length, we need the average insert length, and the standard deviation from the average. The more accurate the model of the sequencing process, the more accurately the assembly likelihoods can be calculated.

In our work, we start with a very simple model of sequencing process, and show how it can be extended to be more realistic. We describe efficient algorithms which generate the probability of a read, given the proposed genome. And finally, in order to make this evaluation even more practical, we show that the likelihoods can be estimated accurately based on a small sample of the assembled reads.

Chapter 2

A new approximate seed

In this chapter we describe a new type of approximate seeds for DNA homology searches, and an algorithm for finding these seeds efficiently. In contrast to previous algorithms that find exact seeds or spaced seeds, our approximate seeds may contain insertions or deletions. We propose a simple algorithm for finding these seeds which uses only suffix arrays.

2.1 Introduction

Finding local matches between two long sequences is a fundamental problem in computational biology. For example one might want to find similar regions in the genome sequence of two organisms. Due to differences in the genome sequence of different organisms, or to sequencing error, we have to look for approximate matches. Many algorithms and tools have been created that address this need, yet finding local alignments still remains one of the most computationally intensive steps in biological sequence analysis.

We define the *local alignment search* problem as follows: Given two strings S_1 and S_2 of lengths n_1 and n_2, a minimum match length l and a maximum distance k, find all pairs of substrings of S_1 and S_2 of length l that are within distance k. We assume all strings are from a finite alphabet Σ of size σ. A few popular distance

metrics are Hamming distance, unit-cost edit distance (Levenshtein distance), and general edit distance based on a substitution cost matrix.

A string matching problem is called *offline* if we are allowed to pre-process the text and make an *index* data structure, and *online* if we are not allowed to pre-process the text. The exact alignment problems can be solved optimally (with running time bounded by a linear function of the size of input strings) using e.g. the KMP algorithm for the online, and suffix trees for the offline case. In this chapter we will focus on the offline problem.

The online local alignment problem can be solved in $O(n_1 \cdot n_2 \cdot l)$ time by using each length l substring of S_2 as a pattern and using a variation of the well known Smith-Waterman algorithm [32]. This running time, however, is impractical for the size of the data-sets commonly encountered in practice.

In practice the most common solution to the inexact local alignment problem is to find *seeds* and try to *extend* them to longer local alignments [2, 5, 7]. A seed is a short substring of one sequence that matches closely to a substring of the other sequence. Seed and extend algorithms are fast because they avoid trying to align every region of the two sequences and only focus on the regions that are likely to align well. Two most common type of seeds used are exact seeds and spaced seeds. Exact seeds need to match perfectly, whereas spaced seeds allow mismatches at some positions. Neither allow insertions and deletions within the seed. In this work we look at more general inexact seeds that are based on the same notion of similarity as in Smith-Waterman [32]. We describe an algorithm for finding these inexact seed, and show that an implementation of this algorithm has good performance

in practice. Since our algorithm is based on dynamic programming it can use a general substitution cost matrix for distance computation, allowing its application to non-standard alignment problems.

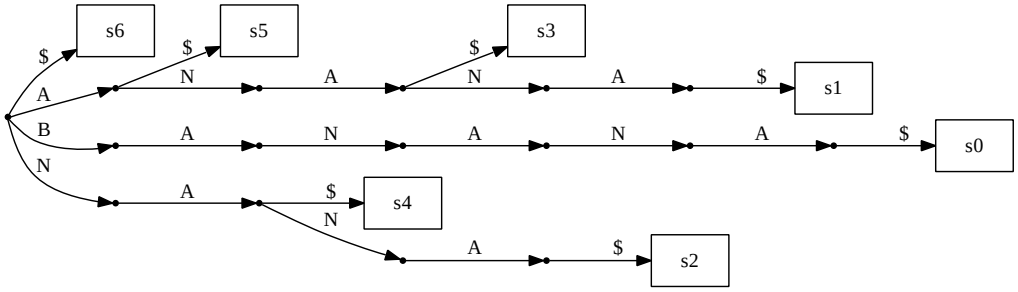
Seed finding algorithms are evaluated by their *sensitivity* and *specificity*. Sensitivity corresponds to the fraction of true local alignments that are found by the seeds. Specificity corresponds to the fraction of reported seeds that can be extended into true local alignments.

In the case of exact seeds, their sensitivity and specificity depends on only one parameter: seed length. Spaced seeds provide an additional parameter - weight - the number of characters within the seed that must match within the aligned strings. For the same weight, spaced seeds achieve better sensitivity than exact seeds without sacrificing specificity. Finding the most sensitive spaced seed is, however, NP-hard [19]. Most spaced seed tools use a pre-computed seed or set of seeds, making it difficult to tune their parameters at run-time in order to achieve the best trade-off between sensitivity and specificity for a given alignment task.

We will prove that our seed finding algorithm is perfectly sensitive, i.e. if a good enough local alignment is present it will be detected by our seeds. Furthermore since we allow inexact seeds, by using longer seeds we can achieve better specificity too. With our algorithm the user can pick any seed length and similarity at running time without the need to re-compute the index.

Most of the tools used in practice today are based on one of two index data structures for large strings; inverted k-mer index and the family of indexes related to suffix trees (suffix arrays and Burrows-Wheeler index.) BLAST [2] uses an inverted

Figure 2.1: Sample suffix trie for string 'BANANA'. Each edge is labeled by a symbol. The labels of the edges leaving each vertex must be different. Each suffix corresponds to a path from the root to a unique leaf.



index of exact seeds. MUMmer [6] is a whole genome alignment tool that uses suffix trees. Bowtie [17] uses a Burrows-Wheeler index of the human or other genome and can align short reads with few differences to the reference. Our algorithm uses a suffix array index.

2.2 Suffix array search algorithm

Our algorithm systematically tries to compute dynamic programming score (similar to Needleman-Wunsch [28]) for aligning every pair of substrings of S_1 and S_2. We assume the "cost" of aligning two characters is zero if they are identical and is some positive number otherwise. To achieve efficiency our algorithm gives up as soon as it notices that the alignment cost is getting "too high".

2.2.1 Recursive search of suffix trie

It is easier to explain our algorithm using the *Suffix Trie* data structure. A suffix trie is a rooted tree data structure that stores every suffix of a string. Unfortunately this representation requires up to $\Theta(n^2)$ space in general which makes it

0	B	0	6	$
1	A	1	5	A$
2	N	2	3	ANA$
3	A	3	1	ANANA$
4	N	4	0	BANANA$
5	A	5	4	NA$
6	$	6	2	NANA$

(a) String in memory (b) Suffix array in memory

Figure 2.2: The suffix array for a string consists of a list of positions of the suffixes. Assuming each position number is stored in 4 bytes, the suffix array requires $4 \cdot n$ bytes. However, in order to be able to perform search, we need to keep the original string also. Therefore, in total, the suffix array index requires $5 \cdot n$ bytes of memory.

impractical for long sequences. In the next section we will adapt this algorithm to use a suffix array which requires only linear space. Let us assume we have built a suffix trie for each of S_1 and S_2. A representation of an example suffix trie is shown in Figure 2.1. Each node of the suffix trie corresponds to a substring of the original string that is spelled out by the labels of the edges on the unique path from root to that node. An *exact* search algorithm starts at the root and follows down the edges *labeled with the same character* in both trees. In other words, if the search function is called on some internal node u from the first trie and v from the second trie, then for each letter of the alphabet if both u and v have a child labeled with that letter the search function is called recursively on the corresponding child nodes. After traversing l edges down from the root (reaching depth l) the leaves under the internal node of each tree represent suffixes from each sequence that share the first l characters exactly. Thus, we find exact "alignments" of length l.

In the case of approximate search however we may have to follow edges labeled

Figure 2.3: Our algorithm maintains a dynamic programming table, which contains the best alignment of the path corresponding to the current node in each suffix trie. For each edge traversal, only the top row (or last column) need to be recomputed. This figure shows a sample dynamic programming table when the algorithm is at depth 3. Only cells marked by ? need be computed when the algorithm follows down an edge to two children of current nodes (i.e. depth 4).

?	?	?	?	?	?		
N	3	2	1	1	?		
N	2	1	0	1	?		
A	1	0	1	2	?		
-	0	1	2	3	?		
	-	A	N	A	?		

with characters that are different in the two suffix tries. Therefore we have to call the search function recursively for *every pair* of children of u and v. Without limiting the number of mismatches allowed this recursive search (which is called for every pair of children) will try to align every substring of length l from S_1 to every substring of S_2. A simple solution is to stop following down branches from a pair of internal nodes as soon as the distance between the two strings corresponding to those nodes is already greater than some threshold. (We also stop when we reach depth l of course). i.e. As soon as the alignment cost becomes "too high" we simply return from recursion and will not call the search function for the children of current nodes. We calculate distance by updating a dynamic programming table on the fly as illustrated in Figure 2.3. We only need to update one row and one column of the dynamic programming table for each edge traversal. If the distance is smaller than the threshold we recursively call the search function for every pair of children of

the two current internal nodes. When we reach depth l we report all pairs of leaves under the two nodes. The cost threshold can either be a constant, or for example, a function of the length of the alignment so far.

A naive choice is to use a constant threshold equal to k, i.e. to stop when the distance between strings corresponding to the two current nodes is greater than k. This strategy is slow in practice however, because for most practical sequences the top levels of the suffix trie are nearly full (each node has nearly σ children), whereas deep nodes (depth $\geq \log_\sigma n$) of the trie have about one child on average. The constant threshold strategy will spend lots of time traversing top levels of the trie allowing many errors in the first few characters of the alignment. In fact, in the unit-cost edit distance model, this approach aligns every substring of length k of the first sequence to every substring of length k of the other sequence. The main intuition behind our algorithm is that we can look for shorter seeds with the desirable property that they do not have many differences in the beginning. This causes the algorithm to backtrack less within the first few levels of the tree and significantly improves the running time.

We show in Lemma 1 that if L_1 and L_2 (e.g. substrings of S_1 and S_2, respectively) of length l match with distance $\leq k$ then there exists a seed E_1 of length e ($1 \leq e < l$) in L_1 that matches a substring E_2 in L_2 in such a way that the cost of edits necessary to match any prefix of E_1 to some prefix of E_2 is never more than $\frac{k}{l-e}$ times the length of the prefix. This is a generalization of exact seeds. This lemma proves that our seed heuristic does not affect the sensitivity of the algorithm at all.

After finding the seeds one should verify that they can be extended to align-

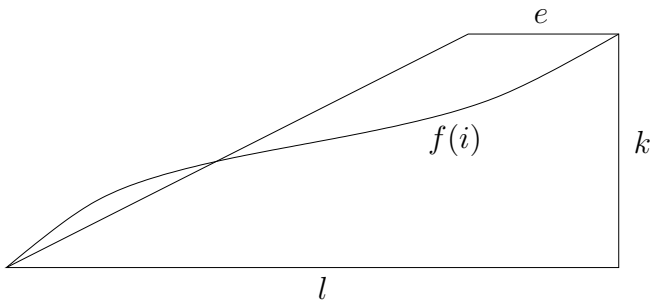


Figure 2.4: If L_1 and L_2 (substrings of the query and reference) have a good alignment (of length l, and distance $\leq k$), we can prove that this alignment must contain an approximate seed of length e, such that the distance of any prefix of length j of the seed is at most $j \cdot \frac{k}{l-e}$. The proof is by contradiction: define $f(i)$ as the edit distance of $L_1[1..i]$ from the prefix of L_2 to which it is aligned. If no approximate seeds exist, we can show that the distance between L_1 and L_2 must be greater than k.

ments of length l. (It is also possible to use these seed directly in a chaining algorithm if we desire a global alignment). This can be done in a post-processing step. But we assume for sufficiently long seeds ($e \geq \log_\sigma n$) the number of seed hits are small enough that the running time will be dominated by the seed search phase. Of course this is not true for highly repetitive sequences.

Lemma 1. *If L_1 of length l matches L_2 with cost $\leq k$, then for any seed length e, $1 \leq e < l$ there exists a seed E_1 a substring of L_1 of length e and E_2 a substring of L_2 such that for every prefix of E_1 of length j, $E_1[1..j]$ matches some prefix of E_2 with $\leq \left(j \cdot \frac{k}{l-e}\right)$ distance.*

Proof. Given an alignment of L_1 to L_2 with distance $\leq k$ let $f(i)$ be a the edit distance of $L_1[1..i]$ from the prefix of L_2 to which it is aligned. See Figure 2.4. It is easy to see that f is a non-decreasing function, and by definition $f(l) \leq k$ and $f(0) = 0$.

We need to show that there exists an $i \in \{0, \ldots, l-e\}$ such that for all

$j = 1 \ldots e$, $f(i+j) - f(i) \leq \left(j \cdot \frac{k}{l-e}\right)$.

As a way of contradiction assume for all $i = 0 \ldots (l-e)$, there exists $j \in \{1 \ldots e\}$ such that $f(i+j) - f(i) > \left(j \cdot \frac{k}{l-e}\right)$ or equivalently $f(i+j) > f(i) + \left(j \cdot \frac{k}{l-e}\right)$.

In particular for $i_1 = 0$ there exists j_1 such that $f(0+j_1) > f(0) + \left(j_1 \cdot \frac{k}{l-e}\right)$, and for $i_2 = (0+j_1)$ there exists j_2 such that $f((0+j_1)+j_2) > f(0+j_1) + \left(j_2 \cdot \frac{k}{l-e}\right) > \left(f(0) + \left(j_1 \cdot \frac{k}{l-e}\right)\right) + \left(j_2 \cdot \frac{k}{l-e}\right)$, and so on up to some z such that $l - e < 0 + j_1 + j_2 + \ldots + j_z \leq l$. We have:

$$
\begin{aligned}
f(l) &\geq f(0 + j_1 + j_2 + \ldots + j_z) \\
&> f(0) + \left(j_1 \cdot \frac{k}{l-e}\right) + \ldots + \left(j_z \cdot \frac{k}{l-e}\right) \\
&= 0 + (j_1 + j_2 + \ldots + j_z) \cdot \left(\frac{k}{l-e}\right) \\
&\geq (l-e) \cdot \left(\frac{k}{l-e}\right) = k
\end{aligned}
$$

Which implies $f(l) > k$ which is a contradiction. □

2.2.2 Adapting the algorithm to suffix arrays

A suffix array [24] is the list of indexes of all suffixes of a string in lexicographically sorted order, as illustrated in Figure 2.2(b). A suffix array can be built in linear time [1] and occupies $n \log_2 n$ bits. Even though the theoretical asymptotic memory requirement of suffix arrays and suffix trees is the same, in practice highly optimized implementations of suffix trees require over 10 bytes of memory per input

[1] Assuming $\log n$ is smaller than the word size of the machine and operations on them takes constant time

character [16] whereas a basic suffix array implementations require just 4+1 bytes per character[2]. Space requirements of suffix arrays can be further reduced to $O(n)$ bits using compressed suffix arrays. Finally it has been shown that by storing some auxiliary tables, Enhanced Suffix Arrays [1] can be used to simulate any type of traversal of suffix trees in the same time complexity.

Our algorithm over suffix arrays is inspired by the simplest exact search algorithm on a suffix array which is in essence a binary search. Suffix arrays have a property useful to our algorithm: if some prefix of the suffix pointed to by ith element of suffix array is equal to that of the suffix pointed to by $j$the element of suffix array, then for every $k \in [i \ldots j]$, every suffix pointed to by the kth element of suffix array shares the same prefix. This property easily follows from the fact that the suffixes are lexicographically sorted. The recursive algorithm described over suffix tries can be adapted to work on suffix arrays with at most a $\log_2 n$ factor increase in running time as will be shown in Lemma 2. The main algorithm is as follows: we maintain two windows of the two suffix arrays and the depth (length of the prefix) that we have aligned so far, if the characters at this depth from both sides of the current window match (for each window of the two suffix arrays), there is only one character to follow down in this window so we update the dynamic programming table and call search for depth+1, otherwise we simply divide the window (for which the characters from both sides at current depth do not match) in two equal windows and recursively find the matches on both halves. The window size will always remain greater than or equal to one.

[2]Assuming length of input string can be stored in a 32 bit integer.

The main search function

```
void sasearch(int l1, int r1,
              int l2, int r2, int depth);
```

recursively finds all approximate matches between two windows of the two suffix arrays ($[l_1 \dots r_1]$ in the first suffix array and $[l_2 \dots r_2]$ in the second) assuming the first "depth-1" characters in each window are the same. And the distance between the shared prefix of suffixes in first window and shared prefix of suffixes in the second window is stored in a global dynamic programming table like Figure 2.3.

Lemma 2. *The number of calls to the recursive suffix array search function is at most* $\log_2 n$ *times the number of suffix trie nodes visited by recursive search algorithm, where* n *is the length of string.*

Proof. let p be the number of nodes visited by the suffix trie recursive search algorithm. The key point is to note that each internal node of the suffix trie corresponds to a window $[i_l \dots i_r]$ on the suffix array. So the p nodes of the suffix trie that contain every path from themselves to the root can only "cut" the suffix array in at most p positions. We need to bound the number of times to recursively cut the suffix array of length n in half that is needed to realize a partitioning of the suffix array in p given positions. (Note that we are free to e.g. continue cutting one half and leave the other half alone). Let us denote the number of cuts made by our recursive suffix array halving algorithm by $g(n, p)$.

$g(n, p)$ is bounded by the following recurrence (where p_r and p_l are two non-

negative integers such that $p_l + p_r \in \{p, p-1\}$)

$$g(n,p) \leq \begin{cases} 0, & n = 1 \text{ or } p = 0 \\ g\left(\frac{n}{2}, p_l\right) + g\left(\frac{n}{2}, p_r\right) + 1, & \text{otherwise} \end{cases}$$

We will prove by induction that $g(n,p) \leq p \log_2 n$. Base case is trivial. For the induction step we have

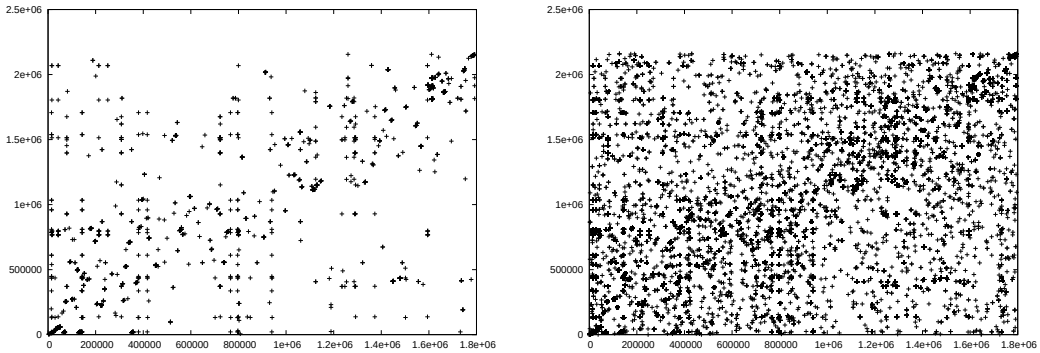
$$\begin{aligned} g(n,p) &= g\left(\frac{n}{2}, p_l\right) + g\left(\frac{n}{2}, p_r\right) + 1 \\ &\leq p_l \log_2 \frac{n}{2} + p_r \log_2 \frac{n}{2} + 1 \\ &= (p_l + p_r) \log_2 \frac{n}{2} + 1 \leq p \log_2 \frac{n}{2} + 1 \\ &= p \log_2 n - p + 1 \\ &\leq p \log_2 n \end{aligned}$$

□

2.3 Experimental results

To examine the practical applicability of this algorithm we have implemented this algorithm in C++. A key advantage of this algorithm is that it is very easy to implement. For example the "sasearch()" function mentioned is implemented in 50 lines of code and the whole seed searching program is less than 200 lines[3]. This is an experimental implementation of our algorithm and is not optimized for running

[3]Source code for our implementation can be found at `http://www.cs.umd.edu/~ghodsi/sasearch/`



(a) MUMmer exact 20bp

(b) Approximate 20bp, 1 error

(c) MUMmer exact 64bp

(d) Approximate 85% 64bp

Figure 2.5: Comparison with MUMmer exact seeds. Each + is a seed match at the corresponding coordinates in the genomes of Streptococcus pneumoniae (vertical, 2.5Mbp) and Streptococcus thermophilus (horizontal 1.8Mbp). Long approximate seeds (d) have better sensitivity (than long exact seeds (c)), and better specificity (than short exact seeds (a)).

time. All tests were run on a single core of a 64-bit Linux PC.

Figure 2.5 shows the dot plot results of running the algorithm on the full sequence of genomes of two bacterial organisms: Streptococcus pneumoniae and Streptococcus thermophilus, in comparison to the exact algorithm of MUMmer [6]. By default MUMmer finds exact matches of length 20 or longer between the two genomes, as can be seen in Figure 2.5(a). For comparison we have included approximate matches of length 20 with one error Figure 2.5(b). Note that the number of false positives increases significantly if we allow just one error in a 20 base pair

Figure 2.6: Comparison of resources used by SASearch and MUMmer for two bacterial genomes. Approximate seeds (being more sensitive and precise) require more computation, compared to exact seeds. Since we use a suffix array index to find approximate seeds, our program requires less memory compared to MUMmer which uses suffix trees.

	MUMmer exact 20bp	SASearch 85% 64bp
Time	2.3s	45s
Memory	35MB	20MB

match. To decrease the number of false positives one can of course increase the length of the match, an exact match of 64 bases is shown in Figure 2.5(c). It is clear that we have decreased sensitivity considerably by increasing the length of the match. Finally our algorithm which looks for approximate (85%) matches of length 64 bases is shown in Figure 2.5(d). It can be seen that our algorithm has simultaneously better sensitivity and better specificity than all variations above. Running time and memory usage for this test are shown in Figure 2.6.

To evaluate the quality of our seeds in a biological sense, we use the ROSETTA's test set [3] (also used to validate LAGAN [5]), which contains 117 orthologous annotated genes with complete intron sequences from human and mouse. These sequences are of interest because they contain conserved coding regions from relatively distant genomes. Such regions are approximately similar overall but may not contain long exact seeds because of silent mutations among other differences. We simply concatenated the orthologous genes in the same order to make a long sequence of total length 602Kbp for human and 578Kbp for mouse. We compare our algorithm labeled SASearch (with different parameters) with MUMmer [7] and nucleotide-BLAST [2] exact seeds and PatternHunter [22] spaced seed as well as the improved spaced seeds

	ESn	GSp	OG	NOG	Ex
PatternHunter spaced seed	99	14	35921	229914	462
BLAST 11bp exact seed	99	9	47918	487846	463
16 spaced seeds weight 11	100	6	165623	2689721	465
MUMmer 20bp exact	51	21	1285	4814	241
Spaced seed of weight 18	78	46	11324	13407	364
SASearch 60bp 85%	64	93	20902	1513	302
SASearch 40bp 80%	92	74	44079	15463	428
SASearch 30bp 70%	99	19	107386	459194	463

Table 2.1: Comparison of different seed finding algorithms on DNA sequence of human and mouse genes. Columns are: ESn: Exon Sensitivity($\frac{Ex}{465}$), GSp: Gene Specificity ($\frac{OG}{OG+NOG}$), OG: Number of seeds that connect *only* true Orthologous Genes (true hits at gene level), NOG: Number of seeds that hit Non-Orthologous Genes (false hits at the gene level), Ex: Number of human exons (of 465) that contain at least one hit.
Approximate seeds found by SASearch (at 80% similarity and 40bp length) have simultaneously better sensitivity and specificity than MUMmer seeds and spaced seeds (weight 18). Also, approximate seeds (at 70% similarity and 30bp length) have equal sensitivity but better specificity than BLAST seeds, PatternHunter seeds, and a set of 16 spaced seeds (weight 11).

suggested in [15]. Default MUMmer and BLAST seeds are exact matches of lengths 20 and 11 respectively. PatternHunter's spaced seed is `111*1**1*1**11*111`. Ilie et. al. suggest multiple spaced seeds and produce a set of 16 spaced seeds of weight 11 as well as single spaced seed of weight up to 18. The single seed of weight 18 that we picked for comparison is `11111*1*11**111*11*11111`.

The seeds found by each algorithm are evaluated using two criteria: exon-sensitivity and gene-specificity, which loosely capture the biological sensitivity and specificity of a set of seeds. Exon-sensitivity is the fraction of 465 human exons that will contain at least one seed. Gene-specificity is the fraction of seeds that connect only pairs of orthologous genes. (We calculate specificity at the gene level because

we do not have a one-to-one correspondace between exons.) The results are shown in Table 2.1.

We observe that BLAST, PatternHunter and weight 11 spaced seeds are all very sensitive (for this data) but also have a very large number of false positives. Essentially they are so short (or low weight) that they hit at many positions by chance. Our inexact seeds of length 30 and with 70% similarity achieve the same sensitivity with a better specificity, because with our longer seed length random hits are less likely.

MUMmer exact seeds of length 20 and the single spaced seed of weight 18 (length 24) seem a better choice for these data. However they both miss a large fraction of exons. Our inexact seeds of length 40 with 80% similarity simultaneously achieve better sensitivity and specificity that both MUMmer seeds and single spaced seeds. This is due to the fact that our inexact seeds are longer and less similar and also allow for insertions and deletions in the seeds.

2.4 Conclusion

For local alignment search there are many different tradeoffs: between specificity and sensitivity, sensitivity and running time, running time and memory, etc. Here we proposed an algorithm for constructing inexact alignment seeds and we show that our algorithm can have better sensitivity (than exact seeds based tools) and lower memory usage (than suffix trees based tools) for the cost of longer running time. The suffix array index we use can be built once and reused while allowing

for full flexibility in the choice of alignment parameters. We have, thus, shown that full flexibility in the choice of alignment seeds can be achieved without a significant penalty in terms of running time. The use of a substitution cost matrix makes our algorithm applicable to seeding protein alignments or for other "non-standard" alignment tasks such as detecting cross hybridization of probes.

Acknowledgments

This was a joint work with Mihai Pop. We thank Niranjan Nagarajan for helpful discussions related to these ideas. We also thank Lucian Ilie and Silvana Ilie for providing us with an improved set of spaced seeds.

This work was supported by grant NIH R01-HG-004885 to MP.

Chapter 3

Clustering DNA sequences

Clustering is a fundamental operation in the analysis of biological sequence data. New DNA sequencing technologies have dramatically increased the rate at which we can generate data, resulting in datasets that cannot be efficiently analyzed by traditional clustering methods.

This is particularly true in the context of taxonomic profiling of microbial communities through direct sequencing of phylogenetic markers (e.g. 16S rRNA) – the domain that motivated the work described here. Many analysis approaches rely on an initial clustering step aimed at identifying sequences that belong to the same operational taxonomic unit (OTU). When defining OTUs (which have no universally accepted definition), scientists must balance a trade-off between computational efficiency and biological accuracy, as accurately estimating an environment's phylogenetic composition requires computationally-intensive analyses.

We propose that efficient and mathematically well defined clustering methods can benefit existing taxonomic profiling approaches in two ways: (i) the resulting clusters can be substituted for OTUs in certain applications; and (ii) the clustering effectively reduces the size of the data-sets that need to be analyzed by complex phylogenetic pipelines (e.g., only one sequence per cluster needs to be provided to downstream analyses).

To address the challenges outlined above, we developed DNACLUST, a fast clustering tool specifically designed for clustering highly-similar DNA sequences.

Given a set of sequences and a sequence similarity threshold, DNACLUST creates clusters whose radius is guaranteed not to exceed the specified threshold. Underlying DNACLUST is a greedy clustering strategy that owes its performance to novel sequence alignment and k-mer based filtering algorithms.

DNACLUST can also produce multiple sequence alignments for every cluster, allowing users to manually inspect clustering results, and enabling more detailed analyses of the clustered data.

3.1 Background

Clustering of sequences (DNA or protein) is a common and basic analysis in bioinformatics that underlies many biological analyses. Clustering can be used to reveal underlying natural groupings of data. Clustering can also be used to simply reduce the size of a large dataset, such that a slower, more accurate, analysis can be applied [21]. The results of the slower analysis can then be carried over to the rest of the sequences. In this dissertation we focus on one application of DNA sequence clustering; namely the analysis of 16S ribosomal RNA (rRNA) data. The algorithms and principles underlying our tool should, however, be applicable to a wider range of sequence clustering tasks.

Sequence analysis of the 16S rRNA is one of the most commonly used methods for measuring microbial diversity and taxonomic composition of an environment.

There are two complementary approaches to the analysis of 16S data: comparative classification, and unsupervised clustering. In the comparative approach, the taxonomic identity of a new sequence can be determined if it is similar to some of the sequences present in a curated database [37]. This approach, however, can not be reliably used for the analysis of novel sequences, thus scientists frequently rely on methods based on the unsupervised clustering of sequences [10, 30, 31]. Our work is specifically targeted at unsupervised methods. Note, however, that clustering of 16S sequences can be used as a pre-processing step even in the case of database-based methods in order to reduce the size of the datasets being analyzed and to speed up the classification process.

The traditional approach for clustering 16S rRNA sequences involves building a multiple sequence alignment (MSA) of all sequences, computing a pairwise distance matrix based on the MSA and clustering the resulting matrix [38]. The clustering algorithm is often a greedy hierarchical clustering algorithm which produces a rooted tree. The tree is then cut at some level, based on a specified similarity threshold, in order to construct a collection of clusters. Alternatively, if the taxonomic annotation of some of the sequences is known, the tree can be used in a more elaborate semi-supervised clustering algorithm [27].

Since the latest DNA sequencing technologies have become faster and cheaper, we are now faced with very large volumes of sequence data. Newer generations of sequencing technologies, e.g., 454 Life Sciences sequencing machines, can generate millions of sequences per run, each of which has a length of hundreds of base pairs. Such datasets cannot be easily clustered using the traditional approach outlined

above.

First of all, finding the best multiple sequence alignment is computationally intractable – this problem falls into the category of NP-hard problems. (NP is the class of problems for which a solution can be verified in polynomial time. NP-hard problems are problems to which all other problems in NP can be reduced in polynomial time. No polynomial time (exact) solution for any of the NP-hard problems is known. If such a solution is found, however, it means that all NP problems can be solved in polynomial time.)

Multiple sequence alignment tools rely on heuristic alignment algorithms that are not guaranteed to generate an optimal alignment (which is not a well defined concept, anyway). The most common heuristic involves building a guide tree (a preliminary hierarchical clustering of the sequences) that then guides the construction of the multiple alignment. Often, the guide tree is constructed from a preliminary distance matrix constructed from pairwise alignments of the sequences – for large data this matrix is impractical (its size, and therefore time needed to construct it, grows with the square of the size of the datasets). Furthermore, determining a guide tree is difficult for large datasets since there could be many trees that fit the distance matrix equally well.

An alternative to the traditional clustering approaches that rely on multiple alignments, is a simple, yet effective, greedy clustering strategy. The process starts by selecting a sequence as a "seed" for a cluster. Additional sequences are added to this cluster if they fall within a certain distance from the seed. The process continues by selecting an unclustered sequence as the seed for a new cluster, and

so on until all sequences have been clustered. This basic approach is employed by the programs CD-HIT [20], UCLUST [9], and our own work. The main difference between these programs is in the way the clusters are constructed, specifically, how a program identifies all sequences that are nearby a cluster seed. As we will describe in more detail below, both CD-HIT and UCLUST search each sequence against a database of all previously constructed clusters. If the sequence does not have a good match against any of the existing clusters, it forms the seed for a new cluster.

The approach we present in this chapter involves searching each cluster seed against a database of all unclustered sequences, thereby "recruiting" a set of sequences to the newly created cluster. We will show that this approach allows us to leverage an efficient search data structure to rapidly cluster large sets of sequences. Our approach is particularly well suited for high-stringency clustering (high similarity between the clustered sequences), intended to remove redundancy in the dataset by co-clustering sequences that are identical or whose differences are primarily due to sequencing errors. Representative sequences from each cluster can then be used as input to more computationally intensive analyses.

3.1.1 Related work

There are two popular tools designed for clustering large number of sequences: CD-HIT and UCLUST. CD-HIT [20] has been widely used in practice and is cited by hundreds of scientific articles. UCLUST [9] is a newer clustering tool. UCLUST is based on a fast sequence search algorithm (which is also used in the related

USEARCH program), and can be more than an order of magnitude faster than CD-HIT. In the following we briefly review the algorithms used by these tools.

3.1.1.1 CD-HIT

CD-HIT uses a greedy incremental clustering algorithm. First the sequences are sorted in non-increasing order of their lengths. The first sequence becomes the first cluster representative. Each consecutive sequence is compared to all previously discovered cluster representatives, and is added to a cluster if it is within a user-selected distance threshold from the corresponding representative. Otherwise the sequence becomes the seed for a new cluster.

CD-HIT uses a "short word filtering" heuristic to avoid computing many of the costly pairwise alignments. Specifically, each sequence is represented as a k-mer spectrum (an array containing the number of occurrences of all substrings of length k in the sequence), and the initial comparison between sequences is performed between the corresponding spectra. If the k-mer counts differ significantly, it is unlikely that the sequences match each other well. CD-HIT relies on a statistical analysis to estimate the minimum number of k-mers that two sequences are expected to have in common, assuming they have a certain similarity to each other.

This filtering approach can be computationally expensive as it requires counting the number of k-mers shared by each sequence and all previously selected cluster representatives.

3.1.1.2 UCLUST

UCLUST follows virtually the same algorithm as CD-HIT, with two major exceptions: (i) sequences can be sorted in different ways, rather than simply by length (as done by CD-HIT); (ii) the mapping of sequences to existing cluster representatives is performed with a new search heuristic called USEARCH.

By default, UCLUST operates in an inexact mode. In the inexact mode each sequence is not aligned to all cluster centers found so far. Instead UCLUST sorts the cluster centers based on the number of "words" they have in common with the query. Each query sequence is thus aligned only with a few of the cluster representatives (up to a predefined constant), which are presumed most likely to be close to it. In the exact mode UCLUST operates more or less like CD-HIT, i.e. each query sequence is aligned to all cluster centers found so far. In this mode the word based filter is not used.

The inexact heuristic guarantees that the number of pairwise alignments is linear (and, thus, the algorithm is fast). In practice, this approximation could result in many more clusters than the exact mode, however the extent of this "blow-up" depends on the stringency of the clustering.

3.2 Results

We describe the algorithms used by our tool in the following section. In the Testing section we evaluate the performance and quality of our implementation.

3.2.1 Algorithm

The main goal in the design of our algorithms is computational efficiency and scalability. In this section we present a simple greedy clustering algorithm which avoids most of the pairwise comparisons in practice. The clustering algorithm uses an alignment search algorithm and a k-mer based filter which are also described in this section. Some definitions and general concepts are covered first.

3.2.1.1 Definitions

Distance measure: Clustering is intimately interconnected with the definition of the distance or similarity measure used to compare the objects being clustered. Several distance measures have been commonly used to compare sequence data, including *edit distance* (also called Levenshtein distance) – a measure that counts the minimum number of insertions, deletions, or substitutions that are required to transform a sequence into the other; *k-mer distance* – a measure of the number of substrings of length k that are shared (or differ) between two sequences; and *evolutionary distance* – an estimate of the number of evolutionary events (usually substitutions) that explain the differences between two sequences. In many cases more than one distance measure is used during clustering, e.g. a k-mer approach can be used to quickly discard sequences that should not belong to a same cluster, then a more precise, but slow, algorithm is applied (this combination is used by our algorithm as well as by CD-HIT and UCLUST).

Our approach defines the distance between two sequences to be the corre-

sponding edit-distance where the cost function is simply unit cost for each gap or mismatch, and zero cost for matches. In the case of sequences of different lengths, we may want to allow gaps at the end and/or the beginning of the shorter sequence in the alignment, without penalty. This type of alignment is referred to as semi-global alignment. The default behavior of DNACLUST is to allow gaps at the end of a sequence but not at the beginning (aligned sequences are anchored at their 5' end), however other alignment policies can be selected through command-line parameters.

Clustering parameters: We define the *diameter* of a cluster as the maximum distance between any two sequences in the cluster. Our algorithm (like CD-HIT and UCLUST) returns one sequence per cluster as *cluster representative*. There is another related but slightly different concept of the *cluster center*, usually picked in such a manner that the maximum or average distance from it to the rest of the items in the cluster is minimized. In the following we sometimes loosely use the term cluster center to refer to the cluster representative. Given a cluster representative, the *cluster radius* can be defined as the maximum distance from the cluster representative to any sequence in that cluster. Given that the edit distance measure is a metric (follows triangle inequality) we also guarantee that the cluster diameter is at most twice the cluster radius.

Sequence similarity and sequence identity: The criterion used by UCLUST and CD-HIT to evaluate distance between sequences is the amount of sequence "identity", i.e. the fraction of characters that match exactly between the two sequences

Figure 3.1: The multiple alignment of a cluster produced by UCLUST. Note that the sequences match at 100% *identity*, despite the long gaps. (In general, sequence identity is not a metric, i.e. it does not satisfy the triangle inequality.)

```
>0|100.0%|nomatch
--GATCTCTCCGGA----------GATCTCGTAGCGCTCATC----------CACTCCTTCACCGGGCTCTGCCCCTTT---
>0|100.0%|dnaGfrag
--------TCCGGAGACGATCGCGCTCGTGAAAGAGCGCACGGATCTCGTAGCGCTCA----------
>0|*|dnaG
ATGATCTCTCCGGAGACGATCGCGCTCGTGAAAGAGCGCACGGATCTCGTAGCGCTCATCGGCGAAAGCGTGCGCCTCGCCCGCCGAGGGCACTCCTTCACCGGGCTCTGCCCCTTTCAC
```

being aligned. This measure also defines the clustering stringency, e.g. a clustering threshold of 99% implies that the sequences within a cluster have 99% or higher identity. In an evolutionary sense identity is a natural definition of the similarity between two sequences, primarily because insertions and deletions are difficult to fit within evolutionary models – DNADIST [10] usually ignores any gaps when computing the distance matrix. The identity measure, however, can lead to unintuitive alignments, especially when comparing sequences of different lengths. In addition, identity is not a transitive measure, i.e. the fact that sequence A and B are identical (have zero distance according to the identity measure), and sequences B and C are identical, does not imply that sequences A and C are also identical. More generally, identity does not follow triangle inequality (A distance measure obeys triangle inequality if, for every three sequences A, B, and C, $\text{dist}(A,B) + \text{dist}(B,C) \leq \text{dist}(A,C)$), implying that even though the cluster radius is small the cluster diameter may be high, i.e. while the distance between all the sequences in the cluster and the cluster representative is small, individual sequences may differ significantly from each other. As a result, the cluster is not as "tight" as would be implied by the distance threshold used. Furthermore, the resulting multiple alignment is of lower quality. (See Figure 3.1.)

To avoid these problems we rely on a different measure of distance between se-

quences: the (semi-)global alignment score. In this context, the "similarity" between two sequences can be defined as:

$$\text{similarity} = 1 - \frac{\text{edit distance}}{\text{length of the shorter sequence}}$$

Here the "length of the shorter sequence" refers to the length before alignment and does not include the gaps induced by the alignment.

Note, however, that due to the different definitions of distance, the clusterings produced by DNACLUST, CD-HIT, or UCLUST cannot be directly compared at a given clustering threshold.

Clustering properties: The clustering problem that we study in this work has the following form: given a set of sequences and a threshold on the cluster radius, group these sequences into clusters, and identify one sequence within each cluster as the cluster representative.

Given a metric distance between a set of items, any clustering of these items can have one or more of the following properties:

1. The radius of every cluster is less than or equal to the specified threshold.

2. The distance between any two cluster centers is strictly greater than the threshold.

3. The distance between any clustered item and any cluster center (except the center of the cluster to which the item belongs) is strictly greater than the threshold. This implies that the closest center to any item is the center of its

cluster.

A clustering is *valid* if it satisfies property 1. A valid clustering with the minimum number of clusters is called an *optimal* clustering. Unfortunately, finding an optimal clustering (assuming a general metric distance between items,) is NP-hard [36]. A clustering that satisfies property 2 in addition to 1 is called an *exact* clustering. Our search and filter algorithms are designed to be able to create exact clusterings.

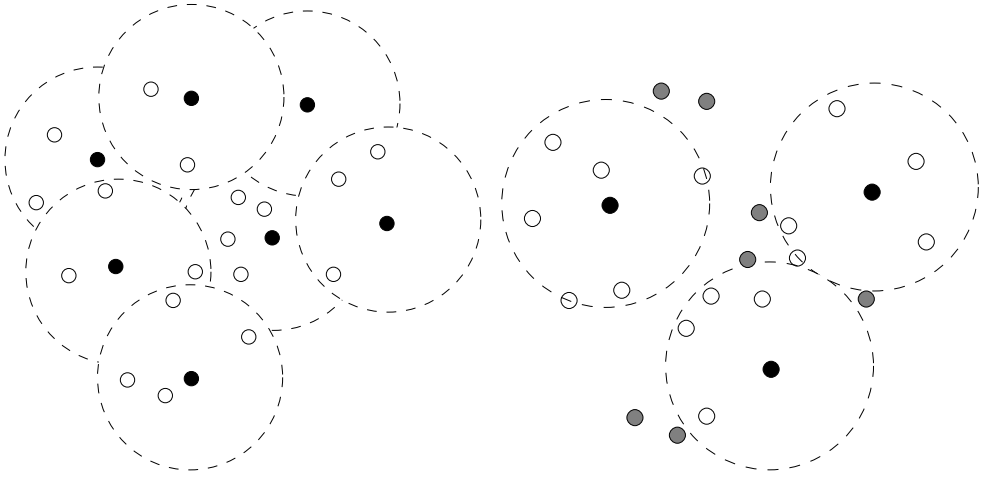
A clustering that satisfies property 3 is called a *well-separated* clustering. (For illustrations of exact and well-separated clusterings see Figure 3.2.) Note that an exact clustering does not guarantee that the clusters are well separated. Also it is not always possible to cluster all of the items into well separated clusters. Below, we will describe an algorithm (that can be selected through command-line parameters) that provides well-separated clusterings.

3.2.1.2 Clustering algorithm

The foundation of DNACLUST is a simple greedy clustering algorithm, which is similar to the algorithms used by CD-HIT and UCLUST.

In practice, the sequences are first sorted based on their length in a non-increasing order. Then at each iteration the longest remaining sequence is picked as the new cluster center. We form the largest possible cluster with this cluster center by searching through the set of unclustered sequences for all sequences that are less than a user-selected distance threshold from the cluster center. The clustered

Figure 3.2: Two possible clusterings of a set of points on the plane based on Euclidean distance. The cluster centers are colored in black. In both of these clusterings no two cluster centers are closer than the cluster radius from each other. The well separated clustering guarantees that each point is assigned to nearest cluster center, a guarantee not provided by an exact clustering. Note, however, that while an exact clustering can cluster every point in the data-set, this may not be possible in the well separated clustering (gray points in the figure).



(a) Exact clustering (b) Well separated clustering

Algorithm 1 Greedy Clustering Algorithm

```
i ← 1
while Sequences ≠ ∅ do
    Center_i ← longest(Sequences)
    Cluster_i ← search(Sequences, Center_i, RADIUS)
    Sequences ← Sequences − Cluster_i
    i ← i + 1
end while
```

sequences are marked and are not taken into consideration any longer. (Pseudocode of this algorithm is provided in Algorithm 1.)

Picking the longest unclustered sequence as the cluster center is necessary to ensure the correctness of clustering when the lengths of the sequences are not equal. Specifically if two sequences, both of which are longer than the cluster center, are clustered together, it is not possible to guarantee that they align well to each other, i.e. these sequences could be incorrectly placed in the same cluster even if they differ

significantly.

In the case that all the sequences have (or are trimmed to have) the same length, it has been proposed that ordering the sequences by abundance results in a better clustering [14, 29], especially in the presence of sequencing errors. The abundant sequences can be inferred to be the correct molecules that are surrounded by a "cloud" of imperfect sequences due to sequencing errors. This approach makes most sense if the data contain high coverage and relatively well separated sequences (e.g. data from a low-complexity community). In diverse communities, however, it can be difficult to distinguish between experimental noise and true genomic variation. In addition, determining "abundance" requires some form of clustering, either exact (counting the number of exact copies of each sequence in the data), or by allowing a small amount of error. DNACLUST is specifically targeted at such high stringency clustering applications, and, thus, could be used as an initial step in a more elaborate clustering scheme that takes abundance into account.

Note, however, that a full evaluation of the phylogenetic interpretation of clustering strategies is beyond the scope of our work. Our main goal was to develop an efficient and mathematically well-defined clustering approach. More complex analyses of the data that, e.g., take into account phylogenetic signal, can be performed by post-processing the output of our software. In order to generate an exact clustering, the search step must find all the unclustered sequences that are within the specified radius from the cluster center. In this context we refer to the cluster center as the *query sequence*. The distance measure can be based on either global alignment cost, or semi-global alignment cost, in which case the gap costs at one or both ends of

the shorter sequence are ignored. Either of these policies can be picked by the user using the command line options. This search is the most time consuming step of the algorithm, and is described in more detail in the next section.

This algorithm can be easily modified to construct well separated clusters (property 3 in the previous section) as follows. Each time a new cluster of radius r is constructed, we also flag every unclustered sequence within distance $2r$ from the center of the new cluster. The flagged sequences can not be picked as cluster centers in subsequent iterations of the greedy algorithm, but may be included in a cluster constructed around an unflagged cluster center. This approach ensures that the distance between two cluster centers can not be less than two times the cluster radius. In order to implement this algorithm all we need to do is to double the search radius but only cluster the sequences that fall within the clustering threshold.

3.2.1.3 Alignment search algorithm

Our alignment search algorithm is designed to find all good (semi-) global alignments of the query sequence to a large set of sequences simultaneously. The main speed up is achieved by taking advantage of the fact that we are only interested in alignments that are high-quality: have a cost which is less than or equal to a certain threshold.

It is easier to explain this algorithm if we assume that all the sequences are stored in a trie data structure [11]. We traverse the trie using a depth first search (DFS) algorithm. At each internal node of the trie we compute the cost of best

alignment of the query sequence (the representative for the current cluster) to the sequence corresponding to the path from root to the current internal node in the trie [35]. This corresponds to simultaneously matching the (identical) prefixes of all the sequences sorted in the trie within the subtree rooted at the current node with the query.

The pairwise alignment and the cost are computed using a dynamic programming algorithm , a variation of the NeedlemanWunsch algorithm [28].

First we describe the alignment algorithm for two complete sequences. Assume we are trying to align two sequences S_1 and S_2 of lengths n_1 and n_2. We fill an $n_1 \times n_2$ table of numbers, such that the element at position (i, j) of the table – $T_{(i,j)}$ – contains the score (i.e. cost) of the "best" alignment of the first i characters of S_1 to the first j characters of S_2. In our case, the types of differences that are allowed are insertions, deletions and substitutions, and $T_{(i,j)}$ depends on only three other elements of the table:

$$T_{(i,j)} = \min \begin{cases} T_{(i-1,j)} + \text{cost}(S_1[i], \text{“gap”}) \\ T_{(i,j-1)} + \text{cost}(\text{“gap”}, S_2[j]) \\ T_{(i-1,j-1)} + \text{cost}(S_1[i], S_2[j]) \end{cases}$$

If the table is filled row-by-row (or column-by-column), the total amount of computation needed to fill this table (and compute an alignment) is proportional to $n_1 \times n_2$.

The table is initialized as follows: If we are interested in a semi-global alignment (as in Figure 3.3) the first row is initialized to all zeros. On the other hand,

Figure 3.3: Partially filled dynamic programming table. The query sequence is represented on the horizontal axis. At this point the algorithm has computed the alignment costs for a prefix of length 4 of the data sequences – which is shown on the vertical axis. Since we are calculating semi-global alignment the first row is initialized to all zeroes, i.e. The alignment of the shorter data sequence can start at any position of the longer query sequence without any penalty. In this figure, the distance threshold is 2, and any values larger than this threshold are set to the maximum value represented by ∞. To optimize the running time, since there are only three valid values on the last finished row, only the values for the three gray cells need to be computed on row right above it.

sequence$_i$									
	?	∞							
	G	∞	∞	2	1	2	∞		
	T	∞	2	1	2	∞			
	C	2	1	1	2	2	2	.	.
	A	1	0	1	1	1	1	.	.
	-	0	0	0	0	0	0	0	0
		-	A	T	G	G	T	.	.
					query				

if we assume all the sequences start at the same position (i.e. global alignment) the first row of the dynamic programming table is initialized with the cost of gaps required at the beginning of the alignment. The first column is always initialized with the cost of the required gaps.

Since we are trying to simultaneously align a query sequence to a set of sequences, we think of one of the sequences as growing (and shrinking) as we backtrack through the common prefixes of sequences in a suffix trie, and update the table as necessary. On the horizontal axis of the dynamic programming table (Figure 3.3) the query sequence is fixed. On the vertical axis we have the prefixes of the sequences in the trie.

As we go deeper in the trie, the dynamic programming table is filled one row at a time. Each time the depth-first search of the trie traverses an edge, we only need to update one row of the table, namely the current top row. Also note that

at each point the cost of the best semi-global alignment of the current path to the query is the minimum value on the top row.

Also, assuming unit cost for each gap (insertion or deletion), we do not even have to update all of the cells on the current row in the dynamic programming table after traversing an edge in the trie [13]. As is shown in Figure 3.3, the dynamic programming table updates need be performed only for the cells whose bottom-left neighbor contains a value no larger than the specified cluster radius.

This is due to the fact that for any two sequences S_1 and S_2, and any i and j, $\text{dist}(S_1[1..i], S_2[1..j]) \geq \text{dist}(S_1[1..i-1], S_2[1..j-1])$ where dist() is the edit distance.

The main heuristic that speeds up this algorithm relies on the observation that it is not always necessary to compute the alignments of the query to all paths in the trie all the way to each leaf. Instead, during the depth-first search, if at any point the alignment cost of the prefix is too high, the recursive search terminates without further exploring the children of the current node in the trie.

The trie data structure described above is not explicitly built in our implementation. Instead we keep a list of the sequences *in lexicographically sorted order*. As we proceed to align one of the sequences (S) to the query we are also implicitly aligning all adjacent sequences that share a common prefix with S. In other words, each internal node of the virtual trie corresponds to a unique consecutive sub-list of the sorted list of sequences, in which the shared prefix of the sequences corresponds to the path from the root of the trie to that internal node.

Effectively we first try to align the first sequence to the query (in our case the cluster representative). If the alignment is good then the first sequence is added to

the search results. When aligning the second sequence, and so on, we can avoid re-aligning its common prefix with the previously aligned sequence, thereby reducing the cost of computation. Note that if we fail to align the first say α characters of sequence i (ith sequence in the sorted list), and the common prefix of sequence i and sequence $i+1$ is longer than α, we do not need to try to align sequence $i+1$ at all. The search algorithm is very similar to the algorithm in [12], except that the backtracking threshold is fixed as the given radius of the clusters.

We use the ternary quick sort algorithm [4] once, in the beginning, to sort the sequences lexicographically. The time spent for sorting the sequences is, however, much less than the time spent during clustering.

3.2.1.4 Star multiple sequence alignment

One valuable output of a sequence clustering algorithm is a multiple sequence alignment representing the global relationship between the sequences present in the cluster. Such a multiple alignment can be used by users to manually inspect the quality of the clustering, and can also represent the substrate for more complex analyses (e.g. computation of evolutionary distances between the sequences).

We rely on a "star" multiple alignment heuristic that computes the multiple alignment from the pairwise alignments between each of the sequences and the cluster representative. The pairwise alignments between the sequences and the representative are a byproduct of our search algorithm. To construct the multiple alignment we reconcile the differences between these pairwise alignments by inserting

gap characters as necessary.

Consider two consecutive letters of the cluster center sequence; in each pairwise alignment, there could be different numbers of gaps between these two letters. In the multiple alignment we need the number of gaps between these two letters to be the maximum of the numbers of gaps inserted between these characters in any of the pairwise alignments.

This approach guarantees that the pairwise distance between any two sequences in the alignment is at most twice the cluster radius (maximum distance between any sequence and the cluster representative).

3.2.1.5 Word-based filter

Finding the best pairwise alignment of two sequences using dynamic programming is computationally intensive. It requires quadratic time in the length of the input sequences, in the general case. In our clustering application, however, it is possible to avoid calculating a pairwise alignment altogether, if we are certain that no good alignment exists. We use k-mer based filtering [20] to speed up the search for sequences with good alignment.

Given a sequence S of length n (e.g. one of the sequences we are trying to cluster), a k-mer is a substring of S of length k, where k is chosen to be much smaller than n. Sequence S contains $n - k + 1$ overlapping k-mers, some of which may be identical.

The filter is based on the key intuition that if two sequences are within a small

edit distance from each other, they must share most of their k-mers. In the following we formalize this idea.

Let us assign numbers from 1 to 4^k to the possible k-mers. Given a sequence s, by counting how many times each one of the 4^k k-mers appears in the sequence we obtain a vector of non-negative integers of dimension 4^k. Namely the ith element of this vector, v_i, counts the number of times that k-mer number i appears in s. We call this vector the *k-mer spectrum* of the sequence s, and it is is denoted by $\text{spectrum}_k(s)$.

Given a vector of integer numbers, v, define $\text{pos}(v)$ to be the sum of the positive values in the vector. Similarly define $\text{neg}(v)$ to be the sum of the negative values. For example, for any sequence s of length n, we have $\text{pos}(\text{spectrum}_k(s)) = n - k + 1$ and $\text{neg}(\text{spectrum}_k(s)) = 0$.

Consider two sequences, s_1 and s_2, that are close to each other. The following observation bounds the maximum difference between their k-mer spectra.

Observation 1. *If s_2 has edit distance d from s_1, then, for all k,*

$pos(spectrum_k(s_1) - spectrum_k(s_2)) \leq k \times d$ and

$neg(spectrum_k(s_1) - spectrum_k(s_2)) \geq -k \times d$.

Proof. Assume $d = 1$. Then s_2 can be obtained from s_1 by one insertion or one deletion or one substitution. Consider $\text{spectrum}_k(s_1)$. It is easy to check that in all three cases, at most k new k-mers are created, also at most k existing k-mers are eliminated.

If $d > 1$, s_1 can be transformed into s_2 by applying d edits one after the other.

During this process a set of $d-1$ intermediate sequences are obtained. Note that the edit distance between each consecutive pair of the intermediate sequences is exactly 1. Therefore the total number of new k-mers in s_2 is at most $k \times d$. Similarly, the total number of k-mers eliminated from s_1 is also at most $k \times d$. □

Since we want to be able to report sequences that might have a good semi-global alignment to the query sequence, we have to consider the case in which one sequence is within a small edit distance from *a substring of* the other sequence. The following observation helps the handling of this case.

Observation 2. *If a sequence s_1 is a substring of s_2, then, for all k,*

$neg(spectrum_k(s_2) - spectrum_k(s_1)) = 0.$

Lemma 3. *If s_1 has edit distance d from s^* and s^* is a substring of s_2, then, for all k,*

$pos\,(spectrum_k(s_1) - spectrum_k(s_2)) \leq k \times d.$

Proof. By Observation 1 and Observation 2 and algebra:

$$\begin{aligned}
&\mathrm{pos}\,(\mathrm{spectrum}_k(s_1) - \mathrm{spectrum}_k(s_2)) \\
&= -\,(\mathrm{neg}\,(\mathrm{spectrum}_k(s_2) - \mathrm{spectrum}_k(s_1))) \\
&\leq -\,(\mathrm{neg}\,(\mathrm{spectrum}_k(s_2) - \mathrm{spectrum}_k(s^*)) + \mathrm{neg}\,(\mathrm{spectrum}_k(s^*) - \mathrm{spectrum}_k(s_1))) \\
&\leq -(0 + (-k \times d)) \\
&= k \times d
\end{aligned}$$

□

Using Lemma 3, given a query sequence q, a sequence s, and a distance threshold d, if *for any* k we have

$$\text{pos}\left(\text{spectrum}_k(s) - \text{spectrum}_k(q)\right) > k \times d$$

then we can be certain that no semi-global alignment of s to q exists which corresponds to a distance less than or equal to d.

In the following, we will extend Lemma 3 to quickly determine whether none of the sequences in a collection has a good alignment to the query sequence. We will then describe how to build a binary search tree of the sequences, and using their k-mer counts, quickly discard subtrees that do not have any sequence close to the query.

Given a set of n vectors which have the same dimension define $\min_1(v_1, v_2, \ldots, v_n) = v$ to be the vector of equal dimension, such that $v[i] = \min(v_1[i], v_2[i], \ldots, v_n[i])$, for all i. Given a set of n sequences, define

$$\text{min\_spectrum}_k(s_1, \ldots, s_n) =$$

$$\min_1(\text{spectrum}_k(s_1), \ldots, \text{spectrum}_k(s_n))$$

Similarly we define $\max_1()$ and $\text{max\_spectrum}_k()$. The following corollary follows from Lemma 3.

Corollary 1. *Given a query sequence q and a set $S = \{s_1, \ldots, s_n\}$ of sequences, for*

any k*, if*

$$pos(min\_spectrum_k(s_1, \ldots, s_n) - spectrum_k(q)) > k \times d$$

Then none of the sequences in S *is within edit distance less than or equal to* d *from* q*.*

We use this result to quickly discard sequences that could not possibly be close to a given query sequence. The maximum length of k-mers used is denoted by $k_{\max}$. The value of $k_{\max}$ is a parameter that can be specified by the user. At the beginning of the program, a search data structure is built based on the $k_{\max}$-mer spectra of the input sequences. The data structure we use for the filter is a binary tree, in which the (unique spectra of the) input sequences are the leaves. The root of the tree represents the set of all of the sequences. For each internal node let descendants(node) denote all the leaf sequences that are located under this node. At each internal node of this tree the following information is stored: for every $k = 1 \ldots k_{\max}$ we store two vectors: min_spectrum$_k$(descendants(node)) and max_spectrum$_k$(descendants(node)).

Let us first explain how we search for the sequences potentially close to a query sequence based on their k-mer spectrums, if such a binary tree is already constructed. We first calculate the k-mer spectrum of the query for every $k = 1, \ldots, k_{\max}$. We use a recursive search function shown in Algorithm 2. The parameters for the first call of this function are the root node of the search tree and $k = 1$. If at some internal node pos(node.min_spectrum$_k$ − query_spectrum$_k$) $> k \times d$ then by

Algorithm 2 k-mer Filter Search Algorithm

```
function search(Node node, Integer k)

if pos(node.min_spectrum_k − query_spectrum_k) > k × d then
  return ∅
end if
if pos(node.max_spectrum_k − query_spectrum_k) ≤ k × d then
  if k < k_max then
    return search(node , k + 1)
  else
    return descendants(node)
  end if
end if
return search(node.left_child, k) ∪ search(node.right_child, k)
```

Corollary 1, none of the descendants of this node can be close enough to query. If pos(node.max_spectrum$_k$ − query_spectrum$_k$) $\leq k \times d$, then every descendant is acceptable regarding this k value. If $k = k_{\max}$ we return all descendants, otherwise we increment k and continue the search. Finally if none of the cases above happens, we recursively search the two children of the current node, and return the union of results from each one.

Let us now explain how the binary tree is built, to allow the search algorithm to quickly find potentially close sequences. Intuitively the sequences with similar spectra should be grouped together. In case of k-mer spectra we measure similarity by the L_1 distance. We build the tree using a top-down recursive procedure, based on $k_{\max}$-mer spectra. At each step if the number of unique k-mer spectra is greater than one, we divide them into two groups and recursively build a binary search tree of each group. The set of k-mer spectra is divided into two using the 2-means clustering algorithm. This is a special case of the k-means (Note that the variable k

in k-means algorithm denotes the number of clusters, and is unrelated to the variable which denotes the k-mer length.) algorithm with $k = 2$. In our implementation of k-means in L_1 distance we define the centroid of a cluster to be the median of the values in each dimension instead of their mean (which is the more commonly used definition).

As the greedy clustering of Algorithm 1 progresses the number of sequences not yet clustered decreases. In practice the clusters created early in the algorithm are bigger, and most of the clusters created near the end of the algorithm are singletons. Therefore, to speed up the filter, the binary search tree structure is rebuilt, when the number of remaining sequences shrinks sufficiently. Specifically, we rebuild our tree data structure every time the number of sequences not clustered so far drops to less than half of the sequences in the structure.

3.2.2 Testing

In this section we compare clustering speed of DNACLUST with other clustering tools on different datasets. We also evaluate the quality of the multiple sequence alignment that DNACLUST can produce for each cluster.

3.2.2.1 Speed

DNACLUST and UCLUST can produce exact or approximate (i.e. inexact) clusterings. (The definitions of these terms are provided in the Algorithm section – *clustering properties.*) Creating an exact clustering takes more time. For these tools

we have measured the running time in both settings.

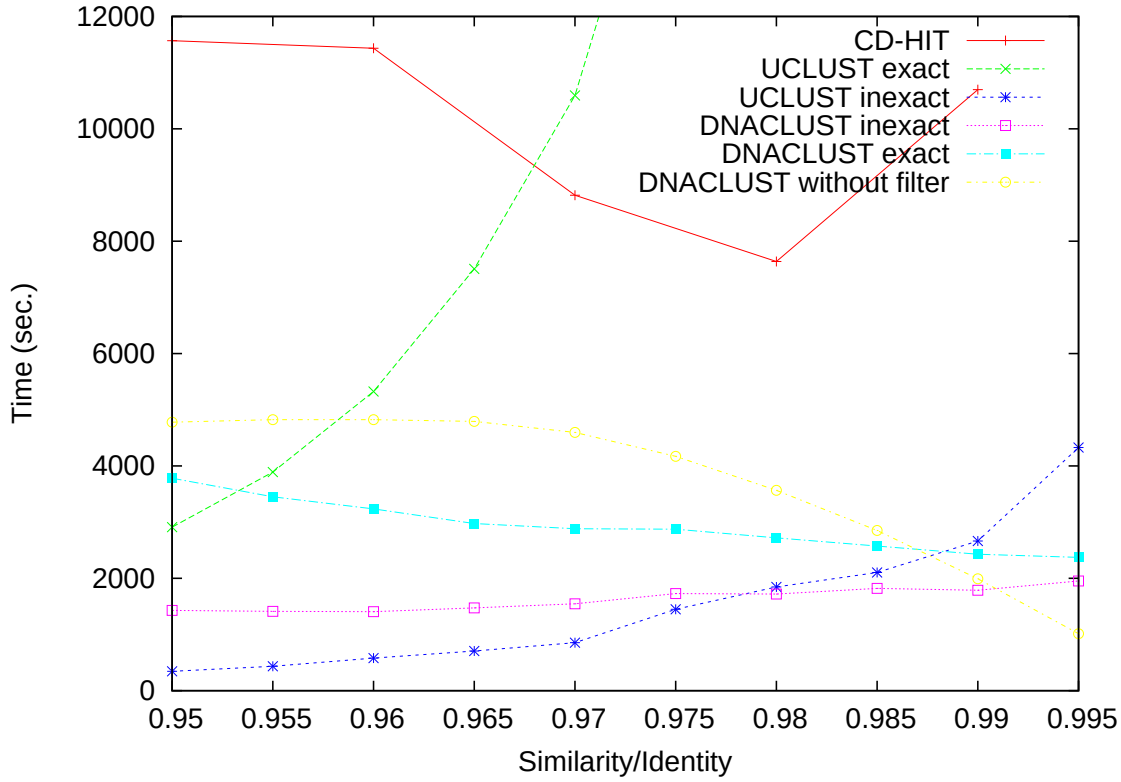
We have created clusterings at different similarity/identity thresholds for each tool and each setting. (Sequence identity and sequence similarity are discussed in detail in the definitions part of the Algorithm section.) The radius of the clusters created range from 0.95 up to 0.995 similarity. (For a list of the parameters that can be set by the user of DNACLUST see Section 3.2.3.2.)

In order to evaluate and compare the performance of our program, we use two publicly available datasets. The first dataset is from the gut microbiome of 154 individuals. These data were generated as part of a project to evaluate the differences in the gut microbiome of obese and lean twins [34]. The dataset contains 1.1 million pyrosequencing reads from the V2 region of the 16S rRNA gene. The reads have an average length of 231 base pairs. We refer to this dataset as the twins dataset. The running time of various clustering tools using different setting and cluster radii on the twins dataset is shown in Figure 3.4. The number of clusters generated for each setting is shown in Table 3.1.

In exact mode the running time of UCLUST increases rapidly as the radius of the clusters is decreased. This is because a smaller cluster radius results in a large number of clusters (and hence cluster centers). In addition, for highly-similar sequences, the search heuristic used by UCLUST becomes less efficient. DNACLUST in exact mode is faster than UCLUST for any similarity threshold above 0.95.

UCLUST in inexact mode is much faster than in exact mode, and thus faster than DNACLUST in most cases. In inexact mode DNACLUST is faster than UCLUST only for similarity thresholds greater than or equal to 0.98. Both DNA-

Figure 3.4: Plot of running time as a function of cluster radius for various tools and settings, on the twins dataset. The dataset contains 1.1 million pyrosequencing reads from the V2 region of the 16S rRNA gene. The reads have an average length of 231 base pairs. The running times were measured on a single 1.8GHz processor of an Intel x86-64 Linux laptop with 4GB RAM.



CLUST and UCLUST in inexact mode are roughly an order of magnitude faster than CD-HIT.

As seen in Table 3.1 the switch from exact to inexact mode leads to a significant change in the number of clusters generated by UCLUST, leading to up to 75% more clusters at the same similarity threshold. In other words, the improvement in speed comes at the cost of reduced cluster quality. In comparison, our k-mer filtering strategy (DNACLUST in inexact mode) leads to only a small increase in the number of clusters ($< 3\%$), i.e. our inexact heuristic is more effective in terms of speeding up the algorithm without significantly affecting the results of the clustering.

Table 3.1: The number of clusters produced by DNACLUST, UCLUST and CD-HIT at various identity/similarity thresholds, on the twins dataset. Since each tool uses slightly different distance measures, the number of clusters can not be directly compared between different tools. (Namely the identity measure used by UCLUST and CD-HIT underestimates the distance between two sequences, as computed by the similarity measure used by DNACLUST). Instead we compare the change in the number of clusters when switching between the exact and inexact modes of each tool – a smaller change indicating better performance.

	0.99	0.97	0.95
DNACLUST exact	233879	73726	28241
DNACLUST inexact	240125	76391	28661
UCLUST exact	144339	48418	20039
UCLUST inexact	253108	71361	26685
CD-HIT	245851	100280	55208

Table 3.2: The running times (minutes) of DNACLUST and UCLUST with various similarity thresholds on the RDP dataset. Note the large increase of the running time of UCLUST in exact mode, compared to inexact mode.

	0.99	0.97	0.95
DNACLUST exact	204	372	960
UCLUST exact		7800	5040
DNACLUST inexact	74	76	150
UCLUST inexact	43	29	16

The second dataset contains all full 16S rRNA sequences from the Ribosomal Database Project [23]. We picked all of the sequences which were between 1300 and 1550 base pairs, covering almost the full length of the gene (480,312 sequences).

This test is meant to evaluate the performance of our algorithm on long sequences such as those that may be generated by future sequencing technologies. The running times of DNACLUST and UCLUST on the RDP dataset are shown in Table 3.2.

The sequences in the RDP dataset are almost three times longer than the earlier dataset, allowing us to evaluate how our approach scales with anticipated increases in read length. The alignment algorithm is slower on these data since, at the same level of similarity/identity between sequences, the total number of differences is higher. The trends in the performance are, however, consistent with the results observed on the twins dataset. DNACLUST still outperforms UCLUST for tighter clustering thresholds (> 0.95): UCLUST in the exact mode takes more than 130 hours to cluster this dataset, in contrast to just over 6 hours for DNACLUST.

Please note that the results shown above ignore any connection between clusters and actual biological entities, i.e. we are primarily concerned with whether the clustering is mathematically consistent instead of whether clustering captures some underlying biological truth. In general, no fixed clustering threshold adequately captures the taxonomic structure in the data [38], in part because "biological truth" is not a well defined concept (at least not in mathematical terms) . Additionally, sequencing errors can blow up the number of clusters especially if the sequencing error rate is of roughly the same order of magnitude as the clustering threshold.

To estimate the true taxonomic composition of a dataset we recommend a two step process that starts by building tight clusters (e.g. at 0.99 similarity) with DNACLUST, then uses the cluster representatives (and the size of the clusters) as input for a more sophisticated but slower algorithm which could not otherwise be applied to the original dataset.

3.2.2.2 Multiple sequence alignment

In the following we compare the quality of the multiple sequence alignment (MSA) produced by DNACLUST and commonly-used multiple alignment algorithms. We are defining the quality of a multiple sequence alignment in a very strict sense, specific to the analysis of 16S rRNA data: we measure how well the distance matrix computed by DNADIST (a commonly used tool that estimates evolutionary distances between sequences in a multiple alignment) matches the clustering criteria: i.e. if the clustering threshold (i.e. radius) is 99% identity, we expect the largest value in the distance matrix (i.e. largest diameter) to be lower than 2%.

To evaluate the quality of the multiple sequence alignment produced by our program we compare it to two of the most popular MSA tools, ClustalW [33] and MUSCLE [8]. Neither of these tools can handle as many sequences as our largest clusters contain (the largest cluster of the twins dataset at 95% similarity contains 74,465 sequences). Both of these tools crash when they fail to allocate enough memory for an $n \times n$ matrix to store the pairwise distances, where n denotes the number of input sequences.

For $n \geq 38,000$ ClustalW, and for $n \geq 22,000$ MUSCLE could not run on a machine with 4 GB of RAM. Note that larger datasets could be aligned using MUSCLE given additional RAM or by specifying the main memory limit (–maxmb parameter). The total memory requirements, however, grow quadratically as a function of the number of sequences being aligned, ultimately limiting the size of datasets that can be analyzed on commodity hardware.

Table 3.3: Time spent building a Multiple Sequence Alignment of a sample cluster using different tools, and the diameter of the MSA produced, as reported by DNADIST. The diameter is expected to be less that or equal to 0.10 .
DNACLUST produces the best MSA according to DNADIST. Traditional MSA tools are slow and produce worse MSAs on this type of data.

MSA method	Time (sec.)	Diameter (DNADIST)
ClustalW	1545.5	0.251
ClustalW -quicktree	87.6	0.264
MUSCLE	197.8	0.198
UCLUST	0.1	0.156
DNACLUST	0.8	0.094

To compare the multiple alignment routines, we selected one of the clusters produced by DNACLUST within the twins dataset. This cluster was constructed with a 95% similarity threshold (5% cluster radius), and contained 1117 sequences. These sequences were aligned using MUSCLE and ClustalW (with and without the "-quicktree" option), resulting in three multiple sequence alignments. We compared these "traditional" MSAs to those generated by DNACLUST and UCLUST. A pairwise distance matrix was obtained based on each MSA using DNADIST [10]. The maximum distance between any pair of the sequences was then reported, which corresponds to the diameter of the cluster in terms of evolutionary distances inferred from the corresponding MSA. The running time and the inferred cluster diameter for different MSA tools are shown in Table 3.3.

This experiment validates the clusters and corresponding MSAs produced by DNACLUST using an independent tool for measuring evolutionary distance between DNA sequences. The DNADIST distance matrices are the underlying data used in the "traditional" 16S rRNA clustering approaches. The results also demonstrate

that commonly used multiple sequence alignment tools are not well suited for the alignment of a large number of sequences.

DNACLUST and UCLUST rely on a star-alignment heuristic, and use the cluster representative as the one sequence against which all the other sequences are aligned. This approach guarantees that the distance between any pair of sequences within the MSA is at most twice the radius of the cluster. Furthermore, this approach is efficient: building a star MSA only requires time linear in the number of sequences, in contrast to the quadratic time needed to construct the guide tree in most traditional multiple alignment approaches.

We further compared multiple sequence alignments produced by DNACLUST and UCLUST. We built clusterings using both programs (and multiple sequence alignments for each cluster) from the twins dataset at the thresholds 95%, 97% and 99% (6 clusterings in total). Since computing pairwise distances is computationally intensive especially for large clusters, from each clustering we randomly selected 50 clusters containing between 100 and 500 sequences. For each cluster we also built a multiple alignment using ClustalW.

We generated the pairwise distance matrix for each alignment using DNADIST, then computed the average pairwise distance of the aligned sequences. Figure 3.5 show the distribution (relative frequency) of the alignments based on their average pairwise distance for each threshold. The clusters produced by DNACLUST are tighter than the clusters produced by UCLUST at the same threshold because of the more stringent definition of distance between sequences (we rely on full alignment score while, by default, UCLUST uses identity). Closer examination of the

Figure 3.5: Figures a, b and c show the distribution of sampled cluster multiple sequence alignments based on their average pairwise distance for thresholds 99%, 97% and 95%, respectively. The figures show that DNACLUST cluster MSAs (thick blue line) are tighter (i.e. have smaller average pairwise distance) than UCLUST cluster MSAs (thick red lines). Furthermore computing a "traditional" MSA using ClustalW from the clusters produced by DNACLUST and UCLUST results in an overestimation of the distances between sequences (dashed lines).

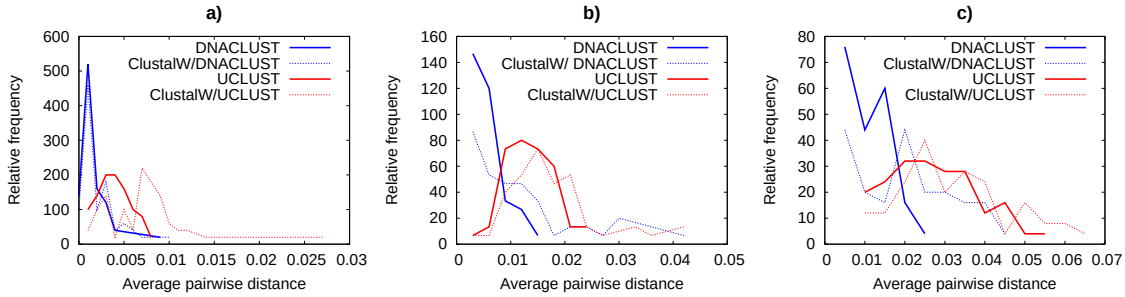


Table 3.4: The average frequency of gaps in multiple sequence alignments for sampled clusters at various similarity thresholds. For each MSA, the frequency of gaps is the number of gaps divided by the total number of characters in the MSA. Gaps before the beginning and after the end of each sequence are excluded. Note that since an insertion in one sequence results in a gap in all other sequences in the MSA, the ratio of gaps may be higher than the clustering threshold. Since the sequence identity measure used by UCLUST does not take gaps into account the number of gaps in UCLUST MSAs are higher than the gaps in DNACLUST MSAs, specially at more stringent thresholds.

	0.99	0.97	0.95
DNACLUST	0.016	0.071	0.103
UCLUST	0.071	0.117	0.146

multiple sequence alignments produced by UCLUST shows that they tend to contain more gaps (which is consistent with the UCLUST definition of distance as sequence identity). Average frequencies of gaps in multiple sequence alignments produced by DNACLUST and UCLUST at various thresholds are shown in Table 3.4. Note that the gaps do not affect the pairwise distance computation, as they are not penalized by DNADIST.

Our results also show that the traditional approach for computing multiple

alignments (here using ClustalW) results in an overestimate of the distances between the aligned sequences (dashed lines in Figure 3.5), confirming the observation that the star alignment strategy is more appropriate for large sets of highly similar sequences.

3.2.3 Obtaining and using DNACLUST

3.2.3.1 Availability and requirements

Project name: DNACLUST

Project home page: `http://dnaclust.sourceforge.net/`

Operating system: Linux – x86/x86-64

Programming language: C++

Other requirements: Boost C++ libraries 1.34 or higher

License: GNU GPL

3.2.3.2 Running DNACLUST:

In its simplest invocation, DNACLUST expects only three parameters: (i) a multi-FASTA file containing the input sequences; (ii) a similarity threshold; and (iii) the name of the output file to contain the resulting clustering. The output file is a simple text file containing one line per cluster. Each line starts with the identifier of the cluster representative, followed by a space-delimited list of the identifiers of all

the sequences in the cluster. This file format is augmented with the FASTA records corresponding to the aligned sequences, if the multiple sequence alignment option is selected.

DNACLUST can be configured through several command line options, briefly described below.

1. *similarity:* Similarity threshold specifies the radius of the clusters created. The threshold is a numerical value between 0 and 1.

2. *k-mer length:* The *maximum* length of k-mers used for filtering. The longer k-mer lengths require more memory to store k-mer counts and the filtering will be slower. However with the longer k-mer length, the filter will be more specific and therefore the sequence alignment search will be faster. If the median length of the input sequences is l then a good choice (and the default value) for k-mer length is $\log_4(l)$.

3. *multiple alignment:* Produces a multiple sequence alignment for each cluster.

4. *allow left gaps:* By default the sequences are anchored at the 5' end. This option allows for the cluster sequence to be aligned to any substring of the cluster center sequence. i.e. The gaps at both ends are not penalized when computing the alignment score.

5. *approximate filter:* Use heuristics to speed up the k-mer filter algorithm. The downside is that the clustering that is produced will not necessarily be an exact clustering. This may slightly increase the number of clusters created.

6. *well separated clusters:* Use a heuristic clustering strategy that guarantees the clusters to be well separated, i.e. each clustered sequences is guaranteed to be closer to the corresponding cluster center than to the center of any other cluster.

3.3 Discussion

Clustering is a basic problem in computational biology and other sciences. It is, however, a difficult problem. Finding the optimal solution, even in very simple models, is computationally prohibitive. Considering the fact that the amount of data created by sequencing machines is growing at a rapid pace (outpacing, in fact, Moore's law at the moment), efficient algorithms for clustering are needed.

In this dissertation we have focused on a simple greedy clustering approach. Our algorithm's running time depends on the data and the cluster radius threshold, and is particularly effective at high clustering thresholds. The efficiency of our algorithm is due to a new k-mer filtering approach, and to an efficient search strategy that allows us to quickly recruit sequences that could be assigned to a cluster. The software implementation is open-source and competitive with other commonly used clustering tools.

It should be noted that our focus has been simplicity, generality and performance. We do not claim to discover the underlying biological "truth". Such analysis require more sophisticated algorithms and are much more computationally intensive. Our tool can help in reducing the size of the data so that these, more intensive,

analyses can be applied. What we provide is a mathematically well defined clustering, which is the best we can hope for given that the biological truth has yet to be unambiguously characterized in mathematical terms.

We have compared our tool to state of the art software for clustering large numbers of sequences. An interesting observation is that the running time of DNACLUST decreases as the radius of the cluster is decreased, whereas the running time of UCLUST increases. Our search algorithm is more effective at higher stringencies, while the heuristics used by UCLUST are more effective when sequences are more dissimilar. This suggests that a faster algorithm could be developed that combines the best properties of both approaches.

Our k-mer based filter and alignment search algorithms can also be used for searching against databases that contain a large number of short sequences. In this scenario, the fact that our filter is completely sensitive is very useful. If the sequence database is static, our search data structures can be built once and stored on the disk for subsequent queries. We hope to add this functionality in future versions.

Finally, it is important to point out that there is still the need for even faster clustering tools. For very large datasets (over several million sequences), such as the ones produced by Human Microbiome Project, all available tools take more than a few days to run. Higher performance can possibly be achieved through parallel computation, and we intend to explore such approaches in future versions of our software.

3.4 Conclusions

The datasets analyzed by biologists are rapidly increasing in size and efficient clustering algorithms are necessary to help reduce the effective size of these datasets. Here we presented a novel approach for sequence clustering that is particularly well suited for high-stringency clustering, outperforming other state of the art clustering tools in this context. While our focus has been the analysis of 16S rRNA sequences, the algorithms we describe can be applied in other contexts as well, e.g. to identify duplicates in high-throughput sequencing data.

While more relaxed clustering thresholds are often used in metagenomic studies ($\leq 97\%$ similarity), using any fixed threshold is not a good approach for creating biologically meaningful clusters [38]. Here we have focused on creating rigorously defined, tight clusters. The representatives of these clusters can be used in further analysis (e.g. to create phylogenetically-informative clusters using more computationally intensive methods), in effect reducing the size of the original dataset.

The software implementing our clustering approach is freely available under an open-source license.

Acknowledgements

This was a joint work with Mihai Pop and Bo Liu. We thank Theodore Gibbons for helpful discussion and a careful review of a draft of this manuscript, and Robert Edgar for valuable input on the characteristics and algorithms underlying the UCLUST program.

We also thank the anonymous reviewers who helped us clarify and improve this manuscript.

This work was supported by the National Institutes of Health [R01-HG-004885 to M.P.]; the National Science Foundation [IIS-0812111 to M.P.]; and the Bill and Melinda Gates Foundation (PI Jim Nataro, subcontract to M.P.).

Chapter 4

Assembly evaluation

The genome of an organism usually consists of one or a few long DNA sequences. Unfortunately, current sequencing technology can only "read" relatively short pieces of DNA. The goal of genome assembly is to reconstruct the original genome, given these sequence fragments.

The above definition of genome assembly is somewhat vague. For example, depending on the sequencing technology and other parameters, some regions of the original genome may not be sampled (covered) at all. Furthermore, two different genomes may be indistinguishable based on short read fragments of fixed length.

There have been several attempts at formulating genome assembly as an algorithmic problem: shortest superstring, string graph, De Bruijn graph, etc. Each formulation tries to optimize a slightly different objective function. Most of these formulations are hard to solve in general [26]. They belong to the NP-hard class of problems, which implies that, as the size of the problem increases, finding a solution becomes intractable. However, given a solution (an assembly), it is relatively easy (polynomial time) to check whether it is correct (e.g. it contains all of the reads and achieves certain value of the objective function).

Even though the assembly problem is NP-hard, there are various heuristic assemblers which produce good solutions for practical instances. The question still

remains as to which assembly is the best for a given set of reads.

4.1 Theory

In this section, we first formalize the definition of the assembly problem as an optimization problem. Our objective function is the likelihood of an assembly being the truth (i.e. the sequenced genome), given our observations (i.e. the set of reads. However, it is easy to use other types of data, such as optical restriction maps, etc.)

The intuition behind our probabilistic criteria is that given a set of reads, we can not determine, with absolute certainty, the correct assembly. In fact, the same set of reads could have been observed from an infinite number of different genomes, with non-zero probability.

There are many other common criteria for evaluating assemblies such as the popular N50. (GAGE uses corrected N50. But to calculate corrected N50, one needs to have the true genome.) These criteria, however, are not suitable because the solution that maximizes them is not the best assembly. For example the concatenation of all of the reads results in an "assembly" with a very large N50, which is obviously not a good solution. We can prove that the true genome maximizes our probabilistic criteria.

4.1.1 Likelihood of an assembly

Let A denote the event that a given assembly is the true genome sequence, and let R denote the event of observing a given set of reads. (We use the same

symbol to denote the assembly and the event of observing the same assembly. The same symbol is also used to denote the set of reads and the event of observing the same set of reads.) Using Bayes' law, the probability of the assembly given the set of reads is

$$\Pr[A|R] = \frac{\Pr[R|A]\Pr[A]}{\Pr[R]} \tag{4.1}$$

First, we briefly discuss the probabilities $\Pr[A]$ and $\Pr[R]$, and then explain the computation of $\Pr[R|A]$. $\Pr[A]$ is the *prior probability* of observing the genome sequence A. In practice, it is hard to approximate the probability of a particular genome. If we have expert knowledge about the genome (e.g. approximate length, `GC`-content, that it must contain certain genes, etc.), it can be included in $\Pr[A]$. However, in absence of such information, and for relatively similar assemblies, we consider $\Pr[A]$ to be approximately a constant. Similarly, $\Pr[R]$ is the prior probability of observing the set of reads R. Since we want to compare assemblies that are constructed from the same set of reads, we can treat $\Pr[R]$ as a constant also. Thus, we can give a relative ranking among a set of assemblies by estimating $\Pr[R|A]$, that is, the probability of the observed read set R being produced from the assembly A.

Calculating $\Pr[R|A]$ is simple if we assume that the event of observing each read is independent. That is, given A, after observing k reads, the probability distribution for observing any sequence as the $k+1$ read, is unaffected by the previous k observations. Then the probability $\Pr[R|A]$ of read set R being produced from genome A, is the product of probabilities $p_i = \Pr[r_i|A]$ of observing reads r_i,

given genome A. That is,

$$\Pr[R|A] = \prod_{r_i \in R} p_i \tag{4.2}$$

(If the set of reads is unordered, we need to multiply $\prod p_i$ by $\left(\frac{|R|!}{\prod |R_j|!}\right)$, to account for different permutations that generate the same set of reads. However, this is a constant factor for a fixed set of reads.)

To compute the p_i probabilities, we need a model for the sequencing process. We discuss how to calculate p_is using increasingly complex models in the remainder of the chapter.

4.1.2 Sampling the reads

If the set of the reads is too large, we can use a random subset (a sample) of the reads, $R' \subseteq R$, to estimate the probability of the reads given the assembly.

The probability of a sample, decreases as the size of the sample is increased, because more terms are multiplied in formula (4.2). To counteract the effect of the size of the sample on our "score", we take the $|R'|$th root of the probability of the sample. In other words we use the geometric mean of the probabilities of the reads.

$$\mathrm{GM}(R') = \left(\prod_{r_i \in R'} p_i\right)^{\frac{1}{|R'|}}$$

The logarithm of this value is reported in our results.

$$\mathrm{LGM}(R') = \log\left(\mathrm{GM}(R')\right) = \frac{\sum_{r_i \in R'} \log p_i}{|R'|} \tag{4.3}$$

If the sample is too small, then the accuracy of the estimation would be low. In general, the mean of the reads probabilities over the sample R' is expected to be equal to the mean of the reads probabilities over R, and the standard deviation of the sample mean is proportional to $\frac{1}{\sqrt{|R'|}}$, and approaches zero as the size of the sample increases.

Proof: Let X be a random variable such that $\alpha \leq X \leq 0$. We want to estimate the mean $\mu = \mathrm{E}[X]$. The variance of X can be bounded by

$$\mathrm{Var}(X) = \mathrm{E}\left[(X-\mu)^2\right] \leq \mathrm{E}\left[\alpha^2\right] = \alpha^2$$

Let $S_n = \frac{\sum_{i=1}^{n} X_i}{n}$ be the mean of a sample of size n of X. By the central limit theorem, as n gets large, the distribution of S_n approximates normal with mean μ and variance $\frac{\mathrm{Var}(X)}{n}$. Therefore

$$\mathrm{stdev}\,(S_n) \approx \frac{\mathrm{stdev}\,(X)}{\sqrt{n}} \leq \frac{|\alpha|}{\sqrt{n}}$$

The probability that $\mu - a < S_n < \mu + a$ is therefore:

$$\Pr\left[|S_n - \mu| < a\right] = \mathrm{erf}\left(\frac{a}{\mathrm{stdev}(S_n)\sqrt{2}}\right) \geq \mathrm{erf}\left(\frac{a\sqrt{n}}{|\alpha|\sqrt{2}}\right)$$

4.1.3 Error-free model

The most basic model for the sequencing process is the *error-free model*. In the error-free model, the reads have a fixed length (This is technically necessary

because if the reads have different lengths, then p_i should include the probability of observing a read of that particular length. However, for assemblies of the same set of reads, these values would amount to a constant factor.), and do not contain any errors (e.g. insertions, deletions or substitutions). A read may be generated, starting at any position of the genome, with equal probability. Furthermore, for the sake of simplicity, we assume that the true genome consists of a single non-circular contiguous sequence, that our assembly is also a single contig, and that every read can be mapped to the assembly. Now calculating p_i is easy. If a read matches to one position in the reference, and its reverse-complement does not match anywhere in the reference, then the probability of the read being produced from the reference sequence is $p_i = \frac{1}{2(L-l)}$ where L is the length of the reference sequence, and l is the length of the reads. The factor 2 is due to the fact that current sequencing technology produces reads from both strands. Generally, if a read and its reverse-complement match at n_i positions on the reference, then

$$p_i = \frac{n_i}{2(L-l)} \tag{4.4}$$

This formulation is identical to the formulation of [25] which was developed for resolving repeat copy numbers in the context of building assemblies.

4.1.4 Maximizing the likelihood function

In this section we show that the true genome maximizes formula (4.3). Assume an error-free sequencing model. Also assume uniform coverage, which means that

the probability of observing a read is proportional to the number of times it appears in the true genome.

There are 4^l possible different reads of length l. Define p_i and q_i as the probability of observing read i, from the assembly, and the true genome, respectively. Therefore

$$\sum_{i=1}^{4^l} p_i = \sum_{i=1}^{4^l} q_i = 1$$

We want to find what p_i values maximize (4.3). Let P and Q denote the probability distributions corresponding to p_is and q_is, respectively. We want to find $\max_P S(Q, P)$ where

$$S(Q, P) = \log \left(\prod_{i=1}^{4^l} p_i^{q_i} \right)$$

Observe that $S(Q, P) = -H(Q) - D_{KL}(Q\|P)$, where H is the Shannon entropy, and D_{KL} is the Kullback-Leibler divergence. $D_{KL}(Q, P) \geq 0$, with equality iff $P = Q$. So $S(Q, P)$ is maximized when $p_i = q_i$.

Therefore the true genome, in expectation, maximizes the probability of observing the reads. Note that, from the point of view of formula (4.3), the assemblies that have the same read probability spectrum are the same.

It follows from our proof that an assembly which collapses a repeat, or in general contains the incorrect number of copies of a (possible) repeat gets a lower probability.

4.1.5 Assemblies containing more than one contig

Assemblers usually fail to produce a single contig. In the extreme case, an "assembler" may return the set of input reads as the assembled genome. We need to define the probability of observing a read given such a fragmented assembly. This probability depends on whether contigs are expected to be non-overlapping, in which case the length of the original genome must be *at least* the sum of lengths of the contigs: $\sum L_j$, where L_j is the length of the jth contig. Therefore, the probability of every read is *at most* $\frac{n_i}{2((\sum L_j)-l)}$. In practice, most assemblers join contigs only if they overlap by more than a certain number of bases. If the contigs are not supposed to overlap by more than o bases, the length of the sequence from which the reads were sampled, must be greater than or equal to $\sum(L_j - o)$. In this case, we need to replace L in (4.4) by $\sum(L_j - o)$.

4.1.6 Read quality values

Quality values are extra information produced by some sequencing technologies. They estimate the probability that a nucleotide was measured correctly. Therefore, the probability of observing a nucleotide depends on the reported quality of that observation.

Thus, for a read sequence r_i and corresponding vector of quality values q_i, we extend our definition of probability of each read to

$$p_i = \Pr[r_i, q_i|A] = \Pr[q_i|A] \Pr[r_i|q_i, A]$$

Probability of observing a sequence of quality values must be modeled based on the sequencing technology and the reference sequence.

Given an alignment of a read with quality values, the probability of each column of the alignment is easy to compute. The probability of observing the same nucleotide is q and the probability of a mismatch is $\frac{1-q}{4}$. The probability of a deletion in the read is not associated with a quality value however, and must be estimated separately.

4.2 Methods

In this section, we first discuss the necessary considerations for applying our formulas to real data. Then we present two methods (exact and approximate) for computing the probabilities of the reads given an assembly.

4.2.1 Practical considerations

Here we relax some of the assumptions we made in previous sections. We discuss how to handle sequencing errors, and what to do with reads that do not align to the assembly. We also show how to incorporate mate pair information in our probability formula (4.2).

4.2.1.1 Sequencing errors

All current technologies for sequencing DNA fragments have a small but significant probability of error. Depending on the sequencing technology, different types

of errors occur. Here we focus on three main types of errors: insertion, deletion, and substitution of a nucleotide.

In the error-free model, the probability of each read having been generated from any position j in the sequence is either zero or one. We now extend this model such that the probability of each read having been generated from any position j of the reference, is a value between zero and one, depending on the number of differences between the sequence of the read and the sequence of the reference at position j.

Let us denote the probability that read i is observed by sequencing the reference *ending* at position j by $p_{i,j}$. Then, the total probability of read i is

$$p_i = \frac{\sum_{j=l}^{2(L)} p_{i,j}}{2(L-l)}$$

For example, if we only allow for substitution errors with probability ϵ, and do not model insertion and deletion errors, then $p_{i,j} = \epsilon^s(1-\epsilon)^{l-s}$ where s is the number of substitutions needed to match read i to position j of the reference sequence. In the following sections we describe two methods for computing $p_{i,j}$ for a model that includes insertions and deletions.

4.2.1.2 Reads that do not align well

In practice, popular assemblers do not incorporate every read in the assembly. Possible reasons include: an assembly error (e.g. insufficient sensitivity of the assembler), erroneous reads , or contamination of the sequencing sample. In any

case, the fact that many reads may not align well to the assembly is a problem for our error-free model, because the probability of any assembly which does not incorporate every read would be zero.

We argue that for any read that does not align well, the overall probability of the assembly should not decrease more than the probability of the same assembly with the missing read appended at the end of its sequence. This is based on the observation that any assembly that does not incorporate all reads is equivalent to: the set of assembled sequences, and the set of "unassembled" reads, where each unassembled read is treated as an individual contig.

Let us calculate the change in the overall probability of an assembly if a new read (which does not align well to the assembly) is added to the set of reads, and also appended to the assembled sequence. Adding and appending a new read has two effects on the overall probability. First, the probability of observing this read exactly $\left(\frac{\Pr[\text{exact match}]}{2(L-l)}\right)$ must be multiplied into the product of the probabilities of all reads. Second, the probabilities of observing any of the other reads are decreased slightly since the total length of the assembled sequence has increased by the length of the appended read. For simplicity, let us assume an error-free model where each read maps to exactly one position on the assembled sequence. Let k denote the number of the original reads, then the ratio between the new probability for all original reads divided by their probability before adding the new read is:

$$\frac{\frac{1}{((L+l)-l)^k}}{\frac{1}{(L-l)^k}} = \left(\frac{L-l}{L}\right)^k = \left(1-\frac{l}{L}\right)^k \approx e^{-\frac{lk}{L}}$$

Therefore, if the probability of observing a read is less than

$$\frac{\Pr[\text{exact match}]}{2(L-l)} e^{-\frac{lk}{L}}, \tag{4.5}$$

we use (4.5) as its probability. We approximate the probability of exact match Pr[exact match] by $(1-\epsilon)^l$ where ϵ is the probability of an error (a mismatch, an insertion or a deletion).

4.2.1.3 Mate pairs

Many of the current sequencing technologies produce paired reads. Some assemblers use these information in building contigs and scaffolds.

If A consists of contigs (or scaffolds), then a pair of mated reads is supported by A if *both* reads match to the *same* contig (or scaffold) with correct orientation and distance. Some alignment search programs (e.g. Bowtie [18]) can align mated reads. If a pair of reads does not align in this way, it is considered unassembled.

4.2.2 Probability calculation via dynamic programming

For a model of sequencing process that allows insertions, deletions and substitutions with specific probabilities, we can exactly compute probability, $p_i = \Pr[r_i|A]$, of observing a read r_i given an assembly A using a dynamic programming algorithm. In general, we want to find the sum of probabilities of all possible alignments of a read to a position of the assembly.

The number of such possible alignments as a function of read length, grows

```
ACCG        ACCG
A-CG        AC-G
```

Figure 4.1: The same read may have multiple alignments to the same region of the reference, with relatively high probabilities. This figure shows two different optimal alignments of the same read to the assembly. Our dynamic programming algorithm calculates the sum of probabilities of all possible alignments, rather than only the probability of the best alignment.

extremely fast. Most of those alignments will have a very small probability. However, several alignments may have probabilities that are equal or close to optimal. For example, the two alignments of the same pair of sequences in Figure 4.1 have the same probability, and are both optimal alignments.

We use a dynamic programming algorithm (similar to the forward algorithm in hidden Markov models) to efficiently calculate the sum of probabilities of all alignments of a read to the assembly.

In general

$$p_i = \sum_{j=1}^{L-l} p_{i,j} + \sum_{j=1}^{L-l} p'_{i,j},$$

where $p_{i,j}$ and $p_{i,j}$ are the sum of the probabilities of *all* possible alignments of read i to, respectively, the reference and its reverse complement, ending at position j.

Let us describe how we use dynamic programming to efficiently calculate the sum of probabilities of all alignments of a read to the reference sequence.

We define $T[x, y]$ as the probability of observing prefix $[1 \ldots y]$ of the read, if y bases are sequenced from the reference ending at position x. Therefore, $p_{i,j} = T[j, l]$.

$T[x, 0]$ represents the probability of observing an empty sequence if we sequence zero bases, and therefore is set to 1. $T[0, y]$ represents the probability of observing prefix $[1 \ldots y]$ of the read, if y bases are sequenced from the reference

ending at position 0 (before the beginning), and is set to 0.

For $x \geq 1$ and $y \geq 1$, $T[x, y]$ is defined recursively:

$$\begin{aligned} T[x,y] =& T[x-1, y-1] \Pr[r[y]|A[x]] \\ &+ T[x, y-1] \Pr[\text{Insert}(r[y])] \\ &+ T[x-1, y] \Pr[\text{Delete}(A[X])], \end{aligned}$$

where $r[y]$ and $A[x]$ represent the nucleotides at positions y and x of the read r and the assembly A, respectively. $\Pr[r[y]|A[x]]$ is the probability of observing nucleotide $r[y]$ by sequencing nucleotide $A[x]$.

In our experimental computations, we do not distinguish between different types of errors and consider their probability to be ϵ, and the probability of observing the correct nucleotide to be $1 - \epsilon$.

This dynamic programming algorithm has a running time of $O(lL)$, per read. Even though the running time is polynomial, it is slow in practice. However, we can speed it up by using alignment seeds. The seeds would give us the regions of the assembly where a read may align with high probability. We can apply the dynamic programming only to those regions and get a very good approximate value of the total probability. We use exact seeds (k-mers) of a given length to build a hash index of the assembly sequence. Then, each read is compared to the regions where it has a common k-mer with the assembly sequence.

4.3 Discussion

There are several underlying assumptions in our framework. We assume the probabilities of observing any of the reads given that the reads were sequenced from a fixed genome/assembly, are independent. That is, we assume that the starting position of the reads, within the true genome, are independent random variables. This assumption is a reasonable approximation, but considering that the reads are obtained by shearing a finite number of copies of the genome, it is not totally accurate. Unfortunately, it is not simple to relax the independence assumption, because that would considerably complicate the probability formulas.

We also assume that the read coverage is uniform. This is highly unlikely for any given set of reads, and unfortunately it is theoretically impossible to get around this problem, assuming finite coverage. However, this is not very critical in practice, assuming relatively high coverage of reads that are *expected* to be uniformly distributed, and assuming that the assemblies are not very degenerate (e.g. some assemblies several times longer than the true genome.)

Even though expected uniform coverage is a popular assumption, it is not entirely accurate. All sequencing technologies have known biases, mostly depending on the sequence of the reads. For example the coverage may depend on the `GC` content of the sequence. However if we know these biases, we can incorporate them into our model for the sequencing process.

Another assumption that we made about the sequencing process is that the sequencing errors are independent (i.e. uncorrelated). One of the reasons that we

observe correlation between "errors" in practice, is sequence contamination. That is some of the reads come from a source which is not the target of sequencing. Again, theoretically there is no solution for this problem in general. In practice, there are two popular heuristic solutions: If the sequence of the source of contamination is known, we can filter those reads out. Otherwise, assuming the amount of contamination is small compared to the target genome, it may be possible to filter those reads out based on empirical analysis. In any case the filtering is a done prior to assembly. The evaluation of the assembly must not depend on the quality of the filtering, therefore if either of these filtering methods is used, we must evaluate the assembly by the set of filtered reads rather than the original set of reads.

Finally it is straightforward to use some other types of data with our framework. For example, we can evaluate assemblies based on optical restriction map fragments. Optical restriction map fragments show the (approximate) positions at which a specific pattern of bases appears in a sequence fragment. The advantage of restriction maps is that their fragments are much longer than DNA reads, and therefore can capture some long-range information that is lost if only DNA reads are used.

In order to calculate the probability of observing a restriction map fragment, we can simply build an in-silico optical map of the assembled contigs, and use a dynamic programming algorithm. This algorithm is a little more complex than the one we used for DNA fragments, because the model of errors for restriction maps is more complex. But the semantics of the probabilities are the same.

Acknowledgments

This was part of a joint work with Chris Hill, Irina Astrovskaya, Henry Lin, Dan Sommer and Mihai Pop. We thank Todd Treangen and Sergey Koren for their assistance with the GAGE dataset. We also thank Hector Corrada Bravo for assistance with the statistics.

Appendix A

Algorithms on strings

Algorithms on strings are fundamental in bioinformatics analysis. Many of these algorithms were developed in different branches of computer science (databases, artificial intelligence, theoretical computer science, etc.), but they have been adapted, improved and extended by bioinformatics researchers. In this section, we review a few of the most basic algorithms and data structures for string search. (We use the terms string search and string matching interchangeably. Also, the term "sequence alignment" sometimes, informally, refers to the search for approximate matches, as well as the computation of the alignment.) For a more detailed description of the algorithms and data structures in this chapter, see [13].

String search algorithms can be broadly classified by two different criteria: 1. whether the search is for exact or inexact matches, and 2. whether the data (the reference string) can be pre-processed before the search.

Before looking at some algorithms for each class, let us elaborate a little on the characteristics of these classes.

The exact search means we are looking for substrings of the reference which are equal to the query string (or in some cases, to a substring of the query string).

In the case of approximate search there are multiple definitions however, and they often depend on some parameters. But in general, an approximate match may

contain certain amount of different types of "errors" (i.e. differences between the query and the reference). If more errors are allowed, potentially more matches may be found, and also more computation is required.

In general, different errors may be weighted differently, and the weights are part of the input parameters. (For more detail, see the discussion of the dynamic programming algorithm for sequence alignment.)

The second classification criterion for string search algorithms is whether we are allowed to pre-process the reference string, before the search. The pre-process time is not included in the search time. During the pre-processing, auxiliary data-structures are created which help the search when the query becomes available. These data structures are collectively referred to as "indexes". Most index data-structures are based on either hashing or sorting.

Without an index the exact search time must be at least linear in terms of the sum of the length of the reference (n), and the length of the query (m). Using an index allows us to perform search in sublinear time in terms of the length of the reference (which is usually much longer than the query).

A.1 Exact string search

Exact string search (with or without an index) can be performed optimally. (i.e. Linear time in terms of the size of the input string(s).) In this section we briefly review a few algorithms and data structures connected with this problem.

A.1.1 Exact search without an index

There are several algorithms which can find exact matches in linear time (in terms of $n + m$). A classic example is the Knuth-Morris-Pratt (KMP) algorithm. Given a query and a reference, the KMP algorithm pre-processes the *query* and builds a deterministic finite automata in linear time. The reference is then fed to the automata, which requires constant amount of operations per symbol of the reference. A match is found if we arrive in a designated state.

A.1.2 Exact search with an index

There are several data structures that can be used for exact string search. Many of these data structures are based on the idea of pre-computing a lexicographically sorted list of the suffixes of the reference sequence. These suffix indexes are closely connected to each other, and it is often possible to transform one to the other, with minimal computation.

In the following we will describe two suffix indexes: the suffix tree and the suffix array. But first, let us describe a simpler structure, called the suffix trie, that allows for many of the same operations (but, in general, requires more space).

A suffix trie (for a string) is a rooted tree, with each edge labeled by a single symbol from the string. Each suffix corresponds to a path from the root of the tree to a unique leaf (and therefore the number of leaves is equal to the number of suffixes, which is equal to the length of the string). Note that the edges leaving any internal vertex must have distinct labels. Therefore, the number of children of each

vertex is less than or equal to the number of symbols in the alphabet.

Given a suffix trie of the reference string, to perform an exact search for a query string, we start at the root of of the trie, and follow the edges with labels corresponding to the symbols in the query. If we get stuck (i.e. there is no edge with the appropriate label), it means that the query does not have an exact match in the reference. On the other hand, if we consume all of the query and reach the end of it while traversing the trie, it means that there are one or more matches in the reference. The position of these matches are the suffixes which correspond to the leaves of the subtree rooted at the current vertex.

As we mentioned above, the space required for a suffix trie is, in general, quadratic in terms of the length of the string. We can decrease the space requirement by removing those internal nodes which have exactly one child, and by labeling the edges with substrings instead of single symbols. (Note that we must avoid storing these labels explicitly, otherwise the space requirement will asymptotically remain the same. The standard method for representing the edge labels is by storing the position of the substring within the original string.)

This representation is called a suffix tree, and requires only linear space. (The constant in the linear function is relatively large however. Because it is necessary to store several pointers per string symbol.) More importantly, it is possible to build a suffix tree in linear time also.

Given the suffix tree, the running time for exact search is linear in terms of the length of the query, and does not depend on the length of the reference. (Assuming that pointers and array indexes in the reference can be manipulated and accessed

in constant time.)

Another popular suffix index of a string is the suffix array. A suffix array is, simply, a list of all of the positions in the original string, such that the suffixes starting at these positions in the list, are in lexicographical order (i.e. as the words in a dictionary). The space requirement for suffix array is one pointer (or array index) per character of the string.

Given the suffix array, we can do exact search using a binary-search algorithm. Specifically, we compare the query string to the suffix which is at the middle of the suffix array, and depending on the results of the comparison, continue the search either within the top or the bottom half of the suffix array. This simple algorithm requires $O(m \log n)$ time. (That is, $\log n$ comparisons of strings of length m.) It is possible to simulate suffix tree operations (and performance, i.e. $O(m)$ search), by adding auxiliary information to the suffix array. Suffix arrays can also be built in linear time. (A naive method is to first build the suffix tree, and compute the suffix array by a depth-first traversal of the tree.)

A.2 Approximate string search

The definition of approximate string search depends on the concept of sequence similarity. Given a model for change in the sequence, two strings are similar if the amount of change between them is small.

The model for change may allow different types of changes. Different types of change may have different weights, which are often based on the probability of the

change, and are part of the model parameters.

If the model for change includes insertion and deletion of a character, the number of possible transformations (alignments) that could turn one string into another becomes exponential (in terms of the lengths of the strings). In this case, to find the best alignment efficiently, we must avoid trying all possible alignments. A general algorithmic technique, known as "dynamic programming", allows us to find the best alignment in polynomial time.

To illustrate the dynamic programming technique, let us use the classic problem of computing the "edit distance" between two strings. The edit distance between two strings is the minimum number of insertions, deletions and substitutions of a single character that transforms one string into the other. This distance (like most of the other sequence similarity measures discussed here) has a close relationship with pairwise sequence alignment. A pairwise sequence alignment is obtained by (potentially) inserting "gaps", within the two strings, such that similar characters from the two strings are "aligned" to each other. The length of the two modified strings must be equal. The characters that appear in at the same position (column) in the two modified strings are aligned to each other. We try to make similar characters appear in the same column of the alignment. The gaps in the alignment correspond to insertions and deletions. A substitution corresponds to two different characters aligned to each other.

The edit distance is the minimum distance induced by any alignment between the two strings. Note that it is possible to have more than one alignment which corresponds to the edit distance.

To compute the edit distance between two strings efficiently, we define it in terms of three smaller problems (of computing the edit distance if we remove either one, or both of the last characters of the two strings.) Given these three results we can compute the edit distance, in constant time, by comparing only the last two characters. The dynamic programming technique solves these sub-problems (edit distance between the prefixes of the strings) in order of increasing size, and stores the results in a two dimensional array (a table). Therefore the total running time is $\Theta(n_1 \times n_2)$.

An alignment corresponding to the edit distance can be found by tracing the cells in the table which lead to the minimum distance.

A.2.1 Alignment search without an index

There are two slightly different variations of this problem: Given a query and a reference, 1. we want to find all substrings of the reference string that have a "good" alignment to the query string, (also known as approximate string matching. The results are called semi-global alignments), or 2. we want to find alignments for any substring of the query (local alignment).

The scores for substitutions and gaps are provided as input parameter. The quality of an alignment is defined by the sum of the scores of its columns.

Both of the problems above can be solved using slight variations of the dynamic programming algorithm that we described for the edit distance problem. For example, in the case of semi-global alignment search, we only need to initialize the

table in the following way: the score of aligning any prefix of the reference to the empty prefix (of the query) is set to zero. (i.e. The alignment can start at any position in the reference sequence without penalty.) Once the table is filled, the best semi-global alignment is found by taking the maximum score for aligning the complete query string to any prefix of the reference.

These dynamic programming algorithms have running time in $O(m \times n)$.

A.2.2 Alignment search with an index

The quadratic running time of the dynamic programming algorithm is slow for very long query and reference sequences. By building an index of the reference (and, possibly, also the query), we hope to avoid comparing every region of the reference to every region of the query.

There are many theoretical algorithms for this problem, which is more commonly known to computer scientists as "approximate string matching". These results often assume a simple distance (e.g. edit distance), and the running time has an exponential dependence on the amount of error which is tolerated.

These theoretical results are focused on the worst case running time. They do not make any assumptions on the structure of the strings. On real life datasets (such as biological sequences), a simple heuristic known as seed-and-extend algorithm is faster in practice.

The main intuition behind seed-and-extend is that, if we can quickly identify potential regions for alignment, we can apply the more time consuming dynamic

programming algorithm to those regions only. For example, if we want local alignments of length 104, with at most 4 errors, then any such alignment must contain a substring of length 25 that appears exactly in both the query and the reference.

Therefore, if we design the alignment seed parameters carefully, we can guarantee that the seeds are completely sensitive. Unfortunately, we can not bound the number of false positives in general (because it depends on the content of the reference and the query), which makes it hard to theoretically bound the running time of the seed-and-extend heuristic.

Bibliography

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[3] S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger, and E.S. Lander. Human and mouse gene structure: comparative analysis and application to exon prediction, 2000.

[4] J.L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms*, pages 360–369. Society for Industrial and Applied Mathematics, 1997.

[5] M. Brudno, C.B. Do, G.M. Cooper, M.F. Kim, E. Davydov, N.C.S. Program, E.D. Green, A. Sidow, and S. Batzoglou. LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA, 2003.

[6] AL Delcher, S. Kasif, RD Fleischmann, J. Peterson, O. White, and SL Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369, 1999.

[7] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478, 2002.

[8] R.C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic acids research*, 32(5):1792, 2004.

[9] R.C. Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460, 2010.

[10] J. Felsenstein. PHYLIP (phylogeny inference package) version 3.6. *Distributed by the author. Department of Genome Sciences, University of Washington, Seattle*, 2005.

[11] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[12] M. Ghodsi and M. Pop. Inexact Local Alignment Search over Suffix Arrays. In *2009 IEEE international conference on bioinformatics and biomedicine*, pages 83–87. IEEE, 2009.

[13] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge Univ Pr, 1997.

[14] S.M. Huse, D.M. Welch, H.G. Morrison, and M.L. Sogin. Ironing out the wrinkles in the rare biosphere through improved OTU clustering. *Environmental microbiology*, 12(7):1889–1898, 2010.

[15] L. Ilie and S. Ilie. Multiple spaced seeds for homology search. *Bioinformatics*, 23(22):2969, 2007.

[16] S. Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 29(13):1149–71, 1999.

[17] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[18] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

[19] M. Li, B. Ma, and L. Zhang. Superiority and complexity of the spaced seeds. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 444–453. ACM New York, NY, USA, 2006.

[20] W. Li and A. Godzik. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658, 2006.

[21] W. Li, L. Jaroszewski, and A. Godzik. Clustering of highly homologous sequences to reduce the size of large protein databases. *Bioinformatics*, 17(3):282, 2001.

[22] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search, 2002.

[23] B.L. Maidak, J.R. Cole, T.G. Lilburn, C.T. Parker Jr, P.R. Saxman, R.J. Farris, G.M. Garrity, G.J. Olsen, T.M. Schmidt, and J.M. Tiedje. The RDP-II (ribosomal database project). *Nucleic acids research*, 29(1):173, 2001.

[24] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 1990.

[25] P. Medvedev and M. Brudno. Maximum likelihood genome assembly. *Journal of computational Biology*, 16(8):1101–1116, 2009.

[26] P. Medvedev, K. Georgiou, G. Myers, and M. Brudno. Computability of models for sequence assembly. *Algorithms in Bioinformatics*, pages 289–301, 2007.

[27] S. Navlakha, J. White, N. Nagarajan, M. Pop, and C. Kingsford. Finding biologically accurate clusterings in hierarchical tree decompositions using the variation of information. In *Proc. 13th Intl. Conf. on Research in Computational Molecular Biology (RECOMB)*, volume 5541, pages 400–417, 2009.

[28] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[29] C. Quince, A. Lanzén, T.P. Curtis, R.J. Davenport, N. Hall, I.M. Head, L.F. Read, and W.T. Sloan. Accurate determination of microbial diversity from 454 pyrosequencing data. *Nature methods*, 6(9):639–641, 2009.

[30] P.D. Schloss and J. Handelsman. Introducing DOTUR, a computer program for defining operational taxonomic units and estimating species richness. *Applied and environmental microbiology*, 71(3):1501, 2005.

[31] P.D. Schloss, S.L. Westcott, T. Ryabin, J.R. Hall, M. Hartmann, E.B. Hollister, R.A. Lesniewski, B.B. Oakley, D.H. Parks, C.J. Robinson, et al. Introducing mothur: open-source, platform-independent, community-supported software for describing and comparing microbial communities. *Applied and environmental microbiology*, 75(23):7537, 2009.

[32] TF Smith and MS Waterman. Identification of common molecular subsequences. *J. Mol. Bwl*, 147:195–197, 1981.

[33] J.D. Thompson, D.G. Higgins, and T.J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic acids research*, 22(22):4673, 1994.

[34] P.J. Turnbaugh, M. Hamady, T. Yatsunenko, B.L. Cantarel, A. Duncan, R.E. Ley, M.L. Sogin, W.J. Jones, B.A. Roe, J.P. Affourtit, et al. A core gut microbiome in obese and lean twins. *Nature*, 457(7228):480–484, 2008.

[35] E. Ukkonen. Approximate string-matching over suffix trees. In *Combinatorial Pattern Matching*, pages 228–242. Springer, 1993.

[36] V.V. Vazirani. *Approximation algorithms*. Springer Verlag, 2001.

[37] Q. Wang, G.M. Garrity, J.M. Tiedje, and J.R. Cole. Naive Bayesian classifier for rapid assignment of rRNA sequences into the new bacterial taxonomy. *Applied and environmental microbiology*, 73(16):5261, 2007.

[38] J.R. White, S. Navlakha, N. Nagarajan, M.R. Ghodsi, C. Kingsford, and M. Pop. Alignment and clustering of phylogenetic markers- implications for microbial diversity studies. *BMC bioinformatics*, 11(1):152, 2010.