Algorithms for clustering highly conserved phylogenetic markers

A prospectus submitted in partial fulfillment of the degree of Doctor of Philosophy

Preliminary Oral Examination for
Christopher Michael Hill

Advisor:
Dr. Mihai Pop

Committee Members:
Dr. Atif Memon
Dr. Héctor Corrada Bravo

Department of Computer Science
University of Maryland, College Park, MD 20742

April 28th, 2014

Abstract

Microbes play a large role in every aspect of our lives. They aid in digestion of food, have been associated with different diseases, and even help identify which keyboard and mouse we use. In the environment, these microbes are responsible for most of the biochemical cycles that allow life to exist on earth, such as the carbon and nitrogen cycles. Despite their importance, these organisms are among the least understood on Earth due in part to the difficult nature of culturing them in a laboratory setting. Fortunately, advances in sequencing technology allow us to capture a small, noisy snapshot of the microbial community. In typical analyses, the short fragments produced from these sequencing technologies first need to be clustered into similar groups based on some definition of similarity.

In this prospectus, we will describe the current approaches and limitations for clustering biological sequences, which are represented electronically as strings of characters. First, we will discuss multi-core algorithms that speedup the runtime of clustering. Then, we will describe efficient string matching algorithms to quickly recruit sequences to potential centers. By sorting highly similar sequences, we provide a novel way to reduce the amount of work done during cluster recruitment. Lastly, we will discuss the problematic issue of sequences that can align equally well to multiple centers and present strategies for handling them.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Microbes play a large role in every aspect of our lives. They aid in digestion of food[15], have been associated with different diseases[32], and even help identify which keyboard and mouse we use[11]. In the environment, these microbes are responsible for most of the biochemical cycles that allow life to exist on earth, such as the carbon and nitrogen cycles[41]. Despite their importance, these organisms are among the least understood on Earth due in part to the difficult nature of culturing them in a laboratory setting.

In metagenomics projects, we study the sequences directly recovered from environmental samples[42]. Advances in sequencing technology allow us to capture a snapshot of the microbial community. Unfortunately, this snapshot only can provide us with a very small, noisy picture of the microbial community. A common genome size of a bacteria is often several million base pairs (bps) long and serves as the blueprint for building that organism. Due to limitations in sequencing technology, we are only able to produce small fragments (called *reads*), a couple hundred base pairs in length, from the organism. These sequences are represented electronically as strings of characters. Depending on the specific sequencing technology used, these fragments can be drawn randomly from the underlying genomes or from a specified region.

Often times, we want to learn what species of bacteria are found in an environment and their relative abundances. Relative bacterial abundances often vary in several orders of magnitude. Using methods to draw reads randomly from the underlying genomes – called whole genome shotgun (WGS) – are very expensive if we want to find the rarer species of an environment. Fortunately, researchers have developed primers which allow us to sequence a specific conserved gene – called the 16s ribosomal RNA (or 16s rRNA) gene) – found in a large number of bacteria. This gene is useful for phylogenetic studies because it is highly conserved between different species of bacteria. 16s rRNA genes contain nine hypervariable regions (V1-V9) that accumulate mutations overtime, providing us with species-specific signatures used for identifying bacteria.

A typical metagenomic sequencing project involves first selecting or designing a primer that starts in a conserved region of the 16s rRNA gene and then allows the sequencing to extend through a couple hypervariable regions. Once the reads have been generated, we now need to determine what reads came from what bacteria. Since many bacterial species we encountered have never been seen before[15], we need to cluster (or group) similar reads together into operational taxonomic units (OTUs) based on sequence similarity.

The problem of clustering is a widely-studied in computer science[10] and serves as the basis of this prospectus. We will first outline the current approaches used to cluster highly similar sequences. Then we will describe our improvements to these existing methods made by utilizing multiple processors and efficient string matching algorithms.

2 Related Work

Sequence clustering tools often fall into two specific categories: *hierarchical* or *greedy*. In this section, we will discuss each clustering paradigm and their popular implementations (Table 1).

2.1 Hierarchical clustering

The traditional approach for clustering sequences has been to use a hierarchical clustering method. These methods start with a pairwise distance matrix and then iteratively merge similar points into clusters. Distance between two sequence can be the minimum number of edits (insertions, deletions, substitutions) to transform one sequence into the other (called *edit distance*), the *similarity* (defined as the $1 - \frac{\text{edit distance}}{|\text{sequence}|}$), and the *identity* (fraction of bases that are identical between the two sequences). In addition to the above distances, we can create a vector of counts for all k-length substrings of the sequences (called k-mers) and calculate some distance measure (such as euclidean) between the resulting vectors. It is important to note that not all distance measures are created equal, though. A distance metric must obey the triangle inequality, i.e., for every sequences A, B, and C, $dist(A, B) + dist(B, C) \leq dist(A, C)$. The identity measure does not follow the triangle inequality.

Once we have selected a distance measure, we need to calculate the pairwise distances for all sequences to create our distance matrix. Lets assume we have selected edit distance for our distance measure. Edit distance between two sequences of length n can be calculated via dynamic programming, a strategy for solving problems by breaking them down into “simpler” subproblems, in $O(n^2)$ time and work. We will discuss the algorithms used in more detail in the next section.

Once the pairwise distance matrix is built, clustering is typically done via hierarchical methods. Agglomerative methods involve a bottom-up approach, where each sequence starts as its own cluster then iteratively

Tool	Description	DNA	Protein	*De novo*	Ref	Exact	Multi-core	Distance measure	Strategy
BLAST[2]	Alignment tool using neighborhood keywords to speed up alignment A script can be used to align to a reference database and filter results.	Y	Y	N	Y	N	Y	N/A	N/A
AbundantOTU[43]	Fast method to build cluster centers from abundant k-mers	Y	N	Y	N	Y	N	Sim	Greedy
CD-HIT[25, 12]	Family of clustering tools. Speed comes from using a short word filter to prune potential alignments.	Y	Y	Y	N	N[1]	Y	Sim	Greedy
DNACLUST[14]	Fast, exact clustering tool that exploits database sequence similarity to reduce work.	Y	N	Y	Y	Y	Y	Sim / Mis	Greedy
DySC[44]	Uses a two-step clustering heuristic to build larger clusters.	Y	N	Y	N	N	N	Sim	Greedy
ESPRIT-Tree[5]	Hierarchical method that represents a cluster of sequences as a single probabilistic sequence and uses that for pairwise alignments of clusters.	Y	N	Y	N	N[2]	N	Sim	Hierarchical
HPC-CLUST[36]	Scalable, and flexible hierarchical method for clustering pre-aligned reads.	Y	N	Y	N	N/A[3]	Y	Sim	Hierarchical
SEED[3]	Very fast clustering of highly similar reads within 3 mismatches.	Y	N	Y	N	Y	N	Mis	Greedy
UCLUST[8]	Fast clustering method that only compares a sequence to a predetermined number of potential centers.	Y	Y	Y	Y	N[4]	Y	Sim	Greedy

Table 1: Clustering tools used in metagenomic studies. DNA and Protein refers to whether the clustering tool can work with DNA and protein sequences, respectively. *De novo* and Ref refers to whether the clustering tool can cluster sequences without and with a reference database, respectively. Exact clustering tools are those that compare a sequence to every cluster center. Multi-core clustering tools are those that support multiple threads. Distance measure refers to tools that define distance between two sequences as percent similarity or mismatch number. Strategy defines how the centers are created with greedy methods having some fixed procedure of selecting the center from the dataset, while hierarchical refers to those that are constructed using a guide tree.

merged with other clusters. ESPRIT-Tree[5] and HPC-CLUST[36] are both fast agglomerative methods. ESPRIT-Tree relies on fast k-mer filtering methods to reduce the amount of pairwise distance computations required to build the tree. HPC-CLUST, unlike ESPRIT-Tree, does not compute the pairwise distances itself, but provides the user with efficient tree building algorithms using a variety of inter-cluster distance measures. Divisive methods, on the other hand, work from a top-down approach, where all sequences belong to a single cluster then are iteratively split into smaller clusters.

[1] Only exact for a preset number of similarities.
[2] Default k-mer length filter does not guarantee correctness.
[3] Takes as input pairwise sequence distances.
[4] Default is inexact, but also offers an exact alignment mode.

2.2 Greedy clustering

Calculating the pairwise edit distance between N sequences of length n will take $O(N^2n^2)$ work. This amount of time is impractical for the large number of sequences produced by sequencing technologies today. 454 sequencing by the 454 Life Sciences company can produce tens of millions of sequences. An alternative approach to hierarchical clustering sequences involves iteratively selecting a sequence to serve as a cluster center and then recruiting all remaining sequences that fall within some given distance of the cluster center. This process is repeated until no more sequences remain or the predetermined list of centers is exhausted. In this section, we will discuss the different strategies for selecting centers and the heuristics used to speed up the alignment of sequences to centers.

2.2.1 Cluster center selection

There are three strategies used for selecting potential cluster centers: ***de novo***, **closed-reference**, and **open-reference**.

In ***de novo*** clustering, centers are selected *only* from the set of input sequences. Selecting which sequence to use as the cluster center is a difficult problem. One strategy is to select the remaining sequence with the longest length, as is done by CD-HIT[25], DNACLUST[14], and UCLUST[8]. An intuitive argument is that a longer sequence would more likely recruit shorter sequences. However, the more rigorous argument is that selecting the longest remaining sequence allows you certain guarantees when aligning and recruiting shorter sequences. For example, given 3 sequences: A, B, and C (Figure 1). If the length of B (our center) is less than A and C and we know *dist(*A,B*)* and *dist*(B, C), we can not say anything for certain about the *dist*(A, C). We do not have any data about the overlapping regions between A and C that extend past the end of B. As we will explain in the next section, this problem exists only for semi-global alignment where gaps at the start and end of the sequences are not penalized. If we used global alignments then overhanging regions of A and C would be penalized.

In addition to sorting by length, UCLUST providers users with alternative methods for selecting the centers. UCLUST assumes that the sequences are sorted in an order such that a *good* centroid is found before other sequences in its cluster. When sequences have a large variation in length, the recommended strategy is to sort the sequences by length first as with CD-HIT and DNACLUST. However, it is possible to assign an abundance to each sequence prior to clustering. The user can run a fast deduplication program prior to

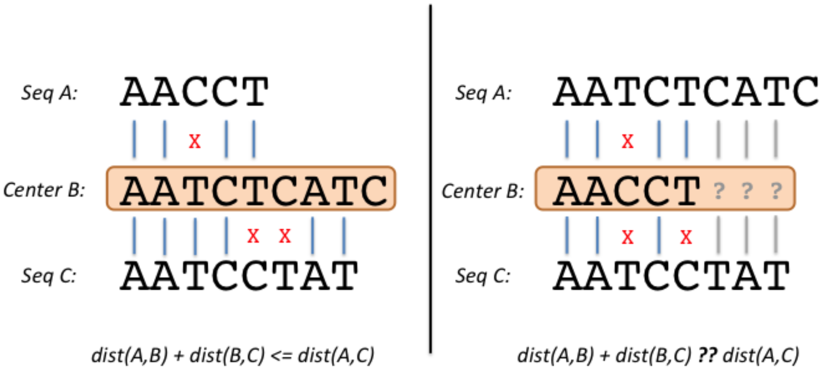


Figure 1: The longest sequence is chosen as the center (left side) so that we can make transitive application of the distances of sequences within the cluster. If the longest sequence is not chosen as the center (right side), then we can not make any claims about distance of the overhanging regions of other sequences in the cluster.

clustering to get a pseudo-cluster count prior to running UCLUST.

AbundantOTU[43] uses a different strategy to construct its cluster centers. It starts by finding the most abundant k-mer (called the seed). Then the seed is greedily extended one nucleotide at a time, selecting the nucleotide that has the highest similarity with the input sequences that share the same seed. AbundantOTU works well at assembling highly abundant OTUs because it is robust to sequencing errors. Errors within OTUs should occur less frequently than their correct bases in abundant clusters. The authors of AbundantOTU recommend using the tool to cluster the abundant OTUs than using another clustering strategy for the remaining sequences.

DySC[44] attempts to have fewer, higher quality clusters by using temporary centers. After a predefined number of sequences are aligned to the temporary center, the cluster becomes *fixed* and a new center is calculated based on shared k-mers of the reads in the cluster. Afterwards, all reads are re-aligned to the new center to insure the similarity constraints still hold.

SEED[3] is a clustering tool that was not designed for taxonomic analysis. Instead, SEED is used to cluster short WGS reads within a few number of mismatches to improve assembly. It performs an approach similar to deduplication, collapsing highly redundant reads. While information is lost, the authors show that they can improve runtime and quality of the resulting assemblies. The cluster centers are chosen arbitrary and finding potential member sequences is done via fast hashing of small seeds.

In **closed-reference** clustering, a list of predetermined centers is given, such as a collection of previously discovered or manually-curated OTUs[7, 33]. 16S rRNA pipelines such as QIIME[6] and Mothur[38] often allow users to supply their own OTU databases. Given a database of sequences, it is possible to use a read alignment tool, such as BLAST[2], to align the sequences. The user could write a script to post-process the alignments to make sure they fit some criteria, e.g., have a certain minimum similarity or identity.

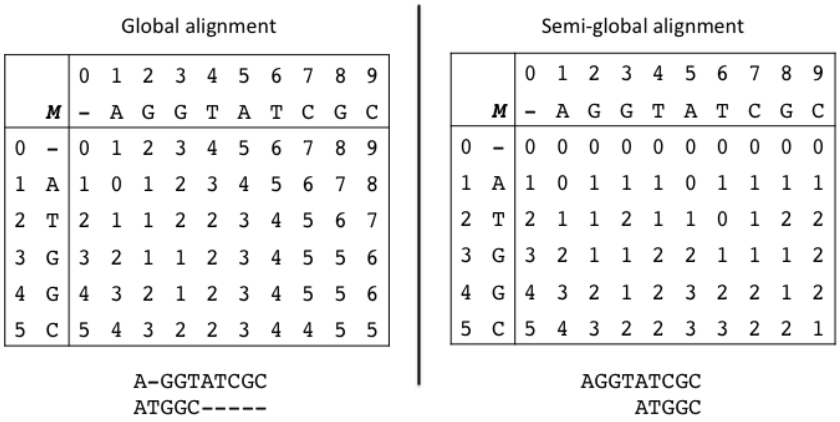
In **open-reference** clustering, sequences are first recruited to a list of predetermined cluster centers. This clustering strategy is provided QIIME. Afterwards, the centers are chosen by the *de novo* methods.

2.2.2 Recruiting sequences to clusters

A key part of any clustering algorithm is how the distance between two objects is computed. In the case of sequence clustering, we need to calculate the distance between two strings. In this section, we will explain how the *edit distance* is calculated.

Definition 2.1. *The edit distance between a text string $t = t_1t_2...t_n$ and pattern string $p = p_1p_2...p_m$ is the minimum number of differences between them such that one string can be transformed into the other. A difference is one of the following:*

1. *A character of the text corresponds to a different character of the pattern.*
2. *A character of the text corresponds to no character (a gap) in the pattern.*
3. *A character of the pattern corresponds to no character (a gap) in the text.*



Global alignment

		0	1	2	3	4	5	6	7	8	9
	M	-	A	G	G	T	A	T	C	G	C
0	-	0	1	2	3	4	5	6	7	8	9
1	A	1	0	1	2	3	4	5	6	7	8
2	T	2	1	1	2	2	3	4	5	6	7
3	G	3	2	1	1	2	3	4	5	5	6
4	G	4	3	2	1	2	3	4	5	5	6
5	C	5	4	3	2	2	3	4	4	5	5

```
A-GGTATCGC
ATGGC-----
```

Semi-global alignment

		0	1	2	3	4	5	6	7	8	9
	M	-	A	G	G	T	A	T	C	G	C
0	-	0	0	0	0	0	0	0	0	0	0
1	A	1	0	1	1	1	0	1	1	1	1
2	T	2	1	1	2	1	1	0	1	2	2
3	G	3	2	1	1	2	2	1	1	1	2
4	G	4	3	2	1	2	3	2	2	1	2
5	C	5	4	3	2	2	3	3	2	2	1

```
AGGTATCGC
    ATGGC
```

Figure 2: Global vs. semi-global alignment. In global alignment, the start and end gaps count as mismatches. In semi-global alignments, the start and end gaps of the text are not penalized.

The Needleman-Wunsch[28] algorithm was the first application of dynamic programming to compare sequences. We can use this algorithm to also calculate the edit distance between two strings. At a high level, we find the optimal alignment of two sequences by looking at the optimal alignment of the prefixes of the two sequences. In our work, the text t is the cluster center and the pattern p is the current sequence in our database. We are trying to find occurrences of the pattern(sequence) in our text (center) with a given number of edits.

Let M be an $(m+1) \times (n+1)$ table. The value at cell $M(i,j)$ is the minimum number of edits to transform $p_1..p_i$ into $t_1..t_j$. We can find this value by examining the three types of edits. In the first case, lets assume we have the edit distance of aligning $p_1..p_{i-1}$ and $t_1..t_{j-1}$ (stored in $M(i-1,j-1)$) and then we add 1 edit if characters p_i and t_j mismatch, 0 otherwise. The second case corresponds to aligning $p_1..p_{i-1}$ and $t_1..t_j$ (stored in $M(i-1,j)$) and then inserting a gap into t, which increases the edit distance by 1. The final case corresponds to aligning $p_1..p_i$ and $t_1..t_{j-1}$ (stored in $M(i,j-1)$) and then inserting a gap into p, which increases the edit distance by 1. The minimum edit distance of aligning $p_1..p_i$ and $t_1..t_j$ is the minimum of the three cases above. These cases can be rewritten as a dynamic programming recurrence:

$$M[i,j] = \quad min \begin{cases} M[i-1,j]+1 \\ M[i,j-1]+1 \\ M[i-1,j-1] + \begin{cases} 0, & \text{if } p_i == t_j \\ 1, & \text{else} \end{cases} \end{cases} \tag{1}$$

The initial conditions of the dynamic programming table depend on whether we want the global or semi-global alignment. Unlike with global alignment, the semi-global alignment of two strings does not penalize start and end gaps of the text (Figure 2). For both global and semi-global alignments, $M(i,0) = i$ for $0 \leq i \leq m$. Global alignment penalizes start gaps in the text, so $M(0,i) = i$ for $0 \leq i \leq n$. For semi-global alignment, $M(0,i) = 0$ for $0 \leq i \leq n$.

Once the dynamic programming table M is populated, we need to specify where to find the solution, i.e., the minimum edit distance between the pattern and text. In the case of global alignment, the solution will be located in cell $M(m,n)$. For semi-global alignments, we do not penalize end gaps in the text, so the solution lies in the cell that contains the minimum value of row m of M. The final algorithm is presented in Algorithm 1.

Algorithm 1 Compute global edit distance between two strings. $O(nm)$ work.

```
procedure ComputeEditDistance($text, pattern$)
    $n \leftarrow |text|$
    $m \leftarrow |pattern|$
    for $i = 0..m$ do
        $M(i, 0) \leftarrow i$
    for $i = 0..n$ do
        $M(0, i) \leftarrow i$
    for $i = 1..n$ do
        for $j = 1..m$ do
            $row \leftarrow M(i-1, j) + 1$            ▷ Number of edits with a gap inserted into $text$
            $col \leftarrow M(i, j-1) + 1$            ▷ Number of edits with gap inserted into $pattern$
            $diag \leftarrow M(i-1, j-1)$            ▷ Number of edits with matching characters $p_i$ and $t_j$
            if $a_i \neq b_j$ then $diag \leftarrow diag + 1$
            $M(i, j) \leftarrow \min(row, col, diag)$
    return $M(n, m)$
```

Since we are populating a table with $(m + 1) \times (n + 1)$ values and doing a constant amount of work per cell (looking up the value of three adjacent cells) the total time and space requirement to calculate the edit distance between two strings is $O(nm)$. The space requirement can be relaxed to $O(n)$ using a divide-and-conquer approach[16]. Similarly, if we set a bounds on the maximum number of edits (k), we can reduce the runtime even further. If we want to find a global alignment within k-differences (edits), we only need to worry about a $2k + 1$ band along the diagonal. This reduces the time complexity from $O(nm)$ to $O(nk)$. However, this assumes we are aligning the two sequences end-to-end. In the next section, we will outline the steps necessary to reduce this to $O(nk)$ for semi-global alignments.

3 Methods

In this section, we will describe our preliminary work parallelizing DNACLUST to work with the ever-increasing amount of sequencing data produced by today's sequencing technologies. Then, we will propose a new clustering tool that uses an efficient k-difference string matching algorithm. Finally, we will discuss how to improve the quality of each cluster by handling the alignment of ambiguous reads, an issue largely ignored by existing clustering tools.

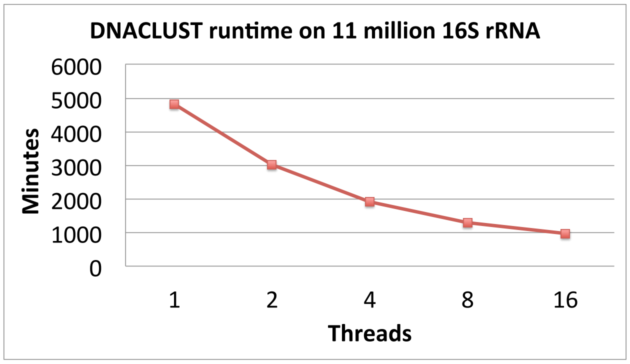


Figure 3: Speedup of DNACLUST using multiple threads.

3.1 Parallelizing sequence recruitment to a cluster center

Due to the large scale of sequencing data produced, clustering tools must utilize multiple processors to process the data in a timely manner. Only half the tools presented in Table 1 are capable of multi-core support. Fast clustering tools such as CD-HIT and UCLUST already implement support for multiple threads. Here, we present two parallel approaches for recruiting sequences to a cluster center within DNACLUST. The first approach (naïve) is based on evenly partitioning the sequences among the processors. The second approach (work-based) involves partitioning the sequences based on the potential work that needs to be done when calculating the edit distance to the center. If the sequences are stored in a trie-like data structure, then it is beneficial to partition highly similar sequences together despite potentially assigning an uneven number of sequences to each processor. We implement these parallel approaches in DNACLUST show the speed-ups when clustering tens of millions of 16S rRNA sequences.

3.1.1 Naïve parallelization strategy

The second step of DNACLUST's algorithm involves recruiting all sequences that lie within a given distance of the current cluster center. Given p processors, we can evenly partition the database into p chunks such that each processor can calculate the edit distance independently in parallel. We have added pthread support to the DNACLUST code, allowing us to assign a contiguous chunk of the sequence database to each thread.

We test this parallelization using 11 million 16S rRNA (average length of 350 bps) (Figure 3). Going from one to two threads, and two to four threads results in a 1.6× speedup.

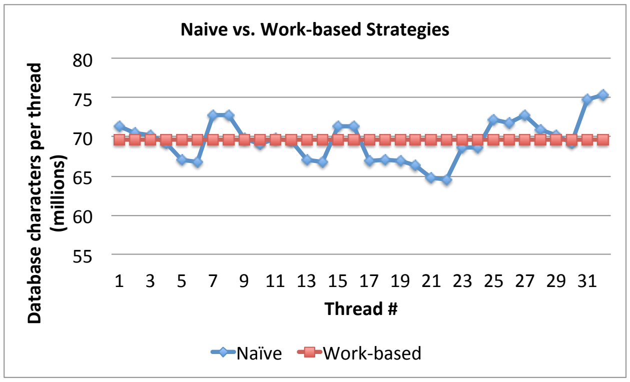


Figure 4: Comparing the per-thread workload of the naïve and work-based parallelization methods.

3.1.2 Work-based parallelization strategy

DNACLUST keeps the database of sequences sorted to speed up the calculation of edit distance between lexicographically adjacent sequences. Adjacent sequences that share a similar prefix allow us to reuse part of the dynamic programming table. Evenly partitioning the sequences may split highly similar sequences into separate threads causing us to repeat work. Instead of evenly dividing the number of sequences between threads, we can evenly divide the amount of potential work (characters we need to examine in the trie).

Definition 3.1. *The* **trie length** *is the total number of characters on the edges in the trie.*

Given a sorted database of sequences, we can calculate the trie length by iterating through each sequence and adding the length of the current sequence minus the longest common prefix of the current sequence and the previous sequence. Once the trie length is calculated, we divide the trie length by the number of threads, giving us the amount of work per thread. To actually assign sequences to threads, we repeat the above procedure, but once the current trie length exceeds the work per thread, we assign those sequences to the specific thread and set the current trie length to 0.

We compare the trie lengths of the naïve and work-based methods using 32 threads (Figure 4). Since we are only as fast as our slowest thread, in the worst case, the naïve method would require 12% more work than the work-based method. Future work involves implementing this feature into DNACLUST.

3.1.3 Parallelizing edit distance computation

Thus far, our parallelization approaches have been focused on partitioning whole sequences. We can parallelize the edit distance calculation since the data dependencies when updating a cell are limited. Each cell only relies on the values stored in the horizontally, vertically, and diagonally adjacent cells. Thus, it is possible to calculate the values along the anti-diagonals in parallel. A group of cells can be updated in a single operation using a feature of most modern processors: SIMD instructions (single instruction multiple data). The speedup is dependent on how many cells we can fit into the registers used for SIMD. Results show that speedups of around six times can be achieved using modern systems[9, 37]. While this speedup has a ceiling (the size of the SIMD register), the speedup will be noticeable in practice. Our proposed work includes implementing this feature into DNACLUST's edit distance computation.

3.2 Efficient data structures for edit distance computation

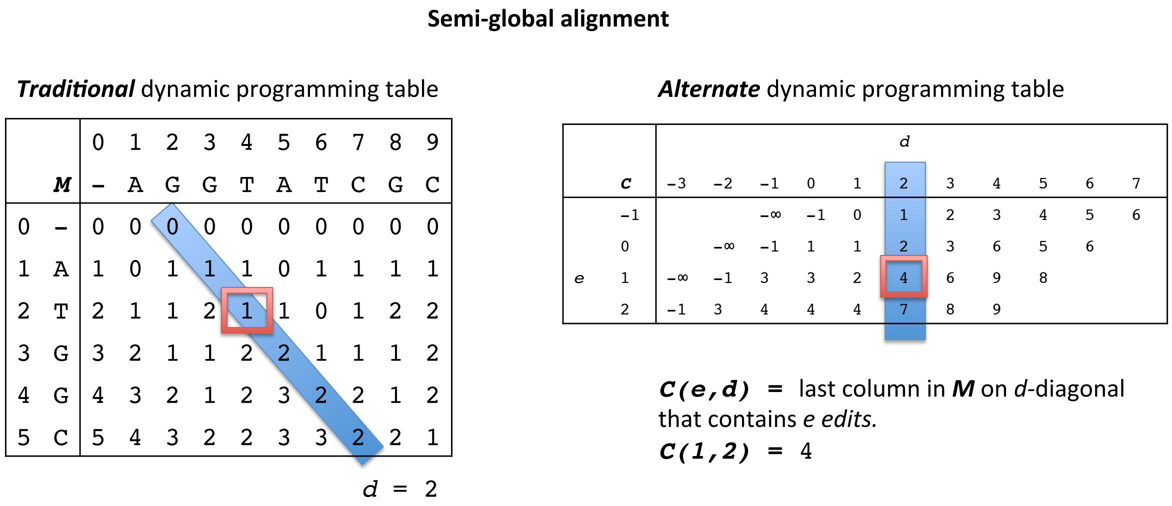
Currently, we use an $O(nm)$ speed and time method to calculate the edit distance between two sequences of lengths n and m (Algorithm 1). This cost is reduced to $O(nk)$ in the specific case of globally aligning two sequences within k edits. Globally aligning sequences may not be ideal in the case where we are aligning short WGS reads to a 16s rRNA database, which is the standard for many tools, including UCLUST. These WGS reads are drawn randomly from the underlying genome so they can occur anywhere within the 16s rRNA gene. Thus, we must implement a fast method to perform semi-global alignment of these sequences.

In this section, we describe how to further improve the runtime to $O(k^2)$ in the case of global alignment and $O(nk)$ for semi-global alignment using a hybrid dynamic programming method developed by Landau and Vishkin[20].

3.2.1 Alternative representation of the dynamic programming table

Thus far, when calculating the edit distance between sequences $t = t_1t_2..t_n$ and $p = p_1p_2..p_m$, cell (i, j) in the dynamic programming table M represents the minimum edit distance between prefixes $t_{1,i}$ and $p_{1,j}$ (Algorithm 1).

Definition 3.2. *Given an n-by-m dynamic programming table, cells on a* **i-diagonal** *are any cells (z,j) where $j - z = i$.*



Semi-global alignment

Traditional **dynamic programming table**

		0	1	2	3	4	5	6	7	8	9
	M	-	A	G	G	T	A	T	C	G	C
0	-	0	0	0	0	0	0	0	0	0	0
1	A	1	0	1	1	1	0	1	1	1	1
2	T	2	1	1	2	1	1	0	1	2	2
3	G	3	2	1	1	2	2	1	1	1	2
4	G	4	3	2	1	2	3	2	2	1	2
5	C	5	4	3	2	2	3	3	2	2	1

d = 2

Alternate **dynamic programming table**

							d					
	C	−3	−2	−1	0	1	2	3	4	5	6	7
	−1			−∞	−1	0	1	2	3	4	5	6
	0		−∞	−1	1	1	2	3	6	5	6	
e	1	−∞	−1	3	3	2	4	6	9	8		
	2	−1	3	4	4	4	7	8	9			

C(e,d)* =** last column in ***M on *d*-diagonal that contains *e edits.*
***C(1,2)* =** 4

Figure 5: Traditional and alternative dynamic programming table used to solve k-difference algorithm.

Definition 3.3. *A* d-**path** *is a path through the dynamic programming table starting at row 0 and ending at a cell with* d *edits. A* d*-path is* **farthest-reaching** *in the* i*-diagonal if it ends in the* i*-diagonal and is greater than or equal to the ending column of any other* d*-path ending in the* i*-diagonal.*

We reduce the storage requirements by only recording the last column with e edits along a given i-diagonal. If we are only interested in finding alignments within k-difference, for each i-diagonal in our table, we need to record at most $O(k)$ entries. We do not need to record more than $O(k)$ entries because in our given formulation of the problem it is impossible to decrease in the amount of edits made. Since we only penalize start gaps in the pattern and not the text, the amount of diagonals we have to compute is $n - m + k$. Thus, we require $O(nk)$ memory to find k-differences between the text and pattern.

At the high level, for every edit number $0 \leq d \leq k$, we will compute the farthest-reaching d-path on the i-diagonal using the farthest-reaching $(d-1)$-paths on diagonals $i-1$, i, $i+1$. Let C be a dynamic programming table where cell (i, j) now refers to the farthest-reaching column in M of the j-diagonal that contains i edits (Figure 5). For simplicity, lets assume we can access the negative j-diagonals of the table with cell $(\_, -j)$. Since accessing negative indexes in arrays are invalid in most programming languages, in practice we offset each diagonal by k.

To begin, we need to initialize the first row of C, where $d = 0$. A 0-path corresponds to a path ending on the i-diagonal with no mismatches. For each column j (j-diagonal in M), cell $C(i, j)$ is set to the *longest common extension* of $p_1..p_m$ and $t_j..t_n$.

Algorithm 2 Landau-Vishkin k-differences algorithm. $O(kn)$ work.

```
procedure ComputeKDifferences($t, p, k$)
    $n \leftarrow |t|$
    $m \leftarrow |p|$
    for $d = 0..n - m + k + 1$ do
        $C(-1, d) \leftarrow i$
                                ▷ Set first row of table $C$ to one minus its diagonal.
    for $d = -(k + 1).. - 1$ do
        $C(|d| - 1, d) \leftarrow -1$
        $C(|d| - 2, d) \leftarrow - \inf$
    for $i = 0..n - m + k$ do          ▷ Process table along the anti-diagonals.
        for $e = 0..k$ do
            $d \leftarrow c - e$
            $col \leftarrow max(C(e - 1, d - 1) + 1, C(e - 1, d) + 1, C(e - 1, d + 1))$
            while $col < n$ and $col - d < m$ and $x_{col+1} == y_{col+1-d}$ do
                $col \leftarrow col + 1$
            $C(i, j) \leftarrow \min(col, m + d, n)$
            if $C(i, j) \geq m + d$ then return Occurrence found
    return No occurrence found”
```

For $d > 0$, to compute the farthest-reaching d-path on the i-diagonal, we first need to consider three different farthest-reaching $(d - 1)$-paths:

- **$(d - 1)$-path on $(i - 1)$-diagonal.** We follow a horizontal edge (a space in p), which increases the column index by 1.
- **$(d - 1)$-path on (i)-diagonal.** We follow an edge corresponding to a mismatch between a character of p and t and increment the column index by 1.
- **$(d - 1)$-path on $(i + 1)$-diagonal.** We follow a vertical edge (a space in t). We do not increment the column index in this case.

Once we have the largest column index (col) of a $(d - 1)$-path, then we find the longest common extension starting from t_{col} and p_{col-d}. An acceptable alignment will be any cell $C(e, d)$ whose value is greater than $m + d$. This means that the full pattern has been matched in $\leq k$ edits.

There are two ways we can calculate the cells of C. Cell $C(i, j)$ relies on the values in adjacent cells: $C(i - 1, j - 1)$, $C(i - 1, j)$, and $C(i - 1, j + 1)$ Once we initialize the first row of C, we can proceed row-by-row. However, in the next section, we will describe why we populate the cells an anti-diagonal at a time (Algorithm 2).

For each edit ($0 \leq e \leq k$) and diagonal ($-k \leq d \leq n - m$), we need to record the longest reaching d-path, so

	Semi-global (k=5)
Naïve	1.022 sec
Landau-Vishkin	0.040 sec
Speedup	25.599x

Table 2: Runtime comparison of the naïve and Landau-Vishkin[20] methods for finding all semi-global alignments within k-differences. The center (a 637bp sequence) was aligned against a database consisting of 1,000 16S rRNA sequences (ranging in lengths from 300-600bp) with $k = 5$. Runtimes were averaged over 100 runs.

the amount of memory required is $O(nk)$. Finding the longest common extension between two substrings in p and t requires $O(n)$ time. However, we can use a suffix tree to store our center (t) and reduce the longest common extension time down to $O(1)$ time[16], reducing the overall runtime down to $O(kn)$.

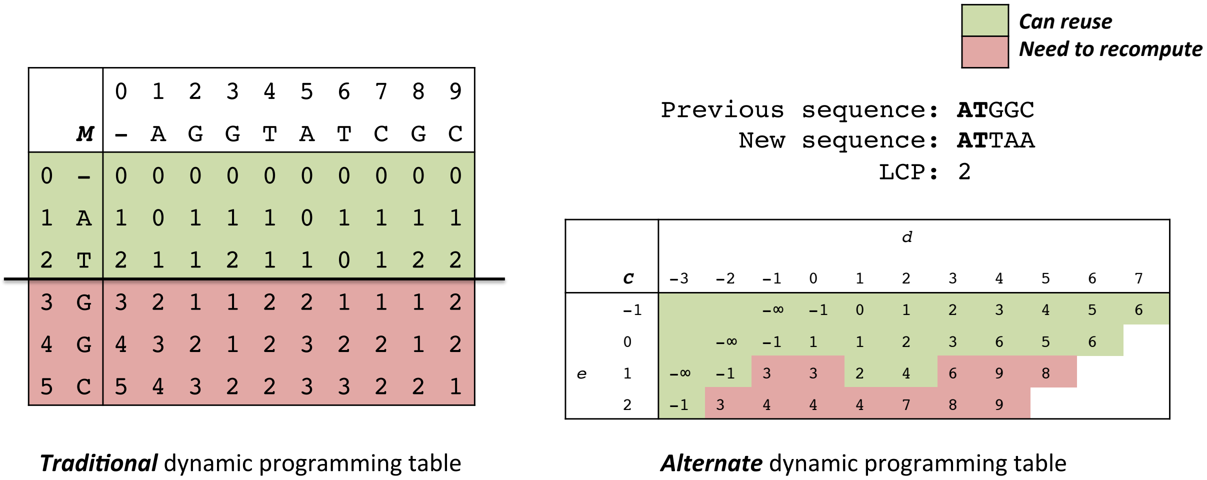
3.2.2 Efficient k-difference implementation

We provide an implementation of both the näive and the Landau-Vishkin algorithms written in C. For our experiment, we created a 1,000 sequence sample from an environmental 16S rRNA data set. The database sequences ranged in length from 300bps to 600bps. We randomly selected a sequence from the database to serve as our center. The center was aligned against the database sequences with $k = 5$ using both methods (Table 2). Runtimes were averaged over 100 runs.

It is important to note that in our current implementation we used an inefficient method for computing the longest common extension between two substrings. Despite this, our implementation of the Landau-Vishkin algorithm is *25x* faster than the naïve method.

3.2.3 Exploiting a sorted sequence database

DNACLUST leverages the fact that the database of sequences are highly similar. By sorting the database sequences, we can reuse the first l rows of the dynamic programming table between database sequences i and $i + 1$, where l is equal to the longest common prefix of i and $i + 1$. Our current implementation does not account for this; however, here we will outline the steps necessary to recreate this feature with our alternative dynamic programming table implementation.



🟩 *Can reuse*
🟥 *Need to recompute*

		0	1	2	3	4	5	6	7	8	9
	M	-	A	G	G	T	A	T	C	G	C
0	-	0	0	0	0	0	0	0	0	0	0
1	A	1	0	1	1	1	0	1	1	1	1
2	T	2	1	1	2	1	1	0	1	2	2
3	G	3	2	1	1	2	2	1	1	1	2
4	G	4	3	2	1	2	3	2	2	1	2
5	C	5	4	3	2	2	3	3	2	2	1

Traditional dynamic programming table

```
Previous sequence: ATGGC
     New sequence: ATTAA
              LCP: 2
```

		d										
	c	−3	−2	−1	0	1	2	3	4	5	6	7
	−1			−∞	−1	0	1	2	3	4	5	6
	0		−∞	−1	1	1	2	3	6	5	6	
e	1	−∞	−1	3	3	2	4	6	9	8		
	2	−1	3	4	4	4	7	8	9			

Alternate dynamic programming table

Figure 6: Exploiting sorted sequence database to speedup k-difference algorithm.

In the traditional dynamic programming table, the i-th row of the table correspond to the i-th character of the pattern and the j-th column corresponds to the j-th character of the text. After populating the full dynamic table, we now attempt to align the new pattern with the text. The new pattern shares l characters with the previously aligned pattern. Here we can start aligning from the $(l+1)$-th row because we have already aligned the prefix ending at position l of the new pattern with the previous pattern.

In our alternative dynamic programming table, the i-th row of the table corresponds to the i-th allowed edit distance and the j-th column corresponds to the j-th diagonal of the traditional dynamic programming table. It becomes trickier to reuse information from this dynamic programming table because edits along a given diagonal in the traditional dynamic programming table may start before the i-th row and extend past the $(l+1)$-th row (Figure 6). We can only reuse the information in this alternative dynamic program table if along a given i-diagonal and edit distance, the farthest-reaching column ends on or before the $(i+l)$-th column.

In Algorithm 2, we calculated the cells in table C starting from the 0 anti-diagonal. Cell $C(e,d)$ lies on the $(e+d)$ anti-diagonal. When calculating $C(e,d)$, we first find the farthest-reaching column (max of $C(e-1,d-1)+1$, $C(e-1,d)+1$, $C(e-1,d-1)$) then extend the match between the pattern and text as far as possible. As we proceed downwards along the given c anti-diagonal, if the value in the cell $C(e,d)-d$ is greater than the l we need to recalculate the cell. We subtract by d to correct for the offset start position of the d-diagonal. A value greater than l indicates that the longest e-path on the d-diagonal extends past the shared prefix of the current and previous patterns. Thus, for a given e and d, if $C(e,d)-d>l$, we need to recalculate the cell. If we have to recalculate $C(e, d)$ then we are forced to recalculate any cells that directly depend on $C(e, d)$, which in this case would be $C(e+1, d-1)$, $C(e+1, d)$, and $C(e+1, d+1)$.

One important thing to note is that if $C(e, d)$ is the first cell in column d that we have to recompute, then we can save this index to speed up finding the first cell to recompute in column $d+1$. Assume $C(e, d)$ does *not* need to be recomputed. That means that the value (farthest-reaching column) in cells $C(e-1, d-1)$, $C(e-1, d)$, and $C(e-1, d+1)$ are all less than $C(e, d)$ and do not need to be recomputed. If any of the three adjacent cells above $C(e, d)$ had to be recomputed, then we would be forced to recompute $C(e, d)$.

Theorem 3.4. *For a given d, let j be the lowest row index ($0 \leq j \leq k$) such that $C(j, d)$ needs to be recomputed, but $C(j-1, d)$ does not need to be recomputed. The lowest row index that needs to be recomputed in column $d+1$ is either $j-1$, j, or $j+1$.*

Proof. Assume not and $j-2$ is the lowest row index that needs to be recomputed in column $d+1$. If $C(j, d)$ is the lowest row index in d that needs to be recomputed, then $C(j-1, d)$ does not need to be recomputed. If $C(j-2, d+1)$ needs to be recomputed, then $C(j-1, d)$ needs to be recomputed as well since $C(j-1, d)$ depends on $C(j-2, d-1)$, $C(j-2, d)$, $C(j-2, d+1)$. However, we have already said that j is the lowest row index such that $C(j-1, d)$ does not need to be recomputed, resulting in a contradiction. A similar reasoning can be done when for $j-3$, $j-4$, ..., 0.

Now assume $j+2$ is the lowest row index that needs to be recomputed in column $d+1$, making $j+1$ the last row index that does not need to be recomputed. If $C(j+1, d+1)$ does need to be recomputed, then the cells it depends on ($C(j, d)$, $C(j, d+1)$, $C(j, d+2)$) do not need to be recomputed. However, we have already said that j is the lowest row index such that $C(j, d)$ needs to be recomputed. Hence, another contradiction. A similar strategy can be used for $j+3$, $j+4$ ..., k.

□

The runtime of finding the starting cells to recompute along on an anti-diagonal takes $O(k)$ work because we start on the first anti-diagonal and proceed downwards until we find the first the cell where $C(e, d) - d > l$. Then when processing the next anti-diagonal, we only need to look at potentially three start positions of the first cell in that column to be recomputed (constant work), taking only $O(k)$ time to find the starting point for recomputing the cells of c. Since the columns are increasing down the anti-diagonal, it would also be possible to perform a binary search to find the appropriate start location, reducing the time to $O(lg(k))$.

Strategy	Clusters
Unique	3,031,761
Ambiguous	5,085,988
Ones	6,201,399
Random	5,183,737
Proportion	6,201,399

Table 3: Comparing the total number of gene clusters found using the different strategies for aligning ambiguous reads on the Qin et al. type 2 diabetes data set.

However, since the values of k are often very small in practice, little practical benefit would be obtained.

3.3 Handling ambiguous reads

Often, a goal of metagenomic studies is to compare the microbial composition across multiple samples and conditions. Qin et al. looked for gut microbial markers that might be useful for classifying type 2 diabetes[32]. Before we can detect differential abundance across samples, the sample sequences have to be clustered into OTUs, counted, and then normalized[30]. Thus far in this prospectus, we have discussed methods for speeding up the alignment of sequences to a center, but we have yet to discuss how to handle ambiguous reads.

When a sequence is being recruited by a center, it is possible that this sequence is within some distance from another potential center. Henceforth, we refer to a sequence that lies within a given distance from multiple centers as *ambiguous.* Currently in DNACLUST, UCLUST, and CD-HIT, an ambiguous sequence is recruited by the first center that it encounters. Depending on the number of ambiguous of sequences, this may affect the resulting cluster abundances. For example, if we are clustering short WGS reads using a database of 16S rRNA genes, the first 16s rRNA gene we select as a center will preferentially recruit all of the WGS reads that happen to fall within a conserved region of the gene. This could drastically affect downstream analyses for detecting differentially abundant OTUs.

Here, we describe different methods for assigning ambiguous reads and constructing the count matrix, where the rows represent different OTUs, the columns represent different samples, and the cell (i, j) corresponds to the count of OTU i in sample j.

In DNACLUST, we mark any sequence *black* that has been assigned to a cluster center, which signifies that the sequence will not be aligned to another center nor chosen as a potential center. We have added the option DNACLUST to print an inverted index mapping sequences to their potential centers. We accomplish this by marking a sequence *gray* when it is aligned to a cluster center, which signifies that the sequence *can* be aligned to other centers, but it cannot be chosen as a potential center. This inverted index allows users the option to their preferred method for assigning ambiguous reads.

The first method is to simply discard any ambiguous reads and only consider reads that can be uniquely aligned to a single center. One potential issue with this approach is when comparing the abundances of OTUs *within* a sample. Two OTUs with equal abundance, but an unequal number of unique regions, will not be equal in the count matrix. This problem can also occur across samples, where one sample contains the two OTUs, and the other sample only contains one of the OUTs. If in reality the shared OTU occurs at the same relative abundance across samples, we may erroneously find an increased abundance in the second sample if we only look at uniquely aligned reads.

Another way to assign ambiguous reads is to randomly assign them to the set of potential centers. An issue with this approach is that we may assign reads to an OTU that does not exist within our sample, the read just happens to align to a conserved region present across multiple OTUs.

Similarly, instead of randomly assigning the reads, we can assign a fractional count to each center. Although, with a large number of ambiguous reads, the fractional counts should be close to the previous randomly assign approach.

Lastly, we can assign a read based on the proportion of uniquely aligned reads to the center. In other words, if a read can align equally well to two different centers, but one center contains uniquely aligned reads and the other contains none, then it is more probable that the read came from the first center. This approach should better handle the potential false OTU detection present with random and fractional assignment of reads. We add a small pseudocount of one to each OTU to help in the case of low coverage or the possibility that the OTU contains no unique regions. An issue with this current approach, however, is that we do not normalized by the possibility of repeats within the center. A more intelligent approach is to use a maximum likelihood method to assign the ambiguous reads with the added constraint being coverage across the center should be uniform[29].

For our proposed work, we want to reanalyze these large publicly available data sets (such as Qin et al's type 2 diabetes data set [32]) and see if there claims still hold depending on their read mapping strategy. As a preliminary experiment, we aligned the reads using Bowtie2 (a fast read alignment tool)[23] to the Human

Microbiome Project's gene clusters (http://hmpdacc.org/HMGC/) using the different read strategies (Table 3). One important thing to note is that when we only count gene clusters that have at least one unique read mapping to it we had nearly 50% less gene clusters than if we randomly assigned ambiguous reads. Since these high profile studies are drawing conclusions from these count matrices, more week needs to be done observing the effect of ambiguous read alignments.

4 Timeline

Parallel DNACLUST	1 months
Fast, exact clustering tool using efficient k-diff algorithm	2-4 months
Handling ambiguous read and their affect on differential abundance	1 months
TOTAL	4-6 months

Paper deadline goals:

- 14th Workshop on Algorithms in Bioinformatics (WABI), May 2014: Parallel DNACLUST and ambiguous read assignment strategy
- Nucleic Acids Research, September 2014: Efficient k-difference clustering tool

5 Conclusion

Microorganisms play an important role in the environment. Despite this, these microbes are among the least understood on Earth due in part to difficulties culturing them in a lab setting. Sequencing technology provides us with a solution, allowing us to get a snapshot of the microscopic communities that inhabit these environments. Unfortunately, this snapshot only can provide us with a very small, noisy picture of the community. In typical analyses, the short fragments of DNA produced from sequencing technologies need to be clustered into similar groups based on some similarity. In this prospectus, we have described the current approaches and limitations for clustering highly similar DNA sequences. By leveraging efficient string matching algorithms and multi-core systems, we have proposed a fast and exact clustering tool for DNA sequences.

6 Acknowledgments

I owe my sincere gratitude to all the people who made this work possible. I'd like to thank my adviser Mihai Pop for giving me the opportunity to work on intellectually challenging and interesting projects for nearly 8 years. His guidance and support has led me to pursue a career in academia and I am forever grateful. Furthermore, I would like to thank my committee members and mentors Atif Memon and Héctor Corrada Bravo for their support. I owe a great thanks to all the members of Pop, Bravo, and Memon labs for their discussions, contribution, and friendship.

A Reading List

A.1 Clustering

[2] Stephen F. Altschul et al. "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs". *Nucleic Acids Research* 25.17 (Sept. 1, 1997). PMID: 9254694, pp. 3389–3402. DOI: 10.1093/nar/25.17.3389.

[3] Ergude Bao et al. "SEED: efficient clustering of next-generation sequences". *Bioinformatics* 27.18 (Sept. 15, 2011). PMID: 21810899, pp. 2502–2509. DOI: 10.1093/bioinformatics/btr447.

[5] Yunpeng Cai and Yijun Sun. "ESPRIT-Tree: hierarchical clustering analysis of millions of 16S rRNA pyrosequences in quasilinear computational time". *Nucleic Acids Research* 39.14 (Aug. 1, 2011). PMID: 21596775, e95–e95. DOI: 10.1093/nar/gkr349.

[8] Robert C. Edgar. "Search and clustering orders of magnitude faster than BLAST". *Bioinformatics* 26.19 (Oct. 1, 2010). PMID: 20709691, pp. 2460–2461. DOI: 10.1093/bioinformatics/btq461.

[10] Daniel Fasulo. *An Analysis of Recent Work on Clustering Algorithms.* 1999.

[12] Limin Fu et al. "CD-HIT: accelerated for clustering the next-generation sequencing data". *Bioinformatics* 28.23 (Dec. 1, 2012). PMID: 23060610, pp. 3150–3152. DOI: 10.1093/bioinformatics/bts565.

[14] Mohammadreza Ghodsi, Bo Liu, and Mihai Pop. "DNACLUST: accurate and efficient clustering of phylogenetic marker genes". *BMC Bioinformatics* 12.1 (June 30, 2011). PMID: 21718538, p. 271. DOI: 10.1186/1471-2105-12-271.

[17] Julia Handl, Joshua Knowles, and Douglas B. Kell. “Computational cluster validation in post-genomic data analysis”. *Bioinformatics* 21.15 (Aug. 1, 2005). PMID: 15914541, pp. 3201–3212. DOI: `10.1093/bioinformatics/bti517`.

[18] Susan M Huse et al. “Ironing out the wrinkles in the rare biosphere through improved OTU clustering”. *Environmental Microbiology* 12.7 (July 2010). PMID: 20236171 PMCID: PMC2909393, pp. 1889–1898. DOI: `10.1111/j.1462-2920.2010.02193.x`.

[19] Victor Kunin et al. “Wrinkles in the rare biosphere: pyrosequencing errors can lead to artificial inflation of diversity estimates”. *Environmental microbiology* 12.1 (2010). PMID: 19725865, pp. 118–123. DOI: `10.1111/j.1462-2920.2009.02051.x`.

[23] Ben Langmead and Steven L Salzberg. “Fast gapped-read alignment with Bowtie 2”. *Nature methods* 9.4 (2012), pp. 357–359.

[24] Weizhong Li and Adam Godzik. “Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences”. *Bioinformatics* 22.13 (July 1, 2006). PMID: 16731699, pp. 1658–1659. DOI: `10.1093/bioinformatics/btl158`.

[25] Weizhong Li, Lukasz Jaroszewski, and Adam Godzik. “Clustering of highly homologous sequences to reduce the size of large protein databases”. *Bioinformatics* 17.3 (Mar. 1, 2001). PMID: 11294794, pp. 282–283. DOI: `10.1093/bioinformatics/17.3.282`.

[26] Weizhong Li et al. “Ultrafast clustering algorithms for metagenomic sequence analysis”. *Briefings in Bioinformatics* 13.6 (Nov. 1, 2012). PMID: 22772836, pp. 656–668. DOI: `10.1093/bib/bbs035`.

[31] Sarah P. Preheim et al. “Distribution-Based Clustering: Using Ecology To Refine the Operational Taxonomic Unit”. *Applied and Environmental Microbiology* 79.21 (Nov. 1, 2013). PMID: 23974136, pp. 6593–6603. DOI: `10.1128/AEM.00342-13`.

[34] Christopher Quince et al. “Accurate determination of microbial diversity from 454 pyrosequencing data”. *Nature Methods* 6.9 (Sept. 2009), pp. 639–641. DOI: `10.1038/nmeth.1361`.

[35] Jens Reeder and Rob Knight. “Rapid denoising of pyrosequencing amplicon data: exploiting the rank-abundance distribution”. *Nature methods* 7.9 (Sept. 2010). PMID: 20805793 PMCID: PMC2945879, pp. 668–669. DOI: `10.1038/nmeth0910-668b`.

[36] Joo F. Matias Rodrigues and Christian von Mering. “HPC-CLUST: distributed hierarchical clustering for large sets of nucleotide sequences”. *Bioinformatics* 30.2 (2014). PMID: 24215029, pp. 287–288. DOI: `10.1093/bioinformatics/btt657`.

[43] Yuzhen Ye. “Identification and quantification of abundant species from pyrosequences of 16S rRNA by consensus alignment”. *2010 IEEE International Conference on Bioinformatics and Biomedicine (BIBM).* 2010 IEEE International Conference on Bioinformatics and Biomedicine (BIBM). Dec. 2010, pp. 153–157. DOI: `10.1109/BIBM.2010.5706555`.

[44] Zejun Zheng, Stefan Kramer, and Bertil Schmidt. “DySC: software for greedy clustering of 16S rRNA reads”. *Bioinformatics* 28.16 (Aug. 15, 2012). PMID: 22730435, pp. 2182–2183. DOI: `10.1093/bioinformatics/bts355`.

A.2 String algorithms

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. “Replacing suffix trees with enhanced suffix arrays”. *Journal of Discrete Algorithms* 2.1 (Mar. 2004), pp. 53–86. DOI: `10.1016/S1570-8667(03)00065-0`.

[4] Jon L. Bentley and Robert Sedgewick. “Fast Algorithms for Sorting and Searching Strings”. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA ’97. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997, 360369.

[9] Michael Farrar. “Striped SmithWaterman speeds database searches six times over other SIMD implementations”. *Bioinformatics* 23.2 (2007). PMID: 17110365, pp. 156–161. DOI: `10.1093/bioinformatics/btl582`.

[13] Zvi Galil and Kunsoo Park. “An Improved Algorithm for Approximate String Matching”. *SIAM J. Comput.* 19.6 (Nov. 1990), 989999. DOI: `10.1137/0219067`.

[16] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, May 28, 1997. 556 pp.

[20] G M Landau and U Vishkin. “Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm”. *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing.* STOC ’86. New York, NY, USA: ACM, 1986, 220230. DOI: `10.1145/12130.12152`.

[21] Gad M Landau and Uzi Vishkin. “Fast parallel and serial approximate string matching”. *Journal of Algorithms* 10.2 (June 1989), pp. 157–169. DOI: `10.1016/0196-6774(89)90010-2`.

[22] Gad M. Landau and Uzi Vishkin. “Fast string matching with k differences”. *Journal of Computer and System Sciences* 37.1 (Aug. 1988), pp. 63–78. DOI: `10.1016/0022-0000(88)90045-1`.

[27] Udi Manber and Gene Myers. “Suffix Arrays: A New Method for On-Line String Searches”. *SIAM Journal on Computing* 22.5 (Oct. 1993), pp. 935–948. DOI: 10.1137/0222058.

[28] Saul B. Needleman and Christian 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 (Mar. 28, 1970), pp. 443–453. DOI: 10.1016/0022-2836(70)90057-4.

[37] Torbjrn Rognes. “Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation”. *BMC Bioinformatics* 12.1 (June 1, 2011). PMID: 21631914, p. 221. DOI: 10.1186/1471-2105-12-221.

[39] E. Ukkonen. “On-line construction of suffix trees”. *Algorithmica* 14.3 (Sept. 1, 1995), pp. 249–260. DOI: 10.1007/BF01206331.

[40] Esko Ukkonen. “Approximate string-matching over suffix trees”. *Combinatorial Pattern Matching*. Ed. by Alberto Apostolico et al. Lecture Notes in Computer Science 684. Springer Berlin Heidelberg, 1993, pp. 228–242.

A.3 Metagenomics

[6] J. Gregory Caporaso et al. “QIIME allows analysis of high-throughput community sequencing data”. *Nature Methods* 7.5 (May 2010), pp. 335–336. DOI: 10.1038/nmeth.f.303.

[7] T. Z. DeSantis et al. “Greengenes, a Chimera-Checked 16S rRNA Gene Database and Workbench Compatible with ARB”. *Applied and Environmental Microbiology* 72.7 (July 1, 2006). PMID: 16820507, pp. 5069–5072. DOI: 10.1128/AEM.03006-05.

[11] Noah Fierer et al. “Forensic identification using skin bacterial communities”. *Proceedings of the National Academy of Sciences* 107.14 (Apr. 6, 2010). PMID: 20231444, pp. 6477–6481. DOI: 10.1073/pnas.1000162107.

[15] Steven R. Gill et al. “Metagenomic Analysis of the Human Distal Gut Microbiome”. *Science* 312.5778 (June 2, 2006). PMID: 16741115, pp. 1355–1359. DOI: 10.1126/science.1124234.

[30] Joseph N. Paulson et al. “Differential abundance analysis for microbial marker-gene surveys”. *Nature Methods* 10.12 (Dec. 2013), pp. 1200–1202. DOI: 10.1038/nmeth.2658.

[32] Junjie Qin et al. “A metagenome-wide association study of gut microbiota in type 2 diabetes”. *Nature* 490.7418 (Oct. 4, 2012), pp. 55–60. DOI: 10.1038/nature11450.

[33] C. Quast et al. “The SILVA ribosomal RNA gene database project: improved data processing and web-based tools”. *Nucleic Acids Research* 41 (D1 2013), pp. D590–D596. DOI: `10.1093/nar/gks1219`.

[38] Patrick D. Schloss et al. “Introducing mothur: Open-Source, Platform-Independent, Community-Supported Software for Describing and Comparing Microbial Communities”. *Applied and Environmental Microbiology* 75.23 (Dec. 1, 2009). PMID: 19801464, pp. 7537–7541. DOI: `10.1128/AEM.01541-09`.

[41] J. Craig Venter et al. “Environmental Genome Shotgun Sequencing of the Sargasso Sea”. *Science* 304.5667 (Apr. 2, 2004). PMID: 15001713, pp. 66–74. DOI: `10.1126/science.1093857`.

[42] John C. Wooley, Adam Godzik, and Iddo Friedberg. “A Primer on Metagenomics”. *PLoS Comput Biol* 6.2 (Feb. 26, 2010), e1000667. DOI: `10.1371/journal.pcbi.1000667`.