# Entropy Measurements and Ball Cover Construction for Biological Sequences

Jeffrey A. Robertson

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Application

Lenwood S. Heath, Chair

Stephen G. Eubank

Madhave V. Marathe

June 22, 2018

Blacksburg, Virginia

Keywords: Bioinformatics, Entropy Scaling, Sequence Search, BLAST

# Entropy Measurements and Ball Cover Construction for Biological Sequences

Jeffrey A. Robertson

(ABSTRACT)

As improving technology is making it easier to select or engineer DNA sequences that produce dangerous proteins, it is important to be able to predict whether a novel DNA sequence is potentially dangerous by determining its taxonomic identity and functional characteristics. These tasks can be facilitated by the ever increasing amounts of available biological data. Unfortunately, though, these growing databases can be difficult to take full advantage of due to the corresponding increase in computational and storage costs. Entropy scaling algorithms and data structures present an approach that can expedite this type of analysis by scaling with the amount of entropy contained in the database instead of scaling with the size of the database. Because sets of DNA and protein sequences are biologically meaningful instead of being random, they demonstrate some amount of structure instead of being purely random. As biological databases grow, taking advantage of this structure can be extremely beneficial. The entropy scaling sequence similarity search algorithm introduced here demonstrates this by accelerating the biological sequence search tools BLAST and DIAMOND. Tests of the implementation of this algorithm shows that while this approach can lead to improved query times, constructing the required entropy scaling indices is difficult and expensive. To improve performance and remove this bottleneck, I investigate several ideas for accelerating building indices that support entropy scaling searches. The results of these tests identify key tradeoffs and demonstrate that there is potential in using these techniques for sequence similarity searches.

# Entropy Measurements and Ball Cover Construction for Biological Sequences

Jeffrey A. Robertson

(GENERAL AUDIENCE ABSTRACT)

As biological organisms are created and discovered, it is important to compare their genetic information to known organisms in order to detect possible harmful or dangerous properties. However, the collection of published genetic information from known organisms is huge and growing rapidly, making it difficult to search. This thesis shows that it might be possible to use the non-random properties of biological information to increase the speed and efficiency of searches; that is, because genetic sequences are not random but have common structures, the increase of known data does not mean a proportional increase in complexity, known as entropy. Specifically, when comparing a new sequence to a set of previously known sequences, it is important to choose the correct algorithms for comparing the similarity of two sequences, also known as the distance between them. This thesis explores the performance of entropy scaling algorithm compared to several conventional tools.

*Soli Deo gloria*

# Acknowledgments

I would like to thank Dr. Heath for his guidance and the opportunity to work on this interesting problem. I would also like to thank my committee members Dr. Eubank and Dr. Marathe for their insight and helpful suggestions. I am thankful to all my Fun GCAT team members listening to my presentations and my lab mates Yanshen, Zhen, and Xiao for the encouraging environment. Most of all, I want to thank my family for their never-ending support and encouragement.

This research was funded by a grant from IARPA.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As DNA sequencing technology improves, biological databases are growing exponentially in size [4, 23]. This advance in technology also provides new opportunities for bad actors to introduce biological threats [3, 34]. If it can be used efficiently, the growing wealth of available data is one of the best resources for combating potential threats. This poses a computational challenge, as traditional sequence search tools were designed for smaller databases and do not always scale well to amounts of data that are orders of magnitude larger. The new sequencing technology rapidly generates sequences, but not all of these additional sequences are equally informative. Constructing a measure of entropy makes it possible to see that these larger sets of biological data tend to display a structure that can be taken advantage of computationally [59]. This is because biology approximately conserves sequences, leading to large amounts of redundancy. There are several data repositories that account for redundancy, such as the NCBI RefSeq Non-Redundant protein database [35], which explicitly combines or removes any sequences that are identical. The UniProt Reference Clusters (UniRef) [49] database also considers this by using the sequence clustering tool CD-HIT [21] to combine similar sequences at varying levels of sequence identity. Designing new analysis and storage tools with this in mind can lead to significantly improved performance [5]. This is particularly demonstrated in the type of taxonomic and functional analysis needed to determine whether a sequence is a potential threat. If a sequence is found to be similar to sequences that are known to be dangerous or to sequences from a dangerous organism, then that sequence can

be categorized as potentially a threat. To perform similarity-based analysis though, these similar sequences have to be searched for in ever expanding reference databases. Traditional search tools such as BLAST [1] and DIAMOND [8] can be used, but they do not intentionally utilize the structure of larger data sets and, therefore, tend to scale linearly with the size of the data set. Using new entropy scaling techniques, it is possible to wrap around these tools and improve their performance.

There are many potential approaches to sequence similarity searches that have query times that scale with the entropy of the data instead of the size of the data. The one that I consider here first constructs an index that consists of a set of balls that contain the reference sequences. The query algorithm can then identify a small number of balls to search in detail. Doing this leads to the execution time of the query being determined by the amount of entropy contained in the sequences as opposed to the number of sequences. In testing multiple implementations of this method, I found that while it can lead to significantly reduced query times, it leads to longer index construction times and has trouble competing with the sensitivity and precision of more conventional tools.

The problem being solved is formally defined in Chapter 2. Chapter 3 describes the necessary background and introduces some useful tools. Chapter 4 covers the applicable theory and math that will be used later. Chapter 5 describes several experimental implementations that make use of entropy scaling behavior. In Chapter 6, I present the results of running these implementations. Chapter 8 concludes by describing the importance of these results in the larger context of bioinformatics research and suggests future research directions.

# Chapter 2

# Problem Statement

To explore the idea of entropy scaling, the algorithms introduced here will solve an indexing and search problem for biological sequences with associated metadata. Specifically, given a biological sequence with unknown function and origin, use the large amounts of curated biological data publicly available to determine the sequence's function and origin. Then, based on this data, determine if this sequence is a potential biological threat.

The problem is broken up into several steps. The first takes as input a database of reference protein sequences and some descriptive metadata. This metadata includes the functional characteristics and taxonomic identity of each of the sequences. More formally, the input is the reference data set $P = \{p_1, p_2, ..., p_n\}$ with metadata $M = \{m_1, m_2, ..., m_n\}$ where $m_i$ is the metadata of sequence $p_i$. This first step produces an index $I$ containing these sequences and their metadata. The input for the second step is a small query protein sequence $q$ and the index $I$ that was constructed in the first step. The metadata $m_q$ for this query sequence is not known or is held back for testing purposes. As output, the second step returns a subset of the set of reference sequences $H_q \subseteq P$ and their associated metadata. These sequences are the ones that this step has determined to be close or similar to the query sequence $d(h, q) < t$, for all $h \in H_q$, where $d(p, q)$ represents the distance between $p$ and $q$ and $t$ is some query distance threshold. The third step uses domain specific knowledge to combine the metadata returned by the second step and propose corresponding metadata including functional characteristics, taxonomic identity, and threat level for the query sequence. The

implementations discussed here focus on exploring possibilities for the first of these steps that would allow the second step to behave in an entropy scaling manner. The third step is not investigated.

# Chapter 3

# Background

Although the idea of using entropy scaling techniques for sequence similarity searches is fairly new [59], there are many conventional tools and resources available that are not intentionally entropy scaling but might still be useful. This chapter covers several such tools and databases and explains what their intended purposes are.

## 3.1 Current Approaches

The problem described above is not entirely new and many techniques have already been developed to solve similar problems. Chapter 5 describes some of the ways that I intend to build off of these ideas to develop a tool that is optimized for the threat analysis use case.

### 3.1.1 BLAST

One well-known related tool is BLAST [1]. The approach BLAST takes is slightly more specific than the problem described in Chapter 2. It searches for result sequences $R$ in a database $P$ such that for each $r \in R$, $r$ aligns well with the query sequence $q$ or $r$ has sections of high local similarity with $q$. To do this, BLAST constructs an index for the database of reference sequences by storing seeds from the sequences in a lookup table. A seed is a short substring from a sequence. The lookup table consists of all the seeds that

appear in the reference sequences. For each of these seeds, it stores all of the locations that the seed was found in the reference sequences. To perform a query on the constructed index, BLAST considers all of the seeds in the query sequence. It uses the lookup table to find sequences in the original database that also contain some of the same seeds. Next, BLAST tries to extend the seeds by matching the parts of the query sequence around the seed to the equivalent parts in the reference sequences. If it finds a reference sequence that has a high similarity to the query sequence, then BLAST reports that sequence in the results of the query. While generic BLAST queries DNA sequences against a DNA index, BLASTP queries protein sequences against a protein index, BLASTX queries DNA sequences against a protein index, and BLASTN queries protein sequences against a DNA index. There are also many variations of BLAST that each have their own more diverse target use cases.

### 3.1.2 DIAMOND

DIAMOND (double index alignment of next-generation sequencing data) [8] is a tool that was developed to improve on BLAST in the common case where there are a large number of queries that need to be searched for in a single database. In particular, it solves the same problem instance as BLASTX (DNA sequence queries with a protein sequence index), but where BLAST would run multiple queries one after another, or potentially in parallel when the hardware allows, DIAMOND is able to combine the queries such that they can all be searched for in one pass through the database. This can lead to several orders of magnitude faster performance.

Fundamentally, DIAMOND uses the same seed and extend algorithm that BLAST uses, but with several new ideas. The first idea is alphabet reduction [31]. Instead of using the full 20 character amino acid alphabet, some groups of characters are combined to a single character.

Based on previous BLASTX results, the authors of DIAMOND found that a non-standard 11 character alphabet gave the best performance. Their reduction is [KREDQN] [C] [G] [H] [ILV] [M] [F] [Y] [W] [P] [STA] where brackets indicate input characters that are represented by a single output character. For example, the sequence VCEDVHAPKDTWMNV would be represented as ICKKIHSPKKSWMKI where each group is represented by its first letter. This can improve performance and storage cost because it is easier to store sequences with a smaller alphabet, but it can also increase the sensitivity because seeds no longer have to be exact matches. Another idea DIAMOND adds is spaced seeds [30]. Instead of normal short seeds where the entirety of the seed is considered, spaced seeds are longer, but ignore some of the positions in the middle of the seed. The pattern of which positions are ignored is called the shape of the seed. The number of positions that are not ignored is called the weight. In its default configuration, DIAMOND uses four different shapes of spaced seeds varying in length between 15 and 24 bp, all with a weight of 12. The most significant additional idea that DIAMOND introduced is double indexing. Based on the premise that memory latency and bandwidth is the bottleneck in other tools such as BLAST, DIAMOND tries to make optimal use of the cache hierarchy found in most hardware. It does this by constructing a seed index for all of the queries together, then sorting both this new query index and the reference index. It then does a single pass through both indices in a similar manner to the merge step in a standard database sort-merge join algorithm [6, 47]. This gives the most noticeable performance benefit when there are a large number of query sequences. Lastly, DIAMOND performs the extension phase on matched seeds using its own streaming implementation of a Smith-Waterman alignment [44].

### 3.1.3 SIFTER

SIFTER [38] is a statistical framework that uses a phylogenetic approach to perform genome wide functional annotations. It attempts to combat what it considers to be prevalent erroneous annotations in databases by considering how phenotypes have diverged and evolved from common ancestors through gene duplication events. SIFTER can use a small number of accurately annotated sequences to statistically model the phylogenetic history of a protein family. It then tries to place unannotated query sequences into this constructed phylogenetic tree. Based on their placement, SIFTER is able to assign functional characteristics to the query sequences.

### 3.1.4 Kraken

Kraken [57] is another popular and efficient tool for doing taxonomic analysis on DNA sequences. Although it uses similar methods to some of the other tools described, Kraken solves a slightly different problem. Given a query DNA sequence $q$, instead of returning a set of result sequences similar to $q$, Kraken returns taxonomic labels that potentially describe the organism that $q$ came from. It does this by using substrings of length $k$ called $k$-mers (default $k = 31$) in a technique called abundance estimation. The idea behind abundance estimation is that one does not need to make use of all reference sequences that are potentially relevant, just a sample of marker genes from each genome and genes that are relatively unique to certain clades in the taxonomy. Doing this can significantly speed up a tool at query time but has the potential to entirely miss some results. If the reference sequences are curated well enough, the results that are found can give a sufficient estimation of the abundance of an organism present in a sample. Kraken first constructs a reference database of all $k$-mers present in the reference sequences, storing each $k$-mer with the least common ancestor of the

organisms of all the sequences that contain it. A query can then do a simple lookup of all the $k$-mers in the query sequence and assign a taxon based on the least common ancestor of the taxa stored in the database. In practice, because Kraken uses a large value of $k$ and a relatively small set of reference sequences, each query has relatively few or no matches and can be either easily assigned a taxon or discarded.

### 3.1.5 MICA

The idea of entropy scaling algorithms for searching biological data sets was introduced by Berger et al. with their tool MICA [59]. This tool wraps around BLAST, Psi-BLAST (a BLAST variant), and DIAMOND to build compressed indices on protein or DNA sequences. MICA does this in an entropy scaling fashion, similar to the method described in Sections 4.4 and 4.5, although there are many parameters and tradeoffs that are left unexplored. In particular, when MICA is constructing the ball covering, it exclusively uses percent identity as its metric and re-implements a BLAST index to facilitate this. Additionally, it uses a fixed 70% sequence identity threshold for the ball radius. Doing this has led to some experimentally good results, but these choices are not necessarily the best for every data set.

## 3.2 Gene Ontology Terms

A Gene Ontology (GO) hierarchy [2, 51] is a directed acyclic graph (DAG) that is used to annotate biological sequences and describe biological ideas. In a GO DAG, each node is a term (such as "protein binding") and each edge is a relationship between terms (such as "regulates" or "is a"). The Gene Ontology Consortium has formalized a structure for this type of ontology, which contains three hierarchies: Cellular Component (CC), Biological

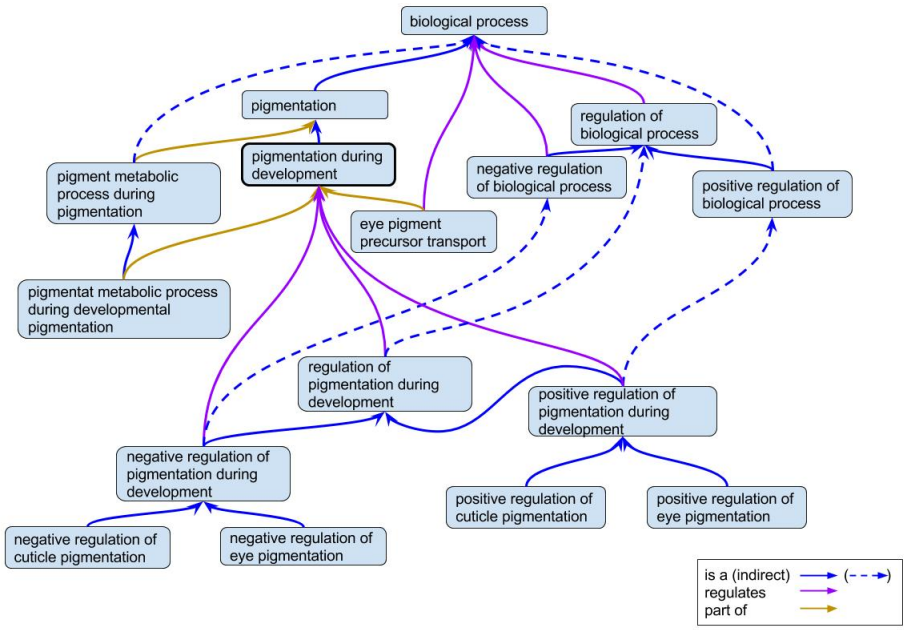


Figure 3.1: An illustration of a small piece of a GO directed acyclic graph which shows the relationships between several terms in the Biological Process hierarchy.

Process (BP), and Molecular Function (MF). The consortium provides a database of the ontology that they have curated. Some GO terms are extremely useful for threat assessments. For example, a protein annotated with GO term 0009403 (toxin biosynthetic process), is very relevant. Other terms such as 0044249 (cellular biosynthetic process) might be less meaningful to threat assessment. Although GO terms are powerful and useful tools, they can be difficult to work with because they are unevenly distributed. For example, some terms are so overused that they are effectively meaningless and others are seldom used. Additionally, GO term annotations of sequences can come from a variety of sources. Some computational sources, such as those described in the previous section, can be useful but are likely not to be as reliable as experimentally determined functional annotations. If a sequence analysis tool is able to take into account the complexities described here, it is more likely that the tool will be able to correctly annotate a query sequence.

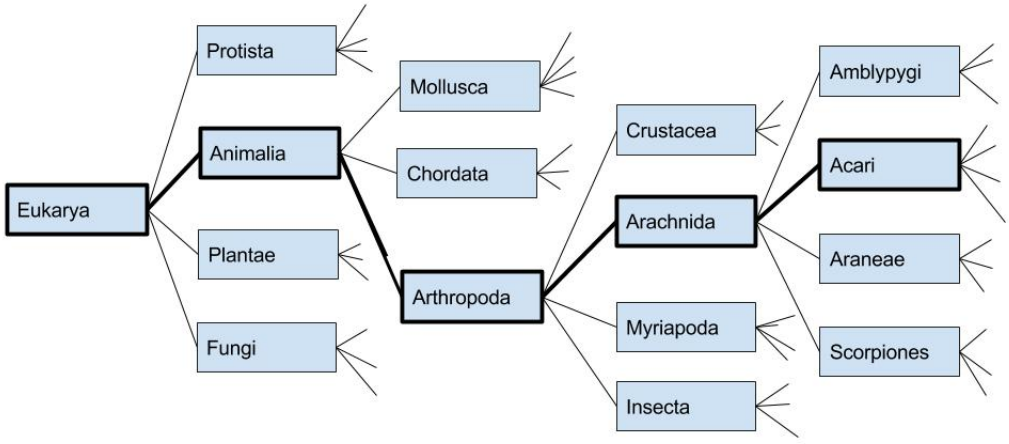


Figure 3.2: Assigning a sequence to a taxonomy

## 3.3 NCBI Taxonomy

The National Center for Biotechnology Information (NCBI) publishes a taxonomy database [14] that contains the taxonomic classification of over five hundred thousand species. A taxonomy is similar to the Gene Ontology explained in Section 3.2, except that it is a tree instead of a DAG, meaning that nodes can only have a single parent. Approaches that make use of the structure of a reliable taxonomy are able to provide a more informative annotation of a query sequence. Figure 3.2 is an illustration of a potential assignment of a sequence to a taxonomy tree. In this example the sequence has been assigned to the Domain Eukarya, the Kingdom Animalia, the Phylum Arthropoda, the Class Arachnidia, and the Order Acari. A sequence can be assigned a taxon as broad or specific as there is data available. In some cases, data is annotated with a reliable source species when it is collected, but often the taxonomy of a sequence is determined by a comparison with or combination of other sequences. In this case the sequence might have a distribution of possible assignments.

## 3.4 UniRef

The Universal Protein Resource (UniProt) [52] provides several databases of protein sequences and their functional and taxonomic annotations. The main one of these is the UniProt KnowledgeBase (UniProtKB) and a secondary one is the UniProt Archive (UniParc). As an additional service, the CD-Hit clustering algorithm [21, 27, 28] is regularly run on the entirety of UniProtKB along with selected sequences from UniParc. The clusters this produces are available in a set of smaller databases (UniRef) [49]. The largest of these, UniRef100, only clusters sequences that share 100 percent identity. Similarly, UniRef90 is built from UniRef100, grouping together any sequences that share at least 90 percent identity. The smallest database is UniRef50, which comes from the clusters in UniRef90, with any sequences sharing at least 50 percent identity being combined. As of the time of this writing, UniProtKB has 115 million sequences and UniParc has 211 million. UniRef100 contains 143 million sequences, UniRef90 contains 73.2 million sequences, and UniRef50 contains 30.3 million sequences.

# Chapter 4

# Theory

The premise of an entropy scaling algorithm is that some property of the algorithm, such as execution time or storage cost, scales with the entropy of a data set instead of with its size [59]. The usefulness of this type of approach depends significantly on the definition of entropy. To begin exploring the possibilities that entropy scaling algorithms provide, it is necessary to clarify some of the theory behind these approaches. Additionally, techniques introduced here can make use of already available tools such as those described in Section 3.1.

Part of the problem defined in Chapter 2 can be abstracted to a mathematical setting to facilitate discussion. Biological sequences (either DNA or protein) can be imagined as points in a high dimensional space. The distance between two points in this space represents how similar the sequences are.

There are two main categories of functions that define how close two points are: metrics and similarity measures.

## 4.1 Metrics

A *metric* [55] is a function that takes two points as parameters and returns a non-negative value representing the distance between them. There are several properties that a metric

$$ACAGTGAC$$
$$AGAGTTA$$

Figure 4.1: Example of comparing two sequences with Hamming distance. There are two characters that do not match (shown in red) and a difference of one between the sequence lengths, meaning that the hamming distance is three.

must satisfy. The first is the triangle inequality. This means that for any three points $a$, $b$, and $c$, the distance from $a$ to $c$ must be less than or equal to the sum of the distances from $a$ to $b$ and from $b$ to $c$, that is,

$$d(a, c) \leq d(a, b) + d(b, c).$$

Metrics must also be symmetric, meaning that the distance from $a$ to $b$ is the same as the distance between $b$ and $a$, that is, $d(a, b) = d(b, a)$. Lastly, metrics must return a distance of zero between points $a$ and $b$ if and only if $a$ and $b$ are identical, that is, $d(a, b) = 0$ is equivalent to $a = b$. When working with biological sequences, there are several metrics that are commonly used.

Hamming distance [18, 20] is one of the simplest metrics. Given two sequences $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$, the Hamming distance between $A$ and $B$ is the number of positions that are not identical plus the difference in the lengths, that is,

$$d(A, B) = \Sigma_{i=1}^{min(m,n)}(0 \text{ if } a_i = b_i \text{ else } 1) + abs(m - n).$$

For example, the Hamming distance between the sequences in Figure 4.1 is 3. Hamming distance was originally developed for use in binary error detection and correction encodings and, as such, has limited use when working with larger alphabets such as those used for

```
SEQUENCE2SET(Sequence S, int k)
1    ▷ S = [s_1 s_2 ··· s_n]
2    kmers ← {}
3    for i ← 1 to n − k
4        kmers = kmers ∪ {s_i ··· s_{i+k−1}}
5    return kmers
```

Figure 4.2: SEQUENCE2SET takes a sequence and an integer $k$ and returns a set of all of the $k$-mers contained in the sequence

DNA or proteins.

The Levenshtein distance (commonly called edit distance) [25] between two sequences is the minimum number of edits (insertions, deletions, and substitutions) needed to change one sequence into the other. It is commonly computed by finding an alignment between the two sequences using the Wagner-Fischer algorithm [53].

Instead of just looking at protein or DNA sequences as strings of characters, they can also be looked at as sets of $k$-mers. Figure 4.2 shows how a sequence can be turned into a set of $k$-mers. With $k = 2$, for example, the sequence $S = AGTGGTC$ is transformed to the set $\{AG, GT, TG, GG, TC\}$, which is the set of all the 2-mers in $S$. Given two such sets, the Jaccard distance [22] between them is defined as one minus the number of elements that they have in common divided by the number of elements that are in either of them. Formally, the Jaccard distance between sets $A$ and $B$ is

$$d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}.$$

This is 0 when the sets are identical (the sequences are made up of the same $k$-mers) and 1 when they do not share any elements.

Sequences can also be transformed into $k$-mer vectors. Again with $k = 2$, the sequence

KMER2INT(*Sequence kmer*, *Alphabet* $\Sigma$)

```
1    ▷ kmer = [s_1 ··· s_k]
2    ▷ Σ = Map from characters to unique integers
3    result ← 0
4    for i ← 1 to k
5        result = result + Σ[s_i]
6    return result
```

SEQUENCE2VEC(*Sequence* $S$, *int* $k$, *Alphabet* $\Sigma$)

```
1    ▷ S = [s_1 s_2 ··· s_n]
2    ▷ Σ = Map from characters to unique integers
3    kmers ← [0 0 ··· 0]
4    ▷ |kmers| = |Σ|^k
5    for i ← 1 to n − k
6        kmer = KMER2INT(s_i ··· s_{i+k}, Σ)
7        kmers[kmer] = kmers[kmer] + 1
8    return kmers
```

Figure 4.3: SEQUENCE2VEC takes a sequence, an integer $k$, and a alphabet $\Sigma$ and returns a vector representing the sequence

$S = AGTGGTC$ becomes the $4^2 = 16$ (alphabet size to the power of $k$) dimensional vector $V = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 1, 1]$ where $v_i$ is the number of times the lexicographically $i$th 2-mer appears in $S$. The first position, $v_1$, is 0 because $S$ does not contain $AA$ (the lexicographically first 2-mer). The third position, $v_3$, is 1 because $S$ contains exactly 1 copy of $AG$. The algorithm for this is shown in Figure 4.3. There are many distance metrics between vectors. One that is made use of here is the vector norm [37, 54] (also called $p$-norm or $L^p$-norm). The $L^p$-norm of a $n$-dimensional vector $X$ is

$$||X||_p = (\Sigma_{i=1}^{n} x_i^p)^{1/p}.$$

The $L^p$-norm distance between two vectors $A$ and $B$ with the same number of dimensions is the $L^p$-norm of the vector representing the distance between them $d(A, B) = ||A - B||_p$ where $X = A - B$ is defined as $x_i = a_i - b_i$. When $p = 1$, this is commonly called Manhattan distance, and, when $p = 2$, it is called Euclidean distance.

It is important to note that these transformations do not preserve all properties of a sequence. What is a metric for vectors or sets might not be a metric for sequences. As a trivial example, the sequences $AGATA$ and $ATAGA$ would both be transformed to the set $\{AG, AT, GA, TA\}$ or the vector $[0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]$. Thus they are identical as sets or vectors and would have 0 Jaccard distance and 0 $L^p$-norm distance, but they are not the same sequence.

## 4.2 Similarity measures

*Similarity measures* [29, 39, 43] are not constrained to have the same properties as metrics. The closer two points are to each other, the higher the similarity between them. Due to this

lack of constraints, similarity measures tend to not give the same guarantees as metrics, but they can be more meaningful for certain applications because they can be directly based on domain knowledge.

Needleman-Wunsch similarity [32] is similar to Levenshtein distance. Instead of counting the number of differences between two sequences it counts the number of similarities. There is a weighted variant in which some edits cost more than others depending on the characters being edited. When working with protein sequences, the BLOSUM [19] and PAM [11] families of scoring matrices are commonly used.

Tools that use other measures and variations of the ones discussed here that were not investigated but are still relevant include BLAST-based edge-weights [16], dna2vec [33], and BBTools [9].

## 4.3 Definitions

The rest of this chapter assumes a metric $d$ is being used, although similarity measures can be used heuristically. As described in [17, 41] and the Supplemental Material from [59], similarity measures can approximate the guarantees made by metrics despite not obeying the triangle inequality. In practice, this potentially affects the sensitivity of the search.

In a metric space with a distance function $d$, the *ball* $B(c, r)$ is defined by a center point $c$ and a radius $r$. The ball contains all points $p$ in the space such that $d(c, p) \leq r$.

Given a finite set of points $P$ in a metric space, a *ball cover* of $P$ of size $n$ is a set of balls $B = \{B(c_1, r_1), B(c_2, r_2), ...B(c_n, r_n)\}$ where $P \subseteq \cup_{i=1}^{n} B(c_i, n_i)$. See Figure 4.4 for an illustration of a ball covering.

*Metric entropy* [50, 56] (also called Kolmogorov entropy [24]) is a measure of the diversity

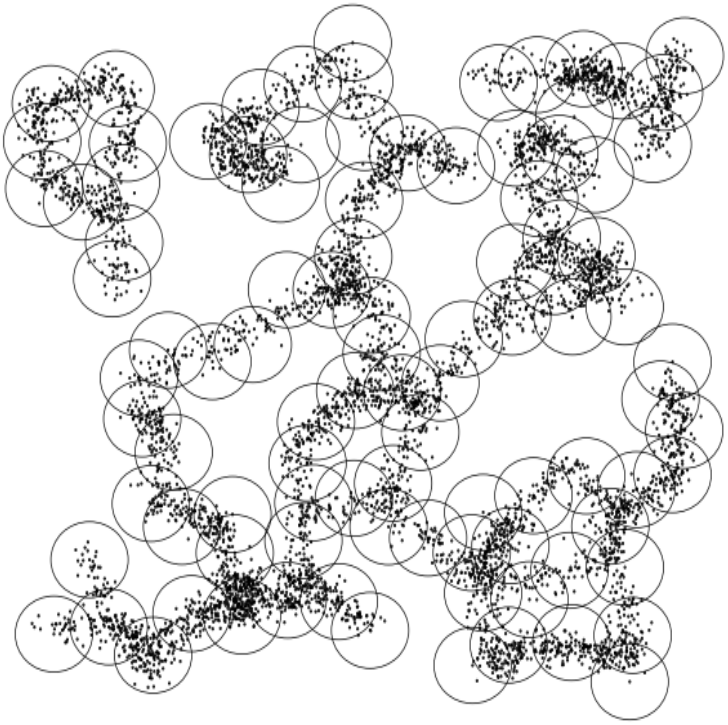

Figure 4.4: A ball covering of points in a 2D space

of a set of points in a metric space. Given a distance metric $d$ and a radius $r$, the metric entropy of a set of points $P$ is equal to the minimum number of balls of radius $r$ needed to cover the set. More formally, the metric entropy $M_r(P) = min_{C \in BC(P)}|C|$ where $BC(P)$ is the set of all possible ball covers of $P$ using balls of radius $r$.

*Fractal dimension* [12] (also called Hausdorff dimension) is a measure of how the metric entropy changes with respect to the ball radius. Formally, it is the exponent $D$ in the expression $n_p = r^D$ where $n_p$ is the number of points from a set of points $P$ that are covered by a ball of radius $r$. More conceptually, it is the dimension of the space with minimum dimensionality required to define $P$. For example, a set of points $P$ might exist in two dimensional space, but only be one dimensional itself. If $P$ consisted of points on a line in two dimensional Euclidean space, then the points in $P$ are represented as a vector of length two, but it would only require a vector of length one (a scalar) to define a point in

$P$. Fractal dimension is just a generalization of this idea to fractional dimensions. It can be helpful to have a heuristic understanding of fractal dimension, but it is prohibitively complex and computationally expensive to empirically measure the fractal dimension of a large finite set of points such as the biological databases that are relevant to the problem described in Chapter 2.

## 4.4 Ball Cover Construction

There are many ways to construct ball covers of $P$. For example, a trivial starting point is to make every point in the set the center of a ball. This is guaranteed to cover $P$, but is not necessarily useful. Particularly, my use case cares more about a minimal set cover with an approximately uniform number of points in each ball, that is, a set cover with a minimal number of balls. In some cases, if the ball radius is sufficiently small, this trivial case might be optimal. A truly optimal solution for this problem is NP-hard [15, 26] and is not practical for this application. Because of this, I make many simplifications that make ball cover construction feasible. The first of these is that only points from the input set are considered as potential ball centers. This makes index construction much simpler, but except in trivial or contrived cases, will at best approximate the optimal ball cover. For example, in a 2-dimensional space with a Euclidean metric, the points $(2, 0)$ and $(0, 2)$ could be covered by a ball with radius 2 centered at $(1, 1)$. However, this set of points could not be covered by a single ball if the ball center was constrained to be one of the input points. With the data sets dealt with here, it is much more straightforward to only look for ball centers in the data points from the input. Because of this, the problem of constructing a ball cover can be reduced to that of selecting a subset of centers from $P$. Chapter 5 describes several of these methods and investigates when they perform well.

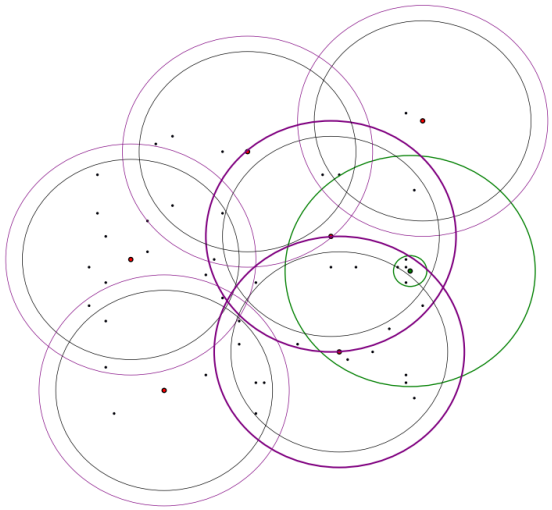

Figure 4.5: A search with query point $q$ and query radius $r_q$ in a set of points that has a ball covering with radius $r_b$. Red points are ball centers and black circles represent balls. The green point is the query point, the small green circle has radius $r_q$ and the large green circle has radius $r_b + r_q$. Purple circles have radius $r_b + r_q$.

## 4.5 Ball Cover Search

Given a set of points $P$ with a covering of balls $B$ with radius $r_b$, a search for all points within query radius $r_q$ of a query point $q$ can be performed as follows: Find all ball centers within the ball radius plus the query radius of the query point $C_q = \{c \in B | d(c, q) < r_b + r_q\}$. These are the balls that intersect with the ball defined by $q$ and $r_q$. Any points that are within $r_q$ of the query point would then have to be contained in the union of these identified balls $\bigcup C_q$. Instead of having to search all of the points in the data set, only the points in this union have to be searched. This is shown in Figure 4.5. If the metric entropy of a data set is sufficiently low, then a ball covering can be constructed such that any query will intersect a small number of balls, the union of which will be small compared to the number of points in the entire data set. This can lead to a significant reduction in query time.

# Chapter 5

# Methods

Several experimental implementations of the entropy scaling system shown in Figure were 5.1 investigated. They all used the same basic strategy of covering a set of data points with balls, then constructing normal indices on the points within each ball and an index of the centers of the balls. These implementations used protein sequences as data points, so the individual indices were created using BLAST. Searches then proceed as described in Figure 5.2.

Perform a coarse search on the index created from ball centers. Use the results of the coarse search to identify which balls contain the best results. Perform fine searches on the indices of the identified balls. The results of these fine searches are the results of the entropy scaling search.

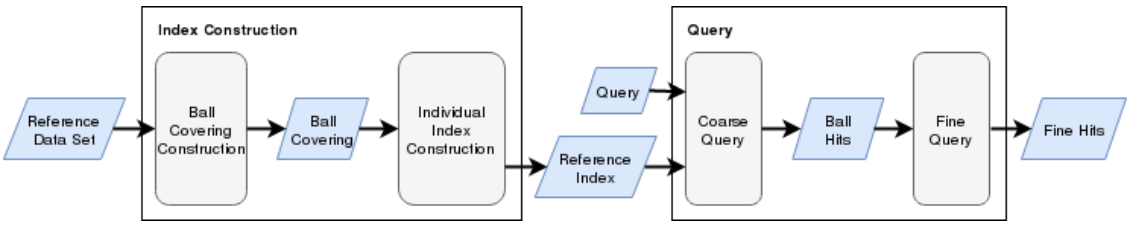


Figure 5.1: System Diagram

```
EntropyScalingSearch(ReferenceIndex R, QueryPoint Q)
    ▷ ReferenceIndex is made up of two parts:
    ▷ centers which is an index on the ball centers,
    ▷ and balls which maps ball center to ball index
    ▷ EntropyScalingSearch returns a set of points from
    ▷ the reference index which are close to the query point
    coarse_hits ← Search(R.centers, Q)
    fine_hits ← {}
    for coarse_hit ∈ coarse_hits
        fine_hits ← fine_hits ∪ Search(R.balls[coarse_hit], Q)
    return fine_hits
```

Figure 5.2: Pseudocode for the entropy scaling search algorithm. When $Search(Index, Query)$ is called as a subroutine, this refers to BLAST, DIAMOND or a comparable non-entropy scaling search algorithm

## 5.1 Measuring Metric Entropy and Fractal Dimension

Before running any of the ball covering construction experiments, it is useful to be able to approximately measure the metric entropy and fractal dimension of a data set. Doing these measurements without having to construct an entire minimal ball covering can facilitate optimizing the parameters for all the other experiments. To achieve this, I first created the Approximate Metric Entropy Measurement (AMEM) tool that estimates the metric entropy and fractal dimension (Figure 5.3). AMEM starts with a range of candidate ball radii $R = \{r_1 r_2 \cdots r_n\}$. Here, $\|R\| = 20$ by default. Then, given a set of data points $P$, AMEM samples a small number $C$ (set at runtime, default 100) of data points as ball centers, that is, $S \subset P$ and $|S| = C$. It then measures how many points are in the balls defined by the ball centers $S$ and candidate ball radii. It reports the average and standard deviation of the number of points in each ball for each radius. Because this is only done with a small number of fixed ball centers, AMEM is relatively fast, but can still give a good indication of the metric entropy and fractal dimension of a data set.

```
ApproximateMetricEntropy(DataPoints P, CandidateRadii R, SampleCount C)
1     ▷ P = {p_1 p_2 ··· p_n}
2     ▷ R = [r_1 r_2 ··· r_m]
3     ▷ 0 < C < |P|
4     counts = []
5     for i ← 1 to m
6         counts[i] ← [0 0 0 ··· 0]
7         ▷ |counts[i]| = C
8     S ← SAMPLE(P, C)
9     for p ∈ P
10        for i ← 1 to m
11            for j ← 1 to C
12                if d(p, s_j) < r_i
13                   then counts[i][j]+ = 1
14    for i ← 1 to m
15        print r_i, mean(counts[i]), std_dev(counts[i]
```

Figure 5.3: Pseudocode for the approximate metric entropy measurement algorithm. the $SAMPLE(P, n)$ function returns a set of $n$ randomly chosen elements from the set $P$.

## 5.2 Naive Ball Cover Construction

The first, most straightforward, test implementation constructs the ball cover using a naive greedy clustering algorithm. A fixed ball radius $r$ is set at the beginning of each run. The main part of the algorithm starts with an empty set of ball centers $B$, adding points to the ball cover in a greedy fashion. It does this by linearly scanning the data points and comparing each data point to all of the centers of the balls that have already been constructed. For each point $p$ from the data set, it uses a linear scan to identify the ball center $c$ that minimizes $d(p, c)$. If $p$ is within the ball radius of $c$ ($d(p, c) < r$), then $p$ is added to the corresponding ball and the algorithm moves on to the next point. If no such ball is found (when the first point is processed the ball set is empty) or $p$ is not within the ball radius of $c$ ($d(p, c) > r$), then $p$ is added to the ball center set $B$.

```
NaiveClusteringBallCover(DataPoints P, BallRadius r)
1    ▷ P = [p_1 p_2 ··· p_n]
2    Centers ← []
3    Balls ← []
4    for i ← 1 to n
5        c_dist ← inf
6        c_i ← −1
7        for j ← 1 to |Centers|
8            if d(p_i, Centers[j]) < c_dist
9                c_dist ← d(p_i, Centers[j])
10               c_i ← j
11       if c_dist ≤ r
12           Balls[c_i].append(p_i)
13       else
14           Balls.append([p_i])
15           Centers.append(p_i)
16   return Centers, Balls
```

Figure 5.4: Pseudocode for the naive clustering ball cover construction algorithm.

Although this method is simple to implement, it does not necessarily have the best performance. There is a single iteration for each data point, and each iteration takes time proportional to the number of balls that have been constructed so far. Thus, the total runtime can be expressed as $O(nE)$, where $n$ is the number of data points and $E$ is the metric entropy of the data set. In the worst case, if a data set has a high metric entropy for a given ball radius then $E$ would go to $n$, meaning that each point is a ball center. In this case, the runtime is $O(n^2)$.

The primary trade-off that this implementation demonstrates is between precision and index building speed. If the measure used is a true metric that obeys the triangle inequality, then perfect precision is guaranteed. If it is not, then the higher the ball radius, the less precise the results will be, but the faster the index can be built.

## 5.3 Random Ball Cover Construction

The second test implementation uses randomized clustering to construct the ball cover. It first scans the points and randomly selects a fixed number $f$ of them as ball centers. See Section 5.1 for the heuristic used to set this number of ball centers. The algorithm then scans the data set, adding each point $p$ to the ball corresponding to the ball center $c$ that is closest to $p$ (minimizes $d(p, c)$).

This implementation has the same theoretic complexity as the one described in Section 5.2, but, because all balls are created at the beginning, it is much easier to parallelize. In the worst case, when a data set has high entropy and the number of balls is directly proportional to the number of points, this implementation still has the same poor runtime as in Section 5.2. However, this is shown closer to the beginning when the ball centers are picked instead of not being apparent until the tool has already been running for a significant amount of time.

## 5.4 Pre-Clustered Ball Cover Construction

Because an efficient ball covering of a set of points roughly corresponds to a particular kind of clustering of those points, the results of an external clustering tool can be used instead of the index construction implementation having to perform the clustering itself. There are many clustering algorithms that are optimized for various applications, but a clustering tool that has been shown to scale well for biological sequence data sets is CD-HIT [27]. This implementation first runs CD-HIT on a data set, then uses the resulting clusters as ball centers. It proceeds to add points to the balls in the same manner as described in Section 5.3.

```
RandomClusteringBallCover(DataPoints P, BallCount f)
1    ▷ P = [p_1 p_2 ··· p_n]
2    Centers ← SAMPLE(P, f)
3    Balls ← [[], [], ··· , []]
4    ▷ |Balls| = f
5    for i ← 1 to n
6        c_dist ← ∞
7        c_i ← −1
8        for j ← 1 to |Centers|
9            if d(p_i, Centers[j]) < c_dist
10               c_dist ← d(p_i, Centers[j])
11               c_i ← j
12       Balls[c_i].append(p_i)
13   return Centers, Balls
```

Figure 5.5: Pseudocode for the random clustering ball cover construction algorithm. The $SAMPLE(P, n)$ function returns an array of $n$ randomly chosen elements from the set $P$

It is important to note here that a mismatch between the metric or measure used for constructing the balls and the metric or measure used for performing the searches can lead to a decrease in sensitivity for the same reason described in Chapter 4. For example, CD-HIT clusters sequences based on percent identity. If the search algorithm uses Euclidean distance, then it might only find results when there is a correlation between percent identity and Euclidean distance. Results can still be fairly good, but this method does not lead to the same guarantees as an implementation that was consistent in its use of a measure.

The performance of this implementation is heavily dependent on the performance and results of the clustering tool used. The more cluster centers that the external tool returns the more work that this implementation has to do to place the remaining points into balls.

```
PRECLUSTEREDBALLCOVER(DataPoints P, Centers)
1    ▷ P = [p_1 p_2 ··· p_n]
2    ▷ C = [c_1 c_2 ··· c_m]
3    Balls ← [[], [], ··· , []]
4    ▷ |Balls| = m
5    for i ← 1 to n
6        c_dist ← inf
7        c_i ← −1
8        for j ← 1 to |Centers|
9            if d(p_i, Centers[j]) < c_dist
10               c_dist ← d(p_i, Centers[j])
11               c_i ← j
12       Balls[c_i].append(p_i)
13   return Centers, Balls
```

Figure 5.6: Pseudocode for the pre-clustered ball cover construction algorithm.

## 5.5 Locality Sensitive Hash Ball Cover Construction

Depending on the similarity measure or distance metric used, it can be fairly easy to speed up the previously described implementations using locality sensitive hashing (LSH) [10, 42]. One of the slowest parts of the ball construction step is having to compare each new point against all of the previously constructed ball centers. LSH identifies a subset of candidate ball centers that, with high probability, contains the desired closest ball center. To do this, LSH adds several steps to the ball construction. It begins by randomly generating $K$ hash functions each of which has $B$ result bins. These hash functions are constructed in such a way that points that are near each other are hashed to the same bin with high probability. For each input point, $p$, each of the $K$ hash functions is computed on $p$ resulting in $K$ bins. All of the cluster centers that have been placed in any of these bins are then considered to be candidate hits. The new point $p$ is compared against each of the candidate hits and the closest is found as before. If this closest center is within the ball radius of $p$, then $p$ is added

to the corresponding ball. If not, a new ball is created with $p$ as its center. Ball creation is more complex than it was in Section 5.2. Instead of just adding $p$ to a set of ball centers, $p$ also has to be added to all of the bins that it hashes to from the $K$ hash functions.

This implementation reveals several trade-offs that were not available in any of the previous implementations. Depending on the values chosen for number of hash functions $K$ and number of bins per hash function $B$, this implementation can have widely varying performance. In particular, increasing $K$ will improve the chance that the set of candidate centers contains the true closest ball center, thus increasing sensitivity. However, it would also require more hashes to be computed and for $p$ to be compared against more candidate centers, increasing run times. Similarly, decreasing $B$ causes there to be more centers in each bin, leading to better sensitivity and longer run times. Alternatively, decreasing $K$ reduces the number of hashes that have to be computed for each point and decreases the size of the candidate hit set, which in turn reduces the number of centers that $p$ is compared to. This potentially decreases sensitivity, but can significantly improve runtime. Lastly, increasing $B$ decreases the number of points in each bin, decreasing the size of the candidate hit set, decreasing sensitivity and runtime.

## 5.6 MinHash Based Ball Cover Construction

A slightly different approach to locality sensitive hashing is using MinHash [7, 36, 58]. This technique was specifically developed to speed up the approximation of Jaccard distance [22], although it can be heuristically used for other metrics. MinHash is explicitly defined for sets of elements. This corresponds more easily with sets of $k$-mers from sequences instead of the abstract point notation used previously. First, a set of $n$ permutations is created over the $k$-mer space. Each permutation is a deterministic ordering of all possible $k$-mers. Then,

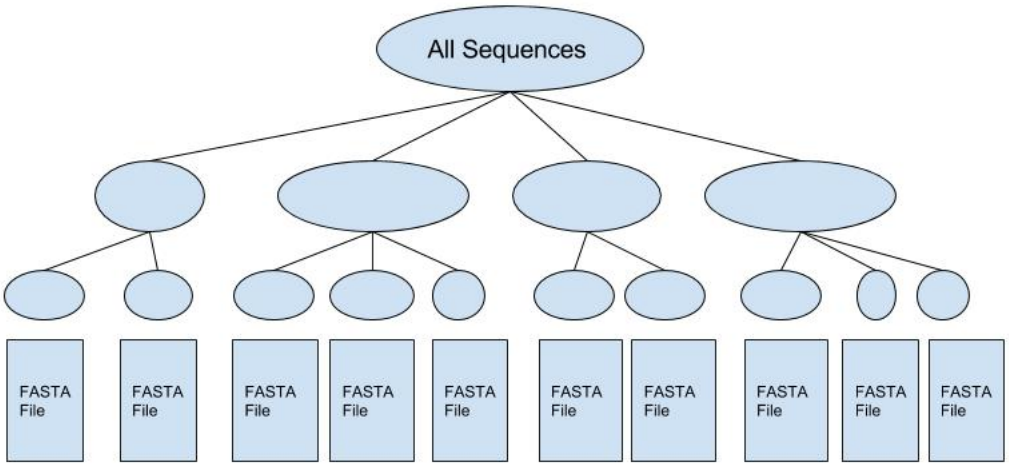


Figure 5.7: Sequence tree representation of a set of sequences

given a set of $k$-mers $S$, a MinHash sketch of $S$ is constructed by saving the minimum $k$-mer from $S$ under each of the $n$ permutations. The Hamming distance between these sketches approximates the Jaccard distance between the original sets, because the expected value of the minimum elements of two sets being equal under a random permutation is equal to the size of the intersection of the sets divided by the size of their union. However, these min-hash sketches can also be used to perform locality sensitive hashing by treating the minimum element under permutations as the hash functions described in Section 5.5.

## 5.7 Sequence Trees

Another approach I investigated was based on Sequence Bloom Trees [46]. It takes the idea of clustering to another level of abstraction. Whereas the clustering methods described above perform clustering once and chose a single set of representative sequences, Sequence Bloom Trees use something similar to hierarchical clustering and allow comparison at each level of the tree (Figure 5.7). In particular, they perform this hierarchical clustering by storing a bloom filter in each node. A bloom filter is a data structure that uses hashing to efficiently

recognize whether or not it has seen a data point (a sequence in this case) before without having to store the entirety of all of the data points that it sees. Due to the nature of the hashing used, it is possible for a bloom filter to give false positives (report that it has seen an item before when it has not), but it can never give false negatives. The larger the bloom filter is proportional to the number of items that it is storing, the less chance it has of giving a false positive.

The index tree starts with just a root and sequences are added one at a time. Each sequence is compared to the bloom filters stored in the children of the root. The root itself represents all sequences that have been processed, so when adding a sequence there is no need to compare it to the root. The sequence is added to any nodes that it matches above some threshold, then this process propagates to their children. If at any point a sequence does not match any of the children of a node, then a new child is added to the node and initialized with the sequence. This tree is built as described to a specified depth. When each sequence reaches that specified depth, instead of creating a new child at a deeper level, it is added to the FASTA file represented by that leaf node. Searches are performed on this index in much the same way that sequences are added. A query sequence is first compared to the bloom filter stored in the root of the tree. If it does not match the root of the tree, then it could not match any of the sequences in the tree. If it does match the root, then it is compared to all the children of the root. The query is then propagated to any nodes that it matches above the specified threshold. When the query reaches a leaf, it can be directly compared to each of the sequences stored in the corresponding FASTA file because each leaf only stores a relatively small number of sequences.

This approach is already being explored in several directions [45, 48]. I looked into using other metrics or measures such as the ones described in Section 4.1 as the basis for constructing the tree. Despite the potential shown by sequence trees' computational complexity, this

approach presented a greater engineering challenge than the other methods described in this chapter. Because of this, I have not focused on testing this implementation.

## 5.8 Evaluation Methods

These methods can be evaluated in several ways. Possibly the most important measures are query time, precision, and sensitivity (recall) [40]. Precision is the fraction of returned sequences that are similar to the query, that is, the number of true positives divided by the total number of results. Sensitivity is the fraction of similar sequences that were in the index which were returned, that is, the number of true positives divided by the total number of sequences in the index which are similar to the query. Two other measures are index build time and index size. Because query time, index build time, and index size are all strongly related to the data set size, and this relationship could be dictated by the relative entropy of the data set, it is important that all methods be tested on data sets of significantly differing sizes. As well as measuring all of these statistics empirically, it is valuable to compute them analytically to estimate how these methods will behave in situations different than what we were able to test explicitly.

# Chapter 6

# Results

The entropy scaling sequence similarity search algorithm (ESSSSA) implementations described in Chapter 5 were tested on sets of both biological and synthetic data of varying sizes. Additionally, three sequence similarity search tools, BLAST [1], MICA [59], and DIAMOND, [8], were run on the same biological sequences.

## 6.1 Test Data

All of the biological data used in these tests came from the UniProt Consortium [52]. Specifically, I used the UniRef100 [49] protein clusters version 2018_01, which contains approximately 133 million sequences varying in length from 2 amino acids to 38105 amino acids. To test how tools scaled with the size of the data, I randomly sampled UniRef100 by taking 1 out of every 100 sequences to give a set of 1.33 million sequences. This set of sequences was then used to create 20 sets that were used to test, where set $i$ had $i$ sequences randomly chosen out of every 20. That is, the first contained 1 sequence out of every 20, the second contained 2 out of every 20, up to the 20th test set which had the full 1.33 million sequences. For each of these 20 data sets, a corresponding random data set was also synthesized. Each of these synthetic data sets was created to have the exact same number of sequences with exactly the same lengths as the biological data sets, just the contents of the sequences were uniformly random over the protein alphabet. All of the test queries consisted of single se-

quences randomly chosen from the largest data set. Not every query sequence is guaranteed to appear in the data set being searched.

## 6.2 Synthetic Data Entropy Comparison

Before constructing a ball cover on a set of biological sequences, it is useful to know what the metric entropy of the set is. The entropy measuring tool described in Section 5.1 was tested with the biological data sets as well as their corresponding randomized synthetic data sets. The results of these runs are shown in Figure 6.1. It is interesting how big a difference is made by which metric is used. Under Jaccard distance, the average number of points per ball in the biological data is notably higher then in the synthetic data for most radii. This implies that with a well chosen ball radius, the metric entropy of the biological data is notably lower than that of the synthetic data.

## 6.3 Performance

These are the results of running the implementations described in Sections 5.2, 5.3, and 5.5 as well as BLAST, DIAMOND, and MICA on the data sets described in Section 6.1. The pre-clustering method described in Section 5.4 was not included because poor performance precluded it from being tested as thoroughly as the other methods.

### 6.3.1 Index Build Time

None of the index construction algorithms are entropy scaling, so the index build times are approximately linear with respect to the size of the reference data set as is shown in Figure

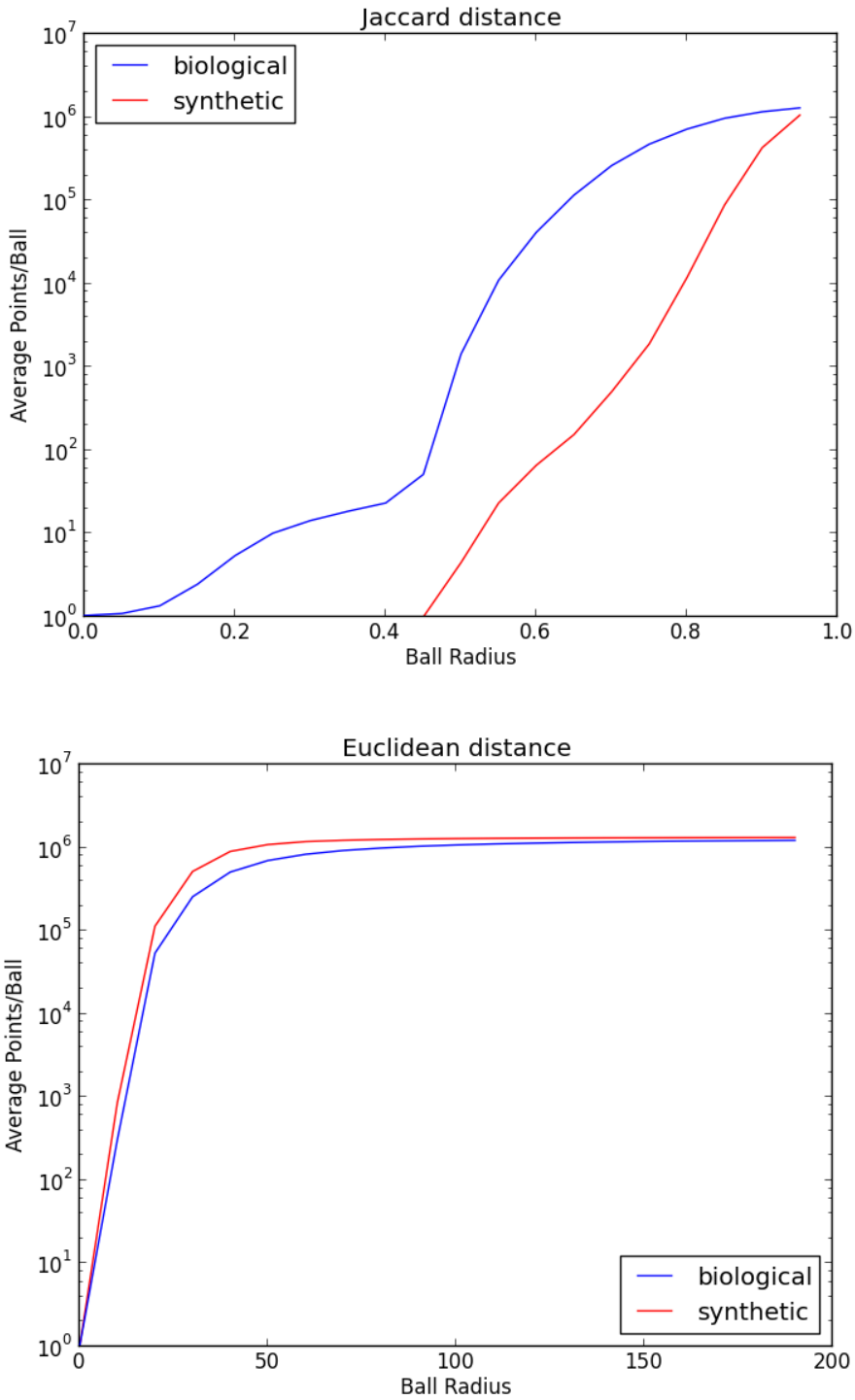


Figure 6.1: Entropy approximations of biological data versus randomized data

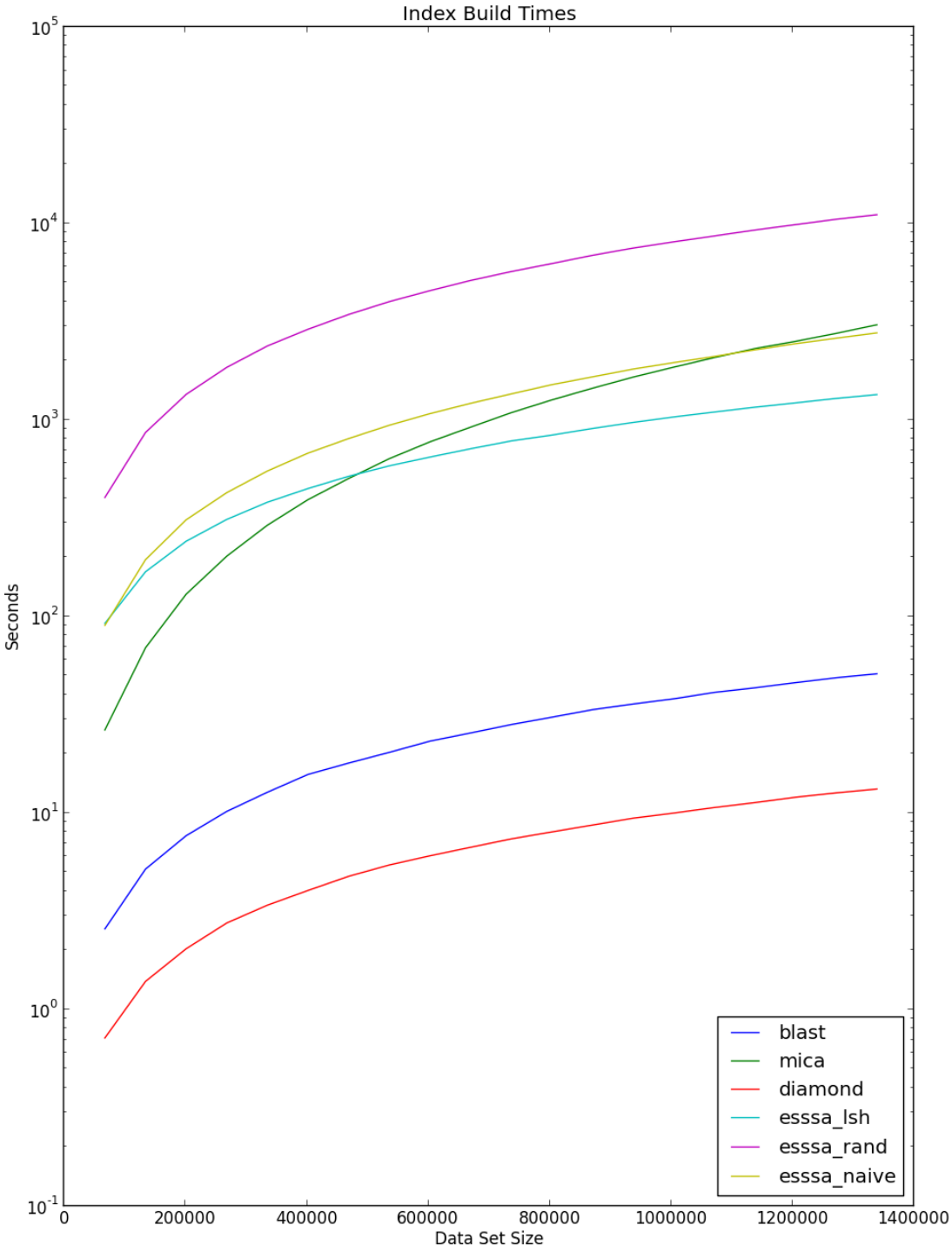


Figure 6.2: Index build times compared to reference data set size. The Entropy Scaling Sequence Similarity Search Algorithm (ESSSSA) shown in red is the implementation of the method described in Section 5.5.

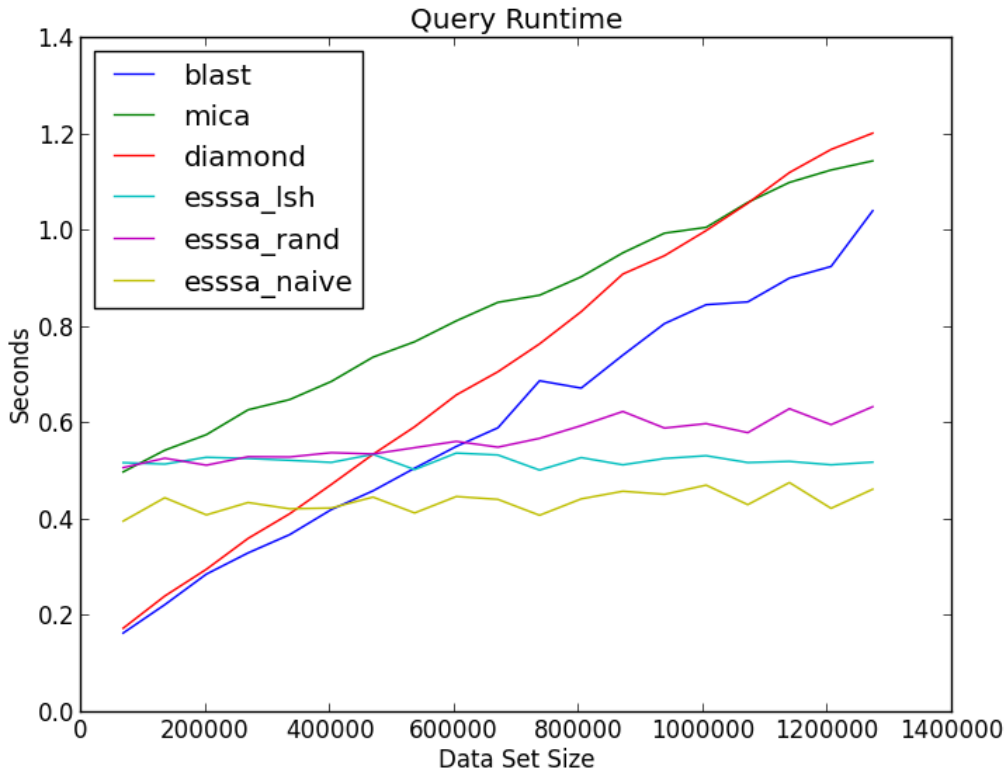


Figure 6.3: Tool query execution times compared to reference data set size. These execution times are averaged across the 100 randomly chosen single sequence queries on each data set.

6.2. MICA and the implementation of the algorithm described in Section 5.5, which are both entropy scaling, take expectedly longer to construct their indices because they have to compute the ball cover before using BLAST or DIAMOND to construct conventional indices.

### 6.3.2 Query Time

Execution times for single sequence queries are shown in Figure 6.3. For the tools which are not entropy scaling such as BLAST and DIAMOND, execution time scales approximately linearly with data sets size. For MICA, the query time does noticeably increase, but not as significantly as the conventional tools. From these tests, none of the three versions of

ESSSSA tested seem to show any strong correlation between query runtime and data set size. The only difference between the three ESSSSA implementations is the index construction method. The queries for all of them follow the entropy scaling search algorithm described in Section 4.5. These execution times are averaged over the 100 queries for each data set.

### 6.3.3 Sensitivity and Precision

Most sequence similarity search tools return a plethora of scores for each result. BLAST and DIAMOND base their scores on alignments such as those described in Sections 4.1 and 4.2. One of the most versatile of these scores is E-value. Given a query and a reference database with $n$ sequences in it, the E-value of a hit $h$ estimates the number of results of the same quality and length as $h$ that would be expected to occur randomly in a database with $n$ sequences [13]. Unfortunately, this means that E-values are only meaningful when comparing results from queries in databases of the same size. Because the ESSSSA tools build indices on and search each ball individually, the E-values that ESSSSA returns do not compare with the E-values returned by BLAST or DIAMOND when run on the entire database. Because of this, bit scores are more useful for comparing these results. Bit scores are log scaled and represent the expected size a database would have to be for a single result of equal length and quality to occur by random. Because bit scores are dependent on query length, I scaled the results of each query to the highest bit-score BLAST returned for that query. This gives scores that mostly fall between zero and one with the occasional score above one representing a result that was better than BLAST. Figures 6.5 and 6.4 show these scaled bit-scores from the results of the queries and data sets described in Section 6.1.

As was mentioned in Section 4.3, because BLAST effectively uses edit distance as its metric and the balls that ESSSSA constructs are based on Jaccard distance, there is an expected

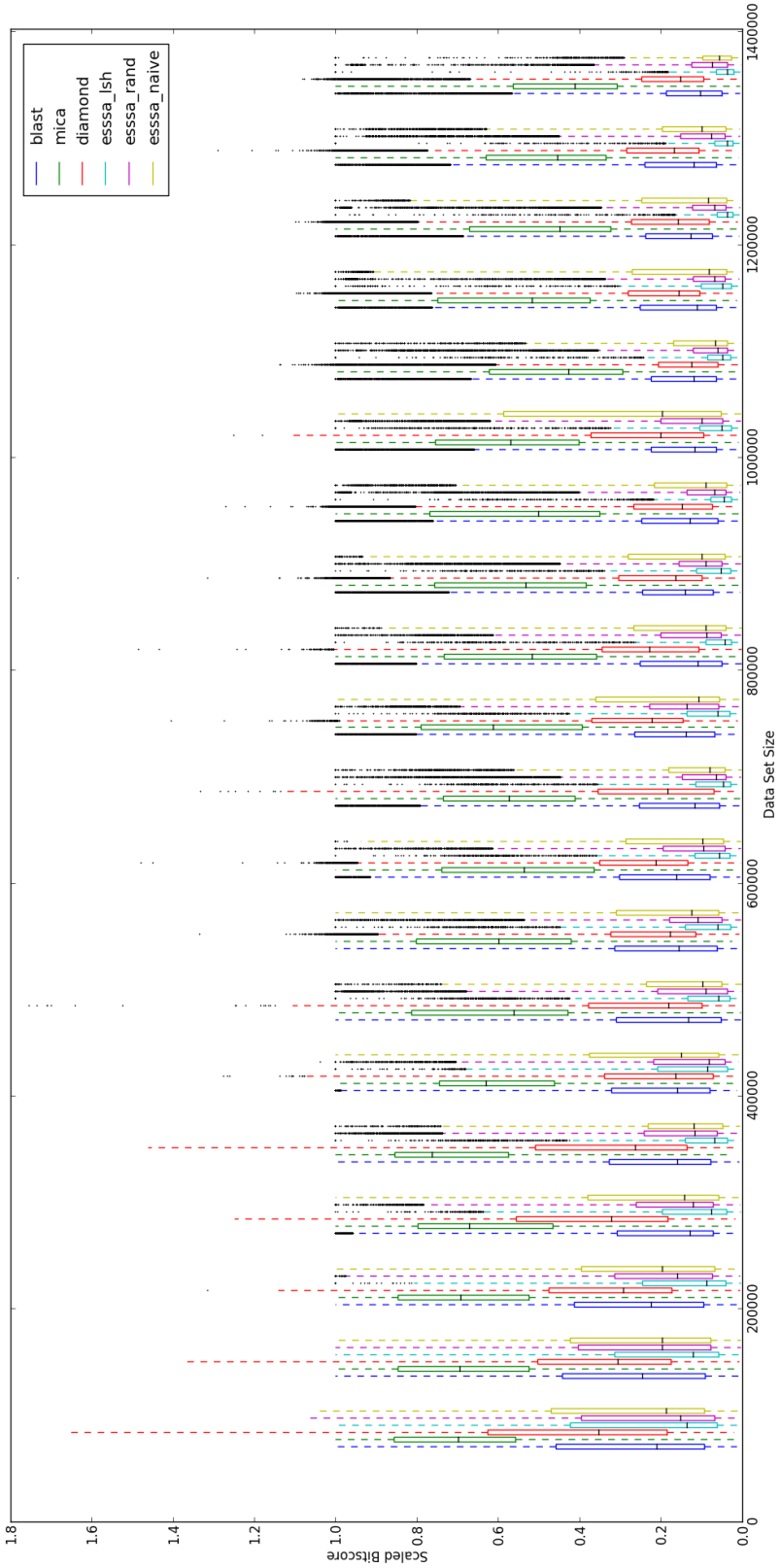


Figure 6.4: Scaled Bit-scores of results from running each tool on all of the 20 datasets. These results are aggregated across 100 randomly chosen single sequence queries on each data set. The scores are scaled so that a value of 1 represents a bit-score equal to the highest bit-score returned by BLAST for that query. One short query is left out that BLAST did not return any results for.

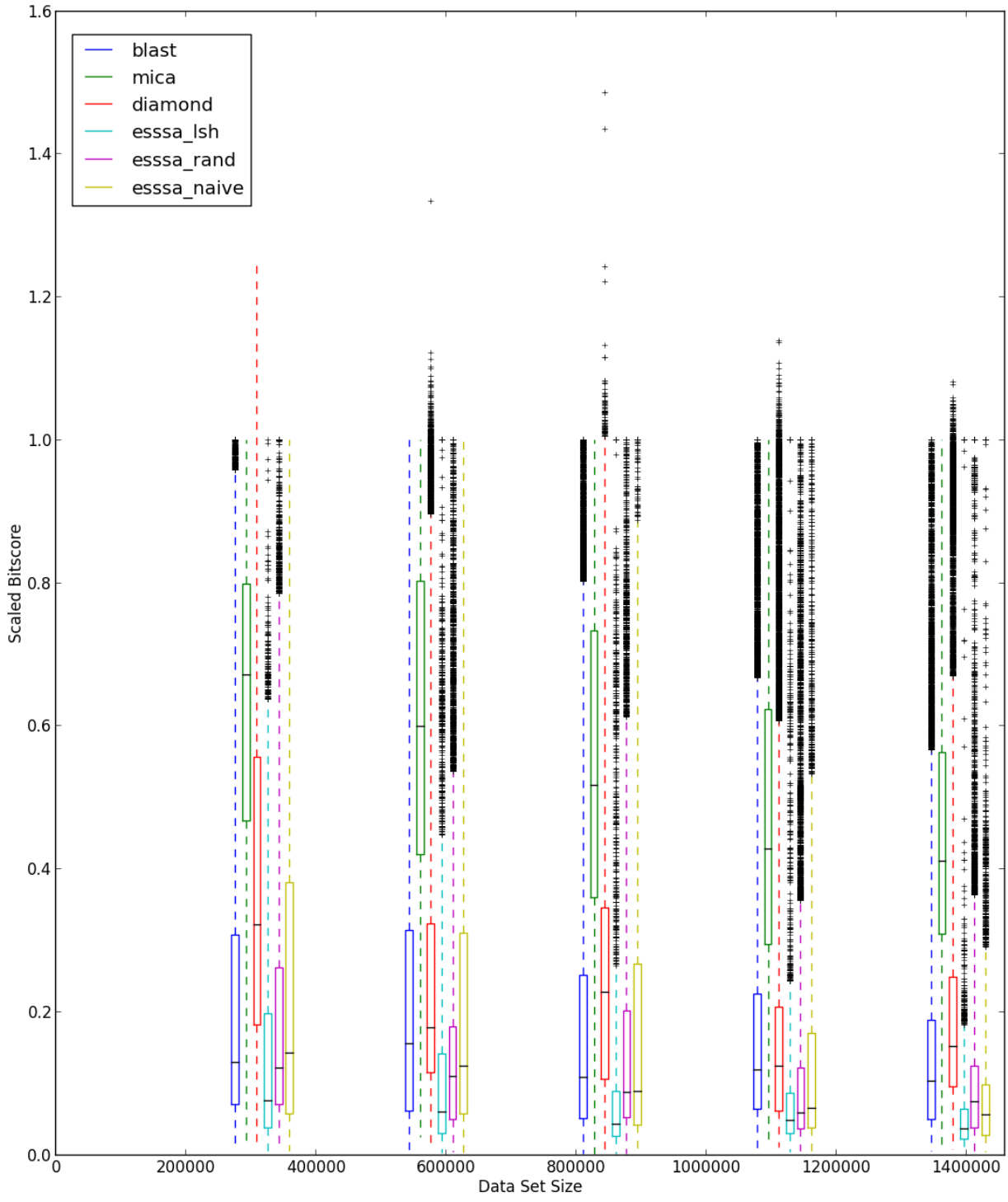


Figure 6.5: Scaled Bit-scores of results from running each tool on a selection of the 20 datasets. This figure is the same as Figure 6.5 but with only 5 of the datasets shown.

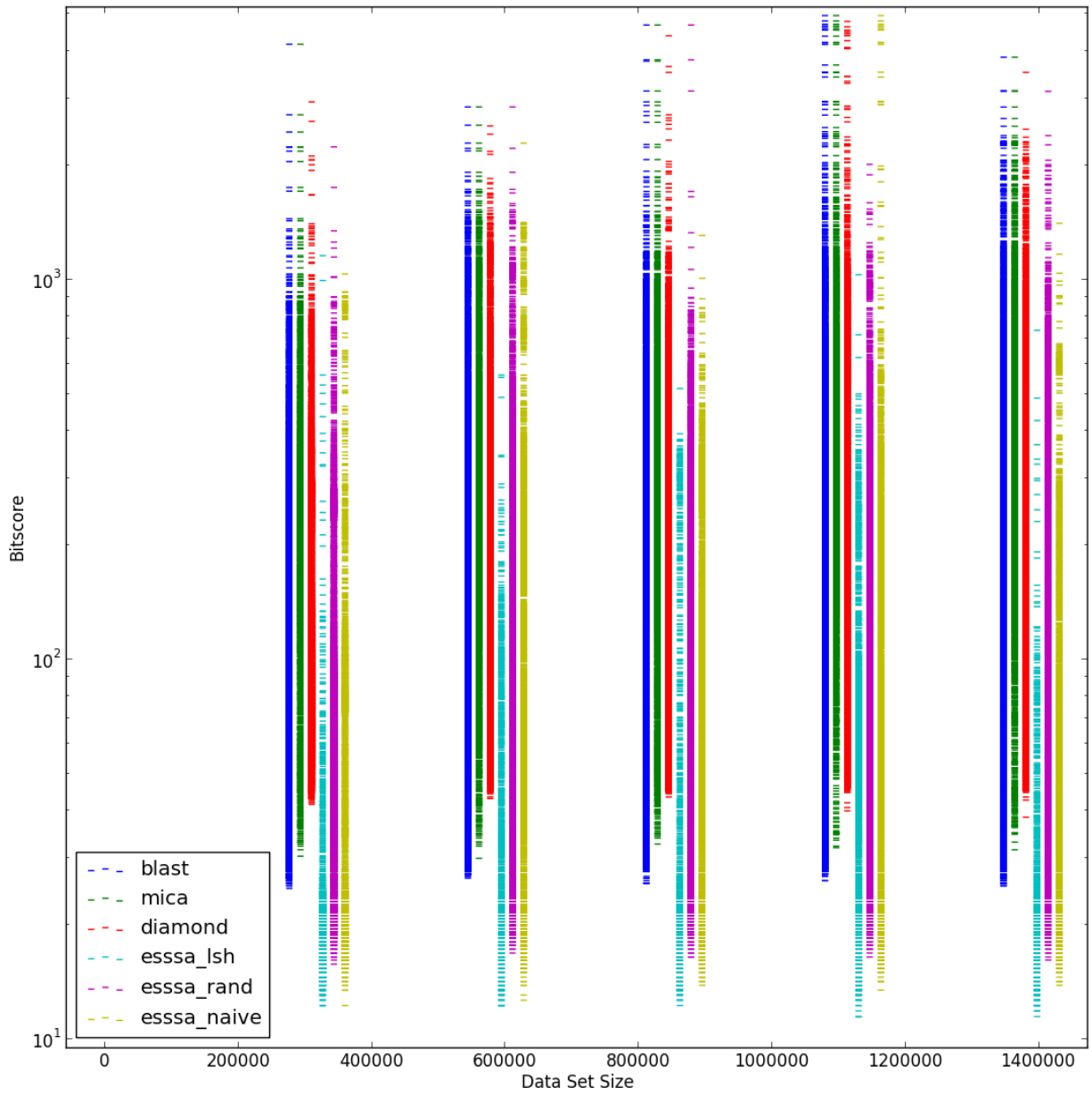


Figure 6.6: Bit-scores of results from running each tool on a selection of the 20 datasets. Each dash is the raw bit-score of a single result. These results are aggregated across 100 randomly chosen single sequence queries on each data set.

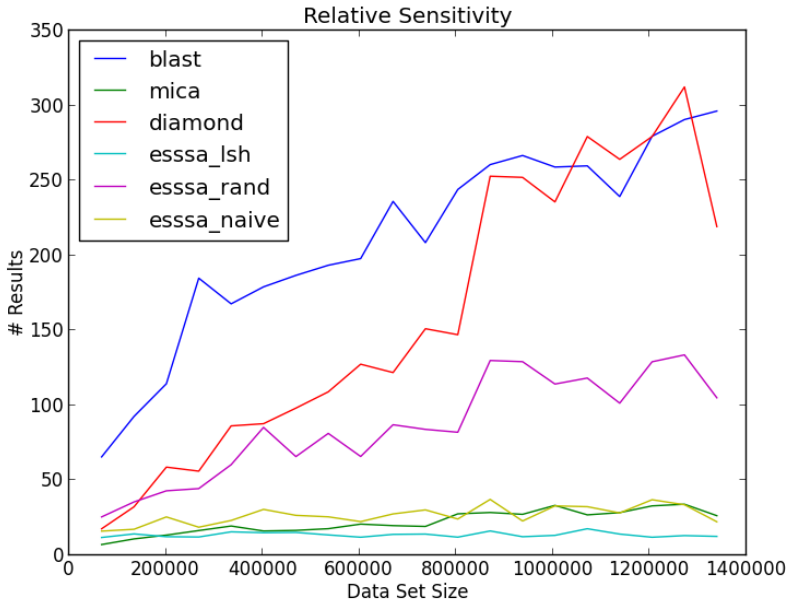


Figure 6.7: Average number of results returned by each tool.

mismatch between what ESSSSA puts in each ball and what BLAST is able to find in those balls. In the ESSSSA implementations that are heavily dependent on ball radius, this is evidenced by a decrease in sensitivity. Implementations such as the randomized one described in Section 5.3 are not as impacted by this discrepancy.

Although Figures 6.4, 6.5, and 6.6 give a good indication of the comparison between the scores of the results that each tool returns, that illustration can make it hard to determine the number of results from each tool. Figure 6.7 shows this more clearly. Table 6.1 shows the average number of top results found for each query by each tool.

| Tool | Top 5 | Top 10 | Top 20 | Top 50 |
|---|---|---|---|---|
| BLAST | 2.305 | 4.8125 | 9.555 | 22.75 |
| DIAMOND | 1.69 | 2.6125 | 4.005 | 7.0825 |
| MICA | 2.01 | 3.64 | 6.5175 | 13.995 |
| ESSSSA naive | 0.0325 | 0.075 | 0.3525 | 2.1425 |
| ESSSSA random | 0.2075 | 0.5075 | 1.385 | 4.125 |
| ESSSSA LSH | 0.0525 | 0.125 | 0.375 | 1.8575 |

Table 6.1: This table shows on average, how many of the top 5, 10, 20 and 50 results were found by each tool. These results are from the test on the largest data set.

# Chapter 7

# Discussion

The starting point for this research was investigating entropy scaling ideas, motivated, in particular, by the approach taken by an existing tool, MICA. With the use case of threat analysis in mind, I set out to explore whether or not there was potential to be gained from implementing an entropy scaling sequence similarity search algorithm (ESSSSA) tool. After researching the mathematical background necessary, discussed in Chapter 4, I implemented an approximation tool that would estimate the amount of entropy present in a data set. Then, I used this tool with three different metrics to determine which of these metrics would be the most useful. Testing with UniRef100 [49] proteins, I found that Jaccard distance was fast and efficient, and appeared to give the best results, leading towards useful ball covers. Another metric I tried, Euclidean distance, was slightly slower to work with but did not behave as well. The structure over the sequence space that it created caused sequences to be more evenly spaced out, leading to higher entropy proportional to data set size. The last metric I tried was edit distance, which constructed reasonably useful ball covers, but was too slow to be able to test on meaningful amounts of data. The results of these approximation tests with Jaccard distance and Euclidean distance are shown in Section 6.2 in Figure 6.1.

Once I had established that my biological sequences did have relatively low entropy if considered under the right metric, as I had hoped, I began implementing several tools to construct ball covers that could be searched in an entropy scaling fashion. The first implementation used naive clustering as explained in Section 5.2. This was the simplest and most straight-

forward approach, but I found that it could be extremely slow when constructing indices. To speed up index construction, I next implemented a randomized approach that I hoped would build a comparable ball cover, but would not take as long to construct. At this point, I decided to see if existing tools that performed biological sequence clustering could be used to assist in this step. I tried using CD-HIT [27], which is what the UniProt Consortium uses to construct the UniRef clusters. However, CD-HIT creates clusters based on percent identity, which I considered to be one of the things that hindered MICA's performance. I did implement a version of ESSSSA that can use the clusters returned from CD-HIT as ball centers, but I found that even at a low percent identity clustering threshold it would construct too many balls, which led to unacceptably long query execution times. The next idea I investigated was locality sensitive hashing (LSH). This method uses hashing to speed up the naive clustering approach at the cost of some of the sensitivity. Some of the sequences that the naive approach would correctly put into the same ball, LSH would overlook and create new balls for. However, the improvement in runtime and cost to sensitivity can be finely tuned by several parameters. The last technique I investigated was using MinHash comparisons which are based on Jaccard Distance to augment the LSH implementation.

To test all of these implementations I used protein sequences from UniProt and sampled them down to usable quantities as described in Section 6.1. I also ran the same tests with the conventional sequence similarity search tools BLAST and DIAMOND, as well as the entropy scaling tool MICA. In these tests I found that index construction was the primary bottleneck for all of the entropy scaling implementations. In all data sets tested, DIAMOND was the fastest at constructing indices, although BLAST was not much slower. In the smaller data sets, MICA had the third fastest index build times, but once data sets got larger than 600,000 sequences, its build time surpassed the LSH ESSSSA implementation and by 1.2 million sequences it also surpassed the naive clustering implementation. Counter to my

expectations and intentions, the randomized ball cover construction algorithm was notably slower than any of the others on all data set sizes. Despite this, I did construct indices for each tool on data sets of a meaningful size and was able to see how query execution times, and the sensitivity and precision of results depended on data set size. As is described in Section 6.3.2, the ESSSSA implementations all achieved query times that did not strongly scale with data set size. As expected, the non-entropy scaling tools BLAST and DIAMOND had query times that did scale approximately linearly with the data set size. MICA, the other entropy scaling tool, did have query execution times that scaled more slowly than BLAST or DIAMOND, but still significantly increased as the data set got larger. Despite having a better complexity relative to data set size, the entropy scaling methods do have a higher overhead. This led to the ESSSSA tools and MICA having significantly longer query execution times compared to BLAST and DIAMOND in the smaller data sets. This overhead was not overcome until the data sets got larger, approximately 400,000 sequences for the naive clustering implementation and between 500,000 and 600,000 sequences for the locality sensitive hashing based and randomized methods, at which point all of the ESSSSA implementations beat BLAST at average query execution time. MICA also demonstrates this larger overhead and takes longer to overcome it due to its increased complexity. In my tests, MICA's average query time was not surpassed by DIAMOND until the data set size was approximately 1.1 million sequences and was not surpassed by BLAST with even my largest 1.3 million sequence data set. Within the ESSSSA implementations, the index built by naive clustering routinely had the best query execution times, followed by the locality sensitive hashing based method. The search on the randomly constructed ball cover did the worst, although it remained fairly close to the LSH method.

The other aspect that is important when determining how well a tool performs is the quality of the results that it returns. There are many metrics that can be used to measure this. I

used bit-score because, unlike other similar metrics such as E-value, it is not dependent on the size of the dataset that the tool was used on. Instead, bit-score is dependent on query length. To account for this and to present scores evenly across queries, I normalized each score by the best bit-score that BLAST returned. Because BLAST is an industry standard that has been around for longer than any of the other tools discussed here, I chose it as a reasonable baseline. There were a few exceptions where another tool, especially DIAMOND, found higher scoring results than BLAST, but the majority of the results justify this decision. These scaled results can be seen in the box and whisker plots in Figures 6.4 and 6.5 in Section 6.3.3. DIAMOND is the only tool that regularly found results better than BLAST's best result, which is shown by DIAMOND having scores above 1. As data sets get larger, the median score of the results goes down across all the tools tested. Figure 6.5 shows that MICA consistently has the highest median bit-score. This is partly because as Figure 6.7 shows, MICA tends to return a relatively small number of results compared to BLAST or DIAMOND. Among the ESSSSA implementations, the naive clustering method achieves the highest median scores for the smaller data-sets. As the number of reference sequences increases, all three ESSSSA tools return results with similar median scores. On average, the LSH-based implementation returns the smallest number of results, although only slightly less than MICA and the naive clustering based ESSSSA. These two return approximately the same number as each other as the data set size gets larger. The ESSSSA implementation which uses a randomized construction approach returns the most results of all the entropy scaling tools, although not nearly as many as BLAST or DIAMOND. In smaller data sets BLAST returns the most results, although as the data sets get larger DIAMOND returns a similar number.

# Chapter 8

# Conclusion

I investigated the potential of using entropy scaling techniques in the sequence similarity search problem motivated by the biological sequence threat analysis problem. To do this, I started by looking at how a previous tool, MICA, tried to achieve improved query execution times based on entropy scaling. In doing this, I identified some ideas that MICA's design did not take advantage of. Based on these ideas, I explored how the metric used affects performance and determined that while some metrics like Jaccard distance show significant potential, entropy scaling might be impossible under other metrics such as Euclidean distance. A significant difficulty that I faced was the bottleneck of index construction. Building an index that allows for entropy scaling searches can be more computationally expensive than building conventional indices. To combat this, I investigated several ways to accelerate the index construction and how these methods affected query execution time and sensitivity. No single method was the best in all regards, but I have identified some of the tradeoffs that various entropy scaling methods demonstrate as summarized in Table 8.1.

## 8.1 Future Work

There are several additional ideas that have demonstrated potential, but I did not fully explore. Many other metrics and variations on the metrics I used either I did not have the chance to investigate or they had a computational cost that was prohibitive to testing with

| Tool | Index Construction Time | Query Time | Query Sensitivity |
|---|---|---|---|
| BLAST | Linear, low overhead | Linear, low overhead | Large number of results, including most high quality results |
| DIAMOND | Linear, lowest overhead of tools tested | Linear with low overhead | Comparable to BLAST, occasionally better |
| MICA | Low overhead, greater than linear complexity | High overhead, low linear complexity | Small number of high quality results |
| ESSSSA naive | Linear, moderate overhead | Constant, moderate overhead | Small number of moderate quality results |
| ESSSSA random | Linear, high overhead | Constant, high overhead | Moderate number of moderate quality results |
| ESSSSA LSH | Linear, moderate overhead | Constant, high overhead | Very small number of results with low median quality |

Table 8.1: Summary of tool performance in tests on data sets sampled from UniRef100

large amounts of data. As discussed in Sections 4.1 and 6.3.3, using a conventional search tool, other than BLAST, that uses a metric that better aligns with whichever metric is used to construct the ball cover could significantly improve sensitivity. Entirely different ball cover construction approaches such as making use of a sequence bloom tree could significantly speed up index construction. If such an approach made it so that index construction was no longer a testing bottleneck, then the search algorithm could be tested on significantly larger data sets, hopefully demonstrating the extent to which entropy scaling would be beneficial. Another idea that would potentially improve sensitivity is to construct a network on top of the ball cover, connecting balls which are close to each other. Doing this would allow the search algorithm to search nearby balls, making it able to find less conserved sequences.

# Bibliography


[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. ISSN 0022-2836. doi: https://doi.org/10.1016/S0022-2836(05)80360-2. URL http://www.sciencedirect.com/science/article/pii/S0022283605803602.

[2] Michael Ashburner, Catherine A. Ball, Judith A. Blake, David Botstein, Heather Butler, J. Michael Cherry, Allan P. Davis, Kara Dolinski, Selina S. Dwight, et al. Gene Ontology: Tool for the unification of biology. *Nature Genetics*, 25(1):25–29, May 2000. ISSN 1061-4036. URL http://dx.doi.org/10.1038/75556. Commentary.

[3] Philip Ball. Synthetic biology: Starting from scratch. *Nature*, 431:624–626, Oct 2004. URL http://dx.doi.org/10.1038/431624a.

[4] Bonnie Berger, Jian Peng, and Mona Singh. Computational solutions for omics data. *Nature Reviews Genetics*, 14(5):333–346, 2013.

[5] Bonnie Berger, Noah M Daniels, and Y William Yu. Computational biology in the 21st century: Scaling with compressive algorithms. *Communications of the ACM*, 59 (8):72–80, 2016.

[6] M. W. Blasgen and K. P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):363–377, 1977. ISSN 0018-8670. doi: 10.1147/sj.164.0363.

[7] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences*, pages 21–29. IEEE, 1997.

[8] Benjamin Buchfink, Chao Xie, and Daniel H. Huson. Fast and sensitive protein alignment using DIAMOND. *Nature Methods*, 12:59, Nov 2014. URL http://dx.doi.org/10.1038/nmeth.3176.

[9] B Bushnell. BBTools software package. *http://sourceforge.net/projects/bbmap*, 2017.

[10] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, pages 253–262. ACM, 2004.

[11] Margaret O Dayhoff. Atlas of Protein Sequence and Structure. 1965. National Biomedical Research Foundation.

[12] Kenneth Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley & Sons, Ltd, 2005. ISBN 9780470013854. doi: 10.1002/0470013850.ch1. URL http://dx.doi.org/10.1002/0470013850.ch1.

[13] John Fassler and Peter Cooper. Blast glossary, 2008. Bethesda (MD): National Center for Biotechnology Information (US) https://www.ncbi.nlm.nih.gov/books/NBK62051/.

[14] Scott Federhen. *Entrez Taxonomy Quick Start*. Taxonomy Help [Internet]. Bethesda (MD): National Center for Biotechnology Information (US), 2011. https://www.ncbi.nlm.nih.gov/books/NBK53758/.

[15] Robert J. Fowler. Optimal packing and covering in the plane are np-complete. *Inf. Process. Lett*, 12(3):133–137, 1981.

[16] Theodore R. Gibbons, Stephen M. Mount, Endymion D. Cooper, and Charles F. Delwiche. Evaluation of BLAST-based edge-weighting metrics used for homology infer-

ence with the Markov Clustering algorithm. *BMC Bioinformatics*, 16(1):218, Jul 2015. ISSN 1471-2105. doi: 10.1186/s12859-015-0625-x. URL https://doi.org/10.1186/s12859-015-0625-x.

[17] Navin Goyal, Yury Lifshits, and Hinrich Schütze. Disorder inequality: a combinatorial approach to nearest neighbor search. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, pages 25–32. ACM, 2008.

[18] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1950.tb00463.x.

[19] Steven Henikoff and Jorja G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992.

[20] Sandeep Hosangadi. Distance measures for sequences. *arXiv preprint arXiv:1208.5713*, 2012.

[21] Ying Huang, Beifang Niu, Ying Gao, Limin Fu, and Weizhong Li. CD-HIT Suite: A web server for clustering and comparing biological sequences. *Bioinformatics*, 26(5):680–682, 2010. doi: 10.1093/bioinformatics/btq003. URL +http://dx.doi.org/10.1093/bioinformatics/btq003.

[22] Paul Jaccard. The distribution of the flora in the alpine zone.1. *New Phytologist*, 11(2):37–50, 1912. ISSN 1469-8137. doi: 10.1111/j.1469-8137.1912.tb05611.x. URL http://dx.doi.org/10.1111/j.1469-8137.1912.tb05611.x.

[23] Scott D. Kahn. On the future of genomic data. *Science*, 331(6018):728–729, 2011. ISSN 0036-8075. doi: 10.1126/science.1197891. URL http://science.sciencemag.org/content/331/6018/728.

[24] Andrei N Kolmogorov. On tables of random numbers. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 369–376, 1963.

[25] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[26] Jian Li and Yifei Jin. A ptas for the weighted unit disk cover problem. In *International Colloquium on Automata, Languages, and Programming*, pages 898–909. Springer, 2015.

[27] Weizhong Li and Adam Godzik. CD-HIT: A fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658–1659, 2006. doi: 10.1093/bioinformatics/btl158. URL +http://dx.doi.org/10.1093/bioinformatics/btl158.

[28] Weizhong Li, Lukasz Jaroszewski, and Adam Godzik. Clustering of highly homologous sequences to reduce the size of large protein databases. *Bioinformatics*, 17(3):282–283, 2001. doi: 10.1093/bioinformatics/17.3.282. URL +http://dx.doi.org/10.1093/bioinformatics/17.3.282.

[29] Dekang Lin. An information-theoretic definition of similarity. In *International Conference on Machine Learning*, volume 98, pages 296–304. Citeseer, 1998.

[30] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002. doi: 10.1093/bioinformatics/18.3.440. URL +http://dx.doi.org/10.1093/bioinformatics/18.3.440.

[31] Lynne Reed Murphy, Anders Wallqvist, and Ronald M. Levy. Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein Engineering, Design and Selection*, 13(3):149–152, 2000. doi: 10.1093/protein/13.3.149. URL +http://dx.doi.org/10.1093/protein/13.3.149.

[32] 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):443 – 453, 1970. ISSN 0022-2836. doi: https://doi.org/10.1016/0022-2836(70)90057-4. URL http://www.sciencedirect.com/science/article/pii/0022283670900574.

[33] P. Ng. dna2vec: Consistent vector representations of variable-length $k$-mers. *ArXiv e-prints*, January 2017. 1701.06279.

[34] Wenfa Ng. Defining and redefining its role in biology: Synthetic biology as an emerging field at the interface of engineering and biology. *PeerJ Preprints*, 4:e2634v1, 2016.

[35] Nuala A O'Leary, Mathew W Wright, J Rodney Brister, Stacy Ciufo, Diana Haddad, Rich McVeigh, Bhanu Rajput, Barbara Robbertse, Brian Smith-White, Danso Ako-Adjei, et al. Reference sequence (RefSeq) database at NCBI: Current status, taxonomic expansion, and functional annotation. *Nucleic Acids Research*, 44(D1):D733–D745, 2015.

[36] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1):132, Jun 2016. ISSN 1474-760X. doi: 10.1186/s13059-016-0997-x. URL https://doi.org/10.1186/s13059-016-0997-x.

[37] Charles R. Johnson Roger A. Horn. *Matrix Analysis*. Cambridge University Press, 1990.

[38] Sayed M. Sahraeian, Kevin R. Luo, and Steven E. Brenner. SIFTER search: A web server for accurate phylogeny-based protein function prediction. *Nucleic Acids Research*, 43(W1):W141–W147, 2015. doi: 10.1093/nar/gkv461. URL +http://dx.doi.org/10.1093/nar/gkv461.

[39] Simone Santini and Ramesh Jain. Similarity measures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(9):871–883, 1999.

[40] Hinrich Schutze, Christopher D Manning, and Prabhakar Raghavan. *Introduction to Information Retrieval*, volume 39. Cambridge University Press, 2008.

[41] Tomáš Skopal and Benjamin Bustos. On nonmetric similarity search problems in complex domains. *ACM Computing Surveys (CSUR)*, 43(4):34, 2011.

[42] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal Processing Magazine*, 25(2):128–131, 2008. doi: 10.1109/MSP.2007.914237.

[43] Temple F Smith and Michael S Waterman. Comparison of biosequences. *Advances in Applied Mathematics*, 2(4):482 – 489, 1981. ISSN 0196-8858. doi: https://doi.org/10.1016/0196-8858(81)90046-4. URL http://www.sciencedirect.com/science/article/pii/0196885881900464.

[44] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981. ISSN 0022-2836. doi: https://doi.org/10.1016/0022-2836(81)90087-5. URL http://www.sciencedirect.com/science/article/pii/0022283681900875.

[45] Brad Solomon and Carl Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *bioRxiv 086561*, 2016. doi: 10.1101/086561. URL https://www.biorxiv.org/content/early/2016/12/02/086561.

[46] Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, 34(3):300–302, Mar 2016. ISSN 1087-0156. URL http://dx.doi.org/10.1038/nbt.3442.

[47] Stanley Y. W. SU. *Database Computers: Principles, Architectures, and Techniques*. New York: McGraw-Hill, 1988.

[48] Chen Sun, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. AllSome Sequence Bloom Trees. *bioRxiv 090464*, 2017. doi: 10.1101/090464. URL https://www.biorxiv.org/content/early/2017/03/23/090464.

[49] Baris E. Suzek, Yuqi Wang, Hongzhan Huang, Peter B. McGarvey, and Cathy H. Wu. UniRef clusters: A comprehensive and scalable alternative for improving sequence similarity searches. *Bioinformatics*, 31(6):926–932, 2015. doi: 10.1093/bioinformatics/btu739. URL +http://dx.doi.org/10.1093/bioinformatics/btu739.

[50] Terence Tao. Product set estimates for non-commutative groups. *Combinatorica*, 28 (5):547–594, Sep 2008. ISSN 1439-6912. doi: 10.1007/s00493-008-2271-7. URL https://doi.org/10.1007/s00493-008-2271-7.

[51] The Gene Ontology Consortium. Expansion of the Gene Ontology knowledgebase and resources. *Nucleic Acids Research*, 45(D1):D331–D338, 2017. doi: 10.1093/nar/gkw1108. URL +http://dx.doi.org/10.1093/nar/gkw1108.

[52] The UniProt Consortium. UniProt: The Universal Protein Knowledgebase. *Nucleic Acids Research*, 45(D1):D158–D169, 2017. doi: 10.1093/nar/gkw1099. URL http://dx.doi.org/10.1093/nar/gkw1099.

[53] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[54] Eric W. Weisstein. *Norm*. MathWorld–A Wolfram Web Resource, . http://mathworld.wolfram.com/Norm.html.

[55] Eric W. Weisstein. *Metric*. MathWorld–A Wolfram Web Resource, . http://mathworld.wolfram.com/Metric.html.

[56] Eric W. Weisstein. *Kolmogorov Entropy*. MathWorld–A Wolfram Web Resource, . http://mathworld.wolfram.com/KolmogorovEntropy.html.

[57] Derrick E. Wood and Steven L. Salzberg. Kraken: Ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, Mar 2014. ISSN 1474-760X. doi: 10.1186/gb-2014-15-3-r46. URL https://doi.org/10.1186/gb-2014-15-3-r46.

[58] Y William Yu and Griffin Weber. HyperMinHash: Jaccard index sketching in loglog space. *arXiv preprint arXiv:1710.08436*, 2017.

[59] Y William Yu, Noah M Daniels, David Christian Danko, and Bonnie Berger. Entropy-scaling search of massive biological data. *Cell Systems*, 1(2):130–140, 2015.