5-2018

Application of Cosine Similarity in Bioinformatics

Srikanth Maturu
University of Nebraska-Lincoln, srikanthmaturu@huskers.unl.edu

APPLICATION OF COSINE SIMILARITY IN BIOINFORMATICS

by

Srikanth Maturu

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Jitender Deogun

Lincoln, Nebraska

August, 2018

APPLICATION OF COSINE SIMILARITY IN BIOINFORMATICS

Srikanth Maturu, M.S.

University of Nebraska, 2018

Adviser: Jitender Deogun

Finding similar sequences to an input query sequence (DNA or proteins) from a sequence data set is an important problem in bioinformatics. It provides researchers an intuition of what could be related or how the search space can be reduced for further tasks. An exact brute-force nearest-neighbor algorithm used for this task has complexity $\mathcal{O}(m * n)$ where n is the database size and m is the query size. Such an algorithm faces time-complexity issues as the database and query sizes increase. Furthermore, the use of alignment-based similarity measures such as minimum edit distance adds an additional complexity to the exact algorithm.

In this thesis, an alignment-free method based similarity measures such as cosine similarity and squared euclidean distance by representing sequences as vectors was investigated. The cosine-similarity based locality-sensitive hashing technique was used to reduce the number of pairwise comparisons while finding similar sequences to an input query. We evaluated our algorithm on a proteins dataset of size 100,000 sequences and found that our cosine-similarity based algorithm is 28 times faster than the exact algorithm and 13 times faster than the BLASTP[3] algorithm for finding similar sequences with percent identity greater than 90%. It also has 99.5% accuracy. We also developed a greedy incremental clustering algorithm based on our cosine-similarity nearest neighbor algorithm for removing redundant sequences in a protein dataset. We compared our clustering algorithm with a popular clustering algorithm CD-HIT. The clustering results on protein dataset of size 100000 show

that our clustering algorithm generated clusters with accuracy almost equal to the CD-HIT algorithm accuracy.

We further demonstrated two bioinformatics application where our cosine-similarity based algorithm can be used: an analysis of assembly data of various assemblers and a clustering of a protein dataset. Using our algorithm, we successfully compared the quality of assembly data of multiple *de novo* and genome-guided assemblers.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my adviser Dr. Jitender Deogun for his constant guidance and support throughout my masters program. I would like to thank him for his kindness and motivation.

I would like to thank the rest of my thesis committee members, Dr. Etsuko Moriyama and Dr. Ashok Samal for their time and valuable guidance during my thesis review. I would also like to thank Sairam Behera for his guidance and help in my research.

Lastly, I would like to thank my friends and family members for their help and support.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Biological sequences which code genes, RNA, and proteins are nothing but the succession of letters from their corresponding Σ, where Σ is the set of symbols or letters. For DNA and RNA sequences $|\Sigma| = 4$ and for protein sequences $|\Sigma| = 20$. A subsequence of length k is called a k-tuple or a k-word. The total number of unique k-tuples that are possible in a sequence depends on both k and Σ, and is exactly equal to $|\Sigma|^k$. For a given k, a sequence can be represented as a n-dimensional vector of these k-tuple frequencies where n, the number of dimensions, equals to $|\Sigma|^k$. For a given set of similar sequences, their corresponding k-tuple frequency vectors tend to close to each other in the n-dimensional vector space. Similarity measures such as cosine-similarity and euclidean distance are applicable for k-tuple frequency vectors in the n-dimensional vector space. Finding nearest k-tuple frequency vectors to an input query k-tuple frequency vector is a nearest-neighbor problem.

There are many nearest-neighbor algorithms that exist in the current literature for n-dimensional vector spaces [6, 17, 23]. However, most of them are limited to smaller dimensions. Biological sequences face the problem of high-dimensional space which is also known as the curse of dimensionality [5]. For an instance of nucleotide sequences

with k values 3, 4, and 5, corresponding k-tuple frequency vector sizes are 64, 512, and 1024. Locality-sensitive hashing techniques have been known for reducing the dimensionality of high-dimensional data. Locality-sensitive hashing techniques use randomized algorithms to reduce the dimensionality [18]. Due to the randomization, an error term will be introduced in the output results, thereby provide approximate solutions rather than exact solutions. Approximate nearest-neighbor algorithms are faster when compared with exact nearest neighbor algorithms as they reduce the dimensionality of biological sequences. Approximate nearest-neighbor algorithms based on locality-sensitive hashing are therefore more appropriate for biological sequences.

Cross-polytope and hyperplane are two locality-sensitive hashing techniques used for cosine-similarity. In both techniques, the n-dimensional space is partitioned across the center and each partition is represented as a bucket. A group of n-dimensional vectors within a partition is hashed into the same bucket. When a query vector is given, it is hashed into a bucket and all vectors that were hashed into that bucket are taken out as candidates for nearest neighbors to the query vector. There is a chance that nearest neighbors are located at boundaries of the partition, in such case candidates from neighboring partitions or buckets are also considered.

In this thesis, we have developed a fast nearest-neighbor search algorithm for biological sequences such as nucleotide sequences (DNA, RNA) and amino acid sequences (proteins) using the cosine-similarity based locality-sensitive hashing technique and then demonstrated two bioinformatics applications of our cosine-similarity based nearest-neighbor algorithm.

The first application is to analyze transcriptome sequences assembled using multiple methods. Many assemblers are available today to assemble large amounts of long and short reads data generated by high-throughput sequencing. The *de novo* assemblers do not use a reference genome to assemble short reads into contigs whereas

the genome-guided assemblers use a reference genome. Different assemblers produce different assembly outputs with different accuracies. Given a set of true positive sequences and assembled contig sequences of an assembler, our cosine-similarity based nearest-neighbor algorithm can be used to determine the accuracy of the assembler output comparing with the true positives for a given input minimum percent identity threshold. For example, for a given input minimum percent identity threshold $th = 70$, we can determine the number of sequences in an assembly output that are similar to the true positives with at least 70 percent identity. Also we can determine the number of true positives that are present in the assembly output with at least 70 percent identity. It can also be used to analyze an ensemble assembly approach where more than one assembler results are combined.

The second application is clustering of amino acid sequences. We developed a greedy incremental clustering algorithm based on our nearest-neighbor algorithm that can be used to reduce the redundancy in the sequences by clustering similar sequences into one cluster. Our clustering algorithm uses an agglomerative approach for clustering amino acid sequences. Initially, amino acid sequences to be clustered are sorted in decreasing length order. The topmost sequence is chosen as the new cluster representative and nearest sequences from remaining amino acid sequences according to a certain input percent identity threshold are removed to form the new cluster. Cluster representatives of all clusters are outputted as a set of non-redundant sequences. Thus the number of output sequences equals to the number of clusters formed. In this thesis, we demonstrated the application of our clustering algorithm by performing clustering on combined assembly data from multiple assemblers. We extracted the cluster representatives and analyzed them by comparing with the true positives. Our results showed that the use the cluster representatives have an accuracy slightly lower than the accuracy of combined assembly data.

1.1 Thesis Outline

Chapter 1 introduces the similarity search problem, outline of this report and contributions of this research. Chapter 2 defines the nearest-neighbor search problem, related algorithms for the nearest-neighbor search problem, introduces the cosine-similarity based locality-sensitive hashing technique for the approximate nearest-neighbor problem, and clustering in bioinformatics. Chapter 3 consists of the problem statement for this thesis, overview of the methodology chapter, defines the similarity measures for biological sequence similarity measurements and vector representation of biological sequences, describes two nearest-neighbor algorithms: one based on brute-force method and the other based on cosine-similarity based locality-sensitive hashing, describes a greedy incremental clustering algorithm and concludes with implementation details of the algorithms. Chapter 4 describes the datasets used for the evaluation of the algorithms, evaluates the alignment-free similarity measures, evaluates the cosine-similarity based approximate nearest-neighbor algorithm and the clustering algorithm, and demonstrated the assembly data analysis application. Chapter 5 summarize, the conclusions and discuss possible directions of future work.

1.2 Thesis Contribution

The following is our contributions in this research:

- We developed a fast approximate similarity search algorithm for biological sequences based on a cosine-similarity locality-sensitive hashing technique. Our search algorithm uses the alignment-free similarity measures to search approxi-

mate nearest neighbors to the input queries. We evaluated our algorithm on the SWISS-PROT protein dataset of size 100,000 sequences. The results demonstrated that our algorithm is 28 times faster than the exact algorithm and 13 times faster than the BLASTP [3] algorithm for finding similar sequences with percent identity greater than 90%, it also has 99.5% accuracy.

- We also developed a greedy incremental clustering algorithm using our fast approximate similarity search algorithm for removing redundant sequences from an input protein database. We evaluated our clustering algorithm on the SWISS-PROT protein dataset of size 100,000 sequences and compared with the CD-HIT [14] algorithm. The clustering results show that our clustering algorithm generated clusters with accuracy almost equal to the CD-HIT algorithm accuracy.

- We also reviewed the correlation between alignment-free similarity measurements and alignment-based similarity measurements of protein sequences. We used three statistical correlation approaches for our correlation analysis. The results demonstrated that these types of similarity measurements correlate highly for percent identities greater than 50.

- We demonstrated two bioinformatics applications of our cosine-similarity based approximate similarity search algorithm and clustering algorithm:

  - The first application is to analyze assembly sequences by comparing with true positive sequences. For a given minimum percent identity threshold *th*, our algorithm can be used to determine the number of true positives that are similar to the assembly sequences with at least percent identity

th and also the number of sequences in the assembly that are similar to the true positives with at least percent identity *th*. We demonstrated this application by analyzing multiple assemblies data of RNA sequencing data. The assemblers accuracy was compared and also the accuracy of combined assembly data of multiple assemblers was analyzed.

– The second application is to cluster combined assembly data of multiple assemblers to remove redundant sequences.

Chapter 2

BACKGROUND & RELATED WORKS

2.1 Nearest-Neighbor Search

The nearest-neighbor search is a similarity problem of finding the closest neighbors for a given object or instance.

Definition

Given a set of n instances $P = \{p_0, p_1, ..., p_{n-1}\}$ in some metric space X, the nearest-neighbor search algorithm finds the nearest neighbor $p' \in P$ to the given query instance q under some similarity measurement function.

The k-nearest neighbors problem:

Definition

Given a set of n instances $P = \{p_0, p_1, ..., p_{n-1}\}$ in some metric space X, the k-nearest neighbors search algorithm finds the k-nearest neighbors $\{p'_0, p'_1, ..., p'_{k-1}\}$, $p'_i \in P$ for the given query instance q under some similarity measurement function.

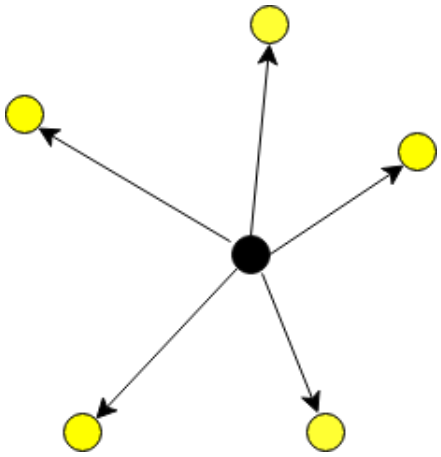

Figure 2.1: 5-nearest neighbors to an object (black circle)

The exact nearest neighbor algorithm is a brute-force algorithm and its time complexity is $\mathcal{O}(m*n*d)$ where m is the number of instances, n is the number of queries and d is the number of dimensions. This algorithm works well for lower dimensions but becomes slow for higher dimensions. This problem of higher dimensions is known as the curse of dimensionality[5] and to solve that these approximate nearest-neighbor algorithms have been introduced[11].

2.2 Approximate Nearest-Neighbor Search

The motivation towards approximate nearest-neighbor algorithms is to reduce the time complexity $\mathcal{O}(m*n*d)$ of the exact nearest-neighbor algorithm. An error term is introduced in order to reduce the time-complexity.

Definition

Given a set of n instances $P = \{p_0, p_1, ..., p_{\text{n-1}}\}$ in some metric space M, find a point $p \in P$ where p is an ϵ-nearest neighbor of the query instance q that $\forall p' \in P, d(p,q) \leq (1+\epsilon)\, d(p',q)$.

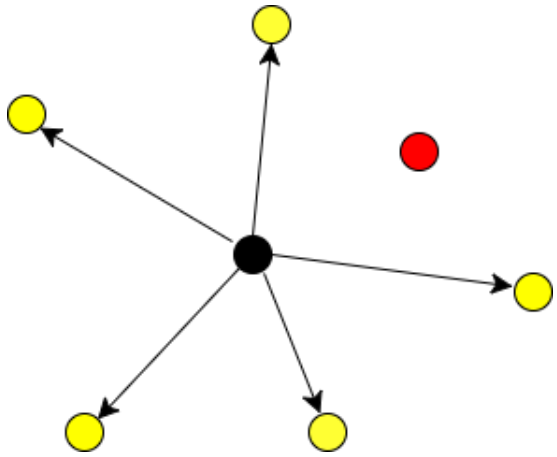

Figure 2.2: 5-approximate nearest neighbors to an object (black circle)

2.2.1 Kd-tree

Kd-tree is a binary tree and stores instances from a k-dimensional space. Kd-trees were first introduced by Bentley in 1975 [6, 12] and first used as a base for the nearest-neighbor search by Friedman [9]. For a d-dimensional set of instances D, the kd-tree construction is started by choosing a dimension from d dimensions and splitting D in to two partitions based on the median of that dimension. All the instances that are less than the median are placed in the left partition and all that are greater than or equal to the median are placed in the right partition. Now for each partition, a kd-tree is further constructed using the remaining *d-1* dimensions recursively. Each leaf node of the kd-tree stores a set of instances. For a given query q, the kd-tree is traversed from the root down to a leaf node by comparing values of the query with the median at each split corresponding to a dimension. All the instances in the leaf node are returned as the nearest neighbors to the query. This method is an approximate nearest-neighbor search method because it can miss some instance when the instance is placed in to another partition.

2.2.2 Ball tree

The approximate nearest-neighbor search using ball trees works similar to kd-trees [17]. For a d-dimensional set of instances D, instead of splitting the points based on some median they are split by computing distances to two centroids. Initially, two centroids are chosen, for every instance distances to the two centroids are computed and is assigned to the cluster with the smaller distance forming two clusters. If the distances are equal then the instance is assigned to the cluster chosen randomly. In turn for each cluster two more centroids are chosen and split again. This is continued recursively until each cluster is containing specified number of points or the number of clusters allowed has reached. For a given query q, recursively computed to find a cluster to which it belongs and all the instances in that cluster are returned as the nearest neighbors to the query. Again, some instance can miss when the instance is placed into another cluster. So this method using ball tree is an approximate nearest-neighbor method.

2.2.3 RPForest

In RPForest [2] (Random projection forest) multiple random projection trees are used. It is a variant of kd-tree. In each random projection tree the given set of instances D are recursively split in to subsets until the number of instances at leaf nodes are at most m. The instances are partitioned based on cosine of angle value of an instance to a randomly chosen hyperplane. The median of cosine of angle values of all instances is computed. Instances with cosine of angle value to the random hyperplane less than or equal to the median goes to the left partition and greater than the median goes to the right partition. For a given query q, every tree in the forest is recursively

traversed down to a leaf node by calculating the cosine of angle values to each random hyperplane. All the instances from all leaf nodes are collected, duplicates are removed and the remaining instances are returned as the nearest neighbors to the given query q.

2.3 Locality-Sensitive Hashing

In locality-sensitive hashing, items are hashed and mapped to the same buckets with high probability when the items are similar and with low probability when the items are dissimilar. Two vectors can be termed as similar if their cosine distance is below a certain threshold and as dissimilar if their cosine distance is above the threshold. Locality-sensitive hashing technique based on cosine distance can be applied to the nearest neighbors search of n-dimensional vectors.

2.3.1 Cosine-similarity

Cosine-similarity is a similarity measure between two vectors that measures the cosine of the angle between them. The cosine similarity value ranges between [0, 1]. It can be derived by computing dot product and magnitudes of two vectors. For a given two vectors a and b:

$$cosine - similarity = cos(\theta) = \frac{\vec{a}.\vec{b}}{\|\vec{a}\|.\|\vec{b}\|} \tag{2.3.1}$$

$$cosine - distance = 1 - cos(\theta) \tag{2.3.2}$$

Two vectors tend to close to each other when their cosine-similarity value is close to 1 and away from each other when their cosine-similarity value is close to 0.

2.3.2 LSH for cosine-distance

In cosine-distance based locality-sensitive hashing [19], for a given set of two instances or vectors, they hash to the same bucket with high probability if they are similar and they hash to the same bucket with low probability when they are dissimilar. The probability of collision between two vectors with angle α between them is equal to $1-\frac{\alpha}{\pi}$. This type of hashing can be applied to approximate nearest-neighbor search of d-dimensional vectors. Given d-dimensional vectors D are hashed to buckets using the LSH. Later when a query vector q is given, it is hashed to a bucket and all the vectors in that bucket are returned as candidates for approximate nearest neighbors to q. The candidates returned may be further filtered by computing the actual cosine distance to the query vector q. Random hyperplanes and cross-polytopes are two types of LSH families based on cosine-similarity[4]. They both can be used for euclidean distance also. Both hyperplane LSH and cross-polytope LSH partition the d-dimensional unit sphere into random partitions with only difference in how granular these partitions are.

2.3.2.1 Hyperplane LSH

In hyperplane LSH [8] a sphere is partitioned in to two parts of equal sizes by sampling a random hyperplane through the center of the sphere. Actually this is achieved by sampling a d-dimensional vector r whose coordinates are i.i.d. standard guassians. For a given vector v, dot product $<r,\ v>$ is computed and sign of the result is used

as a hash of the vector v.

2.3.2.2 Cross-polytope LSH

Cross-polytope is a regular, convex polytope that exists in d-dimensions. Let e_1, ..., e_{2d} be the set of signed bias vectors, i.e., each e_i has exactly one nonzero coordinate that is either +1 or -1. The d-dimensional cross-polytope is the convex hull of these signed standard basis vectors. In 2 dimensions, the cross-polytope is a rotated square and in 3 dimensions, the cross-polytope is the octahedron. All points on the surface of the cross-polytope have l_1-norm 1 which is why the cross-polytope is also known as a l_1-unit ball.

In cross-polytope LSH[22], a random rotation S is sampled to compute hash of a point v. To hash the point v, the rotation S is applied to v and the nearest vertex of the cross-polytope to Sv is found to be its hash value. A cross-polytope hash function partitions the unit sphere corresponding to the Voronoi cells of the vertices of the randomly rotated cross-polytope.

2.4 Clustering

Clustering is grouping similar objects into one group. Clustering is based on two principles:

- Homogeneity: Objects in a cluster are maximally close to each other.
- Separation: Objects in different clusters are maximally far apart from each other.

There are three types of clustering techniques:

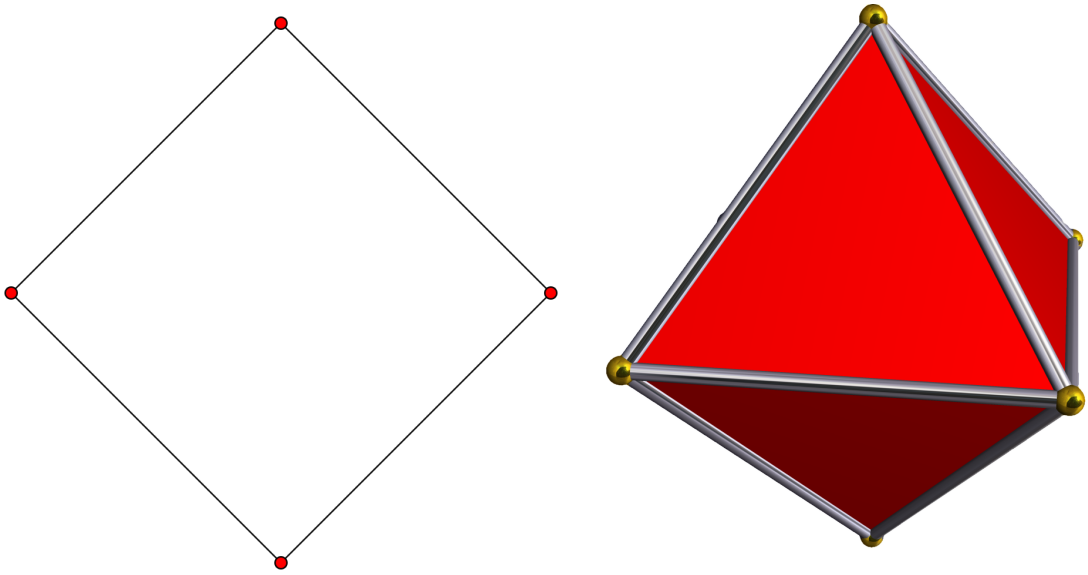

Figure 2.3: 2 dimension cross-polytope (left) & 3 dimensional cross-polytope (right) Source: [1]

- Agglomerative: Initially every object itself is a cluster. Clusters that are similar are joined into one cluster.
- Divisive: Initially all objects are in one big cluster. Then it is iteratively partitioned into smaller clusters.
- Hierarchical: Objects are organized in a tree structure. Leaves represent individual objects, length of the path between leaves represent similarity and similar objects are present in the same subtree.

In this thesis, we used agglomerative clustering to cluster similar biological sequences. One use case of such clustering is to remove redundancy in the sequences.

2.4.1 CD-HIT

CD-HIT is a popular program used for clustering protein or nucleotide sequences[14]. CD-HIT can be used to remove redundant sequences from a database through clustering. It implements a greedy incremental clustering algorithm. All the sequences in a database are clustered such that similar sequences are within a cluster. A cluster representative is chosen from each cluster and outputted as non-redundant sequences. CD-HIT uses heuristics that are short-word filter and banded alignment to approximate the percentage identity between two sequences instead of a dynamic programming algorithm. Although CD-HIT does pairwise comparisons, it is still fast due to the heuristics. In pairwise comparisons of a query to a database, the short-word filter is used initially to filter out the sequences that are possibly not similar to the query at given input percent identity threshold and then the banded alignment algorithm is used to approximate the percent identity between remaining sequences.

2.5 Sequence Similarity Tools

2.5.1 BLAST

The Basic Local Alignment Search Tool (BLAST) [3][21] finds regions of local similarity between biological sequences. It compares nucleotide or protein sequences to sequences databases and computes the statistical significance of the matches. It can also be used for motif searches, gene identification searches, and also for the analysis of multiple regions of matches in long DNA sequences. It produces results quickly by using heuristics. It also calculates an "expect" value that estimates how many matches would have occurred by chance, which can aid a user in deciding how much confidence to have in an alignment.

Chapter 3

METHODOLOGY

3.1 Problem Definition

For a given input nucleotide/protein query sequence, performing a fast approximate query search over a large collection of nucleotide/protein database sequences for finding the similar nucleotide/protein sequences with at least input percent identity threshold *th* using a cosine similarity based locality-sensitive hashing technique. Then compare search performance in terms of both speed and accuracy with pair-wise similarity search algorithms, the brute-force algorithm and the BLASTP[3] algorithm.

3.2 Overview

We first define alignment based and alignment-free similarity measures for biological sequences. Alignment based similarity measures are minimum edit distance and percent identity. Alignment-free similarity measures are cosine-similarity and squared euclidean distance. Sequences will be represented as vectors to compute the align-

ment free-similarity measurements. We then describe two nearest neighbors search algorithms for biological sequences, one using brute-force method and the other using the cosine-similarity based locality sensitive hashing technique. We then define greedy incremental clustering algorithm based on our cosine-similarity approximate nearest neighbors algorithm. We conclude with implementation details of our algorithms.

3.3 Similarity Measure

For similarity measurement between biological sequences such as DNA, RNA or proteins we used both alignment-based and alignment-free similarity measures. The following are the similarity measures:

3.3.1 Minimum edit distance

Minimum edit distance measures the minimum number of edit operations required to transform one sequence to another sequence. Minimum edit distance is also known as levenshtein distance. They are three basic types of edit operations that are involved in transforming a start sequence to the final sequence. They are insertion when a character is inserted in to the start sequence, deletion when a character is removed from the start sequence and substitution when a character is substituted in the start sequence. The Needleman–Wunsch algorithm is an algorithm to compute global alignment between two biological sequences. It uses the dynamic programming approach and can be used to compute minimum edit distance. The algorithm was developed by Saul B. Needleman and Christian D. Wunsch.

3.3.2 Percent Identity

Percent identity of two sequences is calculated by globally aligning the two sequences. Let l be the alignment length of the two sequences including gaps after global alignment and m be the number of matches between the two sequences in the global alignment. The formula for percent identity of the two sequences is

$$PercentIdentity = \frac{m * 100}{l} \tag{3.3.1}$$

3.3.3 Alignment-free Measure

We used an alignment-free method based similarity measurement to measure similarity among nucleotide/protein sequences. We choose a short word length k and map each sequence onto an n-dimensional vector according to its k-length tuple (also called k-tuple or k-word) frequency. In the Section 3.4, we discuss an algorithm for generating k-tuple frequency vector for a given input sequence. Let us consider two sequences S_1 & S_2 and let V_1 & V_2 be their n-dimensional k-tuple frequency vectors. Now we define our two similarity measures for k-tuple frequency vectors.

3.3.3.1 Cosine-Similarity

Cosine-similarity between two given k-tuple frequency vectors is the cosine of the angle between those two vectors.

$$Cosine - Similarity(V_1, V_2) = \frac{\vec{V_1}.\vec{V_2}}{\|\vec{V_1}\|\vec{V_2}\|} \tag{3.3.2}$$

3.3.3.2 Squared Euclidean Distance

Squared euclidean distance between two given k-tuple frequency vectors is square of the euclidean distance between those two vectors.

$$SquaredEuclideanDistance(V_1, V_2) = \sum_{i=0}^{n-1}(V_1[i] - V_2[i])^2 \tag{3.3.3}$$

3.4 K-tuple Frequency Vector

Let Σ be a set of unique letters, where a sequence S of length l over Σ is a succession of l letters $s_0 s_1 ... s_i ... s_{l-1}$ & $s_i \in \Sigma$. For a given sequence S of length l, k-tuples or k-words are obtained by sliding a window of size k from the beginning to the end of sequence. The total size of k-tuples will be l - k + 1 and may contain k-tuples with frequency greater than one. The number of unique k-tuples depend on both k and the size of Σ and exactly equal to $|\Sigma|^k$. Each element in the n-dimensional vector corresponds to an unique k-tuple and stores the k-tuple frequency occurred in the sequence. There can be zero value elements in the vector if their corresponding k-tuple frequencies in the sequence are zero.

For nucleotide sequences $\Sigma = \{A, C, G, T\}$, therefore for k-values 3, 4 and 5 corresponding k-tuple frequency vector sizes are 64, 512 and 1024 respectively. For protein sequences $\Sigma = \{A, R, N, D, C, E, Q, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$, therefore for k-values 3, 4 and 5 corresponding k-tuple frequency vector sizes are 8000, 160,000 and 3,200,000 respectively. We can observe that as the k value increases the vector size is increasing exponentially. However, when the sequence length l is fixed, the k-tuple frequency vector becomes sparser as the k value increases i.e the total

Algorithm 1 COMPUTE-K-TUPLE-POSITION(w, Σ)

INPUT: k-tuple W = $w_0 w_1 ... w_{k-1}$ of length k and Σ where $w_i \in \Sigma$.
//Consider Σ as an array of letters that it contains.
OUTPUT: The corresponding position pos of k-tuple in the k-tuple frequency vector where $0 \leq pos < |\Sigma|^k$.

```
1: k := w.length
2: pos := 0
3: for i = 0 : k - 1 do
4:     w_pos ← position of w_i in Σ
5:     pos ← pos + w_pos * |Σ|^i
6: end for
```

Algorithm 2 COMPUTE-K-TUPLE-FREQUENCY-VECTOR(S, k, Σ)

INPUT: Input sequence **S** = $s_0 s_1 ... s_{l-1}$ of length l, k and Σ where $s_i \in \Sigma$.
OUTPUT: k-tuple frequency vector V where $|V| = |\Sigma|^k$.

```
1: v_size ← |Σ|^k
2: V ← zero-vector Z of size v_size
3: for i = 0 : l - k + 1 do
4:     W ← s_i s_{i+1} ... s_{i+k-1}
5:     pos ← COMPUTE-K-TUPLE-POSITION(W, Σ)
6:     V[pos] ← V[pos] + 1
7: end for
```

number of zero elements in the vector increases. In this thesis, we have used k = 3 for protein sequences, this choice is explained in detail in the section 4.2. We used the algorithm COMPUTE-K-TUPLE-FREQUENCY-VECTOR to generate k-tuple frequency vector for a given input sequence S and Σ.

3.5 Approximate Nearest-Neighbor Search

In this thesis, we developed two algorithms to perform the nearest-neighbor search on biological sequences. Both algorithms accept DNA, RNA or protein sequences as inputs for both database and queries. The first algorithm is an exact nearest-neighbor algorithm that uses the brute-force technique, i.e. every query sequence is compared

Algorithm 3 BRUTE-FORCE-NEAREST-NEIGHBORS(D, Q, th)

INPUT: Database sequences D = $\{S_0, S_1, ..., S_{n-1}\}$ of size n, query sequence Q & percent identity threshold th.

OUTPUT: **R** = $\{S'_0, S'_1, ..., S'_{m-1}\}$ $|\mathrm{R}| \geq 0$ and $S'_i \in D$.

```
1: R ← {}
2: for i = 0 : n - 1 do
3:    Calculate percent identity between sequences Q and S_i using the Needleman–Wunsch algorithm.
4:    percent_identity ← PERCENT-IDENTITY-NW(Q, S_i)
5:    if percent_identity ≥ th then
6:        R ← R ∪ S_i
7: end for
```

to all the sequences in a database. Second algorithm is an approximate nearest-neighbor algorithm that uses the cosine-similarity based locality-sensitive hashing technique to retrieve the candidate sequences that are similar to the input query and further filter them by pairwise comparisons with the query. For pairwise comparisons, we use the Needleman–Wunsch algorithm to global align two biological sequences and then compute the percent identity between them. The objectives for both algorithms is the same that for a given query q, return similar sequences in the database according to the input minimum percent identity threshold. Let D be the set of database sequences, Q be the set of query sequences and the percent identity threshold th.

3.5.1 Brute force method

In brute force method, the inputs database and query sequences are considered as it is. A percent identity threshold th is also given as input to return the similar sequences in the database to a query with percent identity greater than or equal to the threshold th. We used the algorithm 3 for brute force based nearest neighbors search.

3.5.2 Cosine similarity based locality-sensitive hashing method

Nearest neighbor search using the cosine similarity based locality-sensitive hashing method works in the following two phases:

1. Index construction
2. Query processing

3.5.2.1 Index Construction

An index I is constructed for the given input D = $\{S_0, S_1, ..., S_{n-1}\}$ database sequences which later will be used in the query processing phase. Initially, all the database sequences are converted to k-tuple frequency vectors P = $\{V_0, V_1, ..., V_{n-1}\}$. Index construction requires two parameters: number of hash tables l and number of hash functions per table f. The index I consists of multiple hash tables where all vectors in P are stored in buckets in each of the hash tables. Each hash table has a unique hash function that hashes a k-tuple frequency vector and that hash value is used as the key to store the vector in the corresponding bucket. All the vectors that shares the same hash value will be stored in the same bucket in a hash table. A hash function splits the unit sphere in to a fixed number of random partitions based on the type of LSH used, where each partition of the sphere corresponds to a bucket in the hash table. So, all the vectors that are in the same partition are hashed in to the same bucket. Two types of LSH for cosine similarity are hyperplane LSH and cross-polytope LSH.

Hyperplane LSH

In hyperplane LSH the unit sphere is partitioned by sampling a random hyperplane through the center of the sphere. If the number of random hyperplanes sampled is K then the number of partitions in the space created is equal to 2^K.

Cross-polytope LSH

Given a n-dimensional unit sphere, the number of vertices in the n-dimensional cross-polytope is equal to $2n$. In cross-polytope LSH the unit sphere is randomly partitioned into voronoi cells each corresponding to a vertex of the cross-polytope. Therefore the number of partitions created in the n-dimensional sphere is equal to $2n$. Cross-polytope LSH is implemented by sampling a random cross-polytope. To achieve that a random rotation R is sampled and applied to the cross-polytope. In order to compute the hash value of an input vector V, the rotation S is applied to the vector and the nearest vertex of the cross-polytope to the Rv is returned as the vector V hash value.

For given input database sequences D = $\{S_0, S_1, ..., S_{n-1}\}$, window size k, Σ and number of hash tables l, the index construction using cross-polytope LSH is done through the algorithm4.

3.5.2.2 Query Processing

In the query processing phase, for an input query sequence q the set of similar sequences in the database D is returned. Initially, the query is converted to k-tuple frequency vector, then the vector is used to compute hash values corresponding to all hash tables. Those hash values act as keys to the hash tables and the corresponding buckets are retrieved. All buckets from all the hash tables are combined and any duplicates found are removed. The resulting set of values after removing duplicates will be indices of the candidate sequences. The candidates are further filtered in two filtering phases using input squared euclidean distance d and percent identity th

Algorithm 4 CONSTRUCT-LSH-INDEX(D, k, Σ ,l)

INPUT: Database sequences D = $\{S_0, S_1, ..., S_{n-1}\}$ of size n, window size k, Σ & number of hash tables l.

OUTPUT: Index I constructed for D using cross-polytope LSH

```
P ← {}
Iterate through all sequences in D and compute k-tuple frequency vectors for each sequence S_i ∈ D. Store the computed vectors in P such that V_i ∈ P corresponds to S_i ∈ D.
for i = 0 : n - 1 do
    V_i ← COMPUTE-K-TUPLE-FREQUENCY-VECTOR(S_i, k, Σ )
    P ← P ∪ V_i
end for
Initialize l empty hash tables
T = { L_0, L_1, ..., L_{l-1}}
Initialize f hash functions each corresponding to a hash table. Each hash function is initialized by sampling a random rotation R.
H = { F_0, F_1, ..., F_{l-1}}
Iterate through all k-tuple frequency vectors in P, for every V_i ∈ P compute hash values for all hash tables and store the value i in every hash table using the hash values corresponding to the hash tables as the keys.
for i = 0 : n - 1 do
    for j = 0 : l - 1 do
        compute hash value of V_i using the hash function F_j and store i in the hash table L_j in a bucket using the hash value as the key.
        hash ← F_j(V_i)
        L_j(hash) ← L_j(hash) ∪ i
end for
```

thresholds.

- The first filtering phase by squared euclidean distance threshold
  - Squared euclidean distance between candidates to the query vector is computed. Only the candidates whose squared euclidean distance with the query is less than the input squared euclidean threshold d are considered for the second filtering phase.

- The second filtering phase by percent identity threshold
  - The input candidates are filtered by pairwise comparisons with the query sequence q using the Needleman–Wunsch algorithm. The candidates whose percent identity is at least the threshold th are returned as the similar sequences to the query q.

The algorithm 5 describes the query processing phase.

3.6 Clustering Algorithm

In this thesis, we developed a greedy incremental clustering algorithm for biological sequences based on our cosine-similarity nearest-neighbor search algorithm. Our algorithm support both nucleotide and protein sequences as input. One use case of our clustering algorithm is to remove redundancy in large dataset sequences through the clustering approach. A cluster representative or consensus sequence can be picked from a cluster. Therefore the number of output sequences is equal to the number of clusters. Within each cluster, all sequences must be similar to the cluster representative with percent identity greater than or equal to the input percent identity threshold. We initially sort the input sequences dataset in decreasing length order. We go through every sequence from top to bottom in decreasing length order. We pick the first sequence in the list, create the first cluster and set the first sequence as the representative of the first cluster. Now we search all the remaining sequences in the dataset to find the similar sequences that are similar to the first sequence with percent identity greater than or equal to the input percent identity threshold and put them in to the first cluster. Now we pick the next sequence in the list that is not

Algorithm 5 PROCESS-LSH-QUERY(q, k, Σ , I, D, d, th)

INPUT: Query sequence q, window size k, Σ, I, D, squared euclidean distance threshold d and minimum percent identity threshold th.

OUTPUT: R = $\{S'_0, S'_1, ..., S'_{m-1}\}$ $|\mathrm{R}| \geq 0$ and $S'_i \in D$.

$R' \leftarrow \{\}$

Let input LSH index I consists of T = { L_0, L_1, ..., L_{l-1}} hash tables & H = { F_0, F_1, ..., F_{l-1}} hash functions corresponding to the hash tables.

V $\leftarrow$ COMPUTE-K-TUPLE-FREQUENCY-VECTOR(q, k, Σ)

Iterate through all the hash tables and compute hash values of V for all the hash tables. Using the hash values as keys retrieve buckets from all the hash tables.

for $i = 0 : l - 1$ **do**

  compute hash value of V using the hash function F_i and retrieve the hash bucket from the hash table L_i using the hash value as the key.

  hash $\leftarrow F_\mathrm{i}(V)$

  $R' \leftarrow R' \cup L_\mathrm{i}(hash)$

end for

Remove duplicate indices from R'.

$R' \leftarrow remove\_duplicate\_indices(R')$

Perform first filtering phase

$R' \leftarrow filter\_by\_squared\_euclidean\_distance(R', V, d)$

m $\leftarrow |R'|$

R = {}

Perform second filtering phase. For every index $r'_i \in R'$ retrieve corresponding sequence in D, compute the percent identity with the query q and if the percent identity $\leq th$ store it in R.

for $i = 0 : m - 1$ **do**

  $S \leftarrow S_{r'_i} \in D$

  $percent\_identity \leftarrow$ PERCENT-IDENTITY-NW(Q, S)

  if $percent\_identity \geq th$ **then**

    R $\leftarrow$ R $\cup$ S

end for

Algorithm 6 GREEDY-INCREMENTAL-CLUSTERING(q, k, Σ , I, D, th)

INPUT: Input sequences D = $\{S_0, S_1, ..., S_{n-1}\}$ of size n, window size k, Σ, number of hash tables l & percent identity threshold th.

OUTPUT: Output clusters W = $\{C'_0, C'_1, ..., C'_{m-1}\}$ |Ci| ≥ 1 & |W| $\leq$ |D|.

1: $W \leftarrow \{\}$
2: Sort sequences in D in decreasing sequence length order. Let $D' = \{S'_0, S'_1, ..., S'_{n-1}\}$ be the set of sequences after sorting D.
3: I = CONSTRUCT-LSH-INDEX(D', k, Σ , l)
4: Construct LSH index I for the sorted sequences D' with l hash tables and window size k.
5: Let LSH index I consists of T = $\{ L_0, L_1, ..., L_{l-1}\}$ hash tables & H = $\{ F_0, F_1, ..., F_{l-1}\}$ hash functions corresponding to the hash tables.
6: Iterate through the sequences in D' from top to bottom in decreasing sequence length order.
7: **for** $i = 0 : n - 1$ **do**
8: If the sequence S'_i is not present in any cluster of W, make a new cluster and set the sequence S'_i as the cluster representative of the new cluster.
9: **if** S'_i is not in any cluster of W
10: C $\leftarrow \{S'_i\}$ & set S'_i as the cluster representative of C.
11: **else**
12: **continue;**
13: **end if**
14: Retrieve all sequences in D' that are similar to S'_i with at least percent identity threshold th.
15: R $\leftarrow$ PROCESS-LSH-QUERY(S'_i, k, Σ , I, D', th) and let R = $\{S''_0, S''_1, ..., S''_{p-1}\}$
16: **for** $j = 0 : p - 1$ **do**
17: if S''_j is not in any cluster of W
18: C $\leftarrow$ C $\cup \{S''_j\}$
19: **end for**
20: $W \leftarrow W \cup \{C\}$
21: **end for**

clustered and create a new cluster in the same method. If the cluster representative doesn't have similar sequences in the remaining list then the cluster formed contains only the cluster representative. The algorithm 6 describes our clustering technique.

3.7 Evaluation Metrics for Cluster Comparison

3.7.1 Average pairwise distance

For measuring accuracy of a cluster, we perform multiple-sequence alignment using the T-coffee algorithm [15], which is then used to compute pairwise distance matrix and obtained the average pairwise distance between any two sequences in the multiple-sequence alignment of the cluster.

We define pairwise distance as:

$$pairwise - distance = 1 - \frac{percent - identity}{100} \tag{3.7.1}$$

3.7.2 Maximum Average Jaccard Index

We used the maximum average Jaccard Index [13] to assess the performance of CSANN-Clust clusters compared against CD-HIT clusters. Given two sets of protein clusters, A and B from two clustering algorithms, the average maximum Jaccard Index of algorithm A against the algorithm B is given by,

$$S(A,B) = \frac{1}{|A|} \sum_{A1 \in A} max_{(B_1) \in B} \frac{|A_1 \cap B_1|}{A_1 \cup B_1}$$

3.8 Implementation.

All our algorithms are written in C++ for better performance. We used the OpenMP parallel programming library in our algorithms to get speedups on multi-core cpus. We used the following two libraries that underlie our algorithms

- Edlib library [16], is a fast edit distance library written in C++ by Martin ęoĺić and Mile ęikić. The library is packaged with Needleman–Wunsch algorithm and we used it to compute the percent identity between two sequences by obtaining global alignment path.

- FALCONN library, is an LSH library for cosine-similarity. The FALCONN is short for FAst Lookups of Cosine and Other Nearest Neighbors. It was developed and currently maintained by Ilya Razenshteyn and Ludwig Schmidt [20]. It supports both sparse and dense datasets as input. Although it is designed for cosine-similarity it can also be used for nearest-neighbor search under euclidean distance. We call this library in index construction and query processing phases.

Chapter 4

RESULTS

4.1 Datasets

In this thesis, we evaluated our cosine-similarity based nearest-neighbor search algorithm and our greedy incremental clustering algorithm on SWISS-PROT protein sequence dataset. SWISS-PROT dataset (0.4M sequences) was downloaded online from NCBI (ftp://ftp. ncbi.nih.gov/blast/db/FASTA/) on September 10, 2017. The datasets for our assembly data analysis are transcriptome protein sequences assembled by four *de novo* assemblers and two genome-guided assemblers. The *de novo* and genome-guided assemblers were run to assemble short reads generated from synthetic RNA sequences of a maize plant. The original synthetic RNA sequences and assembled contig sequences are nucleotide sequences. We used the GeneMarkS[7] tool to convert RNA sequences to protein sequences. We used the *de novo* and genome-guided assemblers assembly contig sequences as our queries datasets. The RNA based amino acid sequences of synthetic maize are our true positives and is used as the database sequences in our experiments. The table 4.1 shows the list of assembly datasets with dataset sizes and sequences length ranges in each dataset.

Table 4.1: Maize transcriptome assembly protein sequence datasets

		number of sequences	bp range
True positive sequences		117,610	$98bp$ - $5267bp$
denovo	idba	177,126	$98bp$ - $5375bp$
	soapDenovo	374,913	$98bp$ - $4743bp$
	SPADes	486,912	$91bp$ - $5375bp$
	Trinity	120,199	$98bp$ - $5231bp$
Genome-guided	Bayesembler	4,002	$98bp$ - $4927bp$
	Cufflinks	63,246	$98bp$ - $4108bp$

4.2 Evaluation of similarity measures

We represent sequences as k-tuple frequency vectors. Although input sequences are of varied length in base pairs, the length of the resulting k-tuple frequency vectors is constant and equal to $|\Sigma|^k$. We defined two similarity measures cosine-similarity and squared euclidean distance 3.3 for similarity measurement between the k-tuple frequency vectors. When finding similar sequences to an input query sequence, we require that the returned similar sequences percent identity similarities with the query sequence are greater than or equal to the input minimum percent identity threshold. To facilitate such requirement, we needed to enforce definite cosine-similarity or squared euclidean distance thresholds for k-tuple frequency vectors. Therefore it is essential to run correlation analysis between similarity measures of both string representation and k-tuple frequency vector representation of sequences. Our correlation study helped us in determining the appropriate thresholds for cosine-similarity and squared euclidean distance measures. As the sequences are varied in size, we choose the percent identity measure over minimum edit distance.

4.2.1 Experimental Design

Our goal is to determine the correlation between percent identity versus both cosine-similarity and squared euclidean distance among protein sequences. For that purpose, we random sampled 10,000 RNA protein sequences from trinity assembly sequences and used them as the query sequences. Next, we sampled 10,000 RNA protein sequences from the true positive sequences dataset and used them as the database sequences. We computed pairwise comparisons between the database sequences and query sequences. For each pair, we computed the percent similarity, the squared euclidean distance and the cosine-similarity between the two sequences. That resulted in 100,000,000 pairwise comparisons. Although it is a large number of pairwise comparisons, there were many duplicates that we removed before our correlation analysis. We conducted these experiments for only protein sequences with $k = 2$, 3, and 4.

We used box-plots to visualize the pair-wise comparisons and a mean curve is also included as a part of the box-plots. We computed two types of box-plots: one for cosine-similarity versus percent identity and other for squared euclidean distance versus percent identity. Our initial box-plot results for all k values suggested that pairwise comparisons with percent identity less than 50 don't correlate well, thereby we removed all the pairwise comparisons with percent identity less than 50 from the total samples. Such filtering of samples doesn't affect our outcomes as we are only interested in percent identities greater than 50. Next, we performed correlation analysis for remaining sample points using the following three correlation methods:

1. Pearson's product-moment correlation
2. Kendall's rank correlation tau
3. Spearman's rank correlation rho

The following two sections discuss the results of our correlation analysis experi-

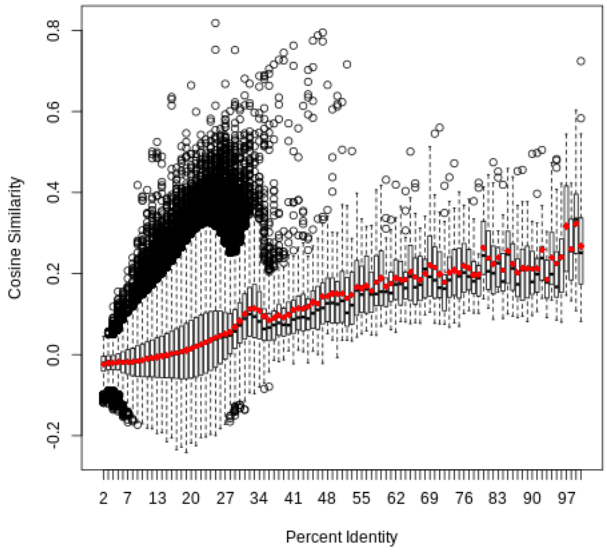

Figure 4.1: Cosine similarity versus percent identity ($k = 2$)

ments.

4.2.2 Cosine-similarity versus percent identity

Figures 4.1 4.2 4.3 shows the cosine-similarity versus percent identity box-plots. From the pairwise comparisons for a given percent identity, it is evident in the box-plot that the sequences have varied cosine-similarity values. It is also seen from the box-plot that there is a good correlation exist for percent identity greater than 50. The red line curve plots the means of cosine-similarity values for each percent identity. It is evident from the box-plots for percent identity > 50, that as the percent identity increase the cosine-similarity between the q-tuple frequency vectors of the sequences also increase in an almost linear manner.

Table 4.2 shows the results of correlation tests for $k = 2, 3$, and 4 that we conducted using the three popular methods. We did our correlation tests only for samples with

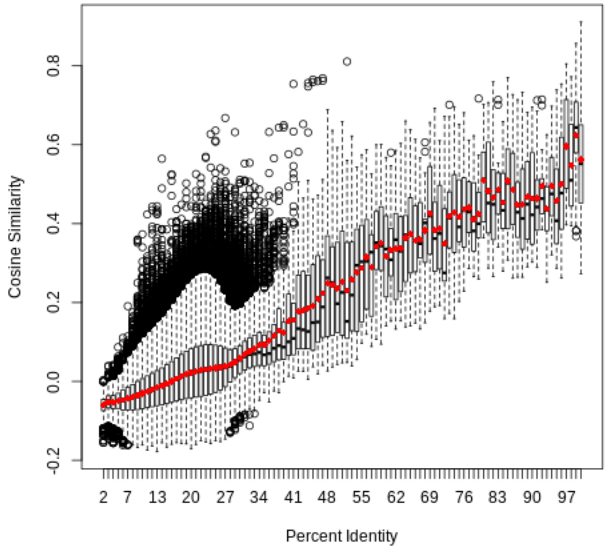

Figure 4.2: Cosine similarity versus percent identity ($k = 3$)

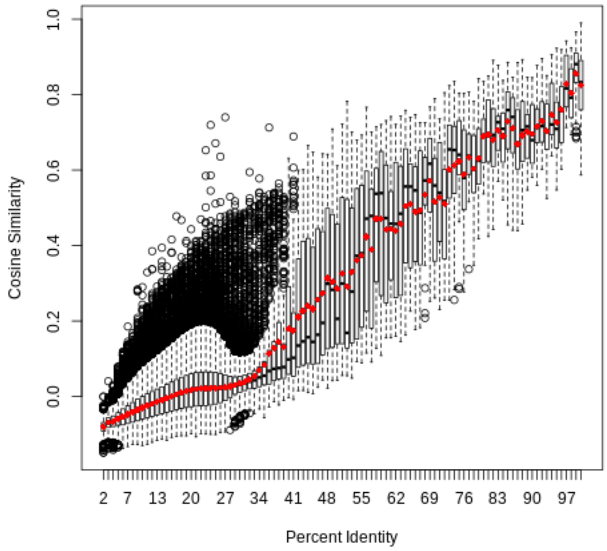

Figure 4.3: Cosine similarity versus percent identity ($k = 4$)

Table 4.2: Cosine-similarity versus percent identity correlation tests

Type of correlation test	k		
	2	3	4
degree of freedom	1821	1820	1822
p-value	$<$ 2.2e-16	$<$ 2.2e-16	$<$ 2.2e-16
Pearson's product-moment correlation	0.3832822	0.6104683	0.738707
Kendall's rank correlation tau	0.2790689	0.4285499	0.5726693
Spearman's rank correlation rho	0.3975879	0.5976841	0.7677395

percent identity greater than 50 as explained in the above.

4.2.3 Squared euclidean distance versus percent identity

Figures 4.4 4.5 4.6 shows the squared euclidean distance versus percent identity box-plots for $k = 2$, 3, and 4. From the pairwise comparisons for a given percent identity it is evident in the box-plot that the sequences have varied squared euclidean distance values. It is also seen from the box-plot that there is a good correlation exist for percent identity greater than 50. The red line curve plots the means of squared euclidean distance values for each percent identity. It is evident from the box-plot for percent identity > 50, that as the percent identity increase the squared euclidean distance between the q-tuple frequency vectors of the sequences decrease in an almost linear manner.

Table 4.3 shows the results of correlation tests for $k = 2$, 3, and 4 that we conducted using the three popular methods. We did our correlation tests only for samples with percent identity greater than 50 as explained in the above.

4.3 Approximate Nearest-Neighbor Algorithm

We evaluated our cosine-similarity based approximate nearest-neighbor algorithm on SWISS-PROT protein sequences. We refer our approximate nearest-neighbor algo-

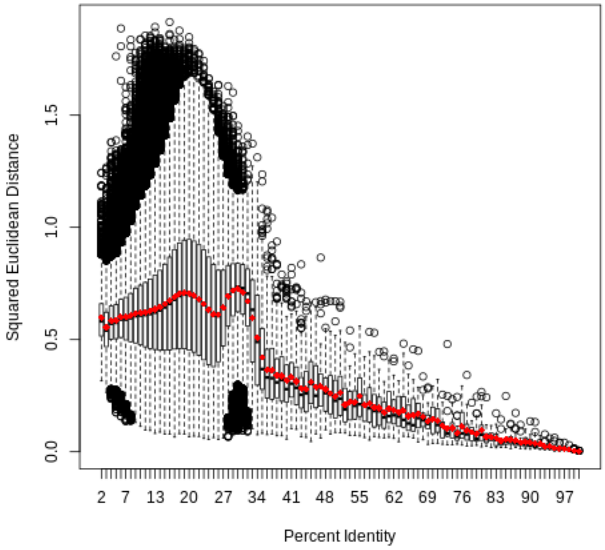

Figure 4.4: Square euclidean distance versus percent identity *(k = 2)*

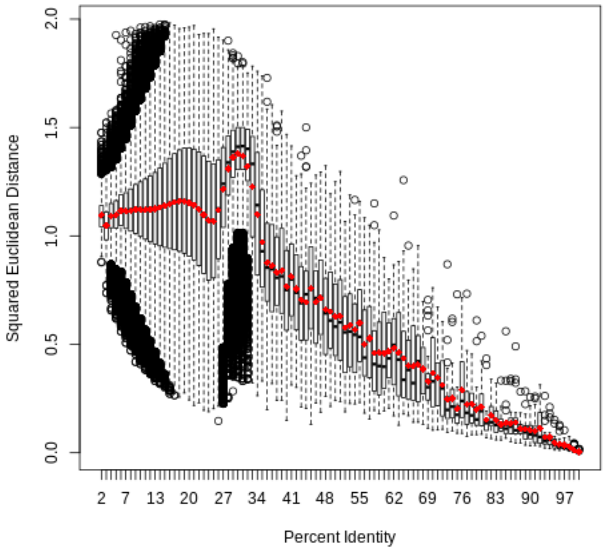

Figure 4.5: Square euclidean distance versus percent identity *(k = 3)*

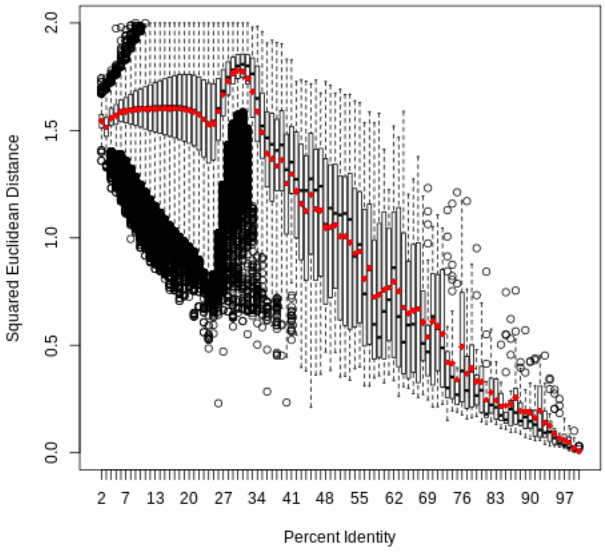

Figure 4.6: Square euclidean distance versus percent identity *(k = 4)*

Table 4.3: Cosine-similarity versus percent identity correlation tests

Type of correlation test	k		
	2	3	4
degree of freedom	1690	1688	1689
p-value	$<$ 2.2e-16	$<$ 2.2e-16	$<$ 2.2e-16
Pearson's product-moment correlation	-0.700298	-0.7463848	-0.749849
Kendall's rank correlation tau	-0.603641	-0.6263307	-0.6276174
Spearman's rank correlation rho	-0.7832556	-0.806429	-0.8054255

rithm as CSANN and our clustering algorithm as CSANN-Clust in the current results section. Our algorithm performance depends on the input parameters: number of hash tables l, squared euclidean distance threshold d and percent identity threshold *th*. When the number of hash tables is reduced, our algorithm performs faster because the number of tables that needed to be queried is reduced, thereby less query time. For higher percent identity thresholds *th*, we can reduce the number of hash tables thereby reduce the query time. Considering the box-plots analysis in section 4.2, we have performed parameter tuning and determined optimal parameter settings

Table 4.4: Optimal parameters setting

Parameter	Value		
k	3		
k-tuple vector size	8000		
LSH type	cross-polytope		
Number of buckets per hash table	16000		
Minimum percent identity threshold (th)	70	80	90
Number of hash tables (l)	32	16	16
Squared euclidean distance threshold (d)	0.7	0.6	0.5

PI 70, PI 80 and PI 90 for percent identity thresholds 70, 80 and 90 respectively. We applied these parameter settings in our experiments according to the percent identity threshold used. The table 4.4 describes the optimal parameter settings that we used in our experiments. We chose k = 3 as optimal parameter by considering the correlation tests that we conducted as presented in the tables 4.2 4.3. We found that higher k values have better correlations when compared with lower k values. However, higher k values increase the k-tuple vector size thereby increase memory usage. For k = 2, 3 and 4, the corresponding k-tuple vector sizes are 400, 8000, and 160,000 respectively. There is a trade-off between memory requirements and correlations for choosing k. We chose $k = 3$ as it has better correlation than $k = 2$ and has lower memory requirement than $k = 4$.

4.3.1 Accuracy

4.3.1.1 Definition

We compute percent identity using the Needleman–Wunsch algorithm. For a given query q and percent identity threshold th, we run the brute-force algorithm to obtain all the true positive sequences from the database db that are similar to the query sequence q i.e., percent identities greater than or equal to the input threshold th. Let

the true positive sequences count be *tp*. Now we obtain similar sequences using our CSANN algorithm and let that count be *c*. Now we define *c/tp* as accuracy of our CSANN algorithm w.r.t query *q*, database *db* and percent identity threshold *th*.

Let a given input query sequences set Q = $\{q_0,\ q_1,\ ...,\ q_{n-1}\}$ of size *n*, corresponding true positive counts be TP = $\{tp_0,\ tp_1,\ ...,\ tp_{n-1}\}$ and corresponding counts returned by CSANN be C = $\{c_0,\ c_1,\ ...,\ c_{n-1}\}$. Now we define accuracy of our CSANN algorithm w.r.t Q and *th* as:

$$accuracy = \frac{\sum_{i=0}^{n-1} c_i}{\sum_{i=0}^{n-1} tp_i} \tag{4.3.1}$$

4.3.1.2 Experimental Design

We evaluated the accuracy of our algorithm in three parameter settings described in 4.4 on SWISS-PROT proteins dataset. We randomly selected subsets of sequences of varying size 5000 - 100,000 as database sets from SWISS-PROT dataset. Now for each subset we further randomly chose a query set of size 5,000 from the subset. We now conducted accuracy tests for these database sets and query sets and plotted the results for each parameter setting.

4.3.1.3 Evaluation

The figure 4.7 shows our accuracy results. As the dataset size varies from 5000 to 100,000 the accuracies of PI 70, PI 80 and PI 90 vary by little. The average accuracies of PI 70, PI 80 & PI 90 are 0.83, 0.91 and 0.995 respectively. The results show that our algorithm has 100% accuracy for PI 90 on SWISS-PROT dataset.

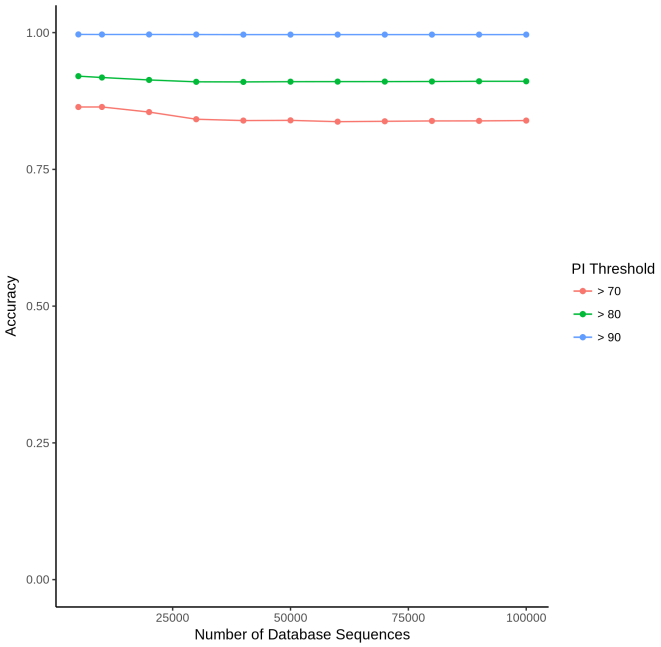

Figure 4.7: Dataset Size versus Accuracy

4.3.2 Query Time

4.3.2.1 Definition

Let a given input query sequences set Q = $\{q_0, q_1, ..., q_{n-1}\}$ of size n, corresponding query times be T = $\{t_0, t_1, ..., t_{n-1}\}$. Now we define average query time of our CSANN algorithm w.r.t Q as:

$$average - query - time = \frac{\sum_{i=0}^{n-1} t_i}{n} \tag{4.3.2}$$

4.3.2.2 Experimental Design

We evaluated the query time of our algorithm in three parameter settings described in 4.4 on SWISS-PROT proteins dataset. We randomly selected subsets of sequences of varying size 5000 - 100,000 as database sets from SWISS-PROT dataset. Now for

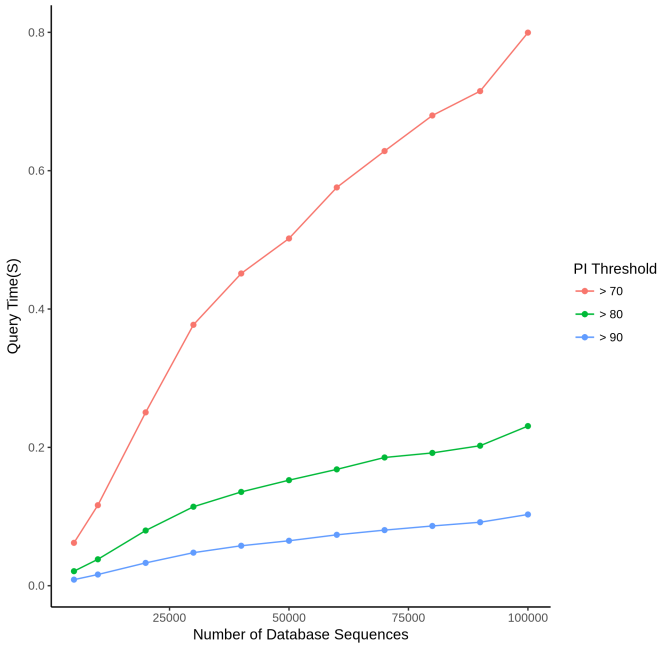

Figure 4.8: Dataset Size vs Query Time

each subset we further randomly chose a query set of size 5,000 from the subset. We now conducted tests for these database sets and query sets and plotted the results for each parameter setting.

4.3.2.3 Evaluation

We evaluated the query time of our algorithm in three parameter settings on SWISS-PROT proteins dataset. The figure 4.8 shows the query times. For PI 70, the query time is raised by a big factor when the dataset size is varied from 5000 to 100,000. However, for PI 80 and PI 90 the query times grow slowly. The results show that PI 80 and PI 90 perform a lot faster than PI 70. The average query times for PI 80 and PI 70 are 0.1382s and 0.06s respectively.

4.3.3 Average Number of Candidates

As described in section 3.5.2.2, query processing by our CSANN algorithm consist two filtering phases. The second filtering phase is an expensive phase as it involves filtering candidates of the first phase by pairwise comparisons using the Needleman–Wunsch (dynamic programming) algorithm, therefore second filtering phase directly affects the query time. We investigated the average number of candidates that need to be processed for an input query q as it impacts the query time of the query.

4.3.3.1 Definition

Let a given input query sequences set Q = $\{q_0, q_1, ..., q_{n-1}\}$ of size n, corresponding candidates counts in the second filtering phase be CC = $\{cc_0, cc_1, ..., cc_{n-1}\}$. Now we define average number of candidates processed by our CSANN algorithm w.r.t Q as:

$$average - number - of - candidates = \frac{\sum_{i=0}^{n-1} cc_i}{n} \tag{4.3.3}$$

4.3.3.2 Experimental Design

We evaluated the average number of candidates of our algorithm in three parameter settings described in 4.4 on SWISS-PROT proteins dataset. We randomly selected subsets of sequences of varying size 5000 - 100,000 as database sets from SWISS-PROT dataset. Now for each subset we further randomly chose a query set of size 5,000 from the subset. We then conducted tests for these database sets and query sets and plotted the results for each parameter setting.

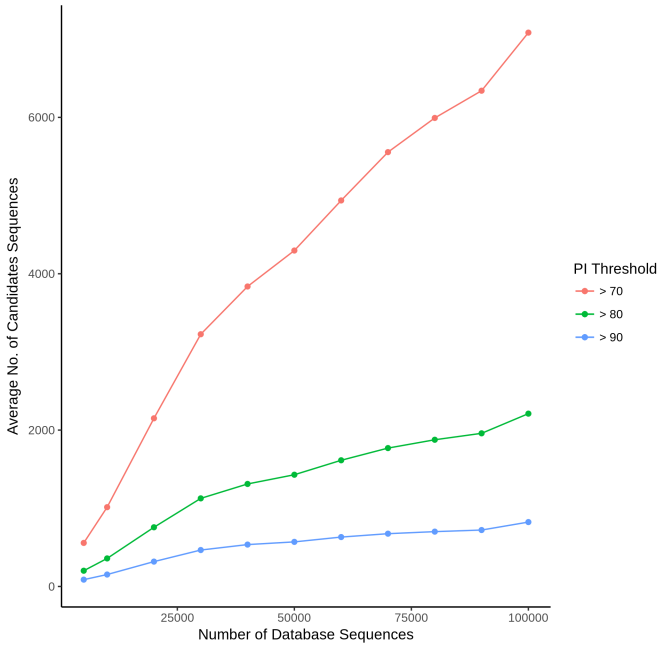

Figure 4.9: Dataset Size vs Number of Candidates in second filtering phase

4.3.3.3 Evaluation

The figure 4.9 describes how the number of candidates grow as the dataset size increases for parameter settings PI 70, PI 80 and PI 90. We can observe from the figure that the number of candidates grows slowly for PI 80 and PI 90 when compared with the PI 70. The reason for the small growth of candidates for PI 80 and PI 90 is the number of hash tables. PI 80 and PI 90 use 16 hash tables which is smaller than 32 which is used in PI 70. The average number of candidates for PI 80 and PI 90 are 1327 and 516 respectively.

4.3.4 Comparison with related algorithms

We compared our cosine-similarity based approximate nearest-neighbor algorithm (CSANN) with two related algorithms. One is Brute-force(NW) described in section 3

	Average query time	Average speed up	Average accuracy
Brute-force(NW)	1.66s	1x	1
BLASTP	0.796s	2x	-
CSANN(PI>70)	0.469s	3.5x	0.83
CSANN(PI>80)	0.1382s	12.5x	0.91
CSANN(PI>90)	0.06s	27.6x	0.995

Table 4.5: Average query times

and the other one is BLASTP [3] 2.5.1. The figure 4.10 shows the comparison in terms of query time between the three algorithms. Both BLASTP and Brute-force(NW) algorithms use pairwise comparisons to determine similar items to a query and their accuracy is very high. For comparison, we included all three configurations of our algorithm: CSANN(PI>70), CSANN(PI>80), and CSANN(PI>90).

4.3.4.1 Experimental Design

We randomly selected subsets of sequences of varying size 5000 - 100,000 as database sets from SWISS-PROT dataset. Now for each subset we further randomly chose a query set of size 5,000 from the subset. We now conducted tests for these database sets and query sets and plotted the results for each algorithm.

4.3.4.2 Evaluation

From the figure 4.10, we can see that the query times of the five algorithms increase as the dataset size increases. Table 4.5 shows the average query times, speed ups and accuracies of all the five algorithms. These averages are computed by varying database size from 10,000 - 100,000. Our CSANN algorithms are faster than Brute-force(NW) and BLASTP on average. Although the speedups of the CSANN(PI>70) and CSANN(PI>80) algorithms are higher, their average accuracies are lower when compared with the BLASTP and Brute-force(NW) algorithms. Our CSANN(PI>90)

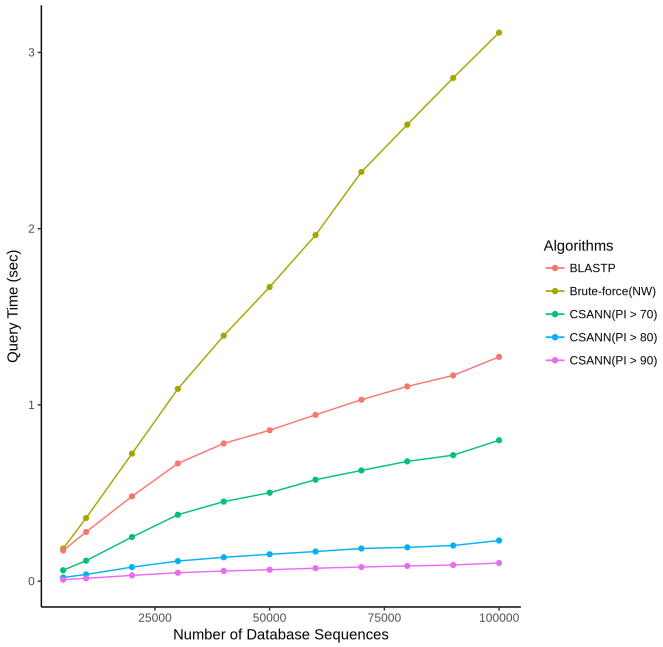

Figure 4.10: Query Time Comparison

has 27.6x speedup with average accuracy at 0.995 thereby outperforms both Brute-force(NW) and BLASTP algorithms.

It is important to note that the BLASTP and Brute-force(NW) algorithms do not use the percent identity threshold as required input. Therefore it is disadvantageous to use those algorithms when we want to search for only highly similar items. Our algorithm is significant in terms that it can speedup when higher percent identity thresholds are provided and maintain its accuracy as well, as shown in the figure 4.10.

4.4 Clustering

We compared our CSANN-Clust clustering algorithm 6 with CD-HIT [14] on SWISS-PROT protein sequence dataset. The objective of the two algorithms is to remove

redundant protein sequences from an input protein sequence data set and output only non-redundant sequences. The number of output clusters is equal to the number of non-redundant sequences. Both algorithms accept minimum percent identity threshold *th* as input, i.e., the sequences in each output cluster are at least percent identity *th* similar to the cluster representative. We compare the performance of CSANN-Clust with CD-HIT in terms of accuracy, number of output clusters generated and total clustering time. Additionally, we also examined the accuracy of CSANN-Clust with the Brute-force-Clust(NW) algorithm.

The clustering approach used in the Brute-force-Clust(NW) algorithm is same as that is used by the CSANN-Clust and CD-HIT algorithms, i.e. greedy incremental clustering algorithm. However, the difference between the three algorithms is that they use different alignment algorithms for computing percent identity between two sequences. The CD-HIT algorithm uses banded alignment algorithm, and the CSANN-Clust and Brute-force-Clust(NW) algorithms uses the Needleman–Wunsch algorithm for computing percent identity between two sequences.

4.4.1 Accuracy

We evaluated the accuracy of our CSANN-Clust algorithm, the CD-HIT algorithm and the Brute-force-Clust(NW) algorithm at three percent identity thresholds *th:* 70, 80 and 90. We used a dataset of size 100,000 protein sequences randomly chosen from SWISS-PROT dataset. All the three algorithms outputted large number of clusters (greater than 60,000 clusters) for each percent identity threshold. Therefore we chose only top 50 clusters from each algorithm by sorting clusters in decreasing size order.

We adopted our approach for testing clustering accuracy from [10]. Now we plotted the average pairwise distances 3.7.1 versus relative frequency. We used the term

	CSANN-Clust	CD-HIT	Brute-force-Clust(NW)
CSANN-Clust	1	0.856	0.935
CD-HIT	0.836	1	0.9
Brute-force-Clust(NW)	0.884	0.86	1

Table 4.6: Average Maximum Jaccard Index comparison (PI>70)

relative frequency to refer to number of sequence pairs with same average pairwise distance. Along with that, we also computed the average maximum Jaccard Index 3.7.2 between all the three clustering algorithms.

4.4.1.1 Evaluation

Figures 4.11 4.12 4.13 plots average pairwise distance versus relative frequency for percent identity thresholds th: 70, 80 and 90. As seen in the plots, the clusters of our CSANN-Clust are slightly more packed than the CD-HIT clusters (especially for PI>70). It is also seen from plots that CD-HIT has few clusters with the average pairwise distance greater than the input percent identity thresholds. The reason is that CD-HIT uses short-word filter, which is not accurate, to compute percent identity between any two sequences whereas our CSANN-Clust uses the global alignment algorithm called Needleman–Wunsch algorithm which is more accurate than the short-word filter. Also, we can see from plots that CSANN-Clust performed very closely to Brute-force-Clust for PI>90.

Tables 4.6 4.7 4.13 shows the average maximum Jaccard Index between the CSANN-Clust, the CD-HIT and the Brute-force-Clust(NW) algorithms for percent identity thresholds: 70, 80 & 90. We see that our CSANN-Clust clusters are almost equal to the Brute-force-Clust clusters for percent identity threshold 90.

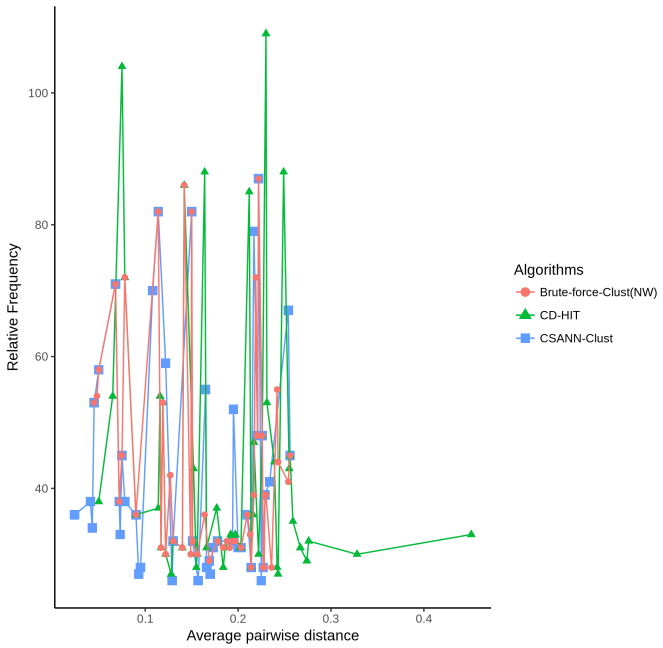

Figure 4.11: Average pairwise distance vs Relative frequency (PI>70)

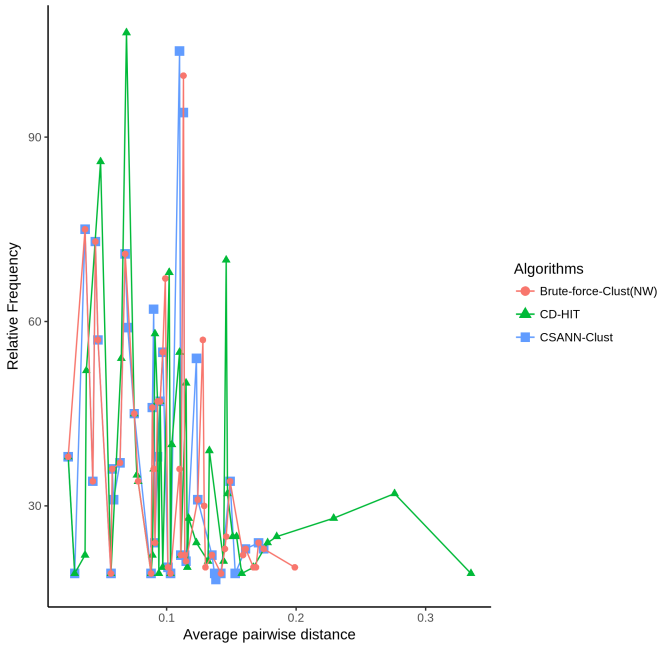

Figure 4.12: Average pairwise distance vs Relative frequency (PI>80)

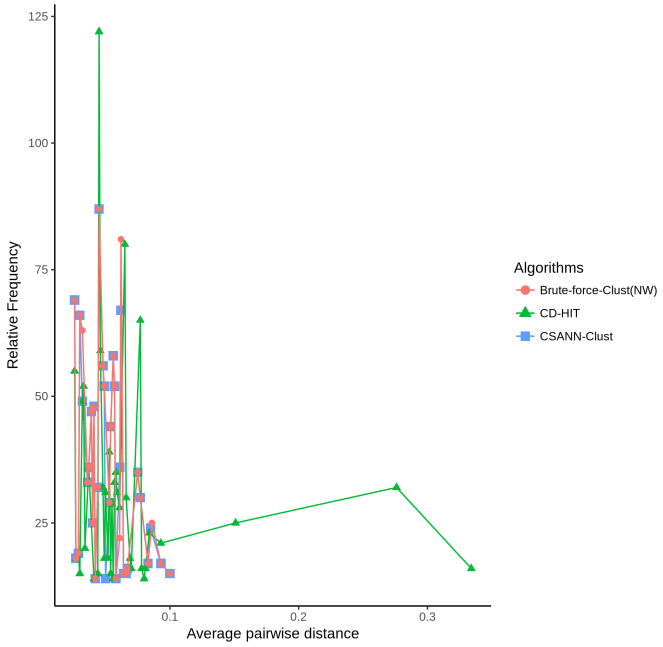

Figure 4.13: Average pairwise distance vs Relative Frequency (PI>90)

	CSANN-Clust	CD-HIT	Brute-force-Clust(NW)
CSANN-Clust	1	0.897	0.984
CD-HIT	0.886	1	0.9
Brute-force-Clust(NW)	0.942	0.89	1

Table 4.7: Average Maximum Jaccard Index comparison (PI>80)

	CSANN-Clust	CD-HIT	Brute-force-Clust(NW)
CSANN-Clust	1	0.914	0.999
CD-HIT	0.892	1	0.891
Brute-force-Clust(NW)	0.999	0.914	1

Table 4.8: Average Maximum Jaccard Index comparison (PI>90)

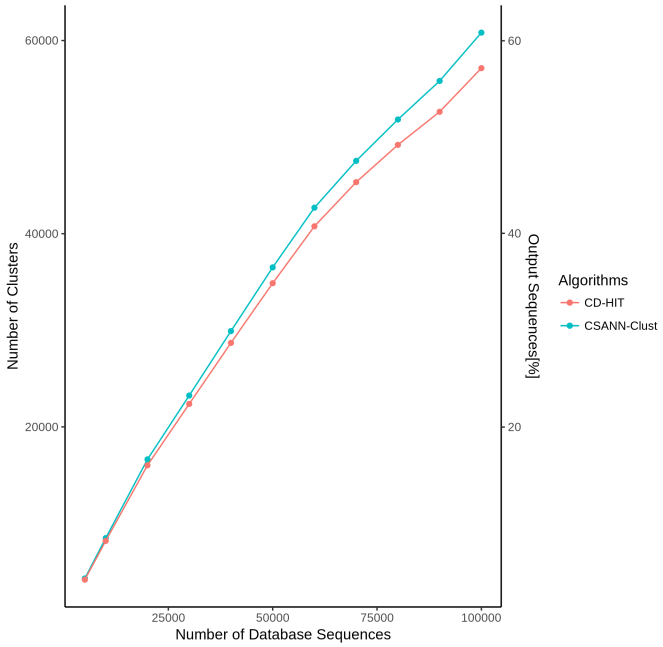

Figure 4.14: Dataset Size vs Number of Clusters (PI>70)

4.4.2 Number of Clusters

4.4.2.1 Experimental Design

We randomly selected subsets of sequences of varying size 5000 - 100,000 from SWISS-PROT dataset. Now for each subset, we ran both CSANN-Clust and CD-HIT algorithms at percent identity thresholds: 70, 80 and 90 and plotted the results.

4.4.2.2 Evaluation

Figures 4.16 4.15 4.14 shows the comparison of CSANN-Clust and CD-HIT in terms of number of clusters formed for percent identity thresholds 70, 80 and 90 respectively. We can see that the number of clusters formed by our algorithm CSANN-Clust is almost equal to the CD-HIT algorithm.

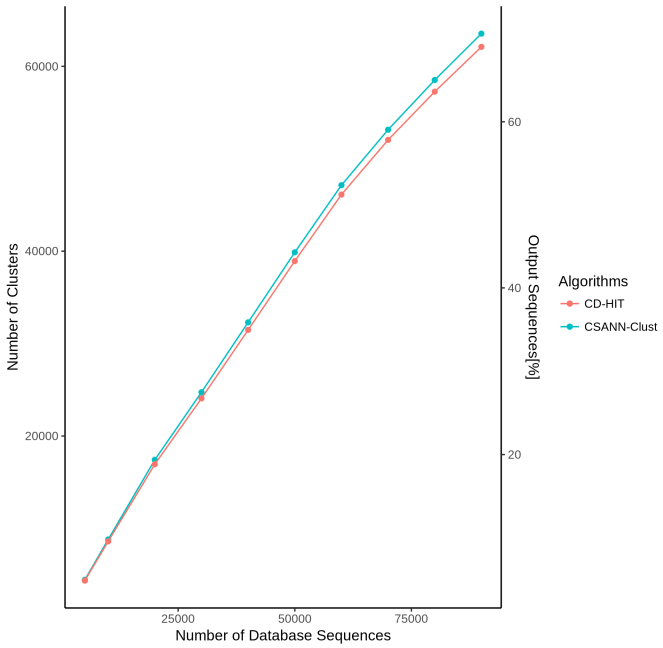

Figure 4.15: Dataset Size vs Number of Clusters (PI>80)

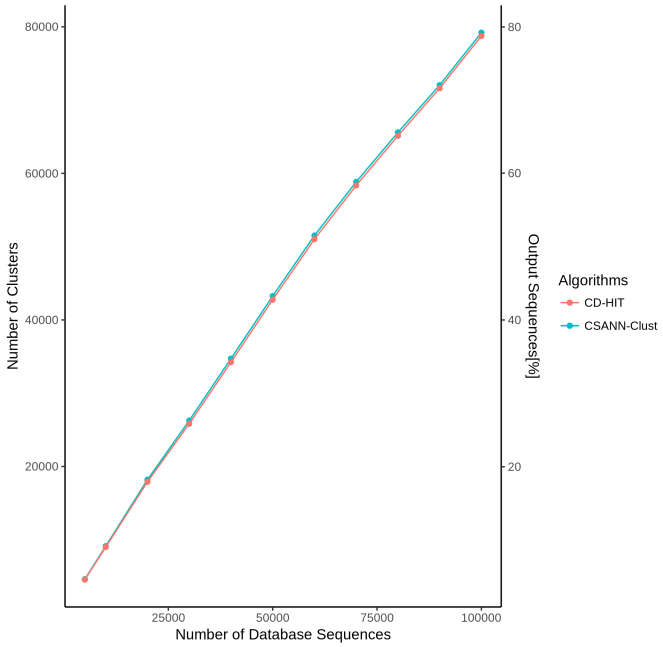

Figure 4.16: Dataset Size vs Number of Clusters (PI>90)

4.4.3 Clustering Time

4.4.3.1 Experimental Design

We randomly selected subsets of sequences of varying size 5000 - 100,000 from the SWISS-PROT dataset. Now for each subset we ran both CSANN-Clust and CD-HIT algorithms at percent identity thresholds: 70, 80 and 90 and plotted the results.

4.4.3.2 Evaluation

Figures 4.17 4.18 4.19 shows the comparison of CSANN-Clust and CD-HIT in terms of total time taken to form clusters for percent identity thresholds 70, 80 and 90 respectively. The CD-HIT algorithm is performing much faster than our CSANN-Clust algorithm for percent identity thresholds 70, 80 and 90. Although CD-HIT does all pairwise comparisons while determining similar items, it uses short-word filter and heuristics to determine the percent identity between two sequences instead of using the dynamic programming algorithm and that is the reason for its higher speedups even for large dataset sizes. Our CSANN-Clust algorithm avoids all pairwise comparisons by using cosine-similarity based LSH but uses the dynamic programming algorithm during the second filtering phase and that is the reason for its lower speed ups for large dataset sizes. However, our CSANN-Clust algorithm is performing very well at percent identity threshold 90 when compared with its performance at percent identity thresholds 80 and 70.

4.5 Assembly Data Analysis

We performed analysis on assembly datasets that we described in 4.1. We used our CSANN algorithm for this analysis. We compared assembly data of each assembler

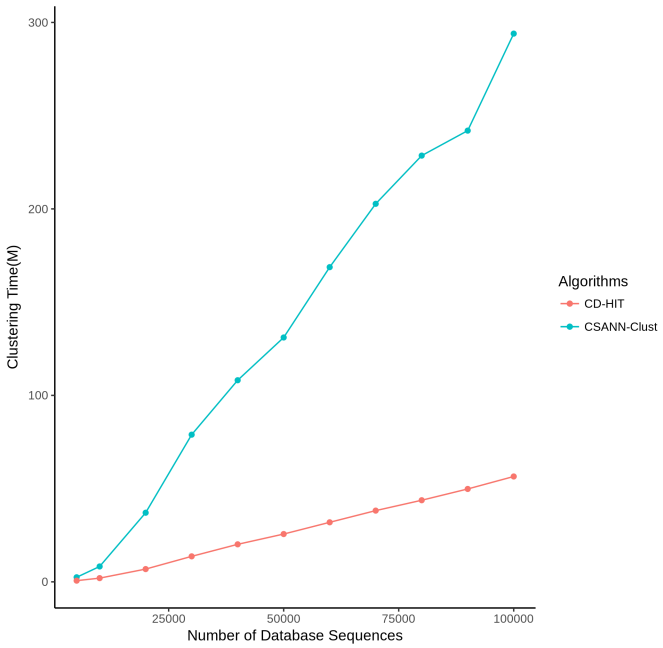

Figure 4.17: Dataset Size vs Clustering Time (PI > 70)

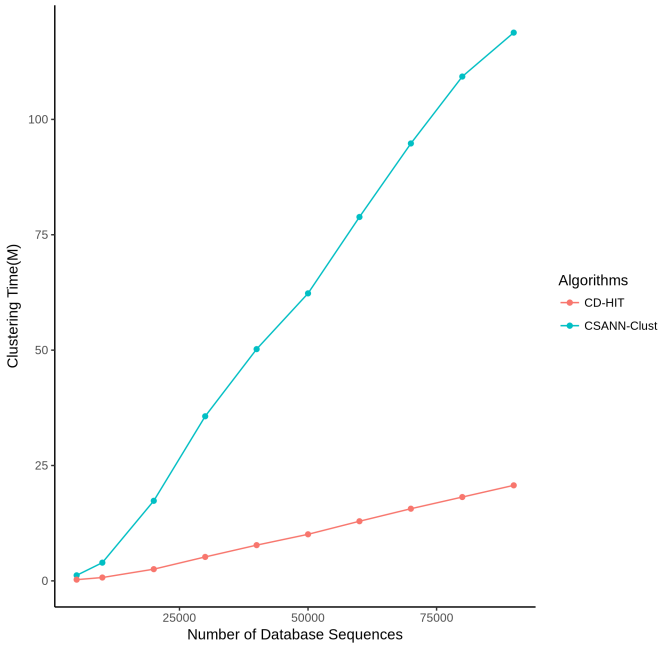

Figure 4.18: Dataset Size vs Clustering Time (PI > 80)

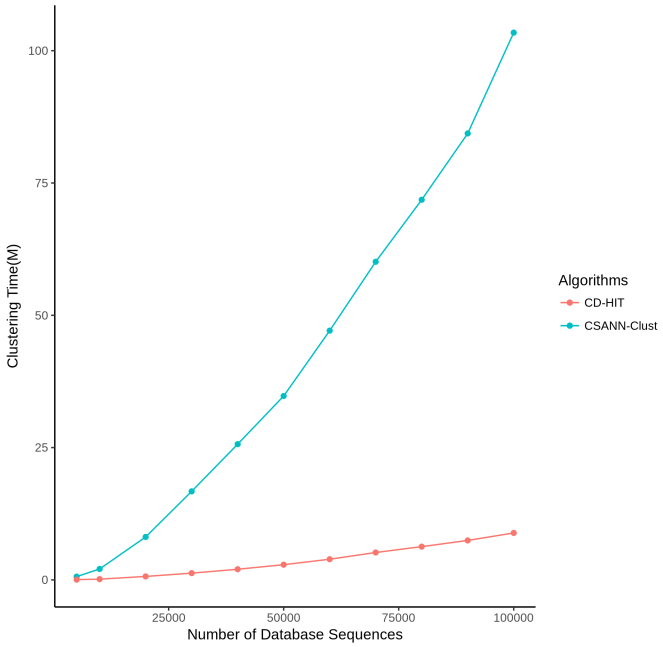

Figure 4.19: Dataset Size vs Clustering Time (PI > 90)

with true positive sequences. For a given input minimum percent identity threshold *th*, we performed two types of analysis:

- Analysis Type 1
  - Find the number of transcripts in an assembly data of an assembler that are similar to the true positive sequences with at least *th* percent identity.
- Analysis Type 2
  - Find the number of true positive sequences that are similar to the assembly data with at least *th* percent identity.

We generated 'All Combined' data by combining all *de novo* assemblers assembly data and 'All Combined-NR(clustered)' data by removing redundant protein sequences from the 'All Combined' data using our CSANN-Clust algorithm. We also analyzed

		IDBA		SOAPdenovo		SPAdes	
Total # of transcripts		177,126		374,913		486,912	
Correct # of transcripts		21,086	11.9%	11,173	3%	26,280	5.3%
# of transcripts with	>99.5 PI	28,373	16%	15,236	4%	39,927	8.2%
	>90 PI	63,494	35.84%	44,674	11.9%	125,256	25.72%
	>80 PI	81,040	45.75%	74,703	19.9%	180,396	37.04%
	>70 PI	94,192	53.17%	104,766	27.94%	228,075	46.84%

Table 4.9: Analysis 1 summary for four denovo assemblers - Part 1

		Trinity		All Combined		All Combined-NR	
Total # of transcripts		120,199		1,035,354		302,813	
Correct # of transcripts		21,945	18.25%	41,072	3.9%	10,900	3.6%
# of transcripts with	>99.5 PI	26,902	22.38%	61,467	5.9%	14,886	4.9%
	>90 PI	53,887	44.83%	221,768	21.4%	33,474	11.05%
	>80 PI	67,305	55.99%	330,329	31.9%	49,844	16.46%
	>70 PI	77,128	64.16%	425,074	41%	69,974	23.107%

Table 4.10: Analysis 1 summary for four denovo assemblers - Part 2

'All Combined' and 'All Combined-NR(clustered)' data by comparing with the true positive sequences.

The tables 4.9 4.10 shows the analysis 1 on assembly data of four *de novo* assemblers. In the tables, the first row represents the total number of transcripts in each assembler data and the second row represents the number of correct transcripts in assembly data that correctly match true positive sequences. As shown in the tables, as the percent identity threshold is decreased the number of transcripts that match true positives increased. The similar analysis on assembly data of two genome-guided assemblers is shown in the table 4.11.

The tables 4.12 4.13 shows the analysis 2 on assembly data of four *de novo* assemblers. The total number of true positives is 117,610. In the tables, the first row represents the total number of true positive sequences that correctly match the sequences in each assembler data. As shown in the tables, as the percent identity

		Bayesembler		Cufflinks		All Combined	
Total # of transcripts		4,002		63246		65,943	
Correct # of transcripts		1,396	34.9%	24,656	39%	25,370	38.47%
# of transcripts with	>99.5 PI	1,641	41%	28,924	45.7%	29,771	45.14%
	>90 PI	2,320	57.9%	38,233	60.45%	39,607	60%
	>80 PI	2,602	65%	43,167	68.25%	44,769	67.89%
	>70 PI	2,812	70.2%	46,593	73.66%	48,379	73.36%

Table 4.11: Analysis 1 summary for two Genome-guided assemblers

threshold is decreased the number of true positives that match the assembly data increased. Among four *de novo* assemblers SPAdes assembly data seem to have a higher number of true positive matches for all percent identity thresholds. In the table 4.13, the 'All Combined' assembly data have a higher number of true positives than any of the individual assembly data. As the 'All Combined' assembly data is a combination of multiple assemblers assembly data, the total number of transcripts are increased and the number of redundant sequences also increased. Our clustering algorithm was used to remove redundant data from the 'All Combined' assembly data and thereby reduced the output sequences size. The 'All Combined-NR' assembly data was created by removing redundant sequences in the 'All Combined' assembly data. Although the 'All Combined-NR' has reduced the number of transcripts, the total number of true positives present is equal to 107,705 (91.57%) with percent identity threshold 70. This demonstrates that our clustering algorithm can be very useful in removing redundant sequences from the combined assembly data analysis and still not lose too many true positives in the reduction process.

The figure 4.20 shows the chart of cumulative number of true positives found in 4 *de novo* assemblers and 'All Combined' assembly data versus percent identity threshold used. We performed analysis 2 using our CSANN algorithm to compute these results. The chart demonstrates the comparison of assembly data quality of 4

		IDBA		SOAPdenovo		SPAdes	
# of TP correctly matched		21,086	17.9%	11,173	9.5%	26,280	22.8%
# of TP matched with	>99.5 PI	26,735	22.7%	15,085	12.82%	34,569	29.39%
	>90 PI	47,851	40.68%	34,085	28.98%	69,940	59.46%
	>80 PI	61,289	52.11%	50,069	42.57%	89,298	75.92%
	>70 PI	72,852	61.94%	65,398	55.6%	101,421	86.23%

Table 4.12: Analysis 2 summary for four denovo assemblers - Part 1

		Trinity		All Combined		All Combined-NR	
# of TP correctly matched		21,945	18.65%	41,072	34.92%	10,900	9.26%
# of TP matched with	>99.5 PI	26,133	22.22%	50,850	43.23%	15,152	12.88%
	>90 PI	53,767	45.7%	88,879	75.57%	46,815	39.8%
	>80 PI	70,791	60.19%	103,620	88.10%	86,058	73.17%
	>70 PI	82,987	70.56%	110,813	94.22%	107,705	91.57%

Table 4.13: Analysis 2 summary for four denovo assemblers - Part 2

		Bayesembler		Cufflinks		All Combined	
Correct # of transcripts		1,396	1.18%	24,656	20.96%	25,370	21.57%
# of TP matched with	>99.5 PI	1,665	1.4%	29,230	24.85%	30,075	25.57%
	>90 PI	2856	2.4%	49,043	41.69%	50,135	42.62%
	>80 PI	3,855	3.27%	64,864	55.15%	65,890	56.02%
	>70 PI	4,869	4.1%	78,205	66.49%	79,036	67.20%

Table 4.14: Analysis 2 summary for two Genome-guided assemblers

de novo assemblers and also the 'All Combined' assembly data in terms of number of true positives found in each assembly data. From the figure we can determine that SPAdes assembly data have higher quality when compared with the remaining assemblers. It is also seen from the plot, that as the percent identity threshold is reduced the percent of true positives found is increased.

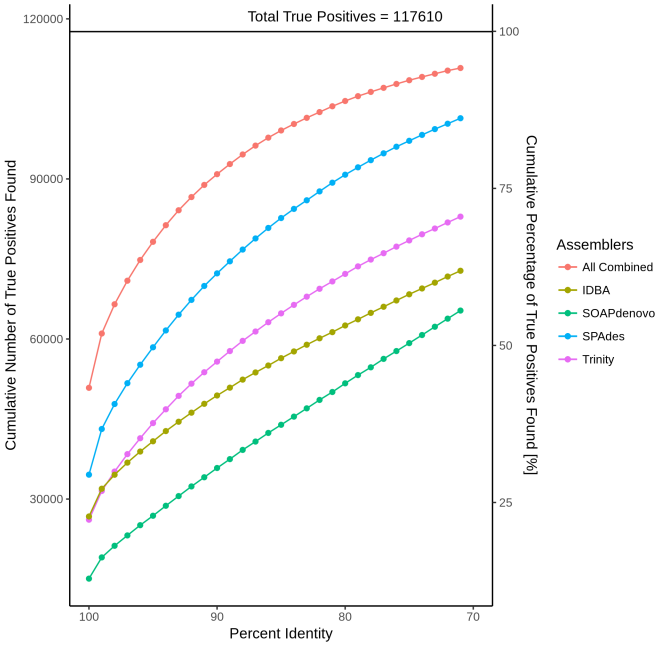

Figure 4.20: Number of true positives found vs Percent Identity of four denovo assemblers using analysis 2

Chapter 5

CONCLUSION AND FUTURE WORK

In this thesis, we developed an approximate similarity search algorithm based on cosine-similarity locality-sensitive hashing technique for biological sequences. We compared our algorithm with the Brute-force(NW) and BLASTP algorithms on SWISS-PROT (100,000 sequences) proteins dataset and the results demonstrated that our cosine-similarity based algorithm is 28 times faster than the Brute-force(NW) algorithm and 13 times faster than the BLASTP algorithm for finding similar sequences with percent identity greater than 90% and have 99.5% accuracy. We also developed a greedy incremental clustering algorithm based on our cosine-similarity nearest neighbors algorithm for removing redundant sequences in a proteins dataset. We compared our clustering algorithm with CD-HIT on SWISS-PROT proteins dataset. The clustering results show that our clustering algorithm generated clusters have accuracy almost equal to CD-HIT, but the total clustering time is higher than the CD-HIT. We demonstrated two bioinformatics applications of our algorithms, one is to perform assembly data analysis and other is for removing redundant sequences in a protein

dataset through clustering.

In this thesis, we evaluated our algorithms only on one protein dataset and k =3 for k-tuple vectors. Evaluation of our algorithms on nucleotide sequences and other datasets is left for future work. Query time of our cosine-similarity based similarity search algorithm depends on the dimension size of k-tuple vectors. As the dimension size increase both the query time and the memory usage increase exponentially. Higher values of k such as 4 and 5 for protein sequences, the dimension sizes are 160,000 and 3,200,000 respectively and the amounts of memory required to store one k-tuple vector in main memory are 625KB, 12.2MB respectively. Such problem is also known as the curse of dimensionality. Dimensionality reduction techniques needs to be investigated for protein sequences in order to use higher k values. Such research is left for future work.

Bibliography

[1] Cross-polytope wiki page. https://en.wikipedia.org/wiki/Cross-polytope. Accessed: 11/18/2017.

[2] rpforest python library on github. https://github.com/lyst/rpforest. Accessed: 2017-11-18.

[3] 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.

[4] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1225–1233. Curran Associates, Inc., 2015.

[5] Richard Bellman. Dynamic programming. princeton landmarks in mathematics, 2010.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[7] John Besemer, Alexandre Lomsadze, and Mark Borodovsky. Genemarks: a self-training method for prediction of gene starts in microbial genomes. implications for finding sequence motifs in regulatory regions, Jun 2001.

[8] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[9] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[10] Mohammadreza Ghodsi, Bo Liu, and Mihai Pop. Dnaclust: accurate and efficient clustering of phylogenetic marker genes, 2011.

[11] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[12] Ashraf M Kibriya and Eibe Frank. An empirical comparison of exact nearest neighbour algorithms. In *PKDD*, volume 7, pages 140–151. Springer, 2007.

[13] Li Li, Yang Guo, Wenwu Wu, Youyi Shi, Jian Cheng, and Shiheng Tao. A comparison and evaluation of five biclustering algorithms by quantifying goodness of biclusters for gene expression data. *BioData Mining*, 5(1):8, July 2012.

[14] 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.

[15] CÃľdric Notredame, Desmond G. Higgins, and Jaap Heringa. T-coffee: a novel method for fast and accurate multiple sequence alignment11edited by j. thornton. *Journal of Molecular Biology*, 302(1):205–217, September 2000.

[16] Martin ÅăoÅąiÄĞ and Mile ÅăikiÄĞ. Edlib: a c/câĂĽ++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017.

[17] Stephen M Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[18] Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348–1358, 2010.

[19] Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. Randomized algorithms and nlp: Using locality sensitive hash function for high speed noun clustering. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 622–629, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[20] Ilya Razenshteyn and Ludwig Schmidt. Falconn library on github. https://github.com/FALCONN-LIB/FALCONN. Accessed on 11/18/2017.

[21] Madden T. The blast sequence analysis tool. The NCBI Handbook [Internet]. 2nd edition. Bethesda (MD): National Center for Biotechnology Information (US), 03 2013. Available from: https://www.ncbi.nlm.nih.gov/books/NBK153387/.

[22] Kengo Terasawa and Yuzuru Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. *Algorithms and Data Structures*, pages 27–38, 2007.

[23] Peter N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, volume 93, pages 311–321, 1993.