

Quality-Based Similarity Search for Biological Sequence Databases

Xuehui Li Tamer Kahveci
CISE Department, University of Florida, Gainesville, FL 32611
{xli, tamer}@cise.ufl.edu

Abstract

Low-Complexity Regions (LCRs) of biological sequences are the main source of false positives in similarity searches for biological sequence databases. We consider the problem of finding similar sequences when the locations of the LCRs are not known precisely. We develop a formulation to measure the quality of each letter in a sequence. The quality value of a letter is the probability for that letter to be in a non-LCR. We show that the quality values can be employed in two fundamental approaches to the sequence search problem to reduce the number of false positives produced by them significantly. The former finds the optimal alignment of two sequences using dynamic programming. The latter computes a suboptimal alignment using hash table. For the latter one, we also develop a randomized memory-resident hash table that indexes k -grams (sequences of length k) probabilistically. The k -grams that are likely to contain LCRs are indexed with lower probabilities. As a result, memory usage and CPU cost are greatly reduced. We also show that this hash table can be used to reconstruct query sequences with negligible information loss. This eliminates the need to store these sequences. Our experiments on real data show that our quality-based similarity search algorithms reduce the number of false positives drastically. In addition, their running times were better than the existing strategies.

1 Introduction

Sequence similarity search algorithms play a very important role in bioinformatics. They can be used to identify candidates of related sequences that form a family or to find candidates of a related gene in an organism. These candidates then go through a costly manual inspection by biologists. Therefore, it is essential that sequence search algorithms return as few false positives as possible.

Low-Complexity Regions (LCRs) are repeating patterns of biased composition [10]. Figure 1 shows two sequences that contain LCRs indicated by the underlined letters. Both of the LCRs in this figure contain a tandem repeat AAT repeated four times. Statistical analyses have shown that approximately one-quarter of the amino acids are in LCRs and more than one-half of proteins have at least one LCR [20].

```
Position #: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25  
Sequence 1: G A A T A A T A A T A A T A W S V W S P T V L L S
```

```
Position #: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20  
Sequence 2: P P Q K M K A A T A A T A A T A A T R D
```

Figure 1: Two sequences that have the same LCR indicated by the underlined letters. Here, the LCRs are composed of a repeating pattern of AAT.

Traditional sequence similarity search methods produce many false positives due to LCRs in biological sequences. We use BLAST [1], one of the popular similarity search algorithms, as an example to illustrate the problem. BLAST uses the maximal segment pair score (MSP) to find the optimal alignment. The theory of MSP finds statistically significant high-scoring alignments under the assumption that letters follow a random distribution. However, biological sequences are very different from random sequences since they contain many LCRs. Statistically significant high-scoring matches due to LCRs, usually, do not indicate genuine relationship between sequences, and hence, are false positives. For example, traditional algorithms produce a high alignment score for the two sequences in Figure 1 even though they are not biologically related. This is because the two sequences contain the same LCR. BLAST returns over 1,000 statistically significant sequences for *Thermus thermophilus seryl tRNA synthetase*, which has only 31 true positive homologs [19]. Such high false positive rates cause enormous amounts of wasted resources and time spent on refuting them.

Existing methods such as BLAST follow one of the two extreme strategies; they either treat LCRs and non-LCRs the same or simply remove all LCRs from the sequences¹. The former strategy produces many false positives. The latter strategy requires the use of an LCR-identification method. A number of computational methods have been developed to identify LCRs such as SEG [21]. Although filtering identified LCRs improves the quality of searches, it is not desirable since no LCR-identification method is 100% accurate. Our experiments on real data showed that the average precision and recall of some well-known methods such as SEG and CARD [15] were as low as

¹Depending on which sequence is masked, the latter one is provided as a “filter low complexity” or “filter lookup table” option

0.2 and 0.3 respectively [10]. Hence, both strategies are problematic. Note that BLAST also has an option where it masks a region specified by a user. This, however, requires the user to have perfect knowledge of the location of such regions. Thus, it is not practical as these regions are not known for many sequences.

Contributions: This paper considers the problem of finding similar sequences when the locations of the LCRs are not known precisely. The goal is to develop algorithms that reduce the number of false positives significantly without losing true positives. There are three main contributions of this paper. 1) We describe a proper way of using LCRs. We develop a formulation to measure the quality of each letter in a sequence. The quality value of a letter is the probability for that letter to be in a non-LCR. We show that the quality values can be used in two fundamental approaches to the sequence search problem. The former finds the optimal alignment of two sequences using dynamic programming. The latter computes a suboptimal alignment using a hash table. 2) We develop a randomized memory-resident hash table that indexes k -grams (subsequences of length k) probabilistically. As a result, the main memory usage and the CPU cost are greatly reduced. 3) We show that this hash table can be used to reconstruct query sequences with negligible information loss. This eliminates the need to store these sequences. Our experiments on real data show that our algorithms reduce the number of false positives significantly. In addition, their running times are better than existing strategies. Note that we use local alignment to define similarity. Extending this paper to global alignment is trivial. *The methods developed in this paper are orthogonal to existing search tools. They can be used along with the existing tools to improve their accuracy and performance.*

Paper Organization: Section 2 discusses the related work. Section 3 illustrates how to assign a quality value to each letter in a sequence. Section 4 introduces our similarity search algorithms. Section 5 shows quality and performance results. Section 6 concludes the paper.

2 Background

Sequence similarity search: A number of sequence similarity search algorithms have been developed [16, 1, 11, 8, 4, 17]. We consider these strategies under two categories. The first one, called DPS, finds the optimal alignment using dynamic programming. The second one, called HTS, finds a suboptimal alignment by employing a hash table. Smith-Waterman algorithm [16] and BLAST are examples of these two strategies respectively.

Dynamic programming solution (DPS): DPS computes a two dimensional matrix M . Each entry corresponds to the best local alignment score between two subsequences being compared. DPS constructs the best alignment by

tracing back from the maximum score in M until reaching a zero.

Hash table solution (HTS): This strategy sacrifices sensitivity (i.e., recall) to improve space and time. For any two k -grams generated from the letter alphabet, if their alignment score is greater than a threshold, they are called *neighbors* of each other. Neighbors of all k -grams are first generated. HTS uses matching k -grams as seeds to find longer alignments. It runs in three steps:

(1) *Building hash table.* Each k -gram in each query sequence is inserted into the hash table.

(2) *Search phase.* The database is scanned to find neighbors of the k -grams from the query set with the help of the hash table. The region is popularly referred as *seed*.

(3) *Alignment phase.* Seeds are extended to find local alignments that satisfy a user-defined score cutoff.

LCR-identification: Several algorithms have been developed to identify LCRs in sequences, such as SEG [21], CARD [15], and GBA [10]. SEG first finds contigs with Shannon Entropy complexity [14] less than a cutoff. It then detects the leftmost and longest subsequences with minimal probability of occurrence as LCRs. BLAST uses SEG as the underlying LCR detection algorithm for protein sequences. CARD identifies LCRs based on the Shannon Entropy complexity analysis of subsequences delimited by a pair of identical repeats. GBA develops a k -gram *complexity* that takes letter similarity, letter order, and sequence length into account. It uses a graph-based method to find potential seeds. It then extends these seeds into longer intervals based on a novel complexity measure. Finally, these longer intervals are postprocessed to find LCRs.

Quality values: Some algorithms, such as CAP3 [6], have used quality values. The meaning of CAP3’s term “quality” differs from that of this paper. In CAP3, each letter is given a quality value, based on the probability that that letter is correctly sequenced. Thus, it does not denote whether a letter is in LCR or not. CAP3’s use of quality values, however, is similar to traditional filters for LCRs. It clips the ends of reads (fragments of sequences) that have quality lower than a cutoff. At the time of assembly, it uses two types of scores, one for letters with quality greater than a given cutoff and another for the rest of the letters. This indicates that the algorithms developed in this paper can be employed for sequence assemblers like CAP3, when the quality values are not 100% correct. This paper, however, focuses on sequence comparison in the presence of LCRs.

3 Quality Value Assignment

A letter in a sequence is correctly masked by the underlying LCR-identification algorithm if the letter is annotated in an LCR according to a gold standard (e.g., Swissprot

```

Position #: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Sequence:  G A A T A A P A A T A A T A W S V W S P T V L L S
Masked:   G x x T x x P x x T x x T x x W S V W S P T V L L S

```

Figure 2: A sequence with an underlined LCR and a masked version of it. Masked letters are replaced by x.

(www.expasy.org/sprot). Let TP (True Positive) be the number of letters that are correctly masked as LCRs by the tool. Similarly, let FP (False Positive) and FN (False Negative) be the number of letters that are incorrectly computed as LCRs and non-LCRs. Figure 2 shows a sequence, its LCR, and its masked version by an LCR-identification algorithm. Masked letters are replaced by x. Here, $TP = 8$, $FP = 1$, and $FN = 4$.

In this section, we introduce how to assign a quality value to a letter of a sequence when the underlying LCR-identification method is inaccurate. During the assignment, we use one or more LCR-identification algorithms. Sections 3.1 and 3.2 discuss the cases when single and multiple LCR-identification tools are employed respectively.

3.1 One LCR-identification tool

For each sequence of length L , two measures, *precision* and *noise* are computed as:

- precision = $TP / (TP + FP)$;
- noise = $FP / (L - TP)$.

Here, precision is the probability that a letter randomly chosen among the masked letters is a true LCR. We assign (1-precision) to masked letters of the sequence as their quality values. Noise is the error rate in unmasked letters. (1-noise) then tells the probability for an unmasked letter to belong to a non-LCR. Hence, we assign (1-noise) to all unmasked letters as their quality values.

We can compute TP, FP, and FN only if we know the true LCRs. This is also true for precision and noise. However, we do not know the true LCRs of most sequences. We propose to solve this problem by approximating to precision and noise with the help of sampling. We randomly sample n sequences that contain known repeats from Swissprot (www.expasy.org/sprot). These manually annotated repeats (http://www.expasy.org/sprot/sprot_details.html) do not have any known functions and are reliable for our purpose of sampling. Running any LCR-identification algorithm on these sequences will give each sequence in this set a precision value and a noise value as computed above. For each masked sequence, we also calculate the k -gram complexity [10] for its masked letters. Hence, we can represent each sequence by a (precision, complexity) tuple. Let $(p_1, c_1), (p_2, c_2), \dots, (p_n, c_n)$ be the precision and complexity values for the sampled sequences where $c_1 \leq c_2 \leq \dots \leq c_n$. We create an equi-depth histogram with m bins as follows. We define the bound-

aries H_0, H_1, \dots, H_m of the bins of the histogram as $H_0 = -\infty, H_m = +\infty, H_i = c_{\lfloor n/m \rfloor \cdot i}$ for $0 < i < m$. The first bin contains the first $\lfloor n/m \rfloor$ lowest-complexity tuples. The second bin contains the next $\lfloor n/m \rfloor$ lowest-complexity tuples and so on. For each bin represented by $[H_{i-1}, H_i]$, we define its precision, denoted by π_i , as the mean of the precision of the tuples in that bin. Let $(p_{(i-1)\lfloor n/m \rfloor + 1}, c_{(i-1)\lfloor n/m \rfloor + 1}, \dots, (p_{i\lfloor n/m \rfloor}, c_{i\lfloor n/m \rfloor}))$ be these tuples. Formally, we compute π_i as:

$$\sum_{j=(i-1)\lfloor n/m \rfloor + 1}^{i\lfloor n/m \rfloor} \frac{p_j}{\lfloor n/m \rfloor}.$$

To assign each letter in a sequence a quality value, we first calculate the k -gram complexity of its masked letters and find the histogram bin that this complexity belongs to. Given a complexity value, denoted by μ , we say that μ belongs to bin i if $H_{i-1} \leq \mu < H_i$. We then use the precision of that bin, π_i , as the *precision* of its masked letters. We assign $(1 - \pi_i)$ to all masked letters of this sequence as their quality values.

Next, we talk about how to assign quality values to unmasked letters. We calculate the average of *noises* from all sample sequences. Since the difference between one and this average tells the overall probability for an unmasked letter to belong to a non-LCR, we assign this difference to unmasked letters as their quality values.

3.2 Multiple LCR-identification tools

No LCR-identification method is 100% accurate. Regions masked by different methods can be different. This is because different LCR-detection methods may be tuned for detecting different types of LCRs. For example, CARD is good at identifying LCRs delimited by a pair of repeating subsequences. SEG depends on the window length and Shannon Entropy complexity measure to identify LCRs. Thus, accuracies of LCR-detection methods can be improved by integrating their results. Figure 3 shows a sequence with an underlined LCR and its masked versions by two LCR-identification tools. Masked letters are replaced by x. Each tool masks different regions as LCRs. If different tools classify the same letter to belong to an LCR, then the chance for this letter to belong to an LCR is higher compared to the case where these tools do not agree. Integrating quality values from multiple tools is a nontrivial task. In principle, a quality value from a high-accuracy LCR-identification algorithm should contribute more to the combined quality value than that from a low-accuracy LCR-identification algorithm.

The average *Jaccard coefficient* reflects the accuracy of an LCR-identification algorithm. For a sequence, Jaccard coefficient is computed as

$$TP / (TP + FP + FN).$$

The average Jaccard coefficient from all sequences reflects how well a classifier mirrors the actual class labels [5]. It approaches to one when the classifier gives all instances

```

Position #: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Sequence:  G A A T A A T A A T A A T A W S V W S P T V L L S
Masked 1:  G x x x x x x x x x x x x W S V W S P T V L L S
Masked 2:  G x x x x x x x x x x A x x V x x P T V L L S

```

Figure 3: A sequence with an underlined LCR and its masked versions by two different LCR-identification algorithms. Masked letters by each algorithm are replaced by x.

true class labels. We propose to compute the combined quality value by taking a weighted average of these quality values. Let JC_1, JC_2, \dots, JC_t be the Jaccard coefficients from the t algorithms respectively. Let q_1, q_2, \dots, q_t be quality values assigned to a letter by t algorithms respectively as discussed in Section 3.1. We compute the quality value of this letter by using these t algorithms as:

$$\frac{\sum_{i=1}^t q_i \cdot JC_i}{\sum_{i=1}^t JC_i}$$

The higher the Jaccard coefficient of an algorithm is, the more its quality value affects the new quality value.

4 Quality-based similarity search

In Section 3, we discussed how letters are assigned quality values. In this section, we introduce our quality-based similarity search algorithms: QDPS (Section 4.1) and QHTS (Section 4.2).

4.1 Using quality values in DPS

Our first algorithm, QDPS (Quality-based Dynamic Programming Similarity search algorithm), aligns two sequences. QDPS guarantees full sensitivity, i.e., it can find all optimal local alignments. This algorithm is similar to DPS. The difference is that QDPS uses quality values of letters. Unlike DPS, QDPS weights the score/penalty of each letter pair, including gaps, by multiplying it with the quality values computed for the letter pair. The higher the quality value of a letter is, the more it contributes to the alignment score. The existing strategy of removing LCRs entirely is a special case of QDPS when the quality values of the LCRs are set to zero. QDPS shows this behavior only when the underlying LCR-identification algorithm is 100% precise for the corresponding complexity (i.e., no false positives). Thus, QDPS is superior to this existing strategy since it adapts to the precision of the underlying LCR-identification algorithm.

The space and time complexity of QDPS is $O(mn)$. This is the same as those of DPS. This is because they both compute dynamic programming matrices of the same size. Note that there are optimized versions of DPS which have $O(\min\{m, n\})$ space complexity at the expense of increased running time [13]. Such optimizations can be applied to QDPS too.

4.2 Using quality values in HTS

4.2.1 The algorithm

Many biological applications require all-to-all comparison of two large sequence sets, say, A and B [9, 3]. In these applications, A and B can both be too big to fit in main memory. We will refer to the sequences in A as query sequences and the sequences in B as database sequences in the rest of this section. A trivial solution to this problem is to compare all pairs of sequences (a, b) where $a \in A, b \in B$, using QDPS. This is however impractical as the number of sequence pairs is $|A| \cdot |B|$ and comparing a pair of sequences is costly.

HTS methods, such as BLAST, alleviate this problem by employing a hash table (see Section 2). In this section, we show how to incorporate quality values to such hash table-based searches. We develop a new method called QHTS (Quality and Hash Table-based Similarity search algorithm). It imitates the well-known HTS methods. QHTS exploits quality values not only to perform quality-based search, but also to further improve the memory usage and the running time of traditional hash table-based sequence similarity searches. Similar to HTS, QHTS has three steps: (1) probabilistic hash table construction, (2) search phase, and (3) alignment phase. Note that some of the existing HTS algorithms deviate slightly from the described three steps by finding multiple seeds [18] or using spaced seeds [12]. It is trivial to extend QHTS to simulate them.

Probabilistic hash table construction: We slide a window of length k on the sequences of one of the datasets, say A . Each window position produces a k -gram. For each k -gram, we calculate a quality value q_{avg} by taking the average of the quality values of all k letters in this k -gram. Let $a_i \dots a_{i+k-1}$ be a k -gram. Let $q_{a_i}, \dots, q_{a_{i+k-1}}$ be the quality values of the letters of this k -gram. Formally, the average quality of this k -gram is computed as

$$q_{avg} = \frac{\sum_{j=0}^{k-1} q_{a_{i+j}}}{k}.$$

The quality value q_{avg} is a real number in $[0, 1]$ interval. We insert this k -gram into the hash table with probability q_{avg} .

The bigger the quality value of a k -gram, the bigger the chance that it belongs to a non-LCR and the bigger the chance that it is inserted into the hash table. Hence, the probabilistic insertion tends to insert k -grams from non-LCRs into the hash table. Letters with low quality values are possible true positives, i.e., LCRs. They tend not to be inserted. Figure 4 illustrates a sequence and its k -grams (for $k = 3$). Only three k -grams are stored in the hash table.

Our probabilistic hash table has two advantages over the traditional strategy where all k -grams are kept:

1. Since k -grams from LCRs tend not to be inserted into the hash table, they will not be identified as seeds.

Thus, it is less likely to produce false positives due to seeds in LCRs.

2. The hash table is smaller compared to the case where all k -grams are inserted. This reduces the I/O cost (see Section 5.2).

Our hash table is also superior to BLAST’s hash table using “filter lookup table” option. This is because the latter one removes all the k -grams in the regions masked by an LCR-identification tool. This is undesirable as LCR-identification tools are highly inaccurate. The former one, on the other hand, includes the k -grams in these regions with probabilities determined by the qualities of the k -grams.

Search phase: In this phase, we discuss how we determine seeds. Let $a_i \cdots a_{i+k-1}$ and $b_j \cdots b_{j+k-1}$ denote k -grams in sequences $a \in A$ and $b \in B$, where the k -gram in b is a neighbor of the k -gram in a (see Section 2 for the definition of neighbor). Let $q_{a_i}, \dots, q_{a_{i+k-1}}$ and $q_{b_j}, \dots, q_{b_{j+k-1}}$ be quality values assigned to the letters in the two k -grams respectively. We calculate their alignment score as

$$\sum_{t=0}^{k-1} s(a_{i+t}, b_{j+t}) \cdot q_{a_{i+t}} \cdot q_{b_{j+t}}.$$

If this alignment score is greater than the neighbor threshold, we call this region a *seed*. During this phase, for each k -gram in B , we find all its neighbors in A with the help of the hash table. We then recalculate their alignment scores to decide seeds.

Alignment phase: We extend each seed, i.e., $a_i \cdots a_{i+k-1}$ and $b_j \cdots b_{j+k-1}$ in both left and right directions with no gap allowed along a and b respectively. Whenever we extend each current subsequence by a new letter, we update the alignment score with quality values included. Let $a_i \cdots a_{i+x}$ and $b_j \cdots b_{j+x}$ where $x \geq k$ be the two current subsequences to be extended respectively. Let m denote the alignment score between them. Let a_{i+x+1} and b_{j+x+1} be the two new letters to be added. Let $q_{a_{i+x+1}}$ and $q_{b_{j+x+1}}$ denote the quality values of the new letters respectively. The alignment score is updated as

$$m := m + s(a_{i+x+1}, b_{j+x+1}) \cdot q_{a_{i+x+1}} \cdot q_{b_{j+x+1}}.$$

We extend in each direction until the difference between the maximum alignment score observed so far and the current alignment score is greater than an extension threshold. The maximum alignment score among all seed extensions between the database and the query sequence is taken as their alignment score.

4.2.2 I/O and CPU computations

During the extension we need to access query sequences. The trivial choice is to store all of them in main memory at all times. This, however, allocates memory redundantly since only the part of the query sequence that takes part in the alignment is needed. Such redundant memory usage is undesirable especially when two very large

G A R A Q A Q A Q K L

G A R A O X X X O K L

Figure 4: An example for probabilistic hash table and reconstruction. The solid lines show the three 3-grams of the sequence GARAQAQAQKL stored in the hash table. The resulting sequence after reconstruction is at the bottom. Letter X denotes an unknown letter.

sequence databases are compared to each other. Another option is to read a subsequence to main memory whenever needed. This, however, can incur many costly page reads if the sequences evicted from memory to store the new subsequences are needed again in later steps. We propose to reconstruct query sequences from the hash table whenever needed. Figure 4 shows a sequence GARAQAQAQKL and its three (out of nine) k -grams ($k=3$) stored in the hash table. In this example, we can reconstruct 73% of the letters of this sequence using these k -grams. The lost letters are given quality values of zero, and they are replaced with the letters “N” and “X” for DNA and amino acid sequences respectively. Three important notes can be made about the performance of QHTS.

1. The CPU time for the search and the alignment phases are reduced. This is because not all k -grams are inserted into the hash table. Hence, fewer seeds are usually found and extended (see Section 5.2).

2. Reconstructing query sequences from the hash table for seed extension reduces I/O cost. This is because query sequences are not read from disk. The hash table is in the memory. Hence, reconstructing query sequences from it does not involve any I/O cost (see Section 4.2.4).

3. The sensitivity may drop due to reconstruction. This is because some letters may not be recovered if none of the k -grams containing those letters is inserted into the hash table. However, the drop in sensitivity is insignificant for two reasons. First, the missing k -grams usually belong to the LCRs that need to be removed. Second, a letter belongs to k different k -grams. A letter can not be recovered during reconstruction only if all the k -grams that contain it are missing from the hash table. Thus, as k increases the probability that a letter can not be recovered decreases exponentially. Hence, the sensitivity drop is small (see Section 5.1).

4.2.3 Cost analysis of QHTS

Let σ denote the letter alphabet size. For protein sequences, $\sigma = 20$. Let L be the average sequence length. Let X and Y denote the number of sequences in the query set and the database respectively. Let k be the k -gram size. Let q denote the average quality of the k -grams in the query set.

Probabilistic hash table construction: QHTS spends time linear in the query set size to build the hash

table, i.e., $O(XL)$. The query set contains $O(XL)$ k -grams. However, QHTS does not insert all these k -grams into the hash table. The number of entries in the hash table is $O(XLq)$.

Search phase: This phase takes $O(Y)$ time since it involves a hash table lookup for each k -gram of the database. Each k -gram in the database can have as many as $O(XL/\sigma^k)$ exact matches in the query set. The hash table keeps only a fraction, q , of the query k -grams. Thus the storage needed to store the seeds of a database sequence is $O(XLq/\sigma^k)$.

Alignment phase: The time complexity of this phase depends on the proximity of the query set and the database as well as qualities of the letters. Let p be the probability that two k -grams fail the neighborhood threshold using QHTS, given that they satisfy the threshold without using quality values. The number of seeds that QHTS extends becomes $O(XYL^2q(1-p)/\sigma^k)$. On the other hand, traditional methods need to extend $O(XYL^2/\sigma^k)$ seeds. Thus, the time complexity of QHTS at this step is much less.

The above analysis shows that QHTS performs better than HTS no masking at each step except the first one where they have the same time complexity. The dominant factor in the overall complexity comes from the search phase and the alignment phase. Hence, we conclude that the speedup of QHTS over HTS no-masking strategy is between $O(1/q)$ and $O(\frac{1}{q(1-p)})$.

4.2.4 Memory allocation

QHTS stands out especially when the query set and the database sizes are much larger than the available main memory. It is very important to have a good memory allocation scheme to minimize I/O cost.

To get a block of data from disk, three kinds of time may be spent: seek time, rotation time, and transfer time, denoted by st , rt , and tt respectively. Let Q and DB be sizes of the query set and the database in pages respectively. Assume that a page is big enough to hold the longest sequence. Let M be the total number of available main memory pages. In QHTS, M is divided into four pieces, say M_1 , M_2 , M_3 , and M_4 . Here M_1 , M_2 , M_3 , and M_4 denote the space allocated to hash table, database sequences, query sequences, and quality values respectively. Let C denote the number of memory pages used to index the k -grams of sequences from one disk page. Hence, M_1 pages can hold hash table for M_1/C pages of sequences. Quality values of all sequences can be stored using a small amount of fixed memory, say assigning one page to M_4 . This can be justified as follows. For each sequence, only a quality value for masked letters, starting positions, and ending positions of masked letter intervals need to be stored. Quality values for unmasked letters are a fixed value (see Section 3). Thus, the total space in-

involved in quality values is small enough to be stored in memory. As mentioned in Section 4.2.2, during seed extension, we can either reconstruct query sequences from the hash table or read them from disk. The latter case involves additional I/O cost. Hence, we discuss the memory allocation scheme based on these two cases.

(1) Reconstruction case: Since the query sequences are reconstructed from the hash table, we only need to assign one page to them (i.e., $M_3 = 1$). M_1/C pages of the query set are brought into memory to build a hash table. The database is then read into memory in chunks of M_2 pages and scanned to find seeds for these query sequences. Extending seeds does not require extra I/O since query sequences are reconstructed in memory. This process is repeated $\frac{Q}{M_1/C}$ times by reading a new subset of the query set into memory. Therefore, the total I/O cost can be formulated as:

$$\frac{Q}{M_1/C} \cdot (st+rt) + Q \cdot tt + \frac{Q}{M_1/C} \cdot \left(\frac{DB}{M-M_1-2} \cdot (st+rt) + DB \cdot tt \right).$$

(2) Reading case: Query sequences are read from disk for seed extension. Thus, M_3 is not fixed any more. The I/O cost to read the query set to build the hash table and to read the database is the same as the reconstruction case. The worst case I/O of reading the query set for seed extension happens when the query set needs to be read once for each M_2 pages of the database currently stored in memory. In this case, the query set needs to be read DB/M_2 times. Hence, the total I/O cost can be formulated as:

$$\frac{Q}{M_1/C} \cdot (st+rt) + Q \cdot tt + \frac{Q}{M_1/C} \cdot \left(\frac{DB}{M_2} \cdot (st+rt) + DB \cdot tt \right) + \left(\frac{Q}{M-M_1-M_2} \cdot (st+rt) + Q \cdot tt \right) \cdot \frac{DB}{M_2}.$$

Taking the derivatives of equations gives us the optimal values of M_1 , M_2 , M_3 , and M_4 , i.e., when the total I/O cost is minimized for each case. As we show later in Section 5, reconstruction significantly reduces the I/O, and thus, the total cost.

5 Experimental Evaluation

This section evaluates QDPS and QHTS. We perform both quality and performance experiments. In quality experiments, we compute the search accuracy and reconstruction error. In performance experiments, we evaluate the space usage and the running time.

LCR-identification algorithms: We used three LCR-identification tools: SEG, CARD, and GBA. We downloaded the first two, and implemented the last one. We used their default parameters. The average Jaccard coefficients of SEG, CARD, and GBA were 0.20, 0.19, and 0.25 respectively.

Sequence comparison algorithms: We implemented three masking strategies: (1) *No masking*: LCRs are not identified. (2) *Boolean masking*: Identified LCRs are completely removed from sequences. (3) *Probabilistic masking*: The former two are the existing strategies

whereas the last one is the proposed strategy. We implemented DPS and HTS as described in Section 2. We name their no masking and boolean masking versions as *NDPS*, *BDPS*, *NHTS*, and *BHTS* respectively. We set the gap open and extension penalties as -10 and -0.5 respectively. We set the neighbor threshold, the extension threshold, and the k -gram length in all the cases as 11, seven, and three respectively as these are the default of BLAST. All programs are coded in Java. All experiments were run on a Linux machine with 1 GB memory and 2.53GHz CPU.

We tested different versions of BDPS, BHTS, QDPS, and QHTS on the three LCR-identification methods. For readability, we only show a small result set. In our plots, we consistently chose one case with one, two, or three LCR-identification algorithms. For one algorithm, we chose GBA, as it had the best Jaccard coefficient and precision. For two algorithms we picked SEG and CARD, as all the combinations of two algorithms were similar. We use the reconstruction strategy in our reported QHTS results unless otherwise stated since reconstruction has better performance. For convenience, we add the name of an LCR-identification method as a suffix with a dash to the masking strategy. For example QDPS-GBA denotes the QDPS masking strategy using GBA as the LCR-identification algorithm. When multiple LCR-identification algorithms are used, we will only use their initial letters. For example SC means using SEG and CARD.

Dataset: For accuracy evaluation, we downloaded 60,000 protein sequences from Swissprot as our database. We randomly picked 50 of them as our query set. According to the InterPro database [2], all these sequences belong to one or more families. We identify two sequences as biologically similar if they belong to the same family according to InterPro. The reason is because biologists have integrated various sequence-cluster databases into InterPro. Thus, InterPro provides a biologically reliable classification as the contributing databases complement each other in grouping protein families [7]. Each sequence in our query set comes from one or two families out of 29 families. The average number of sequences in our database with the same families as a query sequence is 64. The minimum, the maximum, and the standard deviation are 2, 1000, and 142 respectively. For performance comparison, we created seven sets of 6000, 3000, 1500, 750, 375, 188, and 94 sequences from the 60,000 sequences.

We also created a dataset of 20,714 protein sequences from PDB (www.rcsb.org/pdb/) for accuracy evaluation. We randomly picked 50 of them as our query set. According to SCOP database, each query sequence belongs to a different super family. We identify two sequences as biologically similar if they belong to the same super family according to SCOP. The average number of sequences in our database with the same super family as a query sequence is 34. The minimum and the maximum are 10

Table 1: Average total number of database sequences returned and precision (in percentage) for NDPS, BDPS-SEG, QDPS-GBA, QDPS-SC, and QDPS-SCG on the Swissprot dataset. The best result of each threshold is shown in **bold**.

score cutoff		25	50	75	100
Result Set Size	NDPS	55897	7775	708	171
	BDPS-SEG	55205	7611	611	108
	QDPS-GBA	25059	194	50	34
	QDPS-SC	3147	63	36	30
	QDPS-SCG	8861	102	44	34
Precision (%)	NDPS	0.06	0.43	4	17
	BDPS-SEG	0.06	0.44	5	27
	QDPS-GBA	0.13	17	60	85
	QDPS-SC	1.08	47	79	89
	QDPS-SCG	0.39	30	68	82

and 100 respectively.

We randomly sampled 121 sequences from Swissprot and created 24 bins with 5 points in each bin (except the last one). None of the queries is included in the sample set. We used the 2-gram complexities of the identified LCRs of each sample to predict quality values (see Section 3).

5.1 Evaluation of accuracy

We say that a sequence satisfies a threshold for another sequence under a similarity search algorithm if their alignment score returned by this algorithm is greater than the threshold. To evaluate the accuracy of each search algorithm, we used 25, 50, 75, and 100 as the threshold. This is because the default parameters of BLAST returns alignments with score at least 25.

For each query sequence, q , let TP (True Positive) be the number of database sequences that satisfy the threshold with q using the underlying similarity search algorithm and belong to the same family as q . Similarly, let FP (False Positive) be the number of database sequences that satisfy the threshold, but do not belong to the same family as q . Also, let FN (False Negative) be the number of database sequences that do not satisfy the threshold, but belong to the same family as q . We compute two measures: precision and recall as follows:

- precision = $TP / (TP + FP)$;
- recall = $TP / (TP + FN)$.

Evaluation of the DP methods: We compare our proposed strategy to existing strategies when all methods have full sensitivities (i.e., they find the best results according to their underlying scoring schemes).

Table 1 compares the average total number of database sequences returned and precision for various thresholds on the Swissprot dataset. For each threshold, the best result is shown in **bold**. QDPS (the last three methods) return significantly fewer results than DPS (the first two meth-

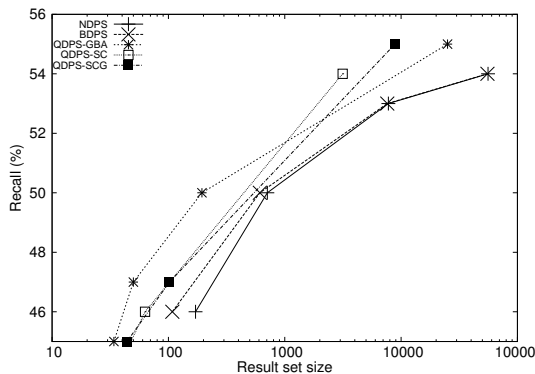


Figure 5: Result set size versus recall (in percentage) for NDPS, BDPS and QDPS on the Swissprot dataset.

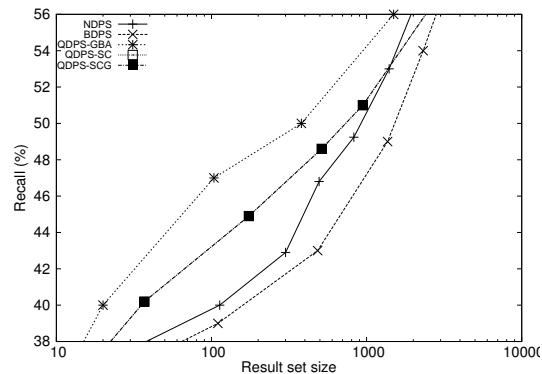


Figure 6: Result set size versus recall (in percentage) for NDPS, BDPS and QDPS on the PDB dataset.

Table 2: Average total number of database sequences returned and precision (in percentage) for NDPS, BDPS-SEG, QDPS-GBA, QDPS-SC, and QDPS-SCG on the PDB dataset. The best result of each threshold is shown in **bold**.

score cutoff		25	50	75	100
Result Set Size	NDPS	16010	1397	113	22
	BDPS-SEG	15808	1365	110	22
	QDPS-GBA	4974	20	13	11
	QDPS-SC	946	13	11	11
	QDPS-SCG	946	13	11	11
Precision (%)	NDPS	0.16	1	12	57
	BDPS-SEG	0.16	1	12	57
	QDPS-GBA	0.42	65	95	95
	QDPS-SC	1.8	92	95	96
	QDPS-SCG	1.8	92	95	96

Table 3: Average total number of database sequences returned and precision (in percentage) for NHTS, BHTS-SEG, QHTS-GBA-Read, QHTS-SC-Read, and QHTS-SCG-Read on the Swissprot dataset. The best result for each threshold is shown in **bold**.

score cutoff		25	50	75	100
Result Set Size	NHTS	43218	198	47	33
	BHTS-SEG	40909	81	31	29
	QHTS-GBA	6979	41	30	26
	QHTS-SC	162	26	23	20
	QHTS-SCG	773	34	27	22
Precision (%)	NHTS	0.08	15	61	81
	BHTS-SEG	0.08	37	90	94
	QHTS-GBA	4	93	100	100
	QHTS-SC	16	93	97	100
	QHTS-SCG	4	83	95	98

ods). The difference grows as threshold decreases. QDPS return 0.8-45 % of the sequences returned by NDPS. As threshold increases, the precision of all the methods increases. QDPS-SC and QDPS-SCG usually have better results in all the experiments than QDPS-GBA. At threshold 50 the precision of QDPS-SC is 30 % better than QDPS-GBA. The superior precision of QDPS indicates that our quality based alignment eliminates almost all the false positives while the traditional methods cannot. Figure 5 shows the relationship between result set size and recall for each method on the same dataset. We see that when all methods have the same recall, QDPS have much smaller result sizes. Hence, QDPS reduces the number of false negatives and false positives over DPS significantly. We obtained similar results on the PDB dataset (Table 2 and Figure 6).

Evaluation of the HT-based methods: We compare our proposed strategy to existing strategies on the Swissprot dataset when they do not have full sensitivities.

Table 3 compares the average total number of database sequences returned and precision for various thresholds. The best result for each threshold is shown in **bold**.

QHTS (the last three methods) return fewer results than HTS (the first two methods). The difference is more significant for small thresholds. At threshold 25 QHTS return 0.4 to 16.1 % of those of NHTS depending on the LCR-identification strategy. QHTS have higher precision than HTS. As threshold increases to large values, all the methods perform roughly the same. Figure 7 shows the relationship between result set size and recall for each method. We can see that when all methods have the same recall, QHTS-GBA has much smaller result size. Hence, it reduces the number of false negatives and false positives over DPS significantly. Although compared with HTS methods, QHTS-SC and QHTS-SCG have large results sizes at the same recall level, the recall drops slightly at the same result size. Hence, we conclude that they eliminate a large number of false positives at the expense of a small number of false negatives.

Figure 8 demonstrates how the proposed strategy compares to BLAST by focusing on a protein, APOA4_MOUSE, from our Swissprot query set. This protein contains 390 amino acids, where 269 of them are annotated as in LCRs. We ran BLAST to align this query to the proteins in our

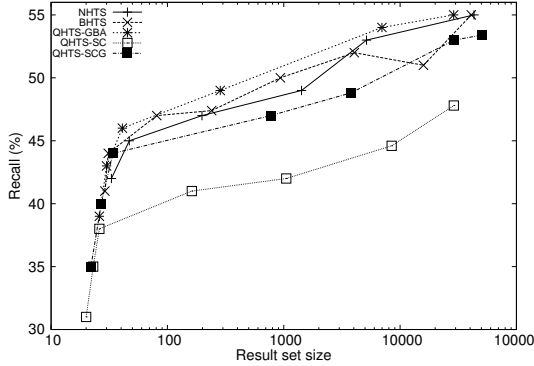


Figure 7: Result set size versus recall (in percentage) for NHTS, BHTS and QHTS on the Swissprot dataset.

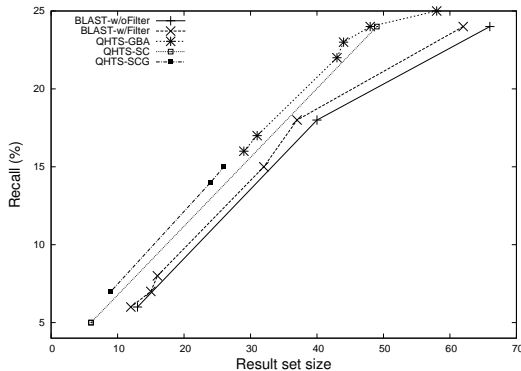


Figure 8: Result set size versus recall (in percentage) for QHTS and BLAST with the LCR-filter off (BLAST-w/oFilter) and on (BLAST-w/Filter) for querying the protein sequence, APOA4_MOUSE, against our Swissprot database.

Swissprot database. We used the default parameters of BLAST with and without using the “filter low complexity” option. We also aligned this query using QHTS methods. The results show that QHTS methods have higher recall at the same result set size. Their differences are more pronounced than those in Figure 7.

Evaluation of reconstruction and reading: As discussed in Section 4.2.2, reconstruction for seed extension may cause sensitivity to drop. Here, we compare three pairs of reconstruction versions and reading versions of QHTS on the Swissprot dataset: (1) QHTS-GBA-Reconstruct, QHTS-GBA-Read, (2) QHTS-SC-Reconstruct, QHTS-SC-Read, (3) QHTS-SCG-Reconstruct, QHTS-SCG-Read.

Table 4 presents the average recall and precision. For each pair, the reading case always has slightly higher recall value than the reconstruction case with a maximum difference of 3%. This is expected since reading case reads sequences from disk and no letter is lost. Reading case has higher precision sometimes, depending on the used LCR-identification tools. However, the differences are small,

Table 4: Average recall and precision (in percentage) of QHTS-GBA-Reconstruct, QHTS-GBA-Read, QHTS-SC-Reconstruct, QHTS-SC-Read, QHTS-SCG-Reconstruct, and QHTS-SCG-Read on the Swissprot dataset. To save space, we omit “QHTS” and use “Rec.” to represent “Reconstruct” in the table.

		score cutoff			
		25	50	75	100
Recall (%)	GBA-Rec.	52	45	42	39
	GBA-Read	53	45	42	39
	SC-Rec.	38	36	32	28
	SC-Read	41	38	35	31
	SCG-Rec.	45	42	38	33
	SCG-read	47	43	40	34
Precision (%)	GBA-Rec.	0.5	73	91	96
	GBA-Read	3.5	92	99	99
	SC-Rec.	21	96	98	99
	SC-Read	16	93	98	99
	SCG-Rec.	5	84	96	98
	SCG-read	4	83	95	98

Table 5: Relative Information Loss regarding the length of k -grams (in percentage).

k	QTS-GBA	QTS-SC	QTS-SCG
2	6.9	7.2	7.3
3	2.9	2.7	2.8
4	1.3	1.3	1.3

especially when score cutoff gets bigger. Thus, the accuracy of reconstruction case is comparable to reading case.

Lost information by reconstruction: As mentioned in Section 4.2.2, we may lose letters during reconstruction. Here, we evaluate the amount of lost information. We define *Relative Information Loss* measure for this purpose. Let $a = a_1 a_2 \dots a_n$ be a sequence. Let q_1, q_2, \dots, q_n be quality values associated with each letter in a . Assume that letters $a_{\pi_1}, a_{\pi_2}, \dots, a_{\pi_m}$ are lost ($\{\pi_1, \pi_2, \dots, \pi_m\} \subseteq \{1, 2, \dots, n\}$). Formally, we define Relative Information Loss as: $\sum_{i=1}^m q_{\pi_i} / \sum_{i=1}^n q_i$.

Table 5 shows how Relative Information Loss changes regarding the length of k -grams on the 1,500-sequence set. As mentioned in Section 4.2.2 of the paper, the bigger the k , the smaller the possibility that a letter is lost. When $k = 3$, the Relative Information Loss is about 3%. As we present in Section 5.2 of the paper, this number is very small compared to the percentage of k -grams evicted from the hash table. Thus, very little information is lost during reconstruction.

5.2 Performance comparison

In this section, we evaluate the performance of our quality based search methods. All tests are self-comparisons.

CPU time comparison: We compare CPU times in

Table 6: CPU times spent in HTC, SP, and AP, and the number of k -grams stored in the hash tables for (I) NHTS, (II) BHTS-SEG, (III) QHTS-GBA-Reconstruct, (IV) QHTS-SC-Reconstruct, and (V) QHTS-SCG-Reconstruct.

		I	II	III	IV	V
CPU time [sec]	HTC	6	5	5	5	6
	SP	11968	8615	8252	7293	8513
	AP	1269	895	343	13	46
	total	13240	9516	8601	7311	8565
# of k -grams		833792	657868	650147	481070	545485

the three phases of hash table-based searches: probabilistic hash table construction (HTC), search phase (SP), and alignment phase (AP). We tested five strategies: NHTS, BHTS-SEG, QHTS-GBA-Reconstruct, QHTS-SC-Reconstruct, and QHTS-SCG-Reconstruct. The CPU time of QHTS for reconstruction and reading cases are almost the same. Thus, we only present the results for reconstruction here.

Table 6 shows the results on the 1,500-sequence set. HTC take much less time than SP and AP in all cases. This is because the running time of HTC is linear in the query set size. Thus, introducing quality values takes HTC negligible time. SP dominates in all cases. Each k -gram in the database requires one or more hash table lookups. We only extend seeds. Hence, AP takes much less time than SP. NHTS has the largest SP and AP time since other methods do not store all k -grams in the hash table. The total time for NHTS is thus significantly larger. BHTS-SEG has the second largest SP and AP, hence the second total. This is because BHTS-SEG eliminates k -grams only from identified LCRs whereas QHTS eliminate k -grams from any place with some probability.

CPU time versus I/O time: For small datasets, CPU time dominates the overall running time of the search tools, including BLAST. Here, we demonstrate that I/O cost increases much faster than CPU cost for growing datasets. Thus, it will be the dominating term in the near future. We also show that I/O cost is significantly reduced by our probabilistic hash table and reconstruction strategy.

We used BLAST with LCR-filter on as a representative to existing BHTS strategies. We ran BLAST on the seven datasets with $n, 2n, \dots, 64n$ sequences by doubling the dataset size. The largest dataset contained 6,000 sequences. We performed a self comparison of each dataset using BLAST and measured the CPU times. Figure 9 plots the CPU times in seconds as a function of the dataset size in log-log scale. We then calculated the optimal I/O costs of reading and reconstructing strategies, for increasing dataset sizes, using the formulas in Section 4.2.4. For this computation, we assumed a fixed

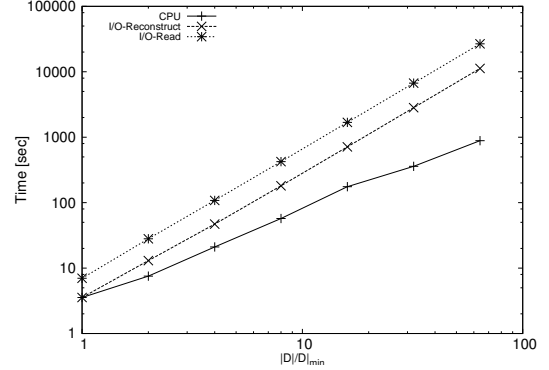


Figure 9: BLAST CPU time and QHTS I/O times for datasets of increasing size.

memory of $M = 100,000$ pages, with page size of 4 kB. We computed the I/O times for self comparisons of the seven datasets of size $M, 2M, \dots, 64M$, by doubling the dataset size. We used $st = 8, rt = 2$, and $tt = 0.09$ (in milliseconds) as I/O parameters for they are typical numbers for current architectures. We experimentally obtained the parameter C as 3.14 from our datasets. In order to visualize the difference between the I/O and the CPU trends better, we scaled all the I/O times down by a constant so that the I/O time of the reconstruction strategy is the same as that of the CPU time for the smallest dataset. The results are presented in Figure 9. Two important observations follow from this figure. First, I/O time grows much faster than CPU time with growing dataset size. Furthermore, Moore’s law suggests that the gap between the increase in I/O and CPU costs will be even more than our results in the future. We conclude that, I/O cost needs to be optimized for large datasets. Second, the I/O cost of the reconstruction strategy is 42 % of that of the reading strategy. Thus, from this figure and Table 4 of the paper, we conclude that the proposed reconstruction strategy reduces the overall running time of existing NHTS strategies by 45 to 58 % depending on the dataset and memory sizes. The improvement is more significant for larger datasets. One important note is that even for the cases where at least one dataset can fit in main memory, QHTS is still cost-effective. The time saving moves one level up along the computer storage hierarchy. Instead of saving the I/O cost, the communication between CPU and main memory is reduced dramatically. The relationship between the CPU cost and this communication cost follows the same pattern as shown in Figure 9.

Hash table size comparison: We evaluate the space usage of our quality based search methods. We measure this usage as the number of k -grams in the hash table as this is the largest data structure to be kept in main memory. A pointer is stored for each k -gram. Thus, memory usage is proportional to the number of k -grams in the hash table.

Table 6 compares the number of k -grams stored in the hash table for NHTS BHTS-SEG, QHTS-GBA, QHTS-SC, and QHTS-SCG on the 1,500-sequence set. NHTS has the largest number. This is because it stores all k -grams unlike other methods. Hash table sizes of QHTS methods are 57-77% of that of NHTS and 73-98% of that of BHTS. This is because BHTS eliminates k -grams only from masked LCRs whereas QHTS eliminates any k -grams with some probability. This probability depends on how the LCR-identification tools perform. This explains why there is a difference between QHTS methods. We conclude that QHTS can answer much larger query sets in main memory than NHTS.

The larger the hash table, the more hash table lookups, the longer SP takes. Table 6 shows that this is true. QHTS-GBA evicts 23% of the k -grams from the hash table, but its information loss is merely 6%. This justifies our reasoning that (1) the chance that all k -grams containing the same letter are deleted is very small, and (2) usually, letters with low quality values are lost during reconstruction.

6 Conclusion

We considered the problem of finding similar sequences when the locations of the LCRs are not known precisely. We developed a formulation to measure the quality of each letter. The quality value of a letter is the probability for that letter to be in a non-LCR. We showed that the quality values can be used in two well known approaches to the sequence search problem. The former finds the optimal alignment of two sequences using dynamic programming. This applies to the case when the sequences are small. The latter computes a suboptimal alignment using a hash table. This applies to the case when the database or both the database and the query set are large. We developed a method that indexes k -grams (sequences of length k) using a hash table probabilistically. As a result, the main memory usage, the CPU cost, and the I/O cost are greatly reduced. We also showed that this hash table can be used to reconstruct query sequences with negligible information loss. This eliminates the need for further disk I/Os to read these sequences. In our experiments on real data, our quality-based similarity search algorithms reduced the number of false positives drastically. In addition, their running times were better than existing strategies. In the future, we would like to explore formulation of P-value for resulting alignments when quality values are involved.

References

- [1] S. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman. Basic Local Alignment Search Tool. *JMB*, 215(3):403–410, 1990.

- [2] R. Apweiler and et al. Interpro—an integrated documentation resource for protein families, domains and functional sites. *Bioinformatics*, 16(12):1145–1150, 2000.
- [3] S. Brenner, T. Hubbard, A. Murzin, and C. Chothia. Gene duplications in *H. influenzae*. *Nature*, 378(9):140, 1995.
- [4] A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. Whited, and D. Salzberg. Alignment of Whole Genomes. *NAR*, 27(11):2369–2376, 1999.
- [5] M. V. M. Halkidi, Y. Batistakis, and M. Vazirgiannis. On clustering validation techniques. *JGIS*, 17:107–145, 2001.
- [6] X. Huang and A. Madan. CAP3: A DNA Sequence Assembly Program. *Genome Research*, 9(9):868–877, 1999.
- [7] E. V. Kriventseva, M. Biswas, and R. Apweiler. Clustering and analysis of protein families. *Current Opinion in Structural Biology*, 11(3):334–339, 2001.
- [8] S. Kurtz and C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, May 1999.
- [9] M. Levitt and M. Gerstein. A unified statistical framework for sequence comparison and structure comparison. *PNAS*, 95(11):5913–5920, 1998.
- [10] X. Li and T. Kahveci. A novel algorithm for identifying low-complexity regions in a protein sequenc. *Bioinformatics*, 22(24):2980–2987, 2006.
- [11] D. J. Lipman and W. R. Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 227(4693):1435–1441, 1985.
- [12] M. Ma, J. Tromp, and M. Li. PatternHunter: Faster and More Sensitive Homology Search. *Bioinformatics*, 18(0):1–6, 2002.
- [13] E. W. Myers and W. Miller. Optimal alignments in linear space. *Comput. Appl. Biosci.*, 4(1):11–17, 1988.
- [14] C. Shannon. Fast Incremental Maintenance of Approximate Histograms. In *Bell Syst. Tech. J.*, pages 50–60, 1951.
- [15] S. W. Shin and S. M. Kim. A new algorithm for detecting low-complexity regions in protein sequences. *Bioinformatics*, 21(2):160–170, 2005.
- [16] T. Smith and M. Waterman. Identification of common molecular subsequences. *JMB*, 147:195–197, 1981.
- [17] D. States and P. Agarwal. Compact Encoding Strategies for DNA Sequence Similarity Search. In *ISMB*, 1996.
- [18] T. Tatusova and T. Madden. BLAST 2 Sequences, A New Tool for Comparing Protein and Nucleotide Sequences. *FEMS Microbiology Letters*, 177:247–250, 1999.
- [19] H. Wan, L. Li, S. Federhen, and J. Wootton. Discovering simple regions in biological sequences associated with scoring schemes. *JCB*, 10:171–185, 2003.
- [20] J. Wootton. Sequences with ‘unusual’ amino acid compositions. *Current Opinion in Structural Biology*, 4:413–421, 1994.
- [21] J. Wootton and S. Federhen. Analysis of compositionally biased regions in sequence databases. *Methods in Enzymology*, 266:554–571, 1996.