FLASH: A Fast Look-Up Algorithm for String Homology

Andrea Califano and Isidore Rigoutsos
IBM T.J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598
acal@watson.ibm.com, rigoutso@watson.ibm.com

Abstract

A key issue in managing today's large amounts of genetic data is the availability of efficient, accurate, and selective techniques for detecting homologies (similarities) between newly discovered and already stored sequences. A common characteristic of today's most advanced algorithms, such as FASTA, BLAST, and BLAZE is the need to scan the contents of the entire database, in order to find one or more matches. This design decision results in either excessively long search times or, as is the case of BLAST, in a sharp trade-off between the achieved accuracy and the required amount of computation.

The homology detection algorithm presented in this paper, on the other hand, is based on a probabilistic indexing framework. The algorithm requires minimal access to the database in order to determine matches. This minimal requirement is achieved by using the sequences of interest to generate a highly redundant number of very descriptive tuples; these tuples are subsequently used as indices in a table look-up paradigm.

In addition to the description of the algorithm, theoretical and experimental results on the sensitivity and accuracy of the suggested approach are provided. The storage and computational requirements are described and the probability of correct matches and false alarms is derived. Sensitivity and accuracy are shown to be close to those of dynamic programming techniques.

A prototype system has been implemented using the described ideas. It contains the full Swiss-Prot database rel 25 (10 MR) and the genome of E. Coli (2 MR). The system is currently being expanded to include the complete Genbank database. Searches can be carried out in a few seconds on a workstation-class machine.

1. Introduction

Conventional techniques for matching arbitrary data structures are based on similarity criteria and usually proceed sequentially by comparing the item of interest to each one of those stored in some database. Recently, a new alternative paradigm has been suggested in the field of computer vision, that of indexing [10, 6]; systems based on this paradigm have successfully dealt with visual databases containing many complex objects.

In domains outside that of computer vision, and especially in the area of string similarity searches, most of the advances to the state of the art have concentrated on speeding up the single item-to-item comparison operation. Pertinent examples include the introduction of: (a) local hash-table techniques [12], (b) ad-hoc heuristics intended to prune the search [2], etc. One problem with the use of heuristics is the resulting degradation of the accuracy and sensitivity of the corresponding techniques. This observation not withstanding, most algorithms require the scanning of the entire collection of data, thus becoming increasingly ineffective with larger, unstructured databases. This is particular true in the field of Genetics where publicly available databases such as *Genbank* already include close to 100,000,000 residues (nucleic acids or aminoacids); the sizes of these databases are expected to exceed the 2 GR mark by the end of the decade.

In this paper we introduce a general probabilistic framework for fast, indexing-based, similarity searches of large databases of strings. Recovery of similar database items matching a test sequence necessitates minimal data access thanks to the use of a redundant table-lookup paradigm. In particular, a large number of very descriptive indices is generated from each portion of a given string. During the creation of the database, references to the string are associated with such indices using a large look-up table structure. During the recall, indices are computed in an identical manner from the test sequence to be matched against the database. All the stored references to the original strings are then recovered from the look-up table; a final histograming step integrates the evidence for different matching sequence hypotheses.

Due to noise and other sources of uncertainty, only a certain percentage of the indices is typically recovered. Theoretical results will be used to highlight the following two major points: first, very descriptive indices (i.e., indices with a large range of possible values) are necessary in order to achieve computational efficiency and reduce the chances of false positives occurring. Second, the use of very descriptive indices requires a high density of indices per string token so that relevant matches will not be missed. The latter condition is the reason why conventional index generation techniques such as those used in FASTA and BLAST cannot be used within this framework: since these techniques form indices by using fixed-length tuples of contiguous characters (residues), they cannot generate the high index density that is necessary for the correct behavior of the technique.

In the sequel, a new index generating mechanism is introduced: k-tuples of tokens are formed by deterministically concatenating non-contiguous sub-sequences of tokens extending over a large portion of the string of interest; finally, the k-tuples are associated with index values. This method effectively produces indices which will reflect any long-range correlated information that may be present in the strings. By appropriately choosing a deterministic or random index generation function, it is possible to achieve arbitrary index densities; this step is crucial to the accuracy and efficiency of the approach.

Once the various algorithm parameters have been determined for an expected maximum database size, the computational requirements for the matching will be linear in the number of tokens stored in the database. The linearity coefficient is very small: indeed, matching a string of a given length to a database containing 4 billion nucleotides would require about 4 times the computation needed for a database containing a single nucleotide.

The algorithm has already been implemented on workstation-class machines and PCs On a single RS6000 530, a 100 residue DNA sequence can be matched against the genome of *E.Coli* in about 15 seconds. The time needed for matching strings to the entire *Genbank* database will be slightly higher, ranging up to 24 seconds. It should be mentioned here that BLAST with default parameters settings currently requires about 5 minutes for a similar search. Aside from its speed, FLASH is garanteed to recognize at least 99% of the correct homologies for a test DNA sequence of 100 nucleotides containing 30 mutations; for comparison purposes, we mention that under identical conditions BLAST is only guaranteed to recognize about 29.4% of them. Similarly a 100 residue protein can be matched against the Swiss-Prot database in about 60 seconds with tolerance for exact and conservative aminoacid matches.

We have also implemented a distributed version of the algorithm to detect protein homologies. With the latter approach, by dividing the database on 8 hard disks and using only about 5% CPU time from 1 to 8 workstations, we can index into the entire Swiss-Prot database in 5 to 8 seconds for matches to a 100 residue strand. Further optimization should bring the time int the subsecond range.

The matching algorithm presented here is of a very general nature and it could be adapted to solve a number of other relevant biology problems that require pattern-matching. By changing the deterministic index generation mechanism to suit specific domain-dependent data structures, the algorithm can be easily extended to other fields and different topologies. As an example, we mention that work is under way to use FLASH in speech recognition, for retrieving programming language structure and *clichés* from partial or similar queries, and in the computer vision domain for fast 3D-object model matching based on graphs of model features and relationships. The two latter problems fall in the category of sub-graph to graph similarity matching.

In section 4 we provide an informal discussion of the approach. An in-depth presentation of the algorithm is presented in sections 5 and 6. Finally, implementation details are given in Section 7.

2. Detecting Homologies

For the purposes of our discussion, let us define χ as a string $(v_0, v_1, \ldots, v_{N-1})$ of tokens, of length N. Each token can have one of τ possible values. For instance, in the case of DNA which is formed exclusively with four nucleic acids (the "bases" A,C,G, and T), $\tau = 4$. For proteins, $\tau = 20$; indeed, the proteins are chains formed using twenty basic aminoacids. In the following, we will use the term "sequence" to indicate one of these strings of nucleotides or aminoacids.

For simplicity, let us assume that the database is structured as a single string of tokens χ_0 of length N_0. The goal here is the determination of one or more locations in χ_0 where sub-sequences matching a given reference sequence χ_r of length N_r can be found. In other words, we wish to determine a set of homology positions $X \equiv \{x_j\}$ in χ_0 with the following property: for all j, χ_r is similar to the sub-sequence of χ_0 which begins at position u_j and has length N_r; we will denote this fact by $(\chi[u_j, N_r])$, for short. By "similar" we mean that a small number $(m_j \le m_0)$ of token mutations (i.e. insertion, deletion or modification) is required to change $\chi[u_j, N_r]$ into χ_r. Here, m_0 is a predefined maximum number of mutations. Additionally, we want the positions x_j to be ranked with respect to the value of m_j in such a way that sequences requiring fewer mutations will have higher rank.

Many techniques are available for performing this type of analysis. However, all of the currently available algorithms, i.e. the original Lipman-Pearson [11], and its more recent incarnations FASTP [12], FASTA [3], and BLAST [2], and Smith-Waterman [5], require a scan of the entire database, χ_0, in order to determine one or more matches.

3. A Probabilistic Indexed Approach

The key idea behind implementing a probabilistic indexing framework is the generation of a large set of highly descriptive indices $\Lambda(x) = \{\lambda_i(x); i = 1 \ldots d_l\}$ for each token position x in χ_0. The corresponding value x is then appended to each entry of a look-up table A indexed by the indices λ_i. The process is repeated for all successive x values until the string χ_0 has been exhausted. The parameter d_l is the index density, i.e. the number of indices generated for each sequence token.

During recognition, given a reference sequence χ_r to be matched against χ_0, the reversed approach is used. For each token position y in χ_r, indices are generated with the same algorithm that was used at storage time. For each generated index, λ_i, the contents of a look-up table location are retrieved. the retrieved entries correspond to the set of token positions $\{x_j\}$ in χ_0 which generated the same index value

AA		CA		GA	11,20	TA	7
AC	1	CC	2,15,16,17	GC	14	TC	
AG		CG		GG	5	TG	4,10,13,19
AT	8,12	CT	18	GT	3,6	TT	9

Table. 1: Contiguous subsequences

at storage time: for each x_j, the value $w_j = x_j - y + 1$ corresponds to the position in χ_0 where the first token of χ_r should be positioned so that the tokens that have produced identical indices are also properly aligned. By histograming the values w_j, we can identify and rank similarity matches between χ_r and χ_0. We will use the term *votes* for a given value w_j to indicate the corresponding height in the histogram.

Consider the nucleotide sequence **ACGTGGTATTGATGCCCCTGA**. For each token position we generate a single index as the value of two consecutive characters starting at that position. The generated position-index pairs will therefore be: (1,**AC**) (2,**CG**), (3,**GT**), ... (20,**GA**). If we arrange these values in a table with 16 entries (one for each possible index value) we obtain the values of Table 1.

If we now consider the reference string **GTACTGA** which is identical to the sub-sequence at position 6 of the original string except for the fourth character, we obtain the following position/index sequence:

[(1,GT) (2,TA) (3,AC) (4,CT) (5,TG) (6,GA)] (3.1)

By retrieving the values in the table above, subtracting the index positions, and histograming the absolute alignment positions we obtain the histogram of Table 2. Notice the peak at position 6: this is clearly the best alignment, with 6 matching characters out of 7 producing 4 matching indices. Notice also the small peak at position 15 (matching on the last 3 characters) with 2 index matches. Unfortunately this index selection strategy is not helpful in general for the following two reasons:

- Indices formed by two four-valued characters have a limited range of values (16 to be exact). Even with a 20-character string in the database, some of the entries will receive many votes (see the entries CC and TG): this observation poses two problems. First, it the algorithm becomes increasingly slower with larger database sizes; indeed, more and more values must be processed and histogramed for each index. Second, it makes the approach less accurate, since most of the values stored in the look-up table entries do not usually correspond to actual matches. As we will see in Section 5, the probability of detecting false positives (i.e. peaks in the histogram not corresponding to effective matches) increases.
- A maximum of one index per token can be obtained. As a consequence, minimal changes in the test string, drastically reduce the probability of correctly detecting a similarity: simply changing the first, fourth and seventh characters in the test string above suffices to destroy all the matching indices. The issue is discussed in detail in Section 5.

Votes	1	1	0	0	1	0	0	4	0	0	1	0	0	0	0	0	2	0	0	0	0
Pos.	-1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Table 2: Contiguous subsequences

To avoid the first problem, algorithms such as the Wilbur-Lipman reverse the mechanism altogether. A different look-up table is generated from the data in each test sequence rather than from the long original one. The entire database is subsequently matched to the test string using this small partial look-up table. The approach requires a scan of the entire database in conjunction with short index lengths in order to maximize the probability of a match; as we will show, this approach is bound to produce suboptimal results.

4. Long-range Indices

The answer to the above double *impasse* is the use of a different mechanism for index generation. This will allow us to use longer, more descriptive indices without any reduction in the probability of a match because of noise. As we will see, this step is crucial in order to guarantee a good behavior of the technique. Conventional techniques associate *k*-tuples with locally connected sub-sequences of *k* tokens (e.g. BLAST and FASTA) in order to perform rapid local hash table alignment. However, such an approach would not work within this framework: this is a direct consequence of the limited number of indices that can be associated with fixed length, contiguous substrings. This may also be the reason why direct indexing into large databases for similarity matches has not been massively employed in practice. This issue is examined in more detail in Section 5.

In order to generate a large number of *k*-tuples and the corresponding indices for a given token position, we define a discrete function which maps a small set of integer values to combinations of contiguous or non-contiguous tokens which start at the selected position. A simple mapping function could generate all possible ordered *k*-token combinations that are contained in an *l*-token interval of the interest string; the *k*-token combinations are assumed to contain the token at the beginning of the *l*-token interval.

Consider the first position on the string (**ACGTGGTATTGATGCCCCTGA**), an interval of $l = 5$ positions, and tuples composed of three tokens (3-tuples). The simple algorithm described above will generate the following six non-contiguous tuples: **ACG**, **ACT**, **ACG**, **AGT**, **AGG**, and **ATG**. Repeating the process over the entire string and considering that the last tokens generate fewer votes because of the finite size of the string, one will obtain a total of 106 tuples and their associated indices. For simplicity, we will use the term index and *k*-tuple interchangeably.

Had we wanted to generate 3-tuples from contiguous subsequences, we would have obtained only 18 of them. Figures 1 and 2 show the histogram values for the cases of non-contiguous and conventional indices respectively: observe how the correct alignment value receives 17 votes in one case and only 2 in the other.

This simple example illustrates the advantages of using higher index densities. Longer indices can be used without any degradation of the accuracy, i.e. without loosing possible matches. Compare the sharp peak of Figure 1 at position 6, clearly separated from the noise background, with the 2 votes in Figure 2 obtained for the same position with conventional indexing techniques. In the latter case, it is very difficult to differentiate between the 6-character match at position 6 and the 4-character match at position 15. Finally, notice how the peak at position -1. which corresponding to a 4-character match with one insertion, is not detected when contiguous k-tuples are used.

4.1. Databases Containing Multiple Strings

The previous discussion has focused on a database that contains one single very long string. The presented approach works in an identical way if the database contains multiple strings. The only difference is that storing the position of the first token of each k-tuple in the look-up table is not sufficient anymore. Indeed, one should store in the corresponding entry of the look-up table a symbolic label that uniquely identifies the string that generated the k-tuple, as well as the position of the first token of the k-tuple in that string. Since the two approaches are identical, in what follows we will concentrate on the first one.

4.2 Extra Information in the Indices.

We can achieve even better results than those described in Section 4. This can be done by considering that the non-contiguous k-tuples generated with the above described procedure contain more information than just the values composing them. To a certain extent, the relative position of the tokens can be used as well to separate accidental matches from real ones. For instance, with the above procedure, both strings AAACT and ACTAA would generate the index AAA. However, such a sequence is hardly indicative of a similarity. In some domains, such as DNA and Protein sequencing, there can be token deletions and insertions which disrupt any technique that relies strictly on the structure of the indices. Depending on the application, some use of index structure can be devised to increase the signal-to-noise ratio of the techniques. For example, in our previous example we could disregard all index matches that would require more than a preset number n_{id} of token insertions or deletions. With n_{id} respectively set to 0 and 1, we obtain the following histograms

Notice that the two peaks in the first image at position 6 and 15 are both indicative of relatively good similarity matches (a 6- and a 4-character match respectively). In cases where resilience to insertions and deletions is desired, a less restrictive use of structural information is required. For instance, the threshold of $n_{id} = 1$ allows also for the detection of the four character match with one insertion at position -1, see figure 4. The overall noise background in the other bins is still reduced with respect to figure 1.

To use this information, the shape of the k-tuples must be stored in the look-up table along with the position of its first token. If each k-tuple structure is the result of a deterministic process with n possible outcomes, one need simply store a numeric label with values between 1 and n in the look-up table; this would require $\log_2 n$ bits. For example, an index density of 32 could be therefore be handled with just five bits of storage.

4.3 Exploiting Domain Dependencies.

The mapping function suggested at the beginning of Section 4 represents a simple example. Algorithms for index generation which are more domain dependent can be identified. The common element of any such algorithm is that it must allow for the generation of a large number of different indices per token. For instance, in the DNA matching domain there is a natural map between *codons*, i.e. groups of three nucleotides, and the aminoacids that would result in the expressed protein. It is therefore convenient to generate the indices by first grouping the nucleic acids into possible codons and then combine (possibly non-connected) groups of k codons to form $3k$-tuples. For proteins, we can use the single aminoacids directly: very descriptive indices can be constructed with relatively few of these tokens. Notice that we made no assumption about the codon reading frame, i.e., the position of the first codon: all the (possibly overlapping) reading frames can be used to create frames, independently of one another.

By selecting $k-1$ pairs $\{(\rho_i^-, \rho_i^+); i = 1 \dots k-1\}$ of minimum and maximum *radii of coherence*, it is possible to limit the index density at an arbitrarily fine grain level. Recall that by index density, we mean the number of indices (k-tuples) that are generated by each token (nucleic acid) in the DNA strand. This control can be achieved as follows: for each token position x_0 in the DNA strand of interest, we

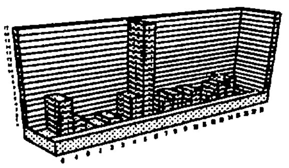


Fig 1: Non contiguous indices

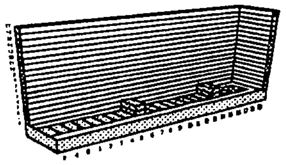


Fig 2: Conventional indices

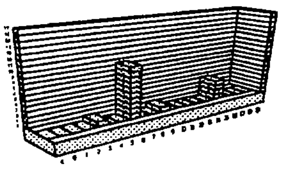


Fig. 3: $n_{id} = 0$

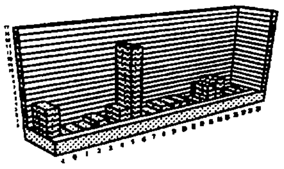


Fig. 4: $n_{id} = 1$

form a codon C_0 using the nucleotides found at positions x_0, x_0+1, x_0+2; we then, select a second position x_1 from the interval $\Delta_1 = [x_0 + \rho_1^-, x_0 + \rho_1^+]$ and form a codon C_1 using the nucleotides found at positions x_1, x_1+1, x_1+2. This procedure is repeated until the last codon C_k has been formed with the nucleotides at positions x_k, x_k+1, x_k+2, where x_k belongs to the interval $\Delta_{k-1} = [x_{k-2} + \rho_{k-1}^-, x_{k-2} + \rho_{k-1}^+]$. The index is formed by concatenating the k-1 codons obtained with the above procedure.

The above stage may be repeated for all the allowable values of the intervals $\Delta_1, \ldots, \Delta_{k-1}$, thus generating a possible grand total of

$$d_I = \prod_{i=1}^{k-1} (\rho_i^+ - \rho_i^-)$$

indices for each nucleotide from the original DNA strand. d_I is the index density.

Clearly, fewer index values are generated towards the end of the string. In our analysis we will ignore this effect of boundary conditions. Furthermore, if index lengths are required which are not multiple of 3, one or two additional nucleotides can be appended from an interval using yet another pair of radii of coherence.

4.4. Using *k*-tuples to Access The Look-Up Table

The k-tuples cannot be used directly to access entries in conventional look-up data structures such as arrays and hash tables. The usual approach calls for a unique integer to be associated with a each different k-tuple; in this way, the entire index set maps into a (possibly) compact subset of integer values. Compactness minimizes the number of unused entries and is thus desirable if the look-up table is structured as an array. If each token at position i of the k-tuple has an associated value $0 \le v_i < \tau$, a unique index γ can be produced as follows:

$$\gamma = \sum_{i=1}^{k} v_i \, \tau^{(i-1)} \qquad (4.1)$$

In the case of DNA strands, for example, $\tau = 4$, $\{A,C,G,T\} \to \{0,1,2,3\}$. The k-tuple **AATCGT**, for instance, would first translate into the string 003123 and then into the index value $0 \cdot 4^0 + 0 \cdot 4^1 + 3 \cdot 4^2 + 1 \cdot 4^3 + 2 \cdot 4^4 + 3 \cdot 4^5 = 3696$. For proteins, $\tau = 20$.

5. A Formal Discussion

The previous Section contained a high level description of the algorithm. In the sequel, we present a detailed analysis of the approach's performance and computational requirements as a function of a number of independent design parameters.

5.1. Storing a Sequence in the Database

Let us assume that the original sequence χ_0 contains N_0 tokens. If k is the length of the k-tuples used to generate the indices and if each token has τ possible values, then we will need a random-access structure that has an entry (bucket) for each possible index value $n_s = \tau^k$. A hash table, or an array **A**, will provide the optimal storage mechanism. Arrays will be more preferable for large databases if the indices are approximately uniformly distributed among the possible values. In this case, once enough k-tuples have been processed, most or all the entries in **A** will contain some information. Use of a sparse structure such as a hash table would prove inefficient. Hash tables are more efficient for small databases because only the entries that are effectively in use will need to be stored. Given the large volume of genomic databases, arrays with buckets of variable length will give the best results. In what follows, **A** will be considered to be an array with variable bucket sizescontaining a total of $n_s = \tau^k$ entries [1].

5.2. A Physical Model

Since the algorithm is of a probabilistic nature, we must clearly state the set of assumptions that it relies upon. First, the distribution of the indices generated from the string under consideration is assumed to be approximately uniform. If the sequence structure is not random, something that would seem logical in the case of DNA and proteins, this assumption might seem unrealistic. However, we must remember that the index generation mechanism effectively hashes the data by forming many independent combinations; as a result, the correlation of the indices is expected to decrease. In [7] we have shown that violations of this assumption do not have any noticeable effect on the performance of the approach.

Assumptions are also needed about the process that is the source of the observed differences among similar sequences. As a first approximation, we will assume that single-token mutations (i.e. insertion, deletion, and value change of individual tokens) are uncorrelated and uniformly distributed. In other words, the mutations are assumed to have been produced by a uniform, uncorrelated random process that operates on the entire string.

5.3. Index Distribution in the Look-Up Table

If d_I represents the index density, then a sequence of length N_0 will generate $N_0 \, d_I$ distinct indices. These indices will be stored in the look-up table **A**. We assume for the moment that the index values are uniformly distributed. Equivalently, the probability p_i that a specific index value i is generated is independent of the value i. For a full discussion of the non-uniform case refer to [7]. In the case of the uniform distribution, the probability $p_e(n)$ that an entry of **A** contains references generated by n independent indices (k-tuples) is given by the binomial distribution:

$$p_e(n) = \binom{N_0 \, d_I}{n} p_i^n \, (1 - p_i)^{N_0 d_I - n} \qquad (5.1)$$

In general, the table **A** will contain τ^k entries, one entry for each possible index value. Here, k is the number of tokens in

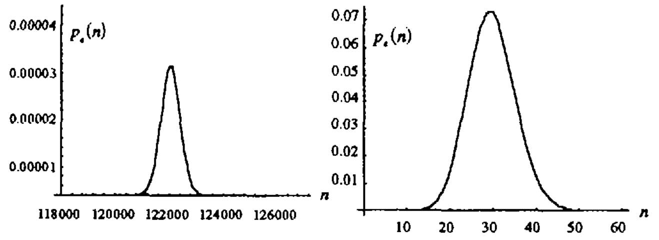


Fig. 5: $p_e(n)$ for k-tuples lengths of 7 and 13 tokens

the k-tuples and τ is the range of values of each token. $p_i = \tau^{-k}$ and the probability $p_e(n)$ can then be rewritten as

$$p_e(n) = \binom{N_0 d_l}{n}\left(\frac{1}{\tau}\right)^{nk}\left(1-\frac{1}{\tau^k}\right)^{N_0 d_l - n} \tag{5.2}$$

Figure 5 shows this distribution for the size of Genbank, and values of k equal to 7, 10, and 13. The average number of references stored in a look-up table entry is given by:

$$\overline{N}_e = \sum_{n=1}^{\infty} n\, p_e(n) = \frac{N_0 d_l}{\tau^k} \tag{5.3}$$

The average number of references per look-up table entry is extremely important for the efficiency of the approach: small values of $\overline{N}_e$ result in fast processing times; indeed, for each index generated during the matching process, only a small number of possible match candidates needs to be processed. This, in turn, helps speed-up the histograming step as well.

6. Retrieving Sequences from the Database.

After the preprocessing of the database has been performed, homologies can be detected by generating k-tuples and indices for a sequence of interest and by retrieving the corresponding references from the look-up table **A**. There are three fundamental elements which are relevant to the good behavior of the technique. These are *sensitivity, accuracy*, and *efficiency*. Sensitivity is the probability that an actual similarity is not detected. Accuracy reflects the probability that an erroneous match is detected. And finally, efficiency expresses how computationally expensive the technique is.

In this Section, we will address these questions in detail. It should be clear that in general, sensitivity, accuracy and efficiency are intimately related. Higher speed will in general result in lower accuracy and/or sensitivity. The reverse holds true as well. In what follows, we show that use of high densities and of highly descriptive indices increases the speed over conventional hashing techniques based on contiguous k-tuples, at no cost to the accuracy.

6.1. Sensitivity

Let us concentrate on mutations that can be modeled by a change to a token's value. As we described in Section 4, appropriate use of the index structure will provide resilience to token insertion and deletion.

Let us assume for the moment that a subsequence of length N_r has been extracted from the original sequence and that n_B of the tokens have been randomly assigned new values that are different from the original ones: we say that the sequence has been affected by n_B token mutations. If we consider all the possible k-tuples that can be formed from the string, the number of the k-tuples that are unaffected by mutations is:

$$T_G = \binom{N_r - n_B}{k} \approx \frac{1}{k!}(N_r - n_B)^k \tag{6.1}$$

The approximation relies on the fact that in most cases $N_r - n_b >> k$. If we denote by $r_g = (N_R - n_B)/N_R$ the ratio of correct tokens in the modified sequence, we can write

$$T_G \approx \frac{N_r^k}{k!} r_g^k \tag{6.2}$$

Then, the number of k-tuples that have different values is

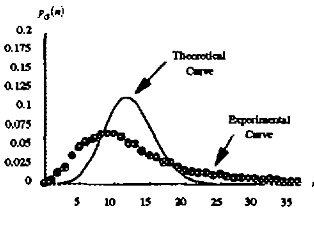
$$T_B = \binom{N_r}{k} - \binom{N_r - n_B}{k} \approx \frac{N_r^k}{k!}\left(1 - r_g^k\right) \tag{6.3}$$

If instead of using all possible k-tuples we randomly select only $N_r d_l$ of them, the probability that n of them will be unaffected by mutations can be written as:

$$p_G(n) \approx \binom{N_r d_l}{n}\left(r_g^k\right)^n\left(1 - r_g^k\right)^{m-n} \tag{6.4}$$

The approximation is valid if we consider that, in general, both T_G and T_B are much larger than m. This is identical to the binomial distribution we would obtain for the probability of having n correct k-tuples, under the assumption that each token remained unaffected with probability $p_G = r_g^k$. Notice that this is equivalent to assuming that the probability p_g that a k-tuple remains unchanged p_G is the product of the probability of each token of the tuple remaining unchanged, with $p_g = r_g$.

The value of p_G decreases exponentially with the length of the k-tuple. We will see that this is a fundamental problem which is also responsible for the observed degradation in sensitivity techniques such as BLAST. On one hand, as can be seen in Equation (5.3), larger values of k are necessary to make the approach work faster, on the other, from Equation (6.4), large values of k decrease the sensitivity of the approach. The solution that we propose is the use of higher



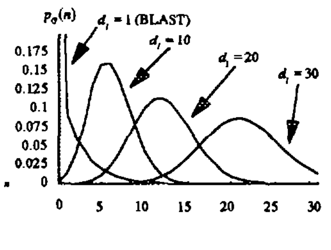


Fig. 6: Probability $p_G(n)$

Fig. 7: d_l = 1, 10, 20, 30

index density values by generating *non-contiguous* indices.

Theoretical values show a good match with experimental ones, see Figure 6 where the probability of getting n matching indices computed from (6.4) (continuous line) is compared with values obtained experimentally (dotted line). In this example, $N_r = 100$, $n_B = 30$, $d_I = 20$, and $k = 12$. The tokens of the 20 k-tuples are spread uniformly over an interval of 35 token positions. This seems to produce one of the best possible outcomes. The small disagreement of the experimental data with the theoretical ones is due to the particular selection of the k-tuples. As demonstrated in [7], quite different results can be expected depending on the particular selection of the d_I index configuration generated by each token. More appropriate strategies for index structure generation will minimize the correlation among the different indices, and are expected to produce values that are even closer to the theoretical curve. Even the simple-minded index selection strategy used in this case, however, leads to almost perfect recognition, as indicated by the experimental curve. That is, even for the chosen considerable ratio of approximately one mutation every 3 tokens, at least three correct indices are obtained in more than 98.5% of the search processes. I.e., less than 1.5% of the correct homologies is lost. As is shown in the next Section, a threshold of three matching indices allows FLASH to easily differentiate among correct and random homologies (see Figure 8). For k=13, FLASH requires a slightly higher index density, $d_I = 40$ in order to achieve experimentally miss rates lower than 1%.

We now compare these results with those obtained using BLAST for DNA matching; for this purpose, BLAST typically uses contiguous tuples of 12 nucleotides. The graph in Figure 7 compares results for BLAST (d_I = 1, leftmost curve) with results from FLASH (d_I = 10, 20, 30 respectively, curves from left to right) for the same values of the other parameter as in the previous example.

The plot clearly shows that, even with k-tuples of identical lengths, FLASH achieves significantly better sensitivity over BLAST. For this number of mutations (30 out of 100 tokens) and with a threshold set at a single correct index match, BLAST would miss more than 70% of the homologies. FLASH with $d_I = 20$ or 30 would recover almost all of them, even with higher match thresholds (e.g., 3). Higher thresholds will be shown to provide increased accuracy (Section 6.3). The complete analysis leading to this results can be found in [7].

6.2. Index Generation Schemes

In the previous sections, we examined the way FLASH exploits non-contiguous k-tuples to achieve better speed and sensitivity.

We next consider a specific example. Without loss of generality, we choose a tuple length k = 13 and index density $d_I = 40$. We form the 13-tuples by choosing 13 tokens from a sub-segment of length r = 40. We will refer to the specific

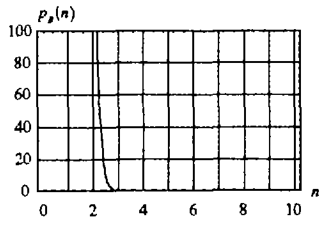


Fig. 8a: FLASH

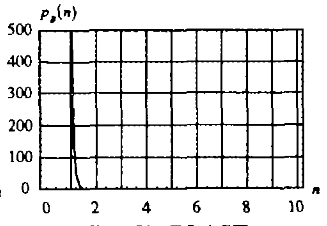


Fig. 8b: BLAST

d_I tuples selected for indexing as "fingerprints." Clearly, these will be chosen out of a total of $C_{13}^{40} = 12{,}033{,}222{,}880$ possible combinations of 13 tokens; these are all the possible ways one can choose 13 unique elements out of 40. The process by which one should select d_I of them is *not* trivial and can lead to very different performances of the method.

For a complete discussion of index generation techniques refer to [7], where we show that random index index selection techniques, in general, gives significantly better results than deterministic ones. Random index selection tends to form k-tuples that are composed of smaller contiguous sub-components. Such sub-components are far more frequent in degraded sequences: the probability of obtaining *unaffected* subsequences of length l out of a longer string, some of whose tokens have uniformly and randomly mutated, decreases monotonically with l.

6.3. Accuracy

We will call *false positives* those sequence homologies that are detected by random cooccurence of fortuitous index matches, and regardless of an actual functional similarity of the sequences. These false positives are due to the finite number of states associated with an index (k-tuple) of a fixed finite length; they are also due to the large number of indices generated from the original long sequence.

In order to better understand the mechanism that leads to the generation of false positives in table look-up based approaches, we will concentrate on the histograming phase. During this phase, votes for candidate matches are accumulated in an array **H**. **H** must have an entry for each possible allowable value of the alignment parameter. That is, one entry for each token in the original string stored in the database. The array **H** is usually structured as a hash table, since only a handful of entries will be used on average.

Suppose we had a uniformly random original sequence of length N_0, stored in **A** as described in Section 5; let us also have a uniformly random test sequence of length N_r. Since we are considering uniformly random sequences, any resulting peak in **H** should be characterized as *false positive*. From (5.3), each index from the random test sequence will produce an average of $\bar{n}_V \approx \frac{N_0 d_I}{\tau^k}$ votes in the histogram **H**. Then, the average number of random votes accumulated in **H** by the entire test sequence will be

$$\bar{N}_V \approx \frac{N_r N_0 d_I^2}{\tau^k} \qquad (6.5)$$

In [7] we demonstrate that probability of obtaining false positives with n index matches can be written as:

$$p_B(n) = \frac{1}{n!}\left(\frac{N_r N_0 d_I^2}{\tau^k}\right)^n \left(\frac{1}{N_0}\right)^n \left(1 - \frac{1}{N_0}\right)^{\frac{N_r N_0 d_I^2}{\tau^k}} \qquad (6.6)$$

Figure 8a plots this function for the parameters of Figure 6. Figure 8b gives the same probability in the case $d_I = 1$; this is equivalent to BLAST. Note that if the threshold is set at $n = 1$, an average of 600 false positives will be detected along with the candidate homologies. Extra processing is necessary to filter out the incorrect matches.

If shorter k-tuples are used, e.g., $k = 2$ to 6 in FASTA, mismatch chances increase significantly. Even for $n = 24$, 15 false positives will be detected in a database the size of Genbank. This means that any homology resulting in less than ~26 matches of 2-tuples will be lost among the large number of random matches.

6.4. Efficiency and Space Requirements

Let t_0 denote the time required to generate a k-tuple and to retrieve the contents of the corresponding entry $\{x_j\}$ from A. We assume that A is implemented as a vector with constant pre-allocated space for each index value t_0. Let t_1 denote the time required to update the histogram for each value of x_j. Since H is implemented as a very sparse hash table, we can assume t_1 to also be constant. The average time $\overline{T}_I$ required to process the $N_r d_I$ k-tuples and to match a given sequence χ_r of length N_r, using an index density d_I, can be demostrated to be given by

$$\overline{T}_I \approx \left[t_0 + \frac{N_o d_I}{\tau^k} t_1 \right] N_r d_I \qquad (6.7)$$

If implemented as an array, b A will occupy an optimal size of

$$S_0 = \tau^k \overline{N}_e S(x) = N_o d_I S(x) \qquad (6.8)$$

bytes, where $S(x)$ is the number of bytes required to store a reference to a specific position in the original sequence. In the case of DNA, using all the data in Genbank (about 10^8 tokens), we have that $S(x) = 4$. Therefore, with an index length of 13 tokens and a density of 20 indices per token we expect an average table size of $S_0 = 10^8 \cdot 24 \cdot 4 \approx 9$ Gigabytes.

7. Detecting Protein Homologies

The method we have described readily extends to the case of proteins. Using aminoacid in the k-tuples, however, would identify only conservative matches. We have therefore extended the technique to use the information contained in mutation matrices such as PAM250 [8]. We reduce the descriptivity of each token from the complete set of aminoacids to a number of amino-classes such that the probability of mutation within a class is maximized. By examining PAM1 to PAM250, seven families of aminoacids clearly stand out [4, 9]. These are: {C}, {S, T, P, A, G}, {N, D, E, Q}, {H, R, K}, {M, I, L, V}, {F, Y}, and {W}. These tend to group aminoacids that are similar in their physical, chemical, or structural properties [9]. One can map each of the aminoacids of a given sequence to a number between 0 and 6 defined to be the *index* of the corresponding class, and use that index as the value v_i in Equation (4.1). Using $k = 9$, this results in an indexing scheme where the look-up table contains 9^7 cells.

8. Results

We have implemented the technique for the approximately 8M residues contained in the Swiss-Prot database and for the 2M nucleotides contained in the public domain version of the *E.Coli* genome.

For DNA, The k-tuples are composed of 13 nucleotides and the index density is equal to 40. The optimal required storage for a 100 million nucleotide database will then be approximately 18 Gigabytes.

In order to minimize processing false positives, an optimal threshold $n_0 = 3$ is selected. The chance of missing a homology then becomes less than 1% for ~30 mutations in a 100 nucleotide string and is practically null if fewer than 28 mutations are present. Current search times average 20 seconds on a workstation class machine (IBM RS/6000, 530) for a 100 nucleotide string. For a database of the size of Genbank, FLASH is expected to be roughly 10 times faster than BLAST.

Accuracy and sensitivity are rival those of some of the best implementations of Smith-Waterman such as BLAZE [5] as can be seen in the successive example on proteins. For instance, the following is an example test on a 100 residues sub-strand of the E.Coli genome perturbed with 40 equiprobable synthetic mutations (insertion, deletion and nucleotide mutations). The result consists of the best recovered alignement and the number of matching indices using FLASH. Notice how no contiguous 12-nucleotide sub-sequences can be found so that BLAST would not recover the homology. This starts to show even with as little as 20 mutations per 100 residues.

```
40% Noise, 373 Votes
AAT-CACCAACCACCTGGTGG-CGATGATTGAAAAAACCATTAGCGGCCAGG-ATGCT
|   ||||| ||   | || |   |||||| ||||| || | ||| |||||| |||||
ATAACACCACCCG--TCGTCG---ATGATT-AAAAA-CCGT-AGCCGCCAGGTATGCT

TT-ACCCAA-TATCAGC-GATGCCGA-A-CGTATTTTTGCCGAACTTTTG
|| |||||| ||||||| |||||||| | ||||  ||||||||  | |||
TTTACCCAATTATCAGCCGATGCCGAGAACGTA-GTTTGCCGA--TATTG
```

Flash can on average detect homologies with far fewer matching residues than BLAST. The values of t_0 and t_1 (approximately 0.005s. and 0.00002s.) can be calculated from the actual performance curve and used in equation (6.7) to show that, even on a billion nucleotide database, the technique will remain extremely efficient. Search time is expected to be around 120 seconds per sequence, or about 35 times faster than BLAST. We also believe that BLAST will run into problems with the number of false alarms on such large databases. Flash, on the other hand, can be structured to minimize such problems as shown in previous sections.

The following is the result of an homology search on the hystone-like HU protein of E.Coli. Notice how almost all the statistically relevant matches were retrieved compared to dynamic programming techniques such as BLAZE. BLAST with default parameters recovered only the first 7 most significant matches.

All results above 1000 are considered statistically significant for FLASH. Final scores can be obtained by taking all candidates above the 1000 threshold and computing the full similarity value. Remember that, although most of the time Flash score will be according to the amount of similarity, they are not guaranteed to be always that way.

With the exception of DBH_THEAC, all the resuls in the following table were the highest returned scores. DBH AERPR was not present in our release of Swiss-Prot and was therefore not found. By checking the results of about 100 scores below the threshold of 1000, it would have been possible to recover also DBH_THEAC. Finally, we must point out that these results were obtained with a random selection of fingerprints. Vast improvments should be expected by determining an optimal set which minimize the correlation between different fingerprints.

```
Sequence Name       Length %Match BLAZE   FLASH
---------------- ---------------------------------
  1. DBHB_ECOLI          90     100     368  374181
  2. DBHB_SALTY          90     100     367  360399
  3. DBHA_ECOLI          90      73     268   27694
  4. DBHA_SALTY          90      73     268   27694
  5. DBH_AERPR           90      70     259 Not in DB
  6. DBH_BACST           90      68     251   21596
  7. DBH_BACSU           92      57     248   20189
  8. DBH_PSEAE           90      66     244   25008
  9. DBH_CLOPA           91      56     206    1998
 10. DBHI_RHILE          91      55     203    3033
 11. DBH5_RHILE          91      53     195    2989
 12. DBH_RHIME           90      53     194    2486
 13. DBH_ANASP           94      51     186    3347
 14. DBH_THETH           95      45     164    1020
 15. IHFB_SERMA          94      41     150    3545
 16. IHFB_ECOLI          94      41     149    5746
 17. IHFA_SERMA          98      38     141    1892
 18. IHFA_SALTY          99      38     138    2493
 19. IHFA_ECOLI          99      37     137    1892
 20. TF1_BPSP1           99      31     115    1242
 21. DBH_THEAC           90      26      97     751
```

Since scores using PAM matrices are computed on the fly, actual search times average around 1 minute for a single workstation running a single process. For proteins, however, we have implemented a cycle-stealing, distributed version of FLASH. Using this approach, the large look-up table can be split on multiple hard-disks so that index retrieval can be performed in parallel. This has the effect of reducing time requirements almost linearly in the number of hard disks used. Network and CPU requirements are minimal. Using 8 disks, either on a single workstation or on multiple workstations on a network protein homologies can be retrieved in an average of 5 to 8 seconds per 100 residues. CPU processing is almost irrelevant compared to disk access time (about 5% ot total CPU). This allows normal usage of the machine to continue at close to full rate.

References

[1] Aho A., J. Hopcroft and I. Ullman, "The design and Analysis of Computer Algorithms." Addison-Wesley, 1984.

[2] Altschul S., W. Gish, W. Miller, E. Myers and D. Lipman, "Basic Local Alignement Search Tool," to appear in *J. Mol. Biol.*

[3] Barsalou T. and D. Brutlag, "Searching Gene and Protein Sequence Databases," *M.D. Computing*, Vol. 8, No. 3, 1991, pp. 144-149.

[4] Brutlag, D.,J-P. Dautricourt, S. Maulik, and J. Relph. ``Improved Sensitivity of Biological Sequence Database Searches." *CABIOS*, 6:3, 1990, pp 237-245.

[5] Brutlag, D., J-P. Dautricourt, R. Diaz, J. Fier, and R. Stamm. ``BLAZE^{TM}: An Implementation of the SMith-Waterman Sequence Comparison Algorithm on a Massively Parallel Computer. *In preparation.*

[6] Califano A. and R. Mohan, "Multi-dimensional indexing for recognizing visual shapes," in Proc. *IEEE Conf. on Comp. Vision and Pattern Rec.*, June, 1991, pp 28-34.

[7] Califano A. and Rigoutsos I., "FLASH: A Fast Look-up Algorithm for String Homology," IBM TR RC18791

[8] Dayhoff M., Schwartz R., and Orcutt B., "A model of Evolutionary Change in Proteins," in *Atlas of Protein Structure*, 5, Suppl. 3, 1979, pp. 345-352.

[9] George, D., W. Barker, and L. Hunt. ``Mutation data Matrix and Its Uses." In *Methods in Enzymology, Volume 183*, R. Doolittle ed. Academic Press, 1990.

[10] Lamdan Y. and H. Wolfson, "Geometric Hashing: a general and efficient model-based recognition scheme," in *Proc. 2nd Int. Conf. on Comp. Vision*, Dec 1988.

[11] Lipman D. and W. Pearson, *Science*, (1985) **227**, pp. 1435-1441.

[12] Pearson W. and D. Lipman, "Improved tools for biological sequence comparison," in *Proc. National Academy of Science: Genetics*, Vol 85, pp. 2444-2448.

[13] Rigoutsos, I. and R. Hummel. ``Implementation of Geometric Hashing on the Connection Machine." In *Proceedings of the IEEE Workshop on Directions in Automated CAD-Based Vision*, June 1991, Maui, Hawaii.