Approximate Multiple String Searching by Clustering

Fei Shi Peter Widmayer
{shi, widmayer}@inf.ethz.ch

Department of Computer Science
Swiss Federal Institute of Technology Zurich
(ETH Zurich)
ETH Zentrum, CH-8092 Zurich, Switzerland

Abstract

We are given a finite set S of text strings and a pattern P over some fixed alphabet Σ. The topic of this paper is the design of a data structure $D(S)$ which supports approximate multiple string searching queries efficiently. Thereby, for a given upper bound $k \in Z^+$ on the allowable distance, $P = p_1 \cdots p_m$ is said to appear approximately in a text $T = t_1 \cdots t_n$, m, $n \in Z^+$, if there exist positions u, v in T such that the edit distance between P and $t_u \cdots t_v$ is at most k. Let N denote the sum of the lengths of all strings in S. We present an algorithm that constructs the data structure $D(S)$ in $O(N)$ time and space. Afterwards, an approximate multiple string search query can be answered in O(N) expected-time if the allowable distance k is bounded above by $O(\frac{m}{\log m})$. The method can be used to search large nucleotide and amino acid sequence databases for similar sequences.

1 Introduction

Multiple string search is a classic topic within the field of algorithms and data structures, and has recently gained great importance in computational biology. This is largely due to the enormous advances in DNA sequencing technology [12]. The current release of GenBank (GenBank Release 95.0, June 1996) alone contains 835,487 sequences of altogether more than 550 million nucleotide base pairs. The size of the sequences databases doubles every 2.5 years [13]. Whenever a sequence investigator determines a new sequence, one of the first things he must do is "to compare it with all available sequences to see if it resembles something already known" [7]. This is the problem of searching for approximate occurrences of a new string among all the known strings and we call it the approximate multiple string searching problem.

Typically, only a handful (if any) of the known sequences will contain an approximate occurrence of the new sequence. Hence, it is undesirable to search through all known sequences explicitly, and we would instead be willing to sacrifice some preprocessing time on the set of known sequences to make subsequent approximate searches faster. The contribution of this paper is to propose a solution to this problem, namely a data structure that supports approximate multiple string searches, together with its preprocessing and searching algorithms.

Before presenting the solution, let us define the approximate multiple string searching problem more formally. We are given a finite set S of strings; each string is a word over Σ^* for some fixed, finite alphabet Σ. In order to perform approximate string searching with a query string $P = p_1 \cdots p_m$ (the pattern) efficiently, we are looking for a suitable data structure $D(S)$ for storing S. The operation that $D(S)$ should support is the approximate string searching query which returns the set of all strings in S in which P occurs approximately, together with all substrings of S that are approximate occurrences of P. For a given upper bound $k \in Z^+$ on the allowable distance, string P is said to occur approximately in $T = t_1 \cdots t_n$, m, $n \in Z^+$, p_i, $t_j \in \Sigma$ for $1 \leq i \leq m$, and $1 \leq j \leq n$, if there are positions u, v in T, $1 \leq u \leq v \leq n$, such that the distance between $t_u \cdots t_v$ and P is at most k. For our purpose, the distance is defined to be the edit distance with unit cost; i.e., the distance between two strings is the minimum number of insertions, deletions and changes of a letter that transforms one string into the other.

The approximate multiple string searching problem differs from the approximate string matching problem (see, for example, [17, 20, 10, 4, 15, 16]), where all approximate occurrences of a pattern string in a *single* text string are sought, quite considerably: our emphasis is on providing a fast, selective filter on all strings that leaves only few strings to be investigated closely, in order to answer a query. Nevertheless, one will probably argue that there exist two sublinear expected-time algorithms, namely Myers' algorithm [15] and Chang and Lawler's algorithm [4], in the literature and one can use any of these two algorithms for the approximate multiple string searching problem by simply concatenating all text strings in the database into a single string. Myers' algorithm assumes that the length m of the pattern P satisfy $m = \log_{|\Sigma|} N$ where N denotes the sum of the lengths of all strings in the database S. It first builds an index structure for the database S in $O(N)$ time and the index occupies $O(N)$ space. This is comparable to our methods for building our index structure and for storing it. But deletion or addition of a string of length r from or into Myers' index can take as much as $O(N)$ time, while one needs only $O(r)$ time with our index structure. Myers' algorithm is termed sublinear in the following sense: it takes $O(k\, N^{pow(\varepsilon)} \log N)$ expected-time to answer a query where $\varepsilon = k/m$ and $pow(\varepsilon)$ is an increasing and concave function [1] that is 0 when $\varepsilon = 0$. Thus Myers' algorithm is superior to existing $O(k\,N)$ algorithms only when ε is small enough to guarantee that $pow(\varepsilon) < 1$. Chang and Lawler's algorithm is purely scanning based, requiring only $O(m)$ working space while both Myers' algorithm and ours require a precomputed index which occupies $O(N)$ space. However, empirical comparisons have shown that this sublinear-expected algorithms is NOT even competitive with an $O(k\,n)$ expected-time algorithm [5]. The expected running time of the sublinear algorithm is $O((k \log m)(n/m))$ for $k < m/(\log m + O(1))$. When k is as large as $O(m/\log m)$, the expected running time of this algorithm is actually *linear* (i.e., $O(n)$). As a matter of fact, experimental results have shown that the actual running time of this algorithm is as much as $160\,(k \log_{|\Sigma|} m\,(n/m))$ [5]. Furthermore, each time when a new query string (pattern) is given, Chang and Lawler's algorithm needs to scan the whole database which is usually stored in a slow secondary storage device and therefore takes a lot

[1]$pow(\varepsilon) = \log_{|\Sigma|} \frac{c+1}{c-1} + \varepsilon \log_{|\Sigma|} c + \varepsilon$ and $c = \varepsilon^{-1} + \sqrt{1+\varepsilon^{-2}}$.

of time; our index structure needs to be constructed only once and although it may need to be stored on a secondary storage device as well, our algorithm needs only to read a very small part of the tree-like index structure to answer a query. Consequently, neither Myers' nor Chang and Lawler's approximate single string matching algorithm is appropriate for the approximate multiple string searching problem.

In previous methods for the approximate multiple string searching problem, the pattern is examined against each of the texts in S, thereby wasting a lot of time with texts that are arbitrarily far from the pattern. In a first approach, we have shown how to use geometric clustering methods to focus on texts that are near to the pattern [3]. This approach follows a different approach: texts in S are clustered according to a non-geometric criterion in such a way that the filter step delivers candidate texts that are likely to match the pattern approximately. In addition, we show how candidates can be tested efficiently.

The next section illustrate the basic idea of our approach. Section 3 presents the details of the proposed data structure, including an algorithm for constructing it, and Section 4 describes the query algorithm. Finally, Section 5 concludes the paper.

2 The basic idea

Let k be the given upper bound on the edit distance in the search for approximate occurrences, and let $P = p_1 \cdots p_m$ be the pattern to be searched for. We partition P into $k+1$ substrings called blocks, denoted by $B_1, \ldots, B_{k+1}$, each of length $r = \frac{m}{k+1}$ (assume that $k+1$ divides m to simplify the presentation), where $B_j = p_{(j-1)r+1}p_{(j-1)r+2} \cdots p_{jr}$, for $1 \leq j \leq k+1$. Then it is obvious that if P matches a substring of a text $T = t_1 t_2 \ldots t_n \in S$ approximately with edit distance at most k, then at least one of the $k+1$ blocks $B_1, \ldots, B_{k+1}$ matches a substring of T exactly. This observation has been used previously in approximate string matching algorithms [2, 22, 16].

To make use of this fact for approximate multiple string searching, we store in a data structure information on all substrings of length r of all strings in S. More precisely, let us call any string from Σ^r, i.e., any string of length r, a r-gram. Let $W_1, \ldots, W_l$ be the set of all r-grams occurring as substrings in some string in S. For each W_i, $1 \leq i \leq l$, we compute a set of "home" strings of W_i, i.e., the set of strings in S that contain W_i as a substring: $h(W_i) = \{T \in S | W_i \subset T\}$ where $\subset$ denotes the substring relation between strings.

To find all approximate occurrences of P in S, it is now sufficient to examine all strings $A = \bigcup_{i=1}^{k+1} h(B_i)$. No other strings can belong to the answer of the query, and the number of strings in A is expected to be much smaller than the number of strings in S.

3 The data structure

For ease of description, let us interpret an r-gram as a positive integer in base-$|\Sigma|$-notation. [2] That is, for $\Sigma = \{a_0, a_1, \ldots, a_{e-1}\}$, $e = |\Sigma|$, a string $X = x_1 x_2 \ldots x_r$ is interpreted as some

[2]An efficient way to convert a string into a positive integer is give by Karp and Rabin [8].

value $f(X)$:

$$f(X) = \sum_{i=1}^{r} \bar{x}_i e^{r-i} \tag{1}$$

where $\bar{x}_i = j$ if $x_i = a_j$, $1 \leq i \leq r$. Note that f is a bijection, i.e., $f(X) = f(X')$ if and only if $X = X'$. when "scanning" a string $T \in S$, i.e., when considering r-grams in T that appear in adjacent positions, the associated integers can be computed quickly. More precisely, for $V_i = t_i t_{i+1} \ldots t_{i+r-1}$, $1 \leq i \leq n - r + 1$, we have:

$$f(V_{i+1}) = (f(V_i) - \bar{t}_i e^{r-1})e + \bar{t}_{i+r} \tag{2}$$

By first computing $f(V_1) = \sum_{i=1}^{r} \bar{t}_i e^{r-i}$ and then computing $f(V_j)$ using (2) for $j = 2, \ldots, n - r + 1$, all associated integers for r-grams in T can be computed in time $O(n)$, assuming that multiplications, additions, and subtractions of integers take constant time each, and that the values of e^1, e^2, ..., e^r can be stored in atable. Let us now define classes of strings in S, according to the r-grams they contain. For an r-gram W, let

$$H(W) = \{(T, i) | T = t_1 \cdots t_n \in S \wedge f(t_i \cdots t_{i+r-1}) = f(W), 1 \leq i \leq n - r + 1\} \tag{3}$$

be the set of home strings of W, augmented by the positions of the occurrences of W.

3.1 Computing home strings of r-grams

The set of home strings, of all possible r-grams W_1, ..., W_l in S, can be computed by scanning all strings in S and keeping track of the r-grams encountered. For the r-gram W_j encountered at position i in string $T \in S$, we add the element (T, i) to the set $H(W_j)$. Apart from the time to access $H(W_j)$ for a given j, and provided that the addition of an element to a set is possible in constant time (which is true, e.g., in a linked list), the computation of all sets of home strings of r-grams takes time $O(N)$, where N is the sum of the lengths of all strings in S.

3.2 Maintaining the home sets

To access $H(W_j)$ for a given j, any of a number of established techniques can be applied. One choice of a data structure for maintaining home sets is a search tree. Due to the high expected number of r-grams, an external search tree, such as B^+-tree [6], should be used. The leaves of the B^+-tree contain the entries of home sets, in the form of pairs (T, i), consisting of a pointer to a string and an integer, ordered according to $H(W_j)$ and interspersed with separation information between different $H(W_j)$. Any access or addition of an entry will then cost time $O(\log N)$, totaling to $O(N \log N)$ for preprocessing.

Another choice of a data structure is a digital tree (trie), where the digits are letters from Σ. The trie has height r, and each node is associated with the substring that can be read off the edge labels on the path from the root to the node. Then, each leaf corresponds to an r-gram W and points to $H(W)$. In this structure, any access or addition of an entry costs time at most $O(r)$, totaling to $O(r \cdot N)$ time for preprocessing.

A more careful investigation shows that a data structure similar to suffix trees [21, 11, 19] can be constructed in $O(N)$ time and space thereby keeping $O(r)$ update time for an access or an addition.

We are in the process of implementing and experimentally comparing the three data structures to see which one is more useful for nucleotide and amino acid sequence databases.

3.3 The expected selectivity of home sets

The number of different r-grams occurring in a data base of s different sequences of average length n is bounded from above by $s \cdot (n - r + 1)$. For GenBank Release 95, June 1996, with $s = 835,487$, $n = 660$, we have for $r = 64$ a bound of $498,785,739$. Note that this is nearly zero compared to e^r (with $e = 4$ for DNA sequences).

If we assume for the moment, that P and T are random strings of length m and n respectively, the probability that a r-gram of P occurs exactly in T is

$$1 - \left(1 - \frac{1}{e^r}\right)^{n-r+1} \tag{4}$$

Therefore, the probability that at least one of the $k + 1$ r-grams of P occurs exactly in T is

$$1 - \left(1 - \frac{1}{e^r}\right)^{(k+1)\times(n-r+1)} \tag{5}$$

For a random query r-gram, the probability to occur in the database is therefore very small. Hence, the selectivity of the home sets method should be very good. Since DNA sequence data will not be rigorously random, the practical selectivity still needs to be determined; first experimental results show that the home sets method will still be very selective.

4 The searching algorithm

An obvious way to search for all approximate occurrences of a pattern $P = p_1 \cdots p_m$ with edit distance at most k is to inspect all home sets of the $k + 1$ r-grams of P, as they are given in the data structure described in the previous section. In order to give this searching process a favourable order, we do not search each candidate text string T immediately after retrieving it, but we instead sort the set of pairs (T, i) according to the T-components. More precisely, we keep track of a set of candidate triples

$$C(P) = \{(T, i, j) | (T, i) \in H(B_j), 1 \le j \le k + 1\} \tag{6}$$

indicating the text strings T matching the j-th block $B_j = p_{(j-1)r+1} p_{(j-1)r+2} \ldots p_{jr}$ of P exactly at position i. The triples in $C(P)$ are sorted according to T. For any $(T, i, j) \in C(P)$, the edit distance d of a substring $t_u \ldots t_v$, with $u \le i \wedge i + r - 1 \le v$, to the pattern P, is bounded from above by the fact that block j of P occurs exactly in $t_u \ldots t_v$. More precisely,

$$\begin{aligned} d(P, t_u \ldots t_v) &\le d(p_1 \ldots p_{(j-1)r}, t_u \ldots t_{i-1}) + d(p_{jr+1} \ldots p_m, t_{i+r} \ldots t_v) \\ &\leftarrow \bar{d}(T, i, j, u, v). \end{aligned} \tag{7}$$

Therefore, for any entry $(T, i, j) \in C(P)$, any pair (u, v) for which $\bar{d}(T, i, j, u, v) \le k$ identifies an approximate occurrence of P in T. Since, on the other hand, each approximate occurrence of P in T must contain a block of P that matches in T exactly, the answer to an approximate searching query is the set of occurrences with $\bar{d}(T, i, j, u, v) \le k$. To find the corresponding u, v for a given $(T, i, j) \in C(P)$, it is sufficient to look k positions around, in the following sense: it suffices to consider $u \ge \alpha$ and $v \le \beta$ where

$$\alpha = \begin{cases} 1 & \text{if } i - (j-1)r - k \le 1 \\ i - (j-1)r - k & \text{otherwise} \end{cases} \tag{8}$$

and

$$\beta = \begin{cases} n & \text{if } m + i - (j-1)r - 1 + k \geq n \\ m + i - (j-1)r - 1 + k & \text{otherwise} \end{cases} \tag{9}$$

Furthermore, if $(j-1)r+1 > i+k$ or $(j-1)r+1 < m-n+i-k$, then we can immediately decide that there exists no pair u, v with $\bar{d}(T, i, j, u, v) \leq k$. Whenever u, v might exist for (T, i, j), we start the dynamic programming algorithm for distance computation (see, e.g., [18]) both at the left end of block j in T, advancing towards the left, and at the right end of block j in T, advancing towards the right. We therefore need at most $(j-1)r((j-1)r+k)$ and $(m-jr)(m-jr+k)$ operations, respectively. In the worst case, we need $(m-r)(m-r+k)+4k^2 < m^2+km+4k^2$ operations to implicitly identify all pairs u, v with $\bar{d}(T, i, j, u, v) \leq k$. The pairs can be made explicit by adding the distances of all combinations of u and v to the dynamic programming matrix, and then extracting all pairs of the solution.

A more detailed look at the selectivity of the home sets algorithm shows that few strings will need to be investigated in a query with a random string. If both T and P are random strings, the expected number of triples $(T, i, j) \in C(P)$ is the number of blocks times the number of r-gram positions times the probability of r-gram equality:

$$\frac{(k+1)(n-r+1)}{e^r} < \frac{(k+1)n}{e^r} \tag{10}$$

Therefore, the expected time needed to find all pairs u, v with $\bar{d}(T, i, j, u, v) \leq k$ is less than

$$\frac{(k+1)n}{e^r}(m^2 + km + 4k^2) \tag{11}$$

This time is linear in the length of T, if $\frac{(k+1)n}{e^r}(m^2 + km + 4k^2) < cn$ for some constant c. Since, $k \leq m-1$, we get the requirement that $n\frac{m}{e^r}(6m^2 - 9m + 4) < cn$. Since $r = \frac{m}{k+1}$, this yields:

$$k \leq \frac{m}{\log \frac{m(6m^2-9m+4)}{c}} - 1 \tag{12}$$

showing that the maximum allowable value for k is $O(\frac{m}{\log m})$, if we desire that the expected time for examining T agains P be linear in the length of T. This bound is quite pessimistic, because we could use a faster $O(kn)$-time algorithm (see, e.g., [9, 14, 20]) instead of the time-consuming dynamic programming algorithm to identify the pairs u, v satisfying $\bar{d}(T, i, j, u, v) \leq k$. In total, we get an expected time of $O(N)$ for finding all approximate occurrences of a pattern P in a set of text strings whose sum of lengths is N.

5 Concluding remarks

A primary motivation for this paper was to be able to efficiently search large genetic sequence databases for sequence homologies. Given a set S of text strings, our data structure can be constructed in $O(N)$ time and space where N denotes the sum of the lengths of the strings in S. With this data structure, an approximate searching query can be answered in $O(N)$ expected time.

Our work presented in this paper is still in its preliminary stage. It remains to be seen what the expected run time will be for real biological sequences, such as the data in GenBank and the data in Brookhaven's Protein Data Bank (PDB) as we understand that the DNA or protein sequences are not completely random strings. We should also be able to distinguish biological significant relationships from chance matches found by our algorithm. An implementation of our method and performance experiments are currently being performed.

Also, we note that the idea of this paper is similar to the essential idea behind a heuristic sequence comparison tool, BLAST [1], now in popular use for protein database searches, although BLAST is a local similarity search algorithm and ours is a global similarity search algorithm. BLAST consists of three stages, namely,

1. for each word W of length w of the query sequence (the pattern), compute a list of words that score at least A when compared to word W;
2. search the database for hits (i.e., subsequences of the database that score at least B when compared to some word in the lists);
3. extend the hits in order to find subsequences of the database that score at least C when compared to some subsequence of the query sequence.

Interested reader is referred to [1] for more information on BLAST.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers and D. Lipman, Basic Local Alignment Search Tool, *J. Mol. Biol.* (1990) 215, pp. 403 - 410.

[2] R.A. Baeza-Yates and C. H. Perleberg, Fast and Practical Approximate String Matching, *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, Vol. 644 of LNCS, Spring-Verlag, 1992.

[3] E. Bugnion, T. Roos, F. Shi, P. Widmayer and F. Widmer, Approximate multiple string matching using spatial indexes. in *Proc. of the first South American Workshop on String Processing*, (eds.) R. Baeza-Yates and N. Ziviani, pp. 43 - 53, 1993.

[4] W. Chang and E. Lawler, Approximate String Matching in Sublinear Expected Time, *Proc. 31st FOCS*, pp. 116-124, St. Louis, MO, Oct. 1990, IEEE.

[5] W. Chang and J. Lampe, Theoretical and Empirical Comparisons of Approximate String Matching Algorithms, *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, Vol. 644 of LNCS, pp. 175-184, Spring-Verlag, 1992.

[6] D. Comer, The ubiquitous B-tree, *ACM Computing Surveys*, 11:2 (June 1979).

[7] R.F. Doolittle, *Of URFs and ORFs: A Primer on How to Analyse Derived Amino Acid Sequences*, University Science Books, Mill Valley, California, 1987.

[8] R. Karp and M. Rabin, Efficient Randomized Pattern Matching Algorithms. *IBM J. Res. Develop.*, Vol. 31, No. 2, March 1987, pp. 249-260.

[9] G. M. Landau and U. Vishkin, Fast String Matching with k Differences, *J. Comp. Sys. Sci.* 37(1988), pp. 63-78.

[10] G. M. Landau and U. Vishkin, Fast Parallel and Serial Approximate String Matching, *J. Algorithms* 10(1989), pp. 157-169.

[11] E.M. McCreight, A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM* 23 (1976), 262-272.

[12] A.M. Maxam, W. Gilbert W., A new method for sequencing DNA. *Proc. Natl. Acd. Sci. USA* 74, 560-564(1977).

[13] H. Mewes, K. Heumann: Genome analysis: Pattern search in biological macromolecules, *Proc. 6-th Annual Symposium on Combinatorial Pattern Matching*, LNCS 939 (1995), Spring-Verlag, pp. 261-272.

[14] E.W. Myers, An O(ND) Difference Algorithm and Its Variations, *Algorithmica* 1(1986), pp. 252-266.

[15] E.W. Myers, A Sublinear algorithm for approximate keyword searching, *Algorithmica*, 12(4/5), 1994, pp. 345-374.

[16] F. Shi, Fast approximate string matching with q-blocks sequences, *Proceedings of the third South American Workshop on String Processing*, Carleton University Press, Ottawa, Canada, 1996, pp. 257-271.

[17] E. Ukkonen, Finding Approximate Patterns in Strings. *J. Algorithms* 6(1985), pp. 132-137.

[18] E. Ukkonen, Algorithms for approximate string matching, *Information and Control*, 64 (1985), pp. 100-118.

[19] E. Ukkonen, On-line construction of suffix-trees. Report No. A-1993-1, Department of Computer Science, University of Helsinki, Finland, 1993.

[20] E. Ukkonen and D. Wood, Approximate String Matching with Suffix Automata, Report A-1990-4, Department of Computer Science, University of Helsinki, April 1990.

[21] P. Weiner, Linear Pattern Matching Algorithm,*Proc. 14th IEEE Symposium on Switching and Automata Theory*, 1973, 1-11.

[22] S. Wu and U. Manber, Fast Text Searching with Errors, Technical Report TR-91-11, Department of Computer Science, University of Arizona, Tucson, Arizona, June 1991.