A Graph Approach to the Threshold All-Against-All Substring Matching Problem

MARINA BARSKY, ULRIKE STEGE, ALEX THOMO, and CHRIS UPTON
University of Victoria, Canada

We present a novel graph model and an efficient algorithm for solving the "threshold all against all" problem, which involves searching two strings (with length M and N, respectively) for all maximal approximate substring matches of length at least S, with up to K differences. Our algorithm solves the problem in time $O(MNK^3)$, which is a considerable improvement over the previous known bound for this problem. We also provide experimental evidence that, in practice, our algorithm exhibits a better performance than its worst-case running time.

1. INTRODUCTION

An important problem in the field of string matching is the extraction of exact and approximate common patterns from two or more strings. In fact, in particular application areas, such as the analysis of biological sequences, the extraction of *approximate* common patterns is of primary (or great) importance. This is because biological sequences often have a high mutation rate and an analysis based only on exact common patterns can miss a great deal of useful information.

Authors' addresses: Marina Barsky, Ulrike Stege, and Alex Thomo, Department of Computer Science, University of Victoria, PO Box 3055, STN CSC, Victoria, BC, V8W 3P6, Canada; email: {mgbarsky,stege,thomo}@cs.uvic.ca; Chris Upton, Department of Biochemistry and Microbiology, University of Victoria, PO Box 3055, STN CSC, Victoria, BC, V8W 3P6, Canada; email: cupton@ uvic.ca.

In contrast to finding exact patterns, the extraction of approximate patterns is computationally much more demanding. An efficient solution to this problem would greatly help important research in biology, such as locating regulatory sites and drug target identification.

In computer science, the problem of finding all approximate common patterns of two given strings is known as "(full) all-against-all approximate substring matching" (see Baeza-Yates and Gonnet [1990]; Gusfield [1997]), and is notorious for its computational difficulty. In practice, various constraints are set for the sought solutions, such as the maximum allowed number of approximations or "errors" and the minimum allowed length of substrings. Formally, the problem we study in this paper is: Given two strings s and t and integer parameters S and K, find all pairs (s', t') of maximal substrings from s and t, respectively, such that their length is at least S, and the edit distance[1] between them is, at most, K.

Our contribution is a fast algorithm for solving the above "threshold all-against-all" problem for two strings.

Pattern discovery methods in the literature may be classified into two groups:

—Probabilistic methods, such as Gibbs motif sampling [Lawrence et al. 1993], expectation maximization [Bailey and Elkan 1995], maximization of the information content [Roth et al. 1998], and Gibbs Sampling over suffix trees [Rocke 2000].
—Deterministic methods, such as [Baeza-Yates and Gonnet 1990, 1999; Vilo 2002] based on a hybrid dynamic programming approach, in which the edit distance computation (using dynamic programming) is combined with (or guided by) the use of suffix trees.

Probabilistic algorithms have a satisfactory running time performance, but there is no guarantee that every solution is produced.

We investigate a new deterministic method. Differently from the methods of the second group, it neither uses suffix trees nor dynamic programming. Instead, our method is based on a novel graph approach and yields feasibility in both time and space without sacrificing the completeness of the solution.

Observe that for two given strings s and t of lengths M and N respectively, a naïve approach to "all-against-all approximate substring matching" is to exhaustively test each pair of substrings from s and t. This approach has a time complexity of $O(M^2 N^2)$, as it requires the computation of $O(MN)$ cells of dynamic programming table for each MN pair of possible starting locations.

A better method for the "threshold all-against-all" problem was first proposed by Baeza-Yates and Gonnet [Baeza-Yates and Gonnet 1990] and [Baeza-Yates and Gonnet 1999]. Recently, Vilo [Vilo 2002] extended the approach [Baeza-Yates and Gonnet 1990, 1999] to better target specific applications, preserving however, the main algorithmic components of [Baeza-Yates and Gonnet 1990, 1999]. Thus, we will mention here some features of the approach by Baeza-Yates and Gonnet. Notably, by avoiding the examination of repeating substrings, their solution is significantly better than

---

[1]The *edit distance* in this paper is a unit-cost edit distance.

the naïve approach with respect to the average time complexity. In their method, each input string is organized as a suffix tree structure and the order of substring comparisons is guided by a depth-first traversal of the nodes of the suffix trees. The average-time complexity lies between $O(MN)$ (best case) and $O(M^2N^2)$ (worst case), but closer to $O(M^2N^2)$ (see Gusfield [1997]).

Setting a threshold criterion bounding the error number, e.g., allowing, at most, K differences in substring matches, significantly improves the performance of Baeza-Yates and Gonnet's algorithm. The reason is that the value of K can be directly incorporated into their algorithm to cut down the depth of the suffix tree traversal. As we verify experimentally, for small values of K, Baeza-Yates and Gonnet's algorithm performs well. However, as K increases, the number of suffix tree nodes that are examined grows very fast.

Our approach bears some similarity with a general technique known as *sparse dynamic programming*, which handles, in a clever way, the sparsity in dynamic programming tables (cf. Eppstein et al. [1992a, 1992b]). The main difference of our approach from sparse DP lies in our novel graph model, which is not based on a dynamic programming table. Our optimized depth-first search also yields a more efficient solution over the dynamic programming method.

1.1 Our Approach

We cast the problem into a new problem of finding "maximal paths" in a special "matching graph." Via a careful study of this graph, we are able to derive interesting and useful properties that help us in devising a highly optimized depth-first search procedure to determine the maximal paths, which correspond to the solutions of the original string problem.

Our proposed algorithm runs in $O(MNK^3)$ time, which is a significant improvement over Baeza-Yates and Gonnet's algorithm. Moreover, we experimentally show that our algorithm scales quadratically in K (without reaching the cubic upper bound) and outperforms Baeza-Yates and Gonnet's algorithm by an order of magnitude.

Also, we stress that our algorithm has an additional nice feature: it reversely depends on the alphabet size. As the size of the alphabet grows, the running time of our algorithm considerably decreases.

We note that [Barsky et al. 2006] is a short version of this paper. However, in [Barsky et al. 2006], we have only presented the high-level ideas without giving proofs, detailed examples, and full experimental evaluations. Besides providing the complete development of our algorithm along with examples and full experimental evaluations, we also accompany the current paper by a software implementation in Java.

The remainder of this paper is organized as follows. Section 2 introduces our new graph model. Section 3 describes the details of our algorithm. In Section 4 we present performance results and compare them with an optimized variant of Baeza-Yates and Gonnet's algorithm. In Section 5 we discuss some further related work. Section 6 concludes the paper.

2. A GRAPH MODEL FOR THE ALL-AGAINST-ALL SUBSTRING MATCHING

Let Σ be a finite alphabet. A sequence of letters $a_1 a_2 \ldots a_N$, where $a_i \in \Sigma$ (for $1 \leq i \leq N$), is called a *string* over Σ. We denote strings with s and t. Given a string s, we denote its ith letter with $s[i]$, and we denote a *substring* of s starting at position i and ending at position j (where $i \leq j$) with $s[i, j]$. The letters of s at positions i and j are included in the substring $s[i, j]$ and thus $s[i, j]$ has length $j - i + 1$.

For any two strings s and t, we can "transform the first string into the second" by applying a sequence of the classical *edit operations*: insertion, deletion, and substitution of a letter by another letter. These edit operations are referred to as *errors*.

Let the *edit distance* for two strings s and t be the minimum number of edit operations needed to transform s into t. Further, we say pair (s, t) is a *K-bounded approximate match* if the edit distance between s and t is, at most, K.

Now consider two strings s and t over Σ, with lengths M and N, respectively. The problem to solve is to find all approximate substring matches with, at most, K errors and of length greater than some threshold S.

Formally we have:

Problem 2.1. ***All error-bounded approximate matches***
INPUT: Strings s and t over alphabet Σ, and positive integers K and S.
OUTPUT: All error-bounded approximate maximal matches $(s[i, j], t[k, l])$ such that

1. the edit distance between $s[i, j]$ and $t[k, l]$ is, at most, K and
2. the lengths of both $s[i, j]$ and $t[k, l]$ are, at least, S.

Observe that in the above definition we are looking for maximal matches, with respect to the length. In other words, if a pair of substrings of sufficient lengths has an edit distance of less than K, we do not report it immediately. Instead, we try to extend the solution as long as the accumulated errors do not exceed K. If it cannot be extended further, we report it as a maximal solution. Note that this is not a limitation, because the full set of all substring pairs can easily be obtained from the maximal solutions.

We solve Problem 1 by casting it to an equivalent problem on graphs induced by a "matching matrix," which is defined as follows.

Let s and t be two given strings. We define the *matching matrix*[2] *of s and t* $(\mathcal{M}_{s,t})$ as

$$\mathcal{M}_{s,t}[i, j] = \begin{cases} 1 & \text{if } s[i] = t[j] \\ 0 & \text{otherwise.} \end{cases}$$

We omit the subscripts whenever they are clear from the context. An example of a matching matrix for strings ababacbbbc and aacbacacab is shown in Figure 1 (left).

---

[2]This matrix is also know as "dot plot" in Bioinformatics literature.

	t	a	a	c	b	a	c	a	c	a	b
s		0	1	2	3	4	5	6	7	8	9
a	0	1	1	0	0	1	0	1	0	1	0
b	1	0	0	0	1	0	0	0	0	0	1
a	2	1	1	0	0	1	0	1	0	1	0
b	3	0	0	0	1	0	0	0	0	0	1
a	4	1	1	0	0	1	0	1	0	1	0
c	5	0	0	1	0	0	1	0	1	0	0
b	6	0	0	0	1	0	0	0	0	0	1
b	7	0	0	0	1	0	0	0	0	0	1
b	8	0	0	0	1	0	0	0	0	0	1
c	9	0	0	1	0	0	1	0	1	0	0

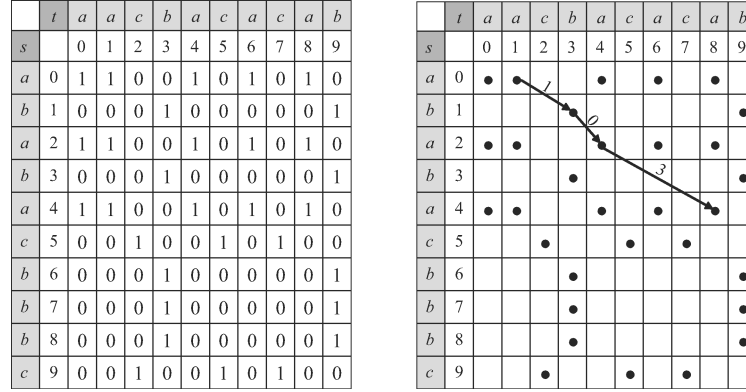|	t	a	a	c	b	a	c	a	c	a	b
s		0	1	2	3	4	5	6	7	8	9
a	0	•	•			•		•		•	
b	1				•						•
a	2	•	•					•		•	
b	3				•				•		•
a	4	•	•			•		•		•	
c	5			•			•		•		
b	6				•						•
b	7				•						•
b	8				•						•
c	9			•			•		•		

Fig. 1. (*Left*) Matching matrix and partial induced graph. (*Right*) Some edges of cost of, at most, 3 are shown.

Based on matching matrix $\mathcal{M}$, we define a weighted directed graph $G_{\mathcal{M}}$ with vertices v_{ij} corresponding to the 1-elements of the matrix, and with (directed) edges defined in a "top-down" and "left-right" fashion as follows: there is an edge $e(v_{ij}, v_{kl})$ iff $i < k$ and $j < l$. Figure 1 (right) illustrates the nodes and some of the edges of the $G_{\mathcal{M}}$ graph based on the matrix $\mathcal{M}$ in the same figure (left).

We define the *cost* $c(v_{ij}, v_{kl})$ of an edge $e(v_{ij}, v_{kl})$ to be $c(v_{ij}, v_{kl}) = max(k - i, l - j) - 1$. A *path* in graph $G_{\mathcal{M}}$ is a sequence of vertices connected by edges. For a path in $G_{\mathcal{M}}$, we define two characteristic properties: the "match length" and the "error number," which are as follows.

Definition 2.2. Let π be a path starting at v_{ij} and ending at v_{kl}. Then:

—The *match length* of π is defined as $ML(\pi) = min(k - i + 1, l - j + 1)$.

—The *error number*, $EN(\pi)$, is defined as the cost of the path π, that is the sum of all costs of edges in π.

Note that $ML(\pi)$ corresponds to the length of the shorter of the two substrings $s[i, k]$ and $t[j, l]$.

Our choice of naming is based on the semantics of the paths between two vertices. In essence, a path from v_{ij} to v_{kl} outlines a sequence of edit operations that realizes an (approximate) match of $s[i, k]$ with $t[j, l]$.

Interestingly, the graph $G_{\mathcal{M}}$ possesses a very desirable property. Namely, for any two vertices v_{ij} and v_{kl}, the error number of the shortest path in $G_{\mathcal{M}}$, going from v_{ij} to v_{kl}, equals the edit distance between the substrings $s[i, k]$ and $t[j, l]$.

Although intuitively right, a formal proof needs special care. Further, observe that $G_{\mathcal{M}}$ is *not* a dynamic programming (induced) graph (also called edit graph in [Gusfield 1997]); DP graphs have been very well studied in the literature. However, to the best of our knowledge there is no work that formally studies the properties of $G_{\mathcal{M}}$ graph.

In order to show the claimed property, we use the notion of an *edit transcript*, defined as a string over the alphabet $\{M, D, I, F\}$, where the letters stand for *match*, *deletion*, *insertion*, and *mismatch*, respectively. An edit transcript

s	abce--d	abce-d	a-bced
t	a-ebced	aebced	aebced
Transcripts	MIFFDDM	MFFFDM	MFMMMM

Fig. 2.   Examples of legal and illegal edit transcripts. M stands for *match*, F stands for *substitution*, I stands for *insertion*, and D stands for *deletion*.

describes a transformation of one string into another [Gusfield 1997] (see Figure 2 as an illustration). Each possible way to transform $s[i,k]$ into $t[j,l]$ using edit operations can be expressed by some edit transcript. In general, for two strings, there may be many different edit transcripts transforming one string into another. Clearly, for any possible pair of substrings in s and t, we are looking for an edit transcript with a minimum possible number of edit operations.

Consider the substrings $s[i,k]$ and $t[j,l]$ with $s[i] = t[j]$ and $s[k] = t[l]$. Such substrings correspond to the path between two matches in the matching matrix (i.e., $\mathcal{M}[i,j]$ and $\mathcal{M}[k,l]$ are equal to 1).

LEMMA 2.3.   *There exists an edit transcript for $s[i,k]$ and $t[j,l]$ with cost (total number of edit operations) at most $max(k-i, l-j) - 1$.*

PROOF.   Without loss of generality let $s[i,k]$ be shorter than $t[j,l]$. Then, $max(k-i, l-j) - 1 = l - j - 1$, i.e., equal to the length of $t[j+1, l-1]$. Now, an edit transcript with the claimed cost can be obtained to reflect the following procedure.

First align $s[i, k-1]$ with the prefix of $t[j,l]$ for a cost of at most $(k-1) - i + 1 - 1 = k - i - 1$ substitutions (recall that $s[i] = t[j]$). Then, insert spaces at the end of $s[i, k-1]$ to fully align it with $t[j, l-1]$. Since $s[k] = t[l]$, we get a full alignment of $s[i,k]$ with $t[j,l]$. It is easy to see that the total cost of the transcript corresponding to this alignment is, at most, $l - j - 1$ edit operations.   □

In line with the above lemma, we define the concept of a legal edit transcript as follows. For this let *consecutive matches* in an edit transcript be two matches ("M's") without any other match in between.

Definition 2.4.   Consider an edit transcript for substrings $s[i,k]$ and $t[j,l]$. Further, let $s[x] = t[v]$ and $s[y] = t[w]$ for $i \leq x < y \leq k$ and $j \leq v < w \leq l$ be two arbitrary *consecutive matches* in this edit transcript. We call the edit transcript *legal* if the number of edit operations between any such pair of consecutive matches does not exceed $max(y-x, w-v) - 1$.

To illustrate, Figure 2 (left) shows an example of an illegal edit transcript for the strings abced and aebced. The transcript is illegal because the number of edit operations (equal to 5) exceeds the total number of letters between the first and last positions of the longer string. The same figure (middle) shows a legal edit transcript for the same strings, while on the right an edit transcript with the minimum possible amount of edit operations (equal to the edit distance) is shown.

We also observe that the number of edit operations between two consecutive matches $s[x] = t[v]$ and $s[y] = t[w]$ in an edit transcript cannot be less than (surprisingly the same bound as before) $max(y - x, w - v) - 1$, regardless of whether the transcript is legal or not. The reason is that there are no other matches between two consecutive matches. Therefore, the total number of edit operations cannot be less than the larger number of characters between two matches.

Thus, in a legal transcript, the number of edit operations between two consecutive matches (M's) $s[x] = t[v]$ and $s[y] = t[w]$ is in fact equal to $max(y - x, w - v) - 1$. We show the following lemma.

LEMMA 2.5. *Let s and t be two strings over alphabet* Σ. *For any legal edit transcript between two substrings $s[i, k]$ and $t[j, l]$, there exists a path π between vertices v_{ij} and v_{kl} in $G_\mathcal{M}$ such that $EN(\pi)$ is equal to the number of edit operations in this edit transcript.*

PROOF. Let us consider some legal edit transcript for $s[i, k]$ and $t[j, l]$. We show that we can find a path in $G_\mathcal{M}$, which passes through vertices corresponding to the transcript matches, and whose error number is equal to the transcript cost (number of edit operations).

Let $s[x] = t[v]$ and $s[y] = t[w]$ be two consecutive matches in this transcript. From the above discussion, we know that the number of edit operations between these two matches (in the transcript) is equal to $max(y - x, w - v) - 1$.

By the construction of $G_\mathcal{M}$, we have that v_{xv} and v_{yw} are vertices in $G_\mathcal{M}$ and $e(v_{xv}, v_{yw})$ is an edge in $G_\mathcal{M}$. The cost of this edge, by definition, is equal to $max(y - x, w - v) - 1$.

Hence, we can always find a path π from v_{ij} to v_{kl} with error number equal to the number of edit operations in our edit transcript. Namely, π consists of the edges connecting the vertices corresponding to the matches in the edit transcript. □

We are ready to prove our *characterization theorem*.

THEOREM 2.6. *The edit distance between $s[i, k]$ and $t[j, l]$ is equal to the error number of the cheapest path(s) from v_{ij} to v_{kl} in $G_\mathcal{M}$.*

PROOF. The edit distance between $s[i, k]$ and $t[j, l]$ corresponds to the number of edit operations in some edit transcript. Such an edit transcript surely is a legal one, and it has the minimum possible number of edit operations. From this and Lemma 2.5, we conclude that the path corresponding to this edit transcript is a cheapest path from v_{ij} to v_{kl} in $G_\mathcal{M}$. □

Clearly, only the cheapest paths from v_{ij} to v_{kl} have an error number which is equal to the edit distance between $s[i, k]$ and $t[j, l]$. We call a path with error number less than or equal to K a *path below (error) threshold*. We present now the following problem.

Problem 2.7. **All paths below threshold**
INPUT: Graph $G_\mathcal{M}$ for two strings s and t, and positive integers K and S.
OUTPUT: All maximal paths below threshold K, and with match length at least S.

Based on Theorem 2.6 we conclude that:

COROLLARY 2.8. *The problem* all error-bounded approximate matches *can be reduced to the* all paths below threshold *problem.*

We show in Subsection 3.1 how to construct, in linear time and space, an instance for *all paths below threshold* from an instance of *all bounded approximate matches*.

How do solutions for *all paths below threshold* look like? We remark that in $G_\mathcal{M}$ there can exist multiple maximal paths below threshold connecting the same two vertices v_{ij} and v_{km}. However, we need to detect only one such path in order to produce $(s[i, k], t[j, m])$ as an approximate match. Based on the properties of $G_\mathcal{M}$ that we show in the next section, we devise a highly optimized depth-first search approach, which takes special care of path expansions and overlap. This way we avoid shortest path methods, which are associated with high overhead and rigidity of path expansions. Still we produce best maximal paths, even without using shortest-path methods.

Further, observe that every solution path π is a path between two character matches. Thus, the transcript for a substring match $(s[i, k], t[j, m])$ starts and ends with a character match, and solutions having an error number less than K can be extended by up to $K - EN(\pi)$ end gaps. It is easy to see that, from the solution set for Problem 2.7, all these solutions for Problem 2.1 can be produced in linear time.

We conclude this section with a remark on the maximality of the paths versus the maximality of the solutions to Problem 2.1. Namely, each maximal solution to Problem 2.1 has a corresponding maximal path, which is a solution to Problem 2.7. However, there are maximal paths produced as solution to Problem 2.7, which correspond only to subsolutions (i.e., nonmaximal solutions) to Problem 2.1. We illustrate such cases in Subsection 3.3. Although these subsolutions can be eliminated in a postprocessing step, we show in Subsection 3.3 how to eliminate the paths producing subsolutions during the main processing of our algorithm.

3. SOLVING "ALL PATHS BELOW THE THRESHOLD"

In this section, we present our algorithm "all paths below threshold" (APBT), which solves Problem 2.7 in $O(MNK^3)$ time.

3.1 Constructing the Matching Matrix

Clearly, we can avoid explicitly building the matching matrix $\mathcal{M}_{s,t}$. For this, when we are about to access, say cell $\mathcal{M}_{s,t}[i, j]$, we only need to test on the fly whether or not $s[i] = t[j]$. However, since the same cell might be accessed multiple times, comparing on the fly characters adds considerable time to the overall execution. For this reason, we present here how to construct the (boolean) matching matrix $\mathcal{M}_{s,t}[i, j]$ in only linear time and using space of $O(N \cdot |\Sigma| + M)$ (where $|\Sigma|$ is the size of the alphabet) as opposed to the naïve construction, which takes quadratic time and space.

We read the string t from left to right. When we read a letter a, say at position i of t, we first check if we have a bit array representing this letter. If not already created, we create a new bit array BA_a of size N, and initialize its bits to 0.

We set to 1 the ith bit in this BA_a array. At the end of the scanning process, each array BA_a records all the positions in t where the corresponding letter a occurs. Formally,

$$BA_a[i] = \begin{cases} 1 & \text{if } t[i] = a \\ 0 & \text{otherwise} \end{cases}$$

After this, we scan the other string s. For each letter a that we read from s, we create a pointer to the previously constructed array BA_a. At the end of scanning we obtain an array of M pointers, each pointing to some bit array. Note that we only scan once both s and t, and the total memory we need for $\mathcal{M}_{s,t}$ is $O(|\Sigma| \cdot N + M)$, where M (the length of s) is the number of pointers to the bit vectors.

Let's turn our attention to graph $G_{\mathcal{M}}$. We stress here that we never explicitly construct and store it, remaining so linear with respect to space. Rather, as we show, we traverse it by constructing the needed paths "on-the-fly."

3.2 A Single-Step Path Expansion

In our search for all maximal paths below threshold K, we use an optimized depth-first search. We scan the matching matrix in row-major order, i.e., we scan the first row from left to right, then the second, etc. When a vertex of $G_{\mathcal{M}}$ is encountered during the scan of $\mathcal{M}$, we initialize a path π with $EN(\pi) = 0$ and $ML(\pi) = 1$. Then, the algorithm recursively builds all the possible expansions of this initial path by adding one vertex at a time and by keeping track of the error numbers of the best paths found so far. Such error numbers are used as bounds for future candidates. As path π is constructed, the algorithm checks the following. If no more vertices can be added without exceeding threshold K, then we stop the further expansion of π, and check whether $ML(\pi) \geq S$. If true, then we consider path π *completed* and report it as a solution. Otherwise, path π is *aborted*. Obviously, we also stop expanding a path when the last row or last column of the matching matrix $\mathcal{M}_{s,t}$ is reached.

Next we describe in detail a single-step path expansion. Since the error number of a path cannot exceed K, an edge to be appended to a path clearly has to have a cost of at most K. As a consequence, all edges in $G_{\mathcal{M}}$ of cost higher than K are excluded from further consideration.

Consider a path π with error number $EN(\pi)$, which ends at vertex v_{ij}. From the above discussion, it is clear that for a single-step expansion of π we need to search (in $\mathcal{M}$) for a possible "next vertex" only inside square $ABCD$, with top-left corner $A = (i + 1, j + 1)$, and bottom-right corner $C = (i + 1 + \kappa, j + 1 + \kappa)$, for $\kappa = K - EN(\pi)$. We call square $ABCD$ the *target square* for path π at vertex v_{ij} (see Figure 3, left).

Note that, as the *error number* of the growing path increases, the area of the corresponding target square decreases (see Figure 3, right).
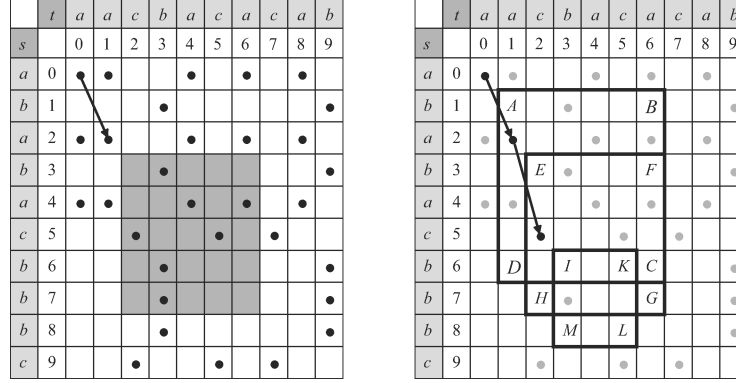
	t	a	a	c	b	a	c	a	c	a	b
s		0	1	2	3	4	5	6	7	8	9
a	0	•	•			•		•		•	
b	1				•						•
a	2	•	•			•		•		•	
b	3				•						•
a	4	•	•			•		•		•	
c	5			•			•		•		
b	6				•						•
b	7				•						•
b	8				•						•
c	9			•			•		•		

	t	a	a	c	b	a	c	a	c	a	b
s		0	1	2	3	4	5	6	7	8	9
a	0	•	•			•		•		•	
b	1		A		•			B			•
a	2	•	•			•		•		•	
b	3		E		•			F			•
a	4	•	•			•		•		•	
c	5			•			•		•		
b	6		D		I		K	C			•
b	7		H		•			G			•
b	8			M	L						•
c	9			•			•		•		

Fig. 3. (*Left*) Target square for path π going from v_{00} to v_{21}, $K = 5$ and $\kappa = 4$. (*Right*) The area of the target square decreases as the accumulated error number increases: *ABCD* is the target square for the path starting at v_{00}; *EFGH* is the target square for the expanded path ending at v_{21}; *IKLM* is the target square for the further expanded path ending at v_{52}.

On the first sight it seems that for any vertex v_{ij} in $G_{\mathcal{M}}$ there are, at most, $(\kappa+1) \times (\kappa+1)$ outgoing edges to be considered. Next, we show how to reduce the number of edges for consideration even further. Toward this end, we introduce the following definitions regarding diagonals in the matching matrix $\mathcal{M}$.

Definition 3.1. Let (i, j) be an arbitrary cell in the matching matrix $\mathcal{M}$.

1. The (i, j)-*main diagonal* for $\mathcal{M}$ is the sequence of $(i + p, j + p)$-cells in $\mathcal{M}$, where $0 \le p \le min\{M - i, N - j\}$.
2. Let q be a value between 0 and $N - j$. The (i, j)-*q-upper diagonal* is the $(i, j + q)$-main diagonal.
3. Let r be a value between 0 and $M - i$. The (i, j)-*r-lower diagonal* is the $(i + r, j)$-main diagonal.

Figure 4 illustrates the above definitions. Note that the (i, j)-main diagonal can be considered as (i, j)-0-upper diagonal, as well as (i, j)-0-lower diagonal.

For the further development of our method, it is useful to visualize the cells on a given diagonal as points on a two-dimensional plane. Then, a (given) diagonal can be visualized as a straight line passing through these points, and forming a 45° angle with the horizontal and vertical axes (cf. Figure 4).

Let π be a path in $G_{\mathcal{M}}$ ending at vertex v_{ij} and with $EN(\pi) < K$.

Now, assume that v_{kl} and v_{mn} are two vertices in the target square for π at v_{ij}, which satisfy the following conditions (illustrated in Figure 5, left):

(**C1**) $i < k < m$ and $j < l < n$,

(**C2**) $l - j \ge k - i$, and

(**C3**) $n - l \ge m - k$. From condition **C1**, we know that the edges $e(v_{ij}, v_{kl})$, $e(v_{kl}, v_{mn})$, and $e(v_{ij}, v_{mn})$ do exist in $G_{\mathcal{M}}$. Condition **C2** is equivalent to $\frac{k-i}{l-j} \le 1$ and, therefore, to $\frac{k-i}{l-j} \le \tan(45°)$. This means, that the line segment connecting (k, l) and (i, j) forms an angle less or equal to 45° with the horizontal axis (see
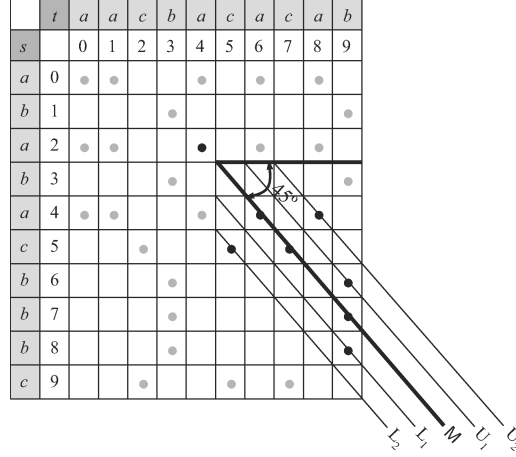
	t	a	a	c	b	a	c	a	c	a	b
s		0	1	2	3	4	5	6	7	8	9
a	0	●	●			●		●		●	
b	1				●						●
a	2	●	●			●		●		●	
b	3				●						●
a	4	●	●			●		●		●	
c	5			●			●		●		
b	6				●					●	
b	7				●				●		
b	8				●			●			
c	9			●			●		●		

Fig. 4. Diagonals for vertex v_{24}. M stands for *main diagonal*, U_1 for $(2, 4)$-1-*upper diagonal*, L_1 for $(2, 4)$-1-*lower diagonal*, etc.
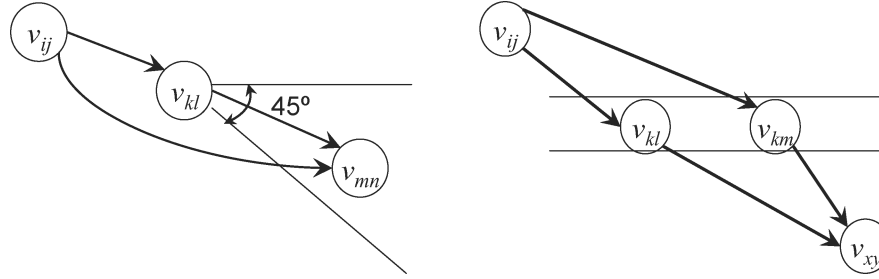
Fig. 5. (*Left*) Reverse Triangle Inequality: $c(v_{ij}, v_{kl}) + c(v_{kl}, v_{mn}) < c(v_{ij}, v_{mn})$ (*Right*) Two-edge paths between v_{ij} and v_{xy}. We show that $c(v_{ij}, v_{kl}) + c(v_{kl}, v_{xy}) = c(v_{ij}, v_{km}) + c(v_{km}, v_{xy})$.

Figure 5, left). We conclude that position (k, l) lies on an upper diagonal with respect to (i, j).

Reasoning similarly, condition **C3** implies that position (m, n) lies on an upper diagonal with respect to (k, l). We show the following theorem.

THEOREM 3.2 (REVERSE TRIANGLE INEQUALITY). *Let π be a path in $G_{\mathcal{M}}$ ending at vertex v_{ij} and with $EN(\pi) \leq K$. Further assume that v_{kl} and v_{mn} are two vertices in the target square for π at v_{ij}, satisfying conditions **C1**, **C2**, and **C3**. Then*

$$c(v_{ij}, v_{kl}) + c(v_{kl}, v_{mn}) < c(v_{ij}, v_{mn}).$$

PROOF. From the definition of $c(\_, \_)$ and based on **C2**, we get

$$c(v_{ij}, v_{kl}) = max\{k - i, l - j\} - 1 = l - j - 1$$

Similarly, **C3** yields

$$c(v_{kl}, v_{mn}) = max\{m - k, n - l\} - 1 = n - l - 1$$

By adding up **C1** and **C2** we obtain $m - i \leq n - j$. Thus,

$$c(v_{ij}, v_{mn}) = max\{m - i, n - j\} - 1 = n - j - 1$$

Finally, adding up the expressions for $c(v_{ij}, v_{kl})$, $c(v_{kl}, v_{mn})$, and $c(v_{ij}, v_{mn})$ yields

$$l - j - 1 + n - l - 1 < n - j - 1 \equiv -1 < 0 \qquad \square$$

Based on the above theorem, we state the following corollary, which captures our first optimization regarding the single-step path expansions.

COROLLARY 3.3. *Let π be a path in $G_{\mathcal{M}}$ ending at vertex v_{ij} and with $EN(\pi) \leq K$. Further assume that v_{kl} and v_{mn} are two vertices in the target square for π at v_{ij}, satisfying conditions **C1**, **C2**, and **C3**. Then extending path π to vertex v_{kl} and then to v_{mn} is cheaper than directly extending π to v_{mn}.*

The above corollary implies that, if we build an edge from v_{ij} directly to v_{mn}, we "ignore" vertex v_{kl} and unnecessarily increase $EN(\pi)$. Rather, we better expand path π to v_{kl} and later on, in the next round, continue to v_{mn}.

We obtain analogous results for the symmetrical conditions when the vertices v_{kl} and v_{mn} are in the lower part of the target square.

Let us take a more careful look at Corollary 3.3. Practically, it says that, if we find a vertex v_{kl}, which lies on an upper diagonal of the target square, then we can exclude from the search for expansion all the triangular area of the target square, which is bounded by

1. row k (exclusive), and
2. the upper diagonal passing through v_{kl} (inclusive).

Symmetrically, if vertex v_{kl} lies on some lower diagonal, then we can exclude from the search for expansion all the triangular area of the target square, which is bounded by

1. column l (exclusive), and
2. the lower diagonal passing through v_{kl} (inclusive).

If vertex v_{kl} lies on the main diagonal of the target square, then both triangular areas are excluded at once.

We strengthen the search-space reduction for path expansions even further. Consider two vertices v_{kl} and v_{km} in the same row of the target square for a path π at vertex v_{ij}, such that

1. $l - j \geq k - i$, i.e., v_{kl} lies on an upper diagonal, and
2. $m > l$, i.e., v_{kl} is closer than v_{km} to the main diagonal (see Figure 5, right).

Let v_{xy} be any vertex inside the target square for some path π', an extension of π that is ending at vertex v_{km}.

THEOREM 3.4. *Let vertices v_{ij}, v_{kl}, v_{km}, and v_{xy} be as above. The error number of the two-edge path from v_{ij} to v_{xy}, which passes through v_{km}, is equal to the error number of the two-edge path from v_{ij} to v_{xy}, which passes through v_{kl}, i.e.*

$$c(v_{ij}, v_{kl}) + c(v_{kl}, v_{xy}) = c(v_{ij}, v_{km}) + c(v_{km}, v_{xy})$$

		t	a	a	c	b	a	c	a	c	a	b
	s		0	1	2	3	4	5	6	7	8	9
a	0		•				•		•		•	
b	1											•
a	2		•	•		•		•		•		
b	3					•						•
a	4		•	•			•		•		•	
c	5				•			•		•		
b	6					•						•
b	7					•						•
b	8					•						•
c	9				•			•		•		

Fig. 6. Search space reduction: None of the shadowed cells of the target square for v_{01} is tested after v_{13} is encountered.

Hence, it is sufficient to collect only the path closer to the main diagonal.

PROOF. $c(v_{ij}, v_{kl}) + c(v_{kl}, v_{xy}) = (l - j - 1) + (y - l - 1) = y - j - 2$ and
$c(v_{ij}, v_{km}) + c(v_{km}, v_{xy}) = (m - j - 1) + (y - m - 1) = y - j - 2.$ □

Observe the following implication based on Theorem 3.4. If a vertex v_{kl} is detected on an upper diagonal while scanning the target square for vertex v_{ij}, then the cells on the right of v_{kl} in row k can be safely excluded from further search for expansions.

Symmetrically, if a vertex v_{kl} is detected on a lower diagonal while scanning the target square for vertex v_{ij}, then the cells below v_{kl} in column l can be safely excluded from further search for expansions.

Based on Corollary 3.3 and the above discussions about Theorem 3.4 (and its symmetrical case), we present the following optimization.

Optimization 3.5. In search for expansions, we scan the cells of the target square in a diagonal-major order, that is: First scan the main diagonal, possibly excluding parts of the target square from further scan. Next, scan the remaining of the target square through the 1-upper and the 1-lower diagonal, possibly excluding other areas of the target square. Then, continue with the 2-upper and the 2-lower diagonal, and so on.

Observe that the scanning of a target square in this order guarantees that the exclusion of triangular areas takes place *as early as possible*. Figure 6 illustrates this search space reduction.

We state two important corollaries that follow from the above discussion.

COROLLARY 3.6. *Single-path extension from an arbitrary vertex v_{ij} in $G_{\mathcal{M}}$ is performed, at most, once for each of the $2K + 1$ diagonals surrounding v_{ij}, and, therefore, the number of possible extensions for v_{ij} is bounded by $2K + 1$.*

COROLLARY 3.7. *An arbitrary cell of a matrix $\mathcal{M}[i, j]$ is accessed from, at most, one node of each of the $2K + 1$ surrounding diagonals. This also implies that an arbitrary vertex v_{ij} serves as a single path extension starting from, at most, $2K + 1$ vertices.*
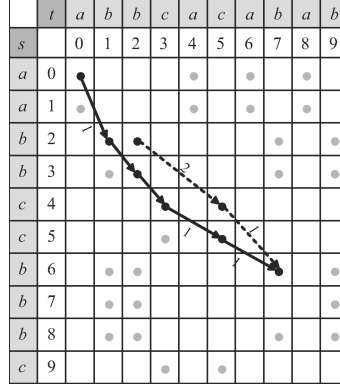
Fig. 7. Interdependence of paths in the graph. The path from v_{00} to v_{67} has $EN = 3$, and the path from v_{22} to v_{67} also has $EN = 3$. We abort the second path.

3.3 Interdependence of Paths in the Graph

We show next how the information from previously explored paths can be reused. Consider the situation that, while scanning the matching matrix, two encountered (sub)paths start at different vertices and end at the same vertex.

Let π_1 be the previously explored path, which connects vertex v_{ij} with v_{mn}. Now, let π_2 be another path that we are currently exploring, which originates in v_{kl}, and is built up to vertex v_{mn}.

The question is whether we should further expand π_2, or safely abort it without loosing any solution. We distinguish three possible cases for π_1 and π_2 with respect to their match length and error number and decide whether to expand or abort π_2.

Case 1.. $EN(\pi_2) < EN(\pi_1)$.

We expand π_2, since π_2 offers a further or better solution than the one discovered by π_1.

Case 2.. $EN(\pi_2) \geq EN(\pi_1)$ and $ML(\pi_2) \leq ML(\pi_1)$

In this case all possible expansions of π_2 from v_{mn} are subsets of already tested expansions of π_1. The part π_2 up to v_{mn} has shorter match length that π_1 and, therefore, no new information can be obtained by expanding π_2. Path π_2 can be aborted.

Note that in this way we may ignore some maximal path, but such a path would only correspond to a nonmaximal solution for Problem 1.

To illustrate consider Figure 7. The path from v_{00} to v_{67} serves as π_1 ($EN(\pi_1) = 3$), which is explored earlier in a row-major order. On the other hand, the path from v_{22} to v_{67} serves to exemplify π_2 ($EN(\pi_2) = 3$). Clearly, path π_2 will only offer a subsolution to the solution corresponding to π_1, since the substring $s[2, 6]$ is a substring of $s[0, 6]$ and $t[2, 7]$ is a substring of $t[0, 7]$.

Case 3.. $EN(\pi_2) \geq EN(\pi_1)$ and $ML(\pi_2) > ML(\pi_1)$

In this case π_2 follows the trails of some already tested expansions of π_1, but by extending π_2 we can obtain a longer path, and possibly contribute a new solution.

Case 3 is the only case where both a path expansion or a path abortion may be the right decision.

We want to ensure that we abort π_2 only if it cannot offer a new solution. We achieve this as follows.

Each path has its starting vertex. All the starting vertices of paths ending at v_{mn} must lie on one of the $2K + 1$ diagonals surrounding the diagonal of v_{mn}. Otherwise, a path ending at v_{mn} will have more than K errors.

Now suppose that paths π_1 and π_2 start on the same diagonal and end at the same vertex v_{mn}. Also, suppose that the starting vertex of π_2 is lower than the starting vertex of π_1. It is easy to see that π_2 corresponds to a pair of substrings, which are suffices of the substrings corresponding to π_1.

Now, because of the row-wise order of processing, π_2 will always be processed after π_1. Clearly, we want to expand π_2 beyond v_{mn} only if it has an error number which is less than the error number of π_1. Otherwise, we can safely abort π_2.

Since all the paths passing through v_{mn} can only start in the $2K + 1$ diagonals surrounding v_{mn}, we store for v_{mn} an array of $2K + 1$ best-error values. We then we expand a path beyond v_{mn} only if this path has an error number which is smaller than the best-error value stored for the diagonal of its starting vertex.

Summarizing, we present the following optimization.

Optimization 3.8.   For each vertex v_{mn} we remember the best error numbers (one per diagonal) of all paths that started at the $2K + 1$ diagonals surrounding v_{mn} and reached this vertex.

Each expansion from v_{mn} starts by checking whether the path constructed so far has an error number greater or equal to the value stored for the diagonal corresponding to the starting vertex of the path. If so, the current path is aborted.

The pseudocode of our algorithm is given in Figure 8.

A last note is about storing the information regarding the "so far" best error number of paths ending at the examined vertices. As we are looking for local similarities, we solved this problem by setting an (artificial) upper bound $S_{max} > S$ for the matches. In such a way, we can use $N \cdot (2K + 1) \cdot S_{max}$ bounded memory[3] in the form of a rotating two-dimensional $N \times S_{max}$ array A (of objects holding up to $2K + 1$ values). Namely, a row of A corresponds to a row of $\mathcal{M}$. Initially, rows 0 to S_{max} of A correspond to rows 0 to S_{max} of $\mathcal{M}$, i.e., the best path error numbers for vertices $v_{i\_}$, where $i < S_{max}$, are stored in row i of A. When we finish processing row 0 of $\mathcal{M}$, we then recycle row 0 of the array to store the best-path errors for the vertices of row S_{max}, and so on. In general, when we ask for some best path error for a vertex v_{ij} we consult the row $i \bmod S_{max}$ of A.

We stop expanding the solution if the length exceeds S_{max}. In practice, we choose $S_{max} = 500$, which seems high enough, since the matches never exceeded it in our experiments. $(1000 + S_{max}) \times S_{max}$, in order to never exceed the linear space while processing two long strings. We stress that setting an artificial upper bound S_{max} is not really a limitation of our algorithm. If there exist

---

[3]This is less than the memory required by Baeza-Yates and Gonnet's algorithm to store the dynamic programming tables.

All_paths_below_threshold($\mathcal{M}$,K,S)
  initialize array *BestErrors*
  scan $\mathcal{M}$ in row major order
  if $\mathcal{M}[i,j] = 1$ **then**
    create path π consisting of single-vertex v_{ij}
    $EN(\pi)$: $= 0$
    $ML(\pi)$: $= 1$
    Expand_path(π)

Expand_path (π)
  if Is_Safe_To_Abort(π) **then**
    return

  Let v_{ij} be the start vertex of π
  Let v_{mn} be the end vertex of π

  κ:= K - $EN(\pi)$
  scan target square $\kappa \times \kappa$ in diagonal-major order
    (starting from the main diagonal)

  if a vertex v_{xy} is encountered **then**
    expand π into π'
    $EN(\pi')$: $= EN(\pi)+max(x-m-1, y-n-1)$
    $ML(\pi')$: $=min(x-i+1, y-j+1)$
    Expand_path(π')

  if $ML(\pi) \geq S$ **then**
    add π to the set of solutions

Is_Safe_To_Abort(π)
  Let v_{ij} be the start vertex of π
  Let v_{mn} be the end vertex of π
  $diagonalID := K + (m-i) - (n-j)$
  /*diagonalID is an identifier of the diagonal passing through v_{ij}.
  Such ids are calculated relative to end point v_{mn}. */

  if $BestErrors[v_{mn}] = null$ **then**
    $BestErrors[v_{mn}]$:= **new array** $[2K+1]$
    $BestErrors[v_{mn}][diagonalID] := EN(\pi)$
    return *false*

  if $BestErrors[v_{mn}][diagonalID] = null$ **or**
    $BestErrors[v_{mn}][diagonalID] > EN(\pi)$ **then**
    $BestErrors[v_{mn}][diagonalID] := EN(\pi)$
    return *false*

  return *true*

Fig. 8.   Pseudocode of the *APBT* algorithm.

oversize matches, then they can be easily obtained as a postprocessing step in linear time with respect to the size of output.

From the above, we can formulate the following lemma.

LEMMA 3.9. *Each vertex v_{mn} in the graph is expanded, at most, $(2K + 1) \cdot (K + 1)$ times.*

PROOF. $K + 1$ is the maximum number of different possible error numbers. Each such error number can occur for each of the $2K + 1$ diagonals that surround v_{mn}. □

THEOREM 3.10. *Algorithm all paths below threshold has a time complexity of $O(MNK^3)$.*

PROOF. During path extension, any cell of the matrix is directly accessed only from at most $2K + 1$ vertices (cf. Corollary 3.7). However, each of those $2K + 1$ vertices gets expanded not more than $(2K + 1)(K + 1)$ times (cf. Lemma 3.9). Therefore, the upper bound of accessing an arbitrary cell of the matrix is $(2K + 1)(2K + 1)(K + 1)$. Since there are, at most, MN many cells in $\mathcal{M}$, the total time complexity is $O(MNK^3)$. □

4. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our *APBT* algorithm, as well as a comparison with the previous algorithm for this problem, namely, the well-known algorithm by Baeza-Yates and Gonnet [Baeza-Yates and Gonnet 1999].

All experiments were performed on the same 3-GHz Pentium 4 PC with 2-GB RAM. We implemented both algorithms in Java 1.5.

4.1 Random Sequences

First, we discuss the average running time of our algorithm with respect to the number of allowed differences K and lengths M and N of the input sequences. For this, we generated random strings with uniform character distribution over two alphabets of sizes 4 and 20. Every point in the following figures corresponds to an average value obtained by running the algorithm on ten different pairs of sequences of the same length and character distribution.

In Figure 9, we observe that although the worst-case bound of *APBT* is cubic in K, on average, *APBT* depends only quadratically on K. Also, we observe that for alphabets of larger size, such as 20, *APBT* performs much better than for alphabets of smaller size, such as 4. This is explained by the fact that with the increase of the alphabet size, the matching matrix used in *APBT* becomes more sparse.

In Figure 10, we show the dependence of *APBT* on the length of sequences. For this experiment, we considered pairs whose first and second sequences were of equal lengths, i.e., $N = M$. It is evident from the presented results that *APBT* is quadratic with respect to length N of input sequences.
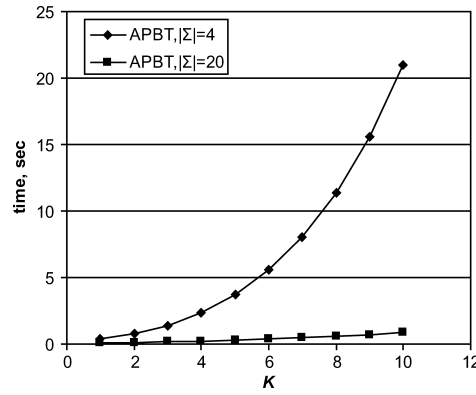
Fig. 9. Dependence of *APBT* on *K* for pairs of randomly generated sequences of length 2000. The minimum pattern length is $S = 30$.
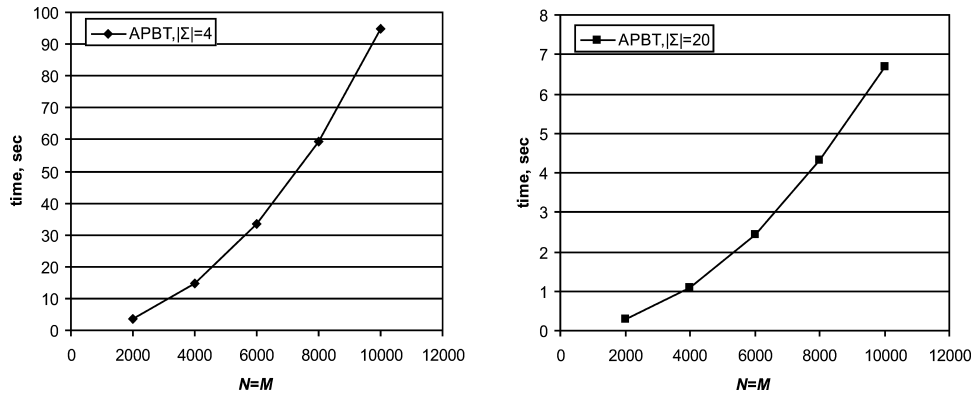

Fig. 10. Dependence of *APBT* on the length of sequences (random strings, $S = 30$, $K = 5$) (*Left*) $|\Sigma| = 4$; (*Right*) $|\Sigma| = 20$.

4.2 Biological Sequences

Next, we evaluate the *APBT* algorithm on biological sequences. In these experiments, we investigate the dependence of the running time on the alphabet size of biosequences, as well as on the similarity of biosequences as measured by the output size produced by *APBT*.

For evaluating the dependence on the alphabet size, we considered DNA and protein sequences, which have alphabets of size 4 and 20, respectively. We are aware that, for proteins, the unit-cost edit distance assumed in *APBT*, might often not be used in practice. Neverthless, it is still interesting to find out if protein sequences contain approximate matches with up to 20% differences (i.e., 80% identical characters).

The considered pairs of DNA sequences were (D1, D2), (D1, D3), (D4, D5), and (D4, D6) where D1, D2, D3, D4, D5, and D6 were the following genomic

Table I. Running Times and Output Sizes for Biological Sequences[a]

K	D1-D2 (T)	D1-D2 (O)	D1-D3 (T)	D1-D3 (O)	D4-D5 (T)	D4-D5 (O)	D4-D6 (T)	D4-D6 (O)
1	1	0	1	0	4	7	4	0
2	3	0	3	0	9	40	9	0
3	5	0	5	0	16	160	17	0
4	9	0	8	0	28	400	28	0
5	15	0	13	0	45	927	45	0
6	22	7	19	1	68	2,225	69	0
7	32	41	28	9	100	4,837	100	0
8	45	146	40	34	141	9,478	143	0
9	63	385	55	143	195	17,309	196	0
10	84	922	74	439	262	29,671	264	0

K	P1-P2 (T)	P1-P2 (O)	P1-P3 (T)	P1-P3 (O)
1	0.3	13,427	0.4	5
2	0.5	40,143	0.5	25
3	0.7	87,518	0.8	58
4	1.0	163,197	1.2	154
5	1.4	272,467	1.7	421
6	1.9	415,222	2.2	965
7	2.5	599,626	2.9	1783
8	3.2	822,541	3.8	3275
9	4.1	1,090,377	4.8	5947
10	5.4	1,403,277	6.0	10263

[a] T is for time (mins) and O is for output size. Parameter S is 50.

sequences:

  D1. Human coronavirus 229E (27,317 bp)

  D2. Human coronavirus OC43 (30,738 bp)

  D3. Avian infectious bronchitis virus strain Cal99 (27,693 bp)

  D4. Mycobacteriophage D29 (49,136 bp)

  D5. Mycobacteriophage Bxb1 (50,550 bp)

  *D6*2 Mycobacteriophage D29 (49,136 bp)

The considered pairs of protein sequences were (P1, P2) and (P1, P3), where P1, P2, and P3 were the following protein sequences:

  P1. Titin isoform N2-A [*Homo sapiens*] (33,423 aa)

  P2. Titin [*Mus musculus*] (26,886 aa)

  P3. Titin A [*Danio rerio*] (32,757 aa)

Sequences D1 and D2 are more biologically related. The same is true for D4 and D5, as well as for P1 and P2. A natural question is whether our *APBT* algorithm depends on the amount of similarity of compared sequences as measured by the size of output. Interestingly, the observed running times do not show any significant dependence on the similarity of the compared sequences.

In Table I, we show the running times of *APBT* as well as the size of output for pairs (D1, D2), (D1, D3), and (D4, D5), (D4, D6) (top) and (P1, P2), (P1, P3) (bottom). We observe that the running times for (D4, D5) and (D4, D6) are almost identical despite the fact that the output sizes differ significantly. We observe a similar phenomenon for pairs (D1, D2), (D1, D3) and (P1, P2),
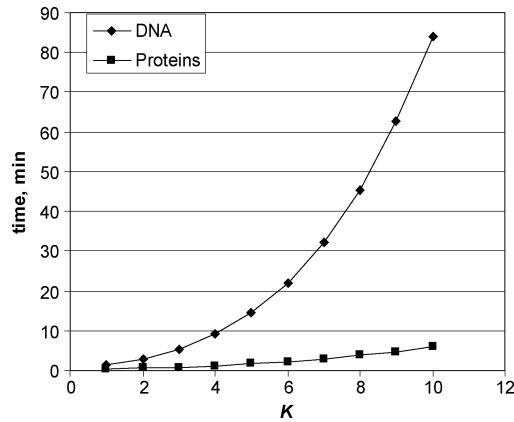
Fig. 11. Dependence of *APBT* on *K* for pairs D1-D2 and P1-P2 of lengths approximately 30,000. Parameter *S* is 50.

(P1, P3). The small differences in the running times for (D1, D2) versus (D1, D3) and (P1, P2) versus (P1, P3) are directly attributed to the differences in length between sequences D2 and D3 and between sequences P2 and P3. For example, for $K = 10$, the running times for (D1, D2) and (D1, D3) are 84 and 74 min, respectively. On the other hand, the length of D2 is 30,738 bp, while the length of D3 is 27,693 bp (D1 is common for both pairs). We can observe that $84/74 \approx 30,738/27,693$. Thus, we conclude that the running time of *APBT* does not (noticeably) depend on the amount of similarities, as measured by the size of the produced output.

In Figure 11, we show running times of the *APBT* algorithm for pairs (D1, D2) and (P1, P2). The sequences of these pairs are of approximately 30,000 characters. The results in this figure confirm that for larger alphabets *APBT* performs much faster. Also, we confirm once more that the dependence on *K* is, in fact, quadratic as opposed to cubic.

4.3 Comparative Performance

Finally, we compare the *APBT* algorithm with the algorithm by Baeza-Yates and Gonnet [Baeza-Yates and Gonnet 1999], which solves exactly the same problem. In fact, we implemented Gusfield's [Gusfield 1997] variant of the Baeza-Yates and Gonnet algorithm.[4] Also, we optimized it by an order of magnitude using Ukkonen's [Ukkonen 1985] error-bounded dynamic programming method.

Briefly described, Gusfield's variant of Baeza-Yates and Gonnet's algorithm is as follows. First, suffix trees are built for strings s and t. These suffix trees are then traversed and, for each pair of nodes (with one node from each tree), the edit distance between the substrings labeling the nodes is computed using dynamic programming (DP). The last row and column of the DP tables are used to initialize the (next) DP tables for the substrings of the child nodes.

---

[4]The original code of [Baeza-Yates and Gonnet 1999] is, unfortunately, not available anymore (personal communications with Baeza-Yates and Gonnet).
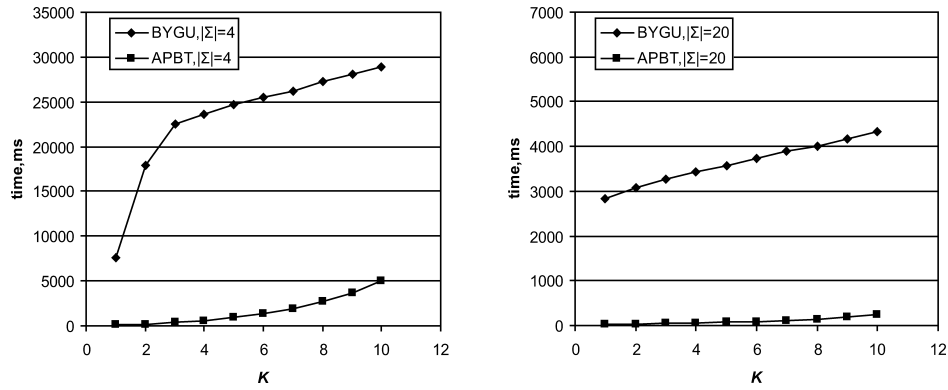
Fig. 12. Comparative performance of *APBT* and *BYGU* on pairs of random strings of length 1000. Parameter S is 30.

In an error-bounded variant, if all the values in the last column and last row of the dynamic programming table for the parent nodes exceed K, then we can safely abort traversing and testing the child nodes. This is true because of the non-decreasing nature of the values in the dynamic programming table. Early stopping this way tremendously improves the average running time of the algorithm for small K.

Regarding the worst case, the use of suffix trees still involves the comparison of (at most) MN different pairs of suffices by calculating values of MN dynamic programming tables of size (at most) $O(MN)$, having so an $O(M^2N^2)$ algorithm (in the worst case).

As mentioned above, since we are interested in an error-bounded version of the problem, we enhanced the original algorithm by using Ukkonen's linear time calculation of the error-bounded edit distance (see [Ukkonen 1985]). In our implementation of the algorithm by Baeza-Yates and Gonnet, we compute cells only in the area of width $2K + 1$ around the main diagonal of the dynamic programming tables. This reduces the worst case time to $O(M^2NK)$ (i.e., $O(MN \times M(2K + 1))$). We abbreviate this version of the algorithm as algorithm *BYGU*.

The major problem we faced in the implementation of the *BYGU* algorithm was its great space demand. This is because for each pair of nodes (with one node from each tree), we need to store the values of the last row and column of the DP table in order to reuse them in the DP calculations for their children. For real DNAs (of length in the order of $30,000 - 50,000$ characters), using any tree traversal strategy, there is an excessive number of such last rows and columns to be remembered, and the needed memory exceeds 2 GB (which was available to us). This fact may lead to the conclusion that our new *APBT* algorithm, to the best of our knowledge, is the only currently known feasible solution (which runs in a main memory of reasonable size) for the all-against-all problem on real DNA sequences.

In Figure 12, we present comparative results on random sequences of length 1000 generated with uniform character distribution (i.i.d.) for alphabets of size 4 and 20. The *BYGU* algorithm, because of its excessive computational demand, was not suitable to run on sequences of greater length.
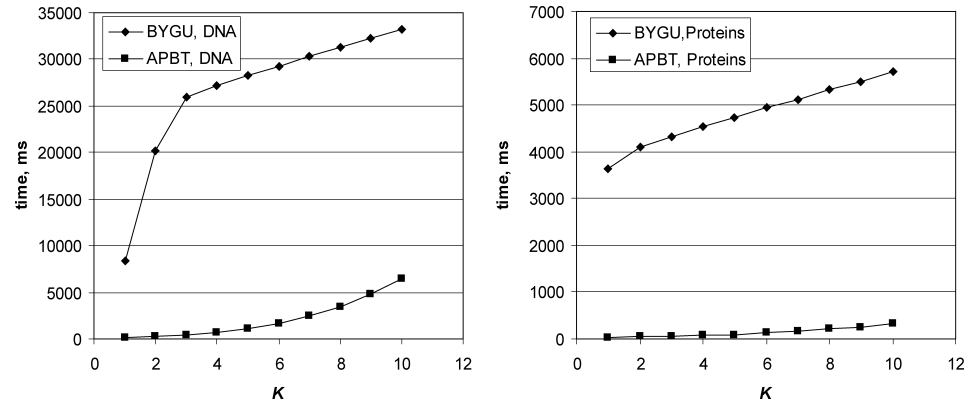
Fig. 13.   Comparative performance of *APBT* and *BYGU* on pairs of strings of length 1000 extracted from D1, D2 and P1, P2. Parameter *S* is 30.

In order to compare the performance of *APBT* with *BYGU* for biological sequences, we show in Figure 13 their running times for substrings of 1000 characters extracted from biological sequences D1, D2 and P1, P2.

We observe that *APBT* outperforms *BYGU* for each *K* and on each pair of sequences as shown by both Figures 12 and 13. For a comparison, *APBT* performs more than five times better than *BYGU* for alphabet of size 4 and more than 20 times better for alphabet of size 20.

5. RELATED WORK

Besides previous work mentioned in the Introduction, in this section, we discuss other related works which address variations of the same problem or use similar techniques.

5.1 Reducing the Search Space—Filtering Algorithms

In [Rasmussen et al. 2005], Rasmussen, Stoye, and Myers propose SWIFT, which is an efficient filtering method for identifying candidate parts of the input strings that can contain an approximate match of length above a given threshold and error number below some other (interrelated) threshold. Their method is based on the observation that if an area $s[i, j] \times t[k, l]$ of the DP table for the given input strings contains an approximate match of length S and error number K, then it must also contain at least $(S + 1) - q(K + 1)$ exact matches of length q (called "q hits"). In this way, all the areas in the DP table containing less than $(S + 1) - q(K + 1)$ q hits can be safely excluded from the search for approximate matches. The remaining areas still need to be searched for approximate matches by using a full sensitivity algorithm. We remark that Rasmussen, Stoye, and Myers [Rasmussen et al. 2005] do not propose such a full sensitivity algorithm. Clearly, SWIFT can be used as preprocessing step before applying our algorithm, which is a full sensitivity algorithm. Thus, SWIFT filtering and our APBT algorithm are complementary of each other rather than competing methods. We further comment on the usuability of SWIFT as follows.
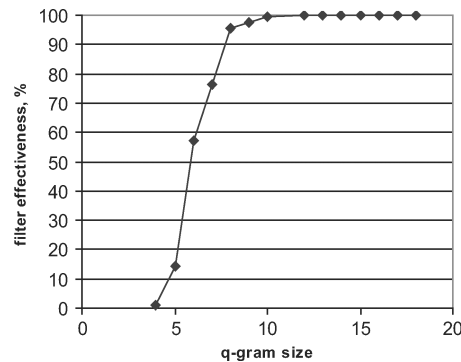
Fig. 14. Effectiveness of q-gram-based filtering. The percentage of the filtered-out entries in the DP table is plotted as a function of q.

SWIFT works well when the value of K is about (or less than) 5% of S, e.g., $S = 50$ and $K = 3$. For such parameters, which imply a search for q hits of length 11, Rasmussen, Stoye, and Myers obtained an impressive filtering rate on large DNA sequences, leaving for a full sensitivity search less than one-thousandth of the search space. Unfortunately, q inversely depends on K and SWIFT quickly deteriorates for $q < 7$, which corresponds to just $K = 7$ and S as above (i.e., 50). In Figure 14, we show the SWIFT filtering rate as a function of q. These rates were obtained on pair (D1,D2). We also experimented with the rest of the pairs considered in the previous section and found very similar filtering rates. The natural observation on Figure 14 is that the effectiveness of q-gram filtering steeply drops for $q \leq 7$.

SWIFT is currently the best known q-gram-based filtering software, which performs better than the previous QUASAR filtering program by Burkhardt et al. [Burkhardt et al. 1999].

5.2 Heuristic Search

Among the widely used local similarity search programs are the well-known BLAST and FASTA. Both are heuristic methods.

We remark that, in fact, BLAST does not solve the problem of threshold all-against-all substring matching. The substring patterns produced by BLAST are chains of exact matches (called *hotspots*) of length at least 7 (for DNAs). Hotspots are then chained together without allowing insertions and deletions. A hotspot is extended by another hotspot if the total score of the corresponding space-free alignment is greater than the scores for each hotspot before chaining. The chains with a score above some similarity threshold are reported as a solution. Unfortunately, BLAST cannot guarantee that the reported regions of compared sequences are, in fact, more similar than other nonreported regions. Also, BLAST misses all the approximate substring matches, which contain insertions and deletions as well as all approximate matches, which do not contain exact matches of length less than the predefined threshold (more or equal to 7 for DNAs).

FASTA exhibits similar issues as BLAST. The first step of FASTA is to find ten diagonal regions in the DP table having the highest density of exact matches for a predefined length. These regions are then further tested. FASTA's first step does not consider insertions and deletions and is based on the density of exact matches. As for BLAST, there is no guarantee that the reported patterns of compared sequences are, in fact, more similar than other nonreported patterns.

The above is summarized by Gusfield [Gusfield 1997, p. 377], as follows: "they [BLAST and FASTA] do not permit precise analysis of their speed and accuracy." Despite the fast performance of BLAST and FASTA, our full sensitivity APBT algorithm cannot be evaluated against them.

5.3 Similar Techniques

Here we highlight some graph models used in the literature for string-searching algorithms, which show some similarity to our graph model.

Kececioglu [Kececioglu 1993] defines an alignment graph. We remark that his so-called supervertices correspond to the vertices of our graph $G_{\mathcal{M}}$. The alignment graph is used for defining multiple sequence traces, which, when restricted to two sequences, coincide with traces as defined by Sankoff [Sankoff and Kruskal 1983]. These traces were also used by Reinert et al. [Reinert et al. 1997] to devise a branch-and-cut algorithm for the multiple-sequence alignment problem.

However, our $G_{\mathcal{M}}$ graph cannot be used to depict a trace. Although the vertices of $G_{\mathcal{M}}$ correspond to supervertices of the alignment graph, the definition of the edges in $G_{\mathcal{M}}$ is different from the one for edges between supervertices of the alignment graph. Namely, an edge is defined from one supervertex X to another supervertex Y iff there exists an element (character) $x \in X$, which immediately precedes (in one of the strings) some element (character) $y \in Y$. For the case of two strings, say s and t, suppose that (i, j) and (k, l) are two supervertices, i.e., $s[i] = t[j]$ and $s[k] = t[l]$. Furthemore, suppose that $k = i + 1$ and $j = l$. Then, there is an edge between supervertices (i, j) and (k, l). However, there is no edge between (i, j) and (k, l) in our $G_{\mathcal{M}}$ graph, because in $G_{\mathcal{M}}$, horizontal edges are not allowed. Similar reasoning holds for the case of vertical edges as well, which are also not allowed in $G_{\mathcal{M}}$. The only edges in common are the diagonal edges between (i, j) and $(i + 1, j + 1)$ in case $s[i] = t[j]$ and $s[i + 1] = t[j + 1]$. On the other hand, in our $G_{\mathcal{M}}$ graph, we have edges between nonadjacent matches as well.

The chaining techniques used in [Abouelhoda and Ohlebusch 2005; Höhl et al. 2002] to compute the heaviest chain are based on a graph model whose edges, in general, are a subset of the edges in $G_{\mathcal{M}}$. To prune, the authors exploit geometrical properties of their graph model, as well as a priority queue-based pruning technique.

We do not see, however, how to solve our problem efficiently using the techniques described in [Abouelhoda and Ohlebusch 2005; Höhl et al. 2002]. Note that both techniques solve variants of the global alignment problem. In general, any technique computing global alignments, certainly can be used to solve

problem 1 by computing such an alignment starting from each pair of positions, for a total cost of $O(N \cdot M \cdot \text{cost\_of\_the\_technique})$.

The chaining techniques [Abouelhoda and Ohlebusch 2005] are also used to solve a variant of the local alignment problem. However, computing local alignments does not provide a solution to problem 1, as already discussed in other works (cf., Arslan et al. [2001]).

6. CONCLUSIONS

In this paper, we focused on the problem of error-bounded all-against-all approximate substring matching for two strings. The main contribution of our work is a new algorithm, which presents a much more feasible solution to this difficult problem. The worst-case running time of our algorithm is $O(MNK^3)$, which is a significant improvement over the well-known algorithm for this problem by Baeza-Yates and Gonnet.

We conclude with the remark that our algorithm can be easily parallelized. This can be achieved by processing different subsets of starting points in the matching matrix, independently and in parallel, in different machines. This should make our algorithm practical for even very long input strings.

REFERENCES

ABOUELHODA, M. I. AND OHLEBUSCH, E. 2005. Chaining algorithms for multiple genome comparison. *J. Discrete Algorithms 3*, 2–4, 321–341.

ARSLAN, A. N., EGECIOGLU, Ö., AND PEVZNER, P. A. 2001. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics 17*, 4, 327–337.

BAEZA-YATES, R. A. AND GONNET, G. H. 1990. All-against-all sequence matching. Tech. rep., Department of Computer Science, Universidad de Chile.

BAEZA-YATES, R. A. AND GONNET, G. H. 1999. A fast algorithm on average for all-against-all sequence matching. In *SPIRE/CRIWG*. IEEE, Los Alamitos, CA. 16–23.

BAILEY, T. L. AND ELKAN, C. 1995. Unsupervised learning of multiple motifs in biopolymers using expectation maximization. *Machine Learning 21*, 1–2, 51–80.

BARSKY, M., STEGE, U., THOMO, A., AND UPTON, C. 2006. A new algorithm for fast all-against-all substring matching. In *SPIRE*, F. Crestani, P. Ferragina, and M. Sanderson, Eds. Lecture Notes in Computer Science, vol. 4209. Springer, New York. 360–366.

BURKHARDT, S., CRAUSER, A., FERRAGINA, P., LENHOF, H.-P., RIVALS, E., AND VINGRON, M. 1999. q-gram based database searching using a suffix array (quasar). In *RECOMB '99: Proceedings of the 3rd Annual International Conference on Computational Molecular Biology*. ACM, New York. 77–83.

EPPSTEIN, D., GALIL, Z., GIANCARLO, R., AND ITALIANO, G. F. 1992a. Sparse dynamic programming i: Linear cost functions. *J. ACM 39*, 3, 519–545.

EPPSTEIN, D., GALIL, Z., GIANCARLO, R., AND ITALIANO, G. F. 1992b. Sparse dynamic programming ii: Convex and concave cost functions. *J. ACM 39*, 3, 546–567.

GUSFIELD, D. 1997. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge.

HÖHL, M., KURTZ, S., AND OHLEBUSCH, E. 2002. Efficient multiple genome alignment. *Bioinformatics 18*, 312–320.

KECECIOGLU, J. D. 1993. The maximum weight trace problem in multiple sequence alignment. In *CPM*, A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, Eds. Lecture Notes in Computer Science, vol. 684. Springer, New York. 106–119.

LAWRENCE, C., ALTSCHUL, S., BOGUSKI, M., LIU, J., NEUWALD, A., AND WOOTTON, J. 1993. Detecting subtle sequence signals: A gibbs sampling strategy for multiple alignment. *Science 262*, 208–214.

RASMUSSEN, K. R., STOYE, J., AND MYERS, E. W. 2005. Efficient q-gram filters for finding all epsilon-matches over a given length. In *RECOMB*, S. Miyano, J. P. Mesirov, S. Kasif, S. Istrail, P. A. Pevzner, and M. S. Waterman, Eds. Lecture Notes in Computer Science, vol. 3500. Springer, New York. 189–203.

REINERT, K., LENHOF, H.-P., MUTZEL, P., MEHLHORN, K., AND KECECIOGLU, J. D. 1997. A branch-and-cut algorithm for multiple sequence alignment. In *RECOMB '97: Proceedings of the 1st Annual International Conference on Computational Molecular Biology*. ACM, New York. 241–250.

ROCKE, E. 2000. Using suffix trees for gapped motif discovery. In *CPM*, R. Giancarlo and D. Sankoff, Eds. Lecture Notes in Computer Science, vol. 1848. Springer, New Yorlk. 335–349.

ROTH, F., HUGHES, D., ESTEP, E., AND CHURCH, M. 1998. Finding dna regulatory motifs within unaligned non-coding sequences clustered by whole genome mrna quantization. *Nature Biotechnology 16*, 10, 939–945.

SANKOFF, D. AND KRUSKAL, J. 1983. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparisons*. Addison-Wesley, Reading, MA.

UKKONEN, E. 1985. Algorithms for approximate string matching. *Information and Control 64*, 1–3, 100–118.

VILO, J. 2002. Pattern discovery from biosequences. Ph.D. thesis, University of Helsinki, Finland.