A New Algorithm for Fast All-Against-All Substring Matching

Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton
University of Victoria, Canada
{mgbarsky, stege, thomo}@cs.uvic.ca and cupton@uvic.ca

Abstract. We present a new and efficient algorithm to solve the 'thresh-
old all vs. all' problem, which involves searching of two strings (with
length $N$ and $M$ respectively) for finding all maximal approximate
matches of length at least $S$ and with up to $K$ differences. The algo-
rithm is based on a novel graph model, and it solves the problem in
time $O(NMK^2)$.

1 Introduction

An important problem in the field of string matching is the extraction of exact
and approximate common patterns from a set of strings. In special application
areas such as biological sequence analysis, finding exact patterns only can miss
a great deal of useful information.

The problem can be defined as “all-against-all approximate substring match-
ing” [12], and is notorious for its computational difficulty [5]. In practice, various
constraints are set for the sought solutions, such as the maximum allowed num-
ber of approximations or “errors” and the minimum length of substrings. Despite
past attempts, this problem is far from being efficiently solved. Our contribution
is a fast algorithm for solving “all-against-all approximate substring matching”
for two strings.

A naive approach to this problem is to exhaustively test each pair of substrings
from $s$ and $t$ respectively. This approach has $O(N^2M^2)$ time complexity.

The best known solution was proposed by Baeza-Yates and Gonnet in [12],
and is widely used [8]. Their solution significantly improved the average time
complexity of the naive approach, by avoiding the examination of repeating
substrings. In their method, the two input strings are organized into a suffix
tree structure, and the order of substrings comparisons is guided by a depth-first
traversal of the suffix tree nodes. The time complexity based on their practical
results lies between $NM$ (best case) and $N^2M^2$ (worst case), but closer to $N^2M^2$
[4]. Setting threshold criteria bounding the error number, i.e., allowing at most
$K$ differences in an approximate substring match, significantly improves the
performance of the Baeza-Yates and Gonnet algorithm. This is because the value
of $K$ can be directly incorporated into their algorithm to cut down the depth
of the suffix tree traversal. As we verify through experiments, for small values
of $K$, the Baeza-Yates and Gonnet algorithm performs very well. However, as
$K$ increases, the number of suffix tree nodes that are examined grows almost exponentially in $K$, which is in accordance with Ukkonen [7].

We cast the original problem into the 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 for finding “maximal paths,” which correspond to the solutions of the original string problem. Our proposed algorithm runs in $O(NMK^2)$ time, which is a significant improvement over the Baeza-Yates and Gonnet algorithm. Moreover, we experimentally show that our algorithm scales linearly (as opposed to quadratically) in $K$ and it outperforms the Baeza-Yates and Gonnet algorithm by an order of magnitude for bigger values of $K$. Finally, our algorithm has an additional nice feature: it reversely depends on the alphabet size. This is contrary to the behavior of the Baeza-Yates and Gonnet algorithm, whose running time worsens with the increase of the alphabet size.

2 A Graph Model for the All-Against-All Substring Matching

Let $\Sigma$ be a finite alphabet. A sequence of letters $a_1 a_2 \ldots a_N$, where $a_i \in \Sigma$ is called a string over $\Sigma$. We denote strings with $s$ and $t$. Given string $s$, we denote its $i$-th letter with $s[i]$, and we denote a substring of $s$ starting at position $i$ and ending at position $j$ with $s[i,j]$. Substring $s[i,j]$ has length $j-i+1$.

Let the edit distance for two strings $s$ and $t$ be the minimum number of edit operations needed to transform $s$ into $t$, as defined in [4]. We say the pair $(s,t)$ is a $K$-bounded approximate match if the edit distance between $s$ and $t$ is at most $K$.

Problem 1. All error-bounded approximate matches

**Input:** Strings $s$ and $t$ over alphabet $\Sigma$, 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 $S$.

We solve Problem 1 by casting it to an equivalent problem on graphs induced by a “matching matrix”.

The matching matrix of $s$ and $t$ ($M_{s,t}$) is defined as

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

Based on matching matrix $M$, we define a weighted directed graph $G_M$ 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 $j < l$ (cf. Fig. 1).

1 The full version of an article can be downloaded from [3].
We define the *cost* of an edge $c(v_{ij}, v_{kl})$ to be $c(v_{ij}, v_{kl}) = \max(k - i, l - j) - 1$.

A *path* in graph $G_M$ is a sequence of vertices connected by edges. For a path in $G_M$, we define two characteristic properties. The *match length* of path $\pi$ between $v_{ij}$ and $v_{kl}$ is defined as $ML(\pi) = \min(k - i + 1, l - j + 1)$. The *error number*, $EN(\pi)$, is defined as the sum of all costs of edges in $\pi$.

Note that $G_M$ is not a dynamic programming (induced) graph (*edit graph* [4]); 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 $G_M$ graph. Graph $G_M$ possesses a very desirable property which is as follows.

**Theorem 1.** 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 $G_M$.

**Problem 2.** All paths below threshold

*Input:* The graph $G_M$ for two strings $s$ and $t$, and positive integers $K$ and $S$.

*Output:* All maximal paths with $EN \leq K$ and with match length at least $S$.

Based on Theorem 1 we conclude that:

**Corollary 1.** The problem all bounded approximate matches can be reduced to the all paths below threshold problem.

We show next how to construct an instance for all paths below threshold from an instance of all bounded approximate matches.

### 3 Solving “All Paths Below the Threshold” (APBT)

We outline the logic flow of algorithm APBT, omitting all formal proofs due to space constraints. In the full version of the paper we give a simple way for building and storing the matching matrix in linear time and space. As for graph $G_M$, we never explicitly construct and store it (remaining so linear w.r.t. space). Rather, as we show, we traverse it by constructing the needed paths “on the fly.”

**Path Expansion.** We scan the matching matrix in row-major order. When a vertex of $G_M$ is encountered, we initialize a path $\pi$ with $EN(\pi) = 0$ and $ML(\pi) = 1$. The algorithm then builds all the possible expansions of this initial path in $G_M$.
path by adding one vertex at a time and by keeping track of the best paths found so far. As paths are constructed, the algorithm examines each partially completed path $\pi$; if no more vertices can be added without exceeding threshold $K$, then we stop the expansion and check whether $ML(\pi) \geq S$. If true, then we report path $\pi$ as a solution.

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 $G_M$ of cost higher than $K$ are excluded from further consideration. Consider a path $\pi$ 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 $\pi$ we need to search (in $M$) for a possible “next vertex” only inside square $ABCD$, where $A = (i + 1, j + 1)$, and $C = (i + 1 + \kappa, j + 1 + \kappa)$, for $\kappa = K - EN(\pi)$. We call square $ABCD$ the target square for path $\pi$ at vertex $v_{ij}$. The area of the target square decreases as the error number accumulated by $\pi$ increases.

On the first sight, for any vertex $v_{ij}$ in $G_M$ there are at most $(\kappa + 1) \times (\kappa + 1)$ outgoing edges to be considered. We show how to reduce the number of edges for consideration. For this, we introduce the following definitions regarding diagonals in the matching matrix $M$.

Let $(i, j)$ be an arbitrary cell in $M$. (1) The $(i, j)$-main diagonal for $M$ is the sequence of $(i + p, j + p)$-cells in $M$, where $0 \leq p \leq \min\{M - i, N - j\}$. (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.

Let $\pi$ be a path in $G_M$ ending at vertex $v_{ij}$ and with $EN(\pi) \leq K$. Now, assume that $v_{kl}$ and $v_{mn}$ are two vertices in the target square for $\pi$ at $v_{ij}$, which lie on one of the upper diagonals w.r.t. $v_{ij}$. In terms of edge cost it means, that $c(v_{ij}, v_{kl}) = l - j - 1$, and $c(v_{ij}, v_{mn}) = n - j - 1$. Now, assume that $i < k < m$ and $j < l < n$. This means 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 $\pi$ to $v_{kl}$ and later on, in the next round, continue to $v_{mn}$. Practically, this means that: if we find a vertex $v_{kl}$ on an upper diagonal of the target square, then we can exclude from the search for single-step 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.

In the full paper, we strengthen the above result by showing that we can safely exclude the row $k$ (or column $l$) from the search for path expansion.

Optimization 1. 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 diagonal and the 1-lower diagonal, possibly excluding other areas of the target square. Then, continue with the 2-upper diagonal 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.

Corollary 2. Single path extension from an arbitrary vertex \( v_{ij} \) in \( G_M \) 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. An arbitrary cell of a matrix \( M[i,j] \) is accessed at most once from each of \( 2K + 1 \) diagonals. This also implies that an arbitrary vertex \( v_{ij} \) serves as a path extension for at most \( 2K + 1 \) vertices.

Interdependence of Paths in \( G_M \). We show next how the information from previously explored paths can be reused.

Let \( \pi_1 \) be a previously explored path, which connects vertex \( v_{ij} \) with \( v_{mn} \). Let \( \pi_2 \) be another path that we are currently exploring, which originates in \( v_{kl} \), and is built up to vertex \( v_{mn} \). Clearly, if \( EN(\pi_2) \geq EN(\pi_1) \) and \( ML(\pi_2) \leq ML(\pi_1) \), we can omit the further expansion of \( \pi_2 \). An illustration is given in Fig. 1, where path \( v_{01}, v_{13}, v_{24}, v_{46} \) serves as \( \pi_1 \), which is explored earlier in a row-major order, and path \( v_{04}, v_{46} \) serves to exemplify \( \pi_2 \). Clearly, path \( \pi_2 \) will only offer a sub-solution to the solution corresponding to \( \pi_1 \), since the substring \( t[4,6] \) is a substring of \( t[1,6] \).

Thus, if we remember the smallest error number among all paths, which reached a particular vertex, then at each vertex, we will do at most \((K + 1)\) expansions. The row major processing order ensures, that if \( ML \) is defined by the length of the vertical substring, then \( ML(\pi_2) \leq ML(\pi_1) \). This is because both paths end at the same vertex, and \( \pi_2 \) starts at the same or later (greater) row than \( \pi_1 \). Notably, if we repeat the computation in a column-major order, all paths where \( ML \) was defined by the horizontal substring will now be defined by the vertical substring, thus \( ML \) of the later path will again be less than \( ML \) of the previously built path. For more detailed explanations see the full version.

The union of the solution sets of the two runs of the algorithm yields the final solution set.

From the above, we can conclude, that each vertex in \( G_M \) is expanded at most \( 2(K + 1) \) times.

Theorem 2. The All Paths Below Threshold algorithm has a time complexity of \( O(NMK^2) \).

Proof. Since during path extension, any cell is accessed only once from at most \( 2K + 1 \) vertices (cf. Corollary 3) and each of these cells, if it is a vertex, is expanded at most \( 2(K + 1) \) times (cf. discussion), the upper bound for traversing a particular cell of the matrix is at most \( 2(K + 1)(2K + 1) \). Since there are at most \( MN \) many cells in \( M \), the total time complexity is \( O(NMK^2) \).  

The pseudocode of our algorithm is given below.
4 Experimental Evaluation

We present an experimental evaluation of our All Paths Below Threshold algorithm as it compares with the algorithm of Baeza-Yates and Gonnet [2].

We implemented Gusfield’s variant of the Baeza-Yates and Gonnet algorithm [3]. We optimized it using Ukkonen’s error bounded dynamic programming method [6]. We abbreviate this optimized variant by BYG + U.

The running time was tested on the same 1.2 GHz PC with 312 MB of RAM.

Fig. 2 represents the running time of BYG + U and APBT on a pair of RNA sequences belonging to viruses from the same family, and where the minimum length of matches is set to $S = 50$. Notably the APBT algorithm outperforms the BYG + U algorithm for values of $K \geq 6$. We also show the size of the output, and this clearly shows that in order to obtain any output at all, even for similar RNA sequences, one has to set a bigger or equal to 6 value of $K$.

Interestingly, for $K \leq 5$, the BYG + U algorithm outperforms the APBT algorithm. This is because the BYG + U algorithm benefits from the early stop of deeply going in the suffix trees, when the accumulated error exceeds $K$. However,

$^2$ The original code of [2] is unfortunately not available anymore.
as $K$ grows the $BYG+U$ algorithm goes deeper in the suffix trees, and we observe an almost exponential in $K$ increase in the running time. In contrast, $APBT$ scales on average linearly with $K$.

Also, we emphasize the fact that for alphabets of bigger size the $APBT$ algorithm performs better than the $BYG+U$ algorithm. The performance of $APBT$ is orders of magnitude better than $BYG+U$ for protein sequences with alphabet size of 20. This can be explained by the greater “bushiness” of the suffix trees (used by $BYG+U$) close to the root, and by the fact that with the increase of the alphabet size, our matching matrix becomes much sparser. The $APBT$ algorithm behaves so much better than the $BYG+U$ algorithm that we had to plot their behavior in different scales (cf. Fig. 3).

![Fig. 3. Effect of alphabet size (random strings pairs of length 1000)](image)

References

1. Baeza-Yates R.A., and Gonnet G.H. All-against-all sequence matching. Rep. Dept. of CS, U. de Chile, 1990.
2. Baeza-Yates R.A., and Gonnet G.H. A fast algorithm on average for all-against-all sequence matching. Proc. SPIRE/CRIWG ’99, pp. 16–23.
3. Barsky M., Stege U., Thomo A., and Upton C.A. A New Algorithm for Fast All-Against-All Substring Matching. http://www.cs.uvic.ca/~mgbarksy/apbt.pdf, 2006.
4. Gusfield D. Algorithms on Strings, Trees and Sequences. Cambridge University Press, 1997.
5. Pevzner P., and Sze S.H. Combinatorial approaches to finding subtle signals in DNA sequences. Proc. ISMB ’00, pp. 269-278.
6. Ukkonen E. Algorithms for approximate string matching. Information and Control 64: 100–18, 1985.
7. Ukkonen E. Approximate string matching over suffix trees. CPM93, LNCS 684, 228–242, 1993.
8. Vilo J. Pattern Discovery from Biosequences. PhD Thesis, Series of Publications A, Report A-2002-3 U. of Helsinki, Finland, 2002.
9. Virus Orthologous Clusters database at http://athena.bioc.uvic.ca Viral Bioinformatics Resource Center, U. of Victoria, Canada.