Fast Approximate Pattern Matching with Few Indels via Embeddings

Mihai Bădoiu* Piotr Indyk*

1 Introduction

The problem of finding a non-exact occurence of a pattern P in a text T over some alphabet Σ is a central problem of combinatorial pattern matching and has a variety applications in many areas of computer science. The classic instance of the problem is when the distance between the pattern P and a substring S of T is measured according to the edit distance. That is, the distance between P and S is equal to the minimum number of insertions, deletions and substitutions of characters that transform P into S. Despite decades of research, the best algorithm for this problem has running time $O(nm)$ if $n = |T|, m = |P|$, even if the algorithm is allowed to report approximate answers. This contrasts with the setting when only substitutions are allowed, where near-linear time approximation algorithms are known. In particular, Karloff [Kar93] showed how to reduce the problem over any Σ to the same problem over $\{0, 1\}$, by mapping each symbol from Σ to a sequence of $O(\log |\Sigma|/\epsilon^2)$ random bits. The binary case can be solved in $O(n \log n)$ time using FFT. Several exact subquadratic-time algorithms exist for general Σ as well (references omitted due to lack of space).

In this paper we investigate searching under the edit distance in the following setting. Let S be a substring of T. We call S a (k, l)-*match* if S can be transformed into P using at most k insertions or deletions and l substitutions.

The key feature in this definition is that the number of insertions and deletions (indels) is separated from the number of substitutions. Since substitutions are "easier" to handle, this allows us to obtain algorithms with better performance than when $k + l$ arbitrary edit operations are allowed. In particular, we show

THEOREM 1.1. *There is an algorithm that, given a pattern P and a text T of length n such that there is a (k, l) match of P in T, it finds a $(k, (1+\epsilon)l)$ match of P in T. The algorithm runs in time $O(n \log^3 n \log |\Sigma|/\epsilon^4 + k^2 n \log^2 n \log l/\epsilon^3)$.*

Thus, we obtain a near-linear approximation algorithm that can handle a small number of indels but an arbitrary number of substitutions. This result can be viewed as a "noisy" version of the $O(kn)$-time exact algorithms that allow at most k indels *and* substitutions.

Our techniques. Our main tool is an approximate substring difference data structure due to [IKM00]. After initial preprocessing, the data structure reports an approximate distance between any two substrings of T in logarithmic time. That data structure uses the dimensionality reduction theorem in l_2 due to Johnson and Lindenstrauss. Combining the data structure with dynamic programming yields our result.

2 The algorithm

For this section we assume we are given an oracle $\mathrm{A}(i, j, r)$ that returns the length of the longest substring S of T, $S = T[i..i + |S| - 1]$, which can be transformed into $P[j..j + |S| - 1]$ using at most r substitutions. Note that, for the sake of simplicity, in this section we assume that the oracle gives the *exact* answer. We show later how to adopt our algorithm to deal with the approximate oracle provided in the next section.

The basic idea behind our approach is as follows. Firstly, we will assume that there exists an (k, l^*)-match of P in T, but there is no $(k, l^* - 1)$-match (we show how to remove this assumption in the full version of this paper). In this case we show that there must exist a much more restricted structure that we call *blockmatch*. Then we show how to search for a blockmatch using a dynamic program that employs a *distance oracle*, i.e., a procedure that returns the Hamming distance between any pair of substrings of P and T. Finally, we show how to implement an approximate version of the oracle and show that the approximate version suffices.

2.1 Blockmatch

The reason for adding extra characters to P and T in the following definition is technical and will become clear in the proof of the next lemma.

DEFINITION 1. *Let $\Sigma = \Sigma \bigcup \{\#, \$\}$, $P' = P \cdot \{\#\}^k$ and $T' = T \cdot \{\$\}^k$. Let S be a substring with the following properties: (1) S contains t disjoint substrings $S_1, S_2, \ldots, S_t$, which will call* blocks *and (2) S can be transformed into P' such that (a) we can perform exactly r substitutions to each block (we are not allowed to substitute a character for itself or perform 2 substitutions on the same position), (b) we delete the characters of S, which are not part of a block, (c) we insert characters between the blocks, (d) we perform at most k indels in (b) and (c) and, (e) we can delete any \$ character and insert any # characted for free. We call S a (k, r, t)-*blockmatch *of P' in T' .*

LEMMA 2.1. *Let $\epsilon > 0$. Let S be the (k, l^*)-match. Then there exists a $(k, \lceil \frac{\epsilon l^*}{k} \rceil, \lceil \frac{(1+\epsilon)k}{\epsilon} \rceil)$-blockmatch.*

Proof: Omitted. ∎

*MIT LCS, 200 Technology Square, Cambridge, MA 02139, USA, {mihai, indyk}@theory.lcs.mit.edu

As stated before, instead of searching for a (k, l^*)-match, we search for a $(k, \lceil \frac{\epsilon l^*}{k} \rceil, \lceil \frac{(1+\epsilon)k}{\epsilon} \rceil)$-blockmatch. Given the blockmatch, and setting ϵ slightly smaller than needed, such that $r = \frac{\epsilon l^*}{k}$ is an integer, we can then construct a $(k, (1+2\epsilon)l^*)$-match for the original problem. The algorithm proceeds by finding the blockmatch of P with a prefix of $T[w..n]$ for $w = 1, 2, \ldots, n$. For each w, we use dynamic programming. Specifically, use a 3-dimensional table v. The entry $v[i][j][b]$ for $i = 0 \ldots k$, $j = -k \ldots k$, $b = 0 \ldots \lceil (1+\epsilon)k/\epsilon \rceil$, denotes the size of the longest prefix P' of P, such that $T[w..w + |P'| + j]$ is a blockmatch of P' that uses i insertions or deletions and has b blocks (each block with r substitutions). We initialize $v[0][0][0]$ to 0 and ∞ to all the other entries. We fill the table using the recurrence, that states that $v[i][j][b]$ is equal to

$$\max(v[i][j][b-1] + A(w + v[i][j][b-1] + j, v[i][j][b-1], r)$$

$$v[i-1][j-1][b], v[i-1][j+1][b] + 1)$$

The table is filled in the increasing order of i and b.

LEMMA 2.2. *$v[i][j][b]$ is equal to the size of the longest prefix P' of P, such that $T[w..w + |P'| + j]$ is a blockmatch of P' that uses i insertions or deletions and has b blocks.*

Proof: We proceed by proving the statement by induction. The base case is given by the initialization of $v[0][0][0]$ to 0. For the inductive step, assume that $v[i][j][b-1], v[i-1][j-1][b]$ and $v[i-1][j+1][b]$ are filled by the definition of the claim. Suppose there is a blockmatch B that matches more characters for $v[i][j][b]$. B must end with either a deletion or an insertion or a block. Let's suppose it ends with a block. If B uses matches up to $v[i][j][b-1]$ and then tries to match a block, then our blockmatch will be just as good as B. Therefore, the B must match less than $v[i][j][b-1]$ and then end with a block. In this case, it is clear that our solution is at least as good as B. By a similar analysis of the insertion and deletion cases, we conclude that B cannot match more characters than $v[i][j][b]$. ■

By filling out this table we will compute a solution that is at least a $(k, \frac{\epsilon l^*}{k}, \frac{(1+\epsilon)k}{\epsilon})$-blockmatch. Our algorithm makes $O(\frac{k^3 n}{\epsilon})$ calls to the oracle A. The number of calls can be reduced by a factor of k; we show it in the full version of this paper.

2.2 Implementing the distance oracle

In this subsection we show how to implement an approximate version of the A(i, j, r) subroutine, which we call A'. The approximate version proceeds as follows. Let l be equal to A(i, j, r). Then A'(i, j, r) returns $l' \geq l$ such that $D(T[i \ldots i + l' - 1], P[j \ldots j + l' - 1]) \leq (1+\epsilon) D(T[i \ldots i + l' - 1], P[j \ldots j + l' - 1])$, where $D(s, s')$ is the Hamming distance between s and s'.

In order to implement A', we show how to implement a subroutine $\text{DIST}(i, j, t)$, that returns a value between $D(T[i \ldots i+t], P[j \ldots j+t])$ and $(1+\epsilon) D(T[i \ldots i + t], P[j \ldots j + t])$. The A' then can be implemented using $\log n$ calls to $\text{DIST}(i, j, t)$ via binary search. It is not difficult to observe that the calls can be arranged in such a way that k is always a power of 2.

An implementation of DIST, for the case when Σ is a set of *real* numbers, $D(\cdot, \cdot)$ is the (square of the) Euclidean distance and k is *fixed* has been given in [IKM00]. It achieves the following parameters: preprocessing time $O(n \log^2 n / \epsilon^2)$ and query time $O(\log n / \epsilon^2)$. For our purpose (i.e., with $D(\cdot, \cdot)$ being the Hamming metric), we first map Σ to $\{0, 1\}^{O(\log |\Sigma| / \epsilon^2)}$ as in [Kar93], obtaining T' and P'. Then we use the fact that, for binary strings, Hamming distance and the square of the Euclidean distance are the same. We build $\log n$ [IKM00] data structures that support operations $\text{DIST}(i, j, k)$ in T' and P' for $k = 2^0, 2^1, \ldots$. The final data structure for A' has the following parameters: preprocessing time $O(n \log^3 n \log |\Sigma| / \epsilon^4)$ and query time $O(\log^2 n / \epsilon^2)$.

2.3 Approximate oracle suffices

It remains to show that if we use the approximate oracle A' instead of the exact one A, then we still solve the $(k, l(1 + O(\epsilon)))$-match problem. For this purpose, we show the following. Let v' be the dynamic programming table filled using A' instead of A.

LEMMA 2.3. *For any i, j, b, we have $v'[i][j][j] \geq v[i][j][b]$.*

The lemma implies that our algorithm will find a (k, r, t)-blockmatch, for $r = \frac{\epsilon l^*}{k}$, $t = \frac{(1+\epsilon)k}{\epsilon}$, such that each block uses at most $r(1+\epsilon)$ substitutions. This will show the main theorem of this paper. *Proof:* The proof proceeds by induction. Assume that the statement is true for all triples $(i-1, j', b)$ and the triple $(i, j, b-1)$; we will show it holds for (i, j, b). Consider the inductive formula for $v'[i][j][b]$. By the inductive assumption we have

$$\begin{aligned} v'[i][j][b] &\geq \max(v'[i-1][j-1][b], v'[i-1][j+1][b] + 1) \\ &\geq \max(v[i-1][j-1][b], v[i-1][j+1][b] + 1) \end{aligned}$$

Thus, it suffices to consider the case $v[i][j][b] = v[i, j, b-1] + A(w + v[i][j][b-1] + j, v[i][j][b-1], r)$. Denote $v = v[i][j][b], v' = v'[i][j][b], v_- = v[i][j][b-1], v'_- = v'[i][j][b-1]$. By definition, $T[w + v_- + j..w + v + j - 1]$ differs in at most r positions with the corresponding prefix of $P[v_- + 1..m]$ of the same length. Therefore, $T[w + v'_- + j..w + v + j - 1]$ also differs in at most r positions with the corresponding prefix of $P[v'_- + 1..m]$. Since $A'(\cdot, \cdot, \cdot) \geq A(\cdot, \cdot, \cdot)$ for all arguments, it follows that $v' \geq v$. ■

References

[IKM00] P. Indyk, N. Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series datasets using sketches. *Proceedings of the 26th International Conference on Very Large Databases (VLDB)*, 2000.

[Kar93] H. J. Karloff. Fast algorithms for approximately counting mismatches. *Information Processing Letters*, 48(2):53–60, 1993.