Università di Pisa

Dipartimento di Informatica

Technical Report: TR-00-13

On 2-Tuple String Matching

Paolo Ferragina S. Muthukrishnan

ADDRESS: Corso Italia 40, 56125 Pisa, Italy. TEL: +39 050 887111 FAX: +39 050 887226

On 2-Tuple String Matching

Paolo Ferragina* S. Muthukrishnan†

1 Introduction

A charm of the combinatorial pattern matching area is that some fairly easy-to-state problems prove to be challenging. Consider the two problems below.

Pair Matching Problem. We are given a collection of documents $\mathcal{D} = \{D_1, \ldots, D_k\}$ that may be preprocessed. Each query is a *pair* of pattern strings p_1 and p_2, and the goal is to determine the set of all documents that contain *both* patterns as substrings. ■

For example, one may seek all documents that contain both strings `ape` and `pea`: this may be documents that contain the two strings separately, overlapping as in `apea` and `peape`), or adjacent as in `apepea` or `peaape`. Although this example is contrived, the problem arises very naturally in many information retrieval scenarios.

A special case of the pair matching problem is one in which there is a single document (i.e., $k = 1$) and one pattern (i.e., p_2 is empty string). Then, the problem can be solved with $O(|\mathcal{D}|)$ time preprocessing and $O(|p_1|)$ time query processing, using suffix trees [W73]. We refer to this as Problem I. A more general case of the pair matching problem is one in which $\mathcal{D}$ is arbitrary, but query contains only a single pattern (i.e., p_2 is empty). Here, there are two variants: find all occurrences of p_1 in $\mathcal{D}$, or find the distinct documents that contain p_1 . We refer to these as Problems II and III respectively. Problem II can be solved in $O(\sum_i |D_i|)$ time preprocessing and $O(|p_1|)$ time query processing, again using [W73]; therefore, it is fully solved. Problem III can be solved in $O(\sum_i |D_i|)$ time preprocessing and $O(|p_1| \log k)$ time query processing [MM+98].

*Univ. of Pisa, Italy `ferragin@di.unipi.it`

†AT&T Labs – Research, Florham Park, NJ. `muthu@research.att.com`.

Note that in Problems I-III, query processing is (essentially) independent of k, $|\mathcal{D}|$. Our goal for solving the pair matching problem would be to decrease the dependence on $|\mathcal{D}|$ and k for query processing since (a) k, $\sum_i |D_i|$ are likely to be large in applications, and (b) our intention is to address the main theoretical challenge, which is the goal stated.

The pair matching problem generalizes Problems I-III and can be solved trivially in time $O(k(|p_1|+|p_2|))$ by checking p_1 and p_2 independently against each D_i as in Problem I, or in $O(k + |p_1| \log k)$ using Problem III. In either case, query processing depends linearly on k, the number of documents, which is undesirable.

open.

2-tuple Matching Problem. We are given a collection of 2-tuples $\mathcal{T} = \{T_1, \ldots, T_k\}$ where each $T_i = (T_i^1, T_i^2)$, for two strings T_i^1 and T_i^2. This collection may be preprocessed. A query comprises a pattern tuple $P = (P^1, P^2)$ and the problem is to determine all tuples T_i in $\mathcal{T}$ for which P^1 is a substring of T_i^1 and P^2 is a substring of T_i^2. We let $M = \sum_i(|T_i^1| + |T_i^2|)$. ∎

For example, we may have a database of first names and last names of people and a query may be to retrieve all names of people containing `ichael` in the first name and `ord` in the second name, `(Michael, Jordan)` being an example.

The special case of the 2-tuple matching problem is one in which all tuples have only one component, that is, T_i^2's and P^2's are null. Then we have Problem III. One way to solve the 2-tuple matching problem is to solve Problem III for each component independently and to combine the results. This gives an $\Omega(M)$ time query processing algorithm (in the worst case) which is undesirable since it depends linearly on M.

To sum, both pair matching and 2-tuple matching problems have the charm of natural, easy-to-state problems that generalize other well-explored string matching problems but have no known nontrivial results.

Our Results. In this paper, we make progress on solving the two problems above and present first nontrivial results.

We first make an observation. The two problems are equivalent. We can reduce pair matching problem to 2-tuple matching problem by setting $T_i^1 = T_i^2 = D_i$ for all i, $P^1 = p_1$, and $P^2 = p_2$. On the other hand, we can reduce the 2-tuple matching problem to pair matching problem by setting $D_i = T_i^1 || \sigma(T_i^2)$, $p_1 = P^1$ and $p_2 = \sigma(P^2)$; here, $||$ denotes string concatenation, and $\sigma(s)$ is a string obtained from s by transforming each symbol in s by σ, a one-to-one mapping of the alphabet set to a new alphabet set (say, by adding $|\Sigma|$ to the original alphabet set $\{1, \cdots, |\Sigma|\}$). Henceforth we will only focus on the 2-tuple matching problem (hence the title of this paper).

Our main result[1] is an algorithm for solving the 2-tuple matching problem that with $O(M^{1.5} \log^2 M)$ preprocessing time and space, takes $O(M^{0.5} + |p_1| + |p_2|)$ time for query processing; Recall that our intention was to address the main theoretical challenge in the area. Our result is significant since it breaks the quadratic time preprocessing bound for the first time and yet achieves sublinear query time. This already represents considerable progress. (A desirable result would be to have query processing near-linear in M, a goal we leave open.)

2 Our Solution

Our solution relies on solving the *colored range query* (CRQ) problem where we are given the set of integers $1, 2, \ldots, N$ each with a color from $1, \ldots, C$. We want to preprocess this set so that the following query can be answered efficiently: Given two non-overlapping intervals I_1 and I_2 in $[1, N]$, list the *distinct* colors which occur in *both* intervals I_1 and I_2.

In what follows, we will present our solution to two TASKS: TASK 1 is to reduce the 2-tuple matching problem to CRQ problem, and TASK 2 is to solve the CRQ problem. All descriptions will be very brief.

TASK 1. Let us assign a distinct color to each tuple. We build one suffix tree ST_1 on the strings T_1^i, and one suffix tree ST_2 on the strings T_2^i. We color each leaf of the two suffix trees according to the color of the tuple its suffix refers to. We number the leaves from left to right. This gives the input to the CRQ problem for preprocessing by appending the two lists of colored leaves in the numbered order.

Query processing works as follows. Given a query tuple $P = (P^1, P^2)$, we search for P^1 in ST_1 and for P^2 in ST_2. The nodes we reach in the two trees (loci, as they are called) identify two intervals I_1 and I_2 of consecutive leaves denoting all the occurrences of P^1 and P^2, respectively. A given color may occur many times in I_1 if P^1 occurs many times in T_1^i for some i (likewise for P^2 and T_2^i's). Finding the distinct colors that occur in *both* I_1 and I_2 is

[1] See the full tradeoff in Theorem 1.

the same as finding the distinct tuples which contain P^1 and P^2 in the first and second component, respectively.

TASK 2. Solving the CRQ problem is bulk of the work. It relies on space partitioning, and a subtle reduction to certain generalized range searching problems in two dimensions. The main improvement is achieved by devising a new solution when the intervals I_1 and I_2 are "small".

Data Structure. We partition the segment $[1, N]$ into N^{1-x} intervals of size N^x each ($x < 1$ is a positive constant). Let us the set of these intervals using $\mathcal{I}$. For each interval I in $\mathcal{I}$ and each color c, we (pre)compute the values $\max\{c, I\}$ and $\min\{c, I\}$ corresponding to the maximum and minimum integer in I colored c (if any), respectively. Then we construct the set P of two-dimensional points (u, v) such that u is an integer in $[1, N]$ and v is equal to either $\min\{I, c\}$ or $\max\{I, c\}$, for any interval I in $\mathcal{I}$ and for color c of u. Altogether, P has size N^{2-x} since u can be chosen in N ways and v can be chosen in N^{1-x} ways (note that the color c of u is fixed, once u is given). The set P is stored in a data structure, let us call it $\mathcal{D}_1$, that answers generalized range-queries in two dimensions. This data structure allows one to report the distinct colors occurring in the rectangle specified by the query. The known algorithmic solution takes $O(Y \log^2 Y)$ space and $O(log^2 Y + \#c)$ time where Y is the total number of colors and $\#c$ is the number of colors in the output to the query [GJS95]. In our case the cost is therefore $O(N^{2-x} \log^2 N)$ space and $O(\log^2 N + \#c)$ time, where $\#c$ is the output size.

We also consider another data structure, let us call it $\mathcal{D}_2$, which allows us to answer the following query in logarithmic time: Does color c occur in an interval I? This can be done in $O(N \log N)$ space and $O(\log^2 N)$ query time, using just binary search trees. (This can be improved, but it is not the bottleneck.)

Query Processing. Consider answering the original query on intervals I_1 and I_2. We distinguish two cases: (1) One of I_1 or I_2 overlaps the boundary of some interval in $\mathcal{I}$; (2) both intervals I_1 and I_2 are entirely contained in an interval of $\mathcal{I}$ each.

To answer the query in case (1) we use $\mathcal{D}_1$ as follows. Assume that I_1 overlaps the right boundary of the interval I in $\mathcal{I}$ and the left boundary of the interval I' in $\mathcal{I}$ adjacent to I. The crucial observation is that *if a color c occurs in I_1, then either* $\max\{I, c\}$ *occurs inside* I_1 *or* $\min\{I', c\}$ *occurs inside* I_1. Therefore, if the color c occurs also in I_2 at position v, then the set P contains either the point $(v, \max\{I, c\})$ or the point $(v, \min\{I', c\})$, or both (if c occurs into both I and I'). In any case, one point colored c among those two points surely lies inside the rectangle $I_2 \times I_1$. A generalized range query $I_2 \times I_1$ on set P will retrieve this color. Clearly, only the colors occurring in both I_1 and I_2 will be retrieved. We answer the query in case (2) by scanning I_1 and checking for every color there if it occurs in I_2 using $\mathcal{D}_2$.

Without considering the additive term linear in the output size, the overall time taken is $O(\log^2 N)$ for case 1 and $O(N^x \log^2 N)$ for case 2, since $|I_1| = O(N^x)$. The overall space required is $O(N^{2-x} \log^2 N)$. This gives,

Theorem 1 *The colored range query problem (hence 2-tuple matching and pair matching problems with suitable N's) can be solved in $O(N^{2-x} \log^2 N)$ preprocessing time and space and $O(N^x \log^2 N)$ query time, for $x < 1$.*

References

[MM+98] Y. Matias, S. Muthukrishnan, S. Sahinalp and J. Ziv. Augmenting suffix trees with applications. *Proc. European Symp. on Algorithms*, 1998.

[W73] P. Weiner. Liner pattern matching algorithms. *IEEE Symp. Switching and Automata Theory*, 1–11, 1973.

[GJS95] P. Gupta, R. Janardhan and M. Smid. Further results on generalized intersection search problems: Counting, reporting and dynamization. *J. Algorithms*, 282-317, 1995.