Succinct 2D Dictionary Matching With No Slowdown

Shoshana Neuburger[1] * and Dina Sokol[2] **

[1] Department of Computer Science, The Graduate Center of the City University of New York, New York, NY, 10016
shoshana@sci.brooklyn.cuny.edu
[2] Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn, NY, 11210
sokol@sci.brooklyn.cuny.edu

Abstract. The dictionary matching problem seeks all locations in a given text that match any of the patterns in a given dictionary. Efficient algorithms for dictionary matching scan the text once, searching for all patterns simultaneously. This paper presents the first 2-dimensional dictionary matching algorithm that operates in small space and linear time. Given d patterns, $D = \{P_1, \ldots, P_d\}$, each of size $m \times m$, and a text T of size $n \times n$, our algorithm finds all occurrences of P_i, $1 \leq i \leq d$, in T. The preprocessing stores the dictionary in entropy compressed form, in $|D|H_k(D) + O(|D|)$ bits. Our algorithm uses $O(dm \log dm)$ bits of extra space. The time complexity of our algorithm is linear $O(|D| + |T|)$.

1 Introduction

Dictionary matching is the problem of searching a given text for all occurrences of any pattern from a given set of patterns. A search for specific phrases in a book, virus detection software, network intrusion detection, searching a DNA sequence for a set of motifs, and image identification, are all applications of dictionary matching. In this work we are concerned with efficiently solving the 2-dimensional dictionary matching problem in *small space.* The motivation for this is that there are many scenarios, such as on mobile and satellite devices, where storage capacity is limited. The added constraint of performing efficient dictionary matching using little extra space is a challenging and practical problem.

Linear-time *single* pattern matching algorithms in both one and two dimensions have achieved impressively small space complexities. For 1D data, we have pattern matching algorithms that require only constant extra space, [10, 8, 18, 11]. For 2D pattern matching, Crochemore et al. present a linear time

* This work has been supported in part by the National Science Foundation Grant DB&I 0542751.
** This work has been supported in part by the National Science Foundation Grant DB&I 0542751 and the PSC-CUNY Research Award 63343-0041.

algorithm that works with *log* extra space for pattern preprocessing and $O(1)$ extra space to scan the text [7]. Such an algorithm can be trivially extended to perform dictionary matching but its runtime would depend on the number of patterns in the dictionary. The goal of an efficient dictionary matching algorithm is to scan the text once so that its running time depends only on the size of the text and not on the size of the patterns sought.

Concurrently searching for a set of patterns within limited working space presents a greater challenge than searching for a single pattern in small space. Much effort has recently been devoted to solving 1-dimensional dictionary matching in small space [6, 14, 15, 3, 13]. The most recent result of Hon et al. [13] has essentially closed this problem, with a linear-time algorithm that uses $|D'|H_k(D')+O(|D'|)$ bits of space, given a dictionary D' of 1D patterns. H_k, i.e. the kth order empirical entropy of a string, describes the minimum number of bits that are needed to encode each symbol of the string within context, and it is often used to demonstrate that storage space meets the information-theoretic lower bounds of data.

In this paper, our objective is to extend succinct 1D dictionary matching to the two-dimensional setting, in a way similar to the Bird and Baker (BB) extension of the Aho-Corasick 1D dictionary matching algorithm (AC). It turns out that this problem is not trivial, due to the necessity to label each position of the text. However, using new techniques of dynamic dueling, we indeed achieve a linear time algorithm that solves the *Small-Space 2D Dictionary Matching Problem*.

Specifically, given a dictionary D of d patterns, $D = \{P_1, \ldots, P_d\}$, each of size $m \times m$, and a text T of size $n \times n$, our algorithm finds all occurrences of P_i, $1 \leq i \leq d$, in T. During the preprocessing stage, the patterns are stored in entropy compressed form, in $|D|H_k(D)+O(|D|)$ bits. $H_k(D)$ denotes the kth order empirical entropy of the string formed by concatenating all the patterns in D, row by row. The preprocessing is completed in $O(|D|)$ time using $O(dm \log dm)$ bits of extra space. Then, the text is searched in $O(|T|)$ time using $O(dm \log dm)$ bits of extra space. For ease of exposition, we discuss patterns that are all of size $m \times m$, however, our algorithm generalizes to patterns that are the same size in only one dimension, and the complexity would depend on the size of the largest dimension. As in [3] and [13], the alphabet can be non-constant in size.

In the next section we give an overview of Bird/Baker [5, 2] and Hon et al. [13] and outline how it is possible to combine these algorithms to yield a small space 2-dimensional dictionary matching algorithm for certain types of patterns. In Section 3 we introduce a distinction between highly periodic patterns and non-periodic patterns, and summarize the algorithm for the periodic case. In section 4 we deal with the non-periodic case, introducing new space-saving techniques including dynamic dueling. We conclude with open problems in Section 5.

2 Overview

The first linear-time 2D pattern matching algorithm was developed independently by Bird [5] and Baker [2]. Although the BB algorithm was initially presented for a single pattern, it is easily extended to perform dictionary matching by replacing the KMP automaton with an AC automaton. In Algorithm 1 we outline the dictionary matching version of BB.

Algorithm 1: Bird/Baker algorithm for 2D Dictionary Matching

1. Preprocess Pattern:
   a) Form Aho-Corasick automaton of pattern rows, called AC1.
   Let ℓ denote the number of states in AC1.
   b) Name pattern rows using AC1, and store a 1D pattern of names for each pattern in D, called D'.
   c) Construct AC automaton of D', called AC2.
   Let ℓ' denote the number of states in AC2.
2. Row Matching:
   Run Aho-Corasick on each text row using AC1.
   This labels positions at which a pattern row ends.
3. Column Matching:
   Run Aho-Corasick on named text columns using AC2.
   Output pattern occurrences.

The basic concept used here is the translation of the 2D patterns into 1D patterns using naming. Rows of each pattern are perceived as metacharacters and named so that distinct rows receive different names. The text is named in a similar fashion and 1D dictionary matching is performed over the text columns and the patterns of names. The linear time complexity of the algorithm depends on the assumption that the label of each state fits into a single word of RAM.

Space Complexity of BB: We use ℓ to denote the number of states in the Aho-Corasick automaton of D, AC1. Note that $\ell \leq |D| = dm^2$. $O(\ell \log \ell)$ bits of space are needed to store AC1, and labeling all text locations uses $O(n^2 \log dm)$ bits of space. Overall, the BB algorithm uses $O(\ell \log \ell + n^2 \log dm)$ bits of space.

Our objective is to improve upon this space requirement. As a first attempt to conserve space, we replace the traditional AC algorithm in Step 1 of BB with the compressed AC automaton of Hon et al. [13]. The algorithm of [13] builds upon the work of Belazzougui [3] which encodes the three functions of the AC automaton (*goto, report, failure*) separately and in different ways. The space complexity is reduced from 0th order empirical entropy to kth order empirical entropy by employing the XBW transform [9] to store the *goto* function. The details of these papers are very interesting, but for our purposes, we can actually use their algorithm as a black box replacement for the AC automata in both Steps 1a and 1c of the BB algorithm.

To reduce the algorithm's working space, we work with small overlapping text blocks of size $3m/2 \times 3m/2$. This way, we can replace the $O(n^2 \log dm)$ bits

of space used to label the text in Step 3 with $O(m^2 \log dm)$, relating the working space to the size of the dictionary, rather than the size of the entire text.

Theorem 1. *We can solve the 2D dictionary matching problem in linear* $O(dm^2 + n^2)$ *time and* $\ell H_k(D) + \ell' H_k(D') + O(\ell + m^2 \log\, dm)$ *bits of space.*

Proof. Since the algorithm of Hon et al. [13] has no slowdown, replacing the AC automata in BB with compressed AC automata preserves the linear time complexity. The space used by the preprocessing is: $\ell H_k(D) + O(\ell) + \ell' H_k(D') + O(\ell')$ bits. The compressed AC1 automaton uses $\ell H_k(D) + O(\ell)$ bits of space and it replaces the original dictionary, while the compressed AC2 automaton uses $\ell' H_k(D') + O(\ell')$ *extra* bits of space. Text scanning uses $O(m^2 \log\, dm)$ extra bits of space to label each location of a text block. □

Although this is an improvement over the space required by the uncompressed version of BB, we would like to improve on this further. Our aim is to reduce the working space to $O(dm \log\, dm)$ bits, thus completely eliminating the dependence of the working space on the size of the given text. Yet, note that this constraint still allows us to store $O(1)$ information per pattern row to linearize the dictionary in the preprocessing. In addition, we will have the ability to store $O(1)$ information about each pattern per text row to allow linearity in text scanning.

The following corollary restates Theorem 1 in terms of $O(dm \log\, dm)$ for the case of a dictionary with many patterns. It also omits the term $\ell' H_k(D') + O(\ell')$, since $\ell' H_k(D') + O(\ell') = O(dm \log\, dm)$.

Corollary 1. *If* $d > m$*, we can solve the 2D dictionary matching problem in linear* $O(dm^2 + n^2)$ *time and* $\ell H_k(D) + O(\ell) + O(dm \log dm)$ *bits of space.*

The rest of this paper deals with the case in which the number of patterns is smaller than the dimension of the patterns, i.e., $d = o(m)$. For this case, we cannot label each text location and therefore the Bird and Baker algorithm cannot be applied trivially. We present several clever space-saving tricks to preserve the spirit of Bird and Baker's algorithm without incurring the necessary storage overhead.

3 Preliminaries

A string S is *primitive* if it cannot be expressed in the form $S = u^j$, for $j > 1$ and a prefix u of S. String S is *periodic* in u if $S = u'u^j$ where u' is a suffix of u, u is primitive, and $j \geq 2$. A periodic string p can be expressed as $u'u^j$ for one unique primitive u. We refer to u as "the period" of p. Depending on the context, u can refer to either the string u or the period size $|u|$.

There are two types of patterns, and each one presents its own difficulty. In the first type, which we call Case 1 patterns, all rows are periodic, with periods $\leq m/4$. The difficulty in this case is that many overlapping occurrences can appear in the text in close proximity to each other, and we can easily have more

candidates than the working space we allow. The second type, Case 2 patterns, have at least one aperiodic row or one row whose period is larger than $m/4$. Here, each pattern can occur only $O(1)$ times in a text block. Since several patterns can overlap each other in both directions, a difficulty arises in the text scanning stage. We do not allow the time to verify different candidates separately, nor do we allow space to keep track of the possible overlaps for different patterns.

In the initial preprocessing step, we divide the patterns into two groups based on 1D periodicity. For Case 1 patterns, the algorithm presented by the authors [17] for LZ-compressed texts can be adapted here to solve the general 2D dictionary matching problem. Since every row of every pattern is periodic with period $\leq m/4$, we have the following.

Observation 1 *Any text row that is included in a pattern occurrence must have exactly one maximal periodic substring of length* $\geq m$.

Hence, we can run the Main and Lorentz algorithm [16] on the text rows, using location m as the 'center.' Once text rows are labeled, we use our innovative naming technique [17] based on Lyndon words to name both pattern rows and text rows. We then run the traditional AC automaton on the 1D patterns of names and 1D text of names to identify candidates for pattern occurrences. Verification of actual occurrences proceeds using similar data structures.

Lemma 1. *[17] 2D dictionary matching for Case 1 patterns can be done in* $O(dm^2 + n^2)$ *time and* $\ell H_k(D) + O(\ell) + O(dm \log m)$ *bits of space.*

For Case 2 patterns, we can use the aperiodic row to filter the text. This simplifies the identification of an initial set of candidates. Yet, verification in one pass over the text presents a difficulty. In dictionary matching, different candidates represent different patterns, and it is infeasible to compute and store information about the relationship between all patterns. In the next section we present an approach to verify a set of candidate positions for a *set of patterns* in linear time using a new technique called dynamic dueling.

4 The Algorithm

Recall that we focus on the case in which the number of patterns in the dictionary is smaller than the dimension of a pattern, i.e. $d < m$. We further assume that each pattern has at least one aperiodic row. The case of a pattern having a row that is periodic with period size between $m/4$ and $m/2$ will add only a small constant to the stated complexities.

The difficulty in applying BB is that we do not have sufficient space to label all text positions. However, we can initially filter the text block using the aperiodic row. Thus, the text scanning stage first identifies a small set of positions that are candidates for pattern occurrences. Then, in a second pass over the text block, we verify which candidates are actually pattern occurrences.

4.1 Pattern Preprocessing

1. Construct (compressed) AC automaton of first aperiodic row of each pattern. Store row number of each of these rows within the patterns.
2. Form a compressed AC automaton of the pattern rows.
3. Construct witness tree of pattern rows and preprocess for LCA.
4. Name pattern rows. Index the 1D patterns of names in a suffix tree.

In the first step, we form an AC automaton of one aperiodic row of each pattern, say, the first aperiodic row of each pattern. This will allow us to filter the text and limit the number of potential pattern occurrences to consider. Since we use only one row from each pattern, using a compressed version of the AC automaton is optional.

In the second step, the pattern rows are named as in BB to form a 1D dictionary of patterns. Here we use a compressed AC automaton of the pattern rows. An example of two patterns and their 1D representation is shown in Fig. 1.

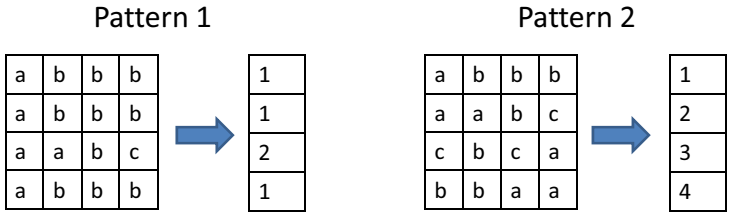


Fig. 1. Two linearized 2D patterns with their 1D names.

Another necessary data structure is the witness tree introduced in [17] and summarized in Appendix A. A witness tree is used to store pairwise distinctions between different patterns, or pattern rows, of the same length. A witness tree provides a witness between pattern rows in constant time if it is preprocessed for Lowest Common Ancestor (LCA).

Preprocessing proceeds by indexing the 1D patterns of names. We form a suffix tree of the 1D patterns to allow efficient computation of longest common prefix (LCP) queries between substrings of the 1D patterns.

Lemma 2. *The pattern preprocessing stage for Case 2 patterns completes in* $O(dm^2)$ *time and* $O(dm \log m)$ *extra bits of space.*

Proof. The AC automaton of the first non-periodic row of each pattern is constructed in $O(dm)$ time and is stored in $O(dm \log m)$ bits, in its uncompressed form. A compressed AC automaton of all pattern rows occupies $\ell H_k(D) + O(\ell)$ bits of space and can then become the sole representation of the dictionary [13].

The witness tree occupies $O(dm \log\ m)$ bits of space [17]. A rooted tree can be preprocessed in linear time and space to answer LCA queries in $O(1)$ time [12, 4]. The patterns are converted to a 1D representation in $O(dm^2)$ time. A suffix tree of the 1D dictionary of names can be constructed and stored in linear time and space, e.g. [19]. □

4.2 Text Scanning

The text scanning stage has three steps.

1. Identify candidates in text block with 1D dictionary matching of a non-periodic row of each pattern.
2. Duel to eliminate inconsistent candidates within each column.
3. Verify pattern occurrences at surviving candidate positions.

Step 1. Identify Candidates We identify a limited set of candidates in the text block using 1D dictionary matching on the first aperiodic row of each pattern. There can be only one occurrence of any non-periodic pattern row in a text block row. Each occurrence of an aperiodic pattern row demarcates a candidate, at most d per row. In total, there can be up to dm candidates in a text block, with candidates for several distinct 1D patterns on a single row of text. If the same aperiodic row occurs in several patterns, several candidates can occur at the same text position, but candidates are still limited to d per row.

We run the Aho-Corasick algorithm over the text block, row by row, to find up to dm candidates. Then we update each candidate to reflect the position at which we expect a pattern to begin. This is done by subtracting the row number of the selected aperiodic row from the row number of its found location in the text block.

Complexity of Step 1: 1D dictionary matching on a text block takes $O(m^2)$ time with the AC method. [3] Marking the positions at which patterns can begin is done in constant time per candidate found; overall, this requires $O(dm) = o(m^2)$ time. The AC algorithm uses extra space proportional to the dictionary, which is $O(dm \log m)$ bits of space for this limited set of pattern rows. The dm candidates can also be stored in $O(dm \log\ m)$ bits of space.

Step 2. Eliminate Vertically Inconsistent Candidates We call a pair of candidate patterns *consistent* if all positions of overlap match. Vertically consistent candidates are two candidates that appear in the same column, and have a suffix/prefix match in their 1D representations. In order to verify candidates in a single pass over the text, we take advantage of the fact that overlapping segments of consistent candidates can be verified simultaneously.

We eliminate inconsistent candidates with a dueling technique inspired by the 2D *single* pattern matching algorithm of Amir et al. [1]. In the single pattern

[3] Hashing techniques achieve linear time complexity in the AC algorithm.

matching algorithm, duels are performed between candidates for the same pattern. In dictionary matching, we perform duels between candidates for *different* patterns.

In general, the dueling paradigm requires that a witness, i.e. position of mismatch in the overlap, be precomputed and stored for all possible overlaps. If no witness exists for a given distance, then such candidates are consistent. During a duel, the text location is compared to the witness, killing one or more of the candidates involved in the duel. To date, the dueling paradigm has not been applied to dictionary matching since it is prohibitive to precompute and store witnesses for all possible overlaps of all candidate patterns in a set of patterns. However, here we show an innovative way of performing 2D duels for a set of patterns.

For our purposes, we need only eliminate vertically inconsistent candidates. Thus, we introduce *dynamic dueling* between two candidates in a given column. In dynamic dueling, no witness locations are computed in advance. We are given two candidate patterns and their locations, candidate A at location (i, j) in the text and candidate B at location (k, j) in the text, $i < k$. Since all of our candidates are in an $m/2 \times m/2$ square, we know that there is overlap between the two candidates.

A duel consists of two steps. In the first step, the 1D representation of names is used for A and B, denoted by A' and B'. An LCP query between the suffix $k - i + 1$ of A' against B' returns the number of overlapping rows that match. If this number is $\geq i + m - k$ then the two candidates are consistent. Otherwise, we are given a "row-witness," i.e. the LCP points to the first row at which the patterns differ. In the second step of the duel, we use the witness tree to locate the position of mismatch between the two different pattern rows, and we use that position to eliminate one or both candidates.

To demonstrate how a witness is found and the duel is performed, we return to the patterns in Fig. 1. Assume two candidates exist; directly below a candidate for Pattern1, we have a candidate for Pattern2. The LCP of *121*, (second suffix of linearized Pattern1) and *1234* (linearized Pattern2) is 2. Since $2 < 3$, the LCP query reveals that the patterns are inconsistent, and that a witness exists between the fourth row of Pattern1 (name 1) and the third row of Pattern2 (name 3). We then procure a witness from the witness tree shown in Fig. 3 by taking the LCA of the leaves that represent names 1 and 3. The result of this query shows that the first position is a point of distinction between names 1 and 3. If the text has an 'a' at that position, Pattern1 survives the duel. Otherwise, if the character is a 'c', Pattern2 survives the duel. If neither 'a' nor 'c' occur at the text location, both candidates are eliminated.

Lemma 3. *A duel between two candidate patterns A and B in a given column j of the text can be performed in constant time.*

Proof. The suffix tree constructed in Step 4 of the pattern preprocessing answers LCP queries in the 1D patterns of names in $O(1)$ time. The witness tree gives a position of a row-witness in $O(1)$ time, and retrieving the text character to perform the actual duel takes constant time. □

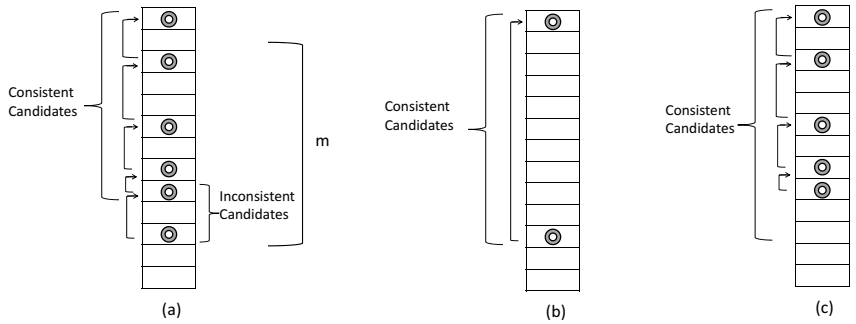


Fig. 2. (a) Duel between vertically inconsistent candidates in a column. (b) Surviving candidates if the lower candidate wins the duel. (c) Surviving candidates if the upper candidate wins the duel.

Verification begins by dueling top-down between candidates within each column. Since consistency is a transitive relation, if the lower candidate is killed, this does not affect the consistent candidates above it in the same column. However, if the lower candidate survives, it triggers the elimination of all candidates within m rows above the witness row. Pointers link consecutive candidates in each column. This way, a duel eliminates the set of consistent candidates that are within range of the mismatch. This is shown in Fig. 2. Note that the same method can be used when two (or more) candidates occur at a single text position.

Complexity of Step 2: Step 2 begins with at most dm candidate positions. Each candidate is involved in exactly one duel, and is either killed or survives. If a candidate survives, it may be visited exactly one more time to be eliminated by a duel beneath it. Since a duel is done in constant time, by Lemma 3, this step completes in $O(dm)$ time. Recall that $d < m$. Hence, the time for Step 2 is $O(m^2)$.

Step 3. Verify Surviving Candidates After eliminating vertically inconsistent candidates, we verify pattern occurrences in a single scan of the text block. We process one text block row at a time to conserve space. Before scanning the current text block row, we label the positions at which we expect to find a pattern row. This is done by merging the labels from the previous row with the list of candidates that begin on the new row. If a new candidate is introduced in a column that already has a label, we keep only the label of the lower candidate. This is permissible since the label must be from a consistent candidate in the same column. Thus, each position in the text has at most one label.

The text block row is then scanned sequentially, to mark actual occurrences of pattern rows. This is done by running AC on the text row with the compressed AC automaton of all pattern rows. The lists of expected row names and actual row names are then compared sequentially. If every expected row name appears

in the text block row, the candidate list remains unchanged. If an expected row name does not appear, a candidate is eliminated. The pointers that connect candidates are used to eliminate candidates in the same column that also include the label that was not found.

After all rows are verified in this manner, all surviving candidates in the text are pattern occurrences of their respective patterns.

Complexity of Step 3: When a text block row is verified, we mark each position at which a pattern row (1D name) is expected to begin. This list is limited by $m/2$ due to the vertical consistency of the candidates. We also mark actual pattern row occurrences in the text row which are again no more than $m/2$ due to distinctness of the row names. Thus, the space complexity for Step 3 is $O(m)$. The time complexity is also linear, since AC is run on the row in linear time, and then two sorted lists of pattern row names are merged. Over all $3m/2$ rows in the text block, the complexity of Step 3 is $O(m^2)$.

Lemma 4. *The algorithm for 2D dictionary matching, when pattern rows are not highly periodic and $d < m$, completes in $O(n^2)$ time and $O(dm \log m)$ bits of space, in addition to $\ell H_k(D) + O(\ell)$ bits of space to store the compressed AC automaton of the dictionary.*

Proof. This follows from Lemma 2 and the complexities of Steps 1, 2, and 3. □

Theorem 2. *Our algorithm for 2D dictionary matching completes in $O(dm^2 + n^2)$ time and $O(dm \log dm)$ bits of extra space.*

Proof. For $d > m$, this is stated in Corollary 1 of Theorem 1. For $d \leq m$, the patterns are split into groups according to periodicity. Case 1 patterns are proven in Lemma 1, and Case 2 patterns are proven in Lemma 4. □

5 Conclusion

We have developed the first linear-time small-space 2D dictionary matching algorithm. We work with a dictionary of d patterns, each of size $m \times m$. After preprocessing the dictionary in small space, and storing the dictionary in a compressed self-index, our algorithm processes the text in linear time. That is, it uses $O(n^2)$ time to search a 2D text that is $O(n^2)$ in size. Yet, our algorithm requires only $O(dm \log dm)$ bits of extra space.

Our algorithm is suitable for patterns that are all the same size in at least one dimension. This property is carried over from the Bird/Baker solution. Small space 2D dictionary matching for *rectangular patterns* remains an open problem. Other interesting variations of small-space dictionary matching include the approximate versions of the problem in which one or more changes occur either in the patterns or in the text.

6 Acknowledgments

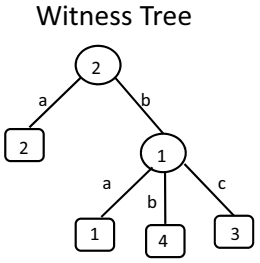
The authors wish to thank J. S. Vitter for elucidating the use of compressed indexes in pattern matching.

A Witness Tree

Given a set S of k strings, each of length m, a witness tree can be constructed to name these strings in linear time and $O(k)$ space [17]. Furthermore, the witness tree can provide the answer to the following query in constant time.

Query: For any two strings $s, s' \in S$, return a position of mismatch between s and s' if $s \neq s'$, otherwise return $m + 1$.

For completeness, we review the construction of the witness tree. We omit all proofs, which have been included in [17].



Name	String
1	abbb
2	aabc
3	cbca
4	bbaa

Fig. 3. A witness tree for several strings of length 4.

Components of witness tree:

- *Internal node:* position of a character mismatch. The position is an integer $\in [1, m]$.
- *Edge:* labeled with a character in the alphabet Σ. Two edges emanating from a node must have different labels.
- *Leaf:* A name (i.e an integer $\in [1, k]$). Identical strings receive identical names.

Construction of the witness tree begins by choosing any two strings in S and sequentially comparing them. When a mismatch is found, comparison halts and a node is added to the witness tree to represent this witness. Each successive string is compared to the witnesses stored in the tree to identify to which name, if any, the string belongs. Characters of a new string are examined in the order dictated by traversal of the witness tree, possibly out of sequence. If traversal halts at an internal node, the string receives a new name. Otherwise, traversal halts at a leaf, and the new string is sequentially compared to the string represented by the leaf as done with the first two strings.

As an example, we explain how the string *bbaa* is named 4 using the witness tree of Fig. 3. Since the root represents position 2, the first comparison finds a 'b' as the second character in *bbaa*. Then, we look at position 1 and find 'b'. Since this character is not a child of the current node, a new leaf is created to represent the name 4.

Proprocessing the witness tree to allow Lowest Common Ancestor (LCA) queries on its leaves allows us to answer the above witness query between any two strings in S in constant time.

References

[1] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two-dimensional pattern matching. *SICOMP: SIAM Journal on Computing*, 23, 1994.

[2] T. J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp*, (7):533–541, 1978.

[3] D. Belazzougui. Succinct dictionary matching with no slowdown. In *CPM*, pages 88–100, 2010.

[4] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *LATIN*, pages 88–94, 2000.

[5] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[6] H. L. Chan, W. K. Hon, T. W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.

[7] M. Crochemore, L. Gasieniec, W. Plandowski, and W. Rytter. Two-dimensional pattern matching in linear time and small space. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 1995.

[8] M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):650–674, 1991.

[9] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *FOCS*, pages 184–196, 2005.

[10] Z. Galil and J. Seiferas. Time-space-optimal string matching (preliminary report). In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 106–113, New York, NY, USA, 1981. ACM.

[11] L. Gasieniec and R. M. Kolpakov. Real-time string matching in sublinear space. In *CPM*, pages 117–129, 2004.

[12] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SICOMP: SIAM Journal on Computing*, 13, 1984.

[13] W.-K. Hon, T.-H. Ku, R. Shah, S. V. Thankachan, and J. S. Vitter. Faster compressed dictionary matching. In *SPIRE*, pages 191–200, 2010.

[14] W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Compressed index for dictionary matching. In *DCC*, pages 23–32, 2008.

[15] W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Succinct index for dynamic dictionary matching. In *ISAAC*, pages 1034–1043, 2009.

[16] M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *ALGORITHMS: Journal of Algorithms*, 5, 1984.

[17] S. Neuburger and D. Sokol. Small-space 2d compressed dictionary matching. In *CPM: 21st Symposium on Combinatorial Pattern Matching*, pages 27–39, 2010.

[18] W. Rytter. On maximal suffixes and constant-space linear-time versions of kmp algorithm. *Theor. Comput. Sci.*, 299(1-3):763–774, 2003.

[19] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.