Better Indexing for Rectangular Pattern Matching

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Adam Górkiewicz
Institute of Computer Science, University of Wrocław, Poland

Abstract

We revisit the complexity of building, given a two-dimensional string of size n, an indexing structure that allows locating all k occurrences of a two-dimensional pattern of size m. While a structure of size $\mathcal{O}(n)$ with query time $\mathcal{O}(m+k)$ is known for this problem under the additional assumption that the pattern is a square [Giancarlo, SICOMP 1995], a popular belief was that for rectangular patterns one cannot achieve such (or even similar) bounds, due to a lower bound for a certain natural class of approaches [Giancarlo, WADS 1993]. We show that, in fact, it is possible to construct a very simple structure of size $\mathcal{O}(n \log n)$ that supports such queries for any rectangular pattern in $\mathcal{O}(m + k \log^{\varepsilon} n)$ time, for any $\varepsilon > 0$.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Pattern matching

Keywords and phrases 2D strings, pattern matching, string indexing

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.33

Related Version *Full Version*: `https://arxiv.org/abs/2508.17365`

Funding Partially supported by the Polish National Science Centre grant number 2023/51/B/ST6/01505.

Acknowledgements We would like to thank Itai Boneh and Panagiotis Charalampopoulos for participating in the initial discussion.

1 Introduction

In the area of algorithms on strings, two basic algorithmic questions are pattern matching and string indexing. In the former, we aim to locate an occurrence of the pattern in the text, while in the latter the goal is to preprocess the text for multiple such queries. The complexity of both problems is well understood, at least for the case of regular strings and exact occurrences. In particular, a text of length n can be indexed in $\mathcal{O}(n)$ space, so that all k occurrences of a pattern of length m can be retrieved in $\mathcal{O}(m+k)$ time: this is a textbook application of suffix trees, already explained in the original article by Weiner [22].

In this paper, we consider two-dimensional strings, which are simply (two-dimensional) arrays of characters. Such a generalisation is naturally motivated by possible applications in image processing, and the complexity of pattern matching for two-dimensional strings has been already considered in the 70s [4,5], and further investigated in the early 90s [1,10]. A common assumption in all of these papers is that both the text and the pattern are square arrays, but this assumption is just to avoid clutter and can be removed without encountering any technical difficulties.

To state the results concerning indexing two-dimensional strings, let the dimensions of the text be $H \times W$, with the total size $n = HW$, and the dimensions of the pattern be $h \times w$, with the total size $m = hw$. Giancarlo [12] introduced the LSuffix tree of a matrix, based on a linearisation of a two-dimensional string (similar concept has been also used by Amir and Farach [2]). This allowed him to design an index of (asymptotically optimal) size $\mathcal{O}(n)$ that supports queries in $\mathcal{O}(m)$ time (for simplicity, we restate the bounds for constant alphabets), but only if the pattern is a square matrix. This assumption was in fact inherent to his approach, as otherwise it is not clear how to linearise the strings. For the general case, he designed another structure, called the submatrix tree [11], of size $\mathcal{O}(\min(H, W)n)$ that supports queries in $\mathcal{O}(m)$ time. In the same paper, he defined an abstract notion of an index for a two-dimensional text, and proved that the size of any such index must be $\Omega(\min(H, W)n)$, making his construction essentially optimal (perhaps up to a logarithmic factor). Consequently, subsequent work focused on the case of square patterns [7, 9, 13–16, 18, 21].

Our contribution. We revisit the complexity of indexing a two-dimensional text for the general case of rectangular patterns. We observe that, in fact, the abstract notion of an index, as defined by Giancarlo [11], is somewhat restricted: he defines what it means for one pattern to be a prefix of another, and then requires that the index has a form of a compacted tree, with every node corresponding to some pattern occurring in the text, and the parent of each node corresponding to its prefix. This is consistent with the notion of one-dimensional suffix trees, but suffix trees are not the only known indexing structures. For example, suffix arrays [20] use $\mathcal{O}(n)$ space and allow retrieving the occurrences in $\mathcal{O}(m + \log n + k)$ time, even though they are not based on storing a compacted trie (although they are of course related to suffix arrays). We show that a very simple idea allows us to obtain the following result.

▶ **Theorem 1.** *For a two-dimensional text of size n, there is an $\mathcal{O}(n \log n)$-space data structure that, given a two-dimensional pattern of size m, reports all k occurrences of the pattern in the text in time $\mathcal{O}(m + k \log^{\varepsilon} n)$, for any constant $\varepsilon > 0$.*

Computational model. In the above theorem we assume the standard word RAM model with words of logarithmic (in the size of the input) length. Basic arithmetic operations on such words (and indirect addressing) are assumed to take constant time. Each character is assumed to fit in a single machine word. We measure the size of our data structures in the number of words.

2 Preliminaries

One-dimensional strings. A (one-dimensional) string is a sequence of characters. We index positions in a string starting from 1. For a string S of length n, we write $S[i]$ to denote its i-th character and $S[i\,..\,j]$ to denote the substring spanning positions i through j, inclusive. If only one endpoint is provided, we interpret the interval as a prefix or suffix: $S[..\,i] := S[1\,..\,i]$ is the prefix of length i; $S[i\,..] := S[i\,..\,n]$ is the suffix starting at position i.

Two-dimensional strings. We define two-dimensional strings as rectangular arrays of characters. We refer to the total number of characters in a two-dimensional string as its *size*. Rows and columns are indexed starting from 1. For a two-dimensional string S we write $S[i]$ to denote its i-th row, interpreted as a one-dimensional string. Consequently, $S[i][j]$ denotes the character in the i-th row and j-th column.

Meta-characters. In our algorithm, we reduce two-dimensional indexing to a collection of problems concerning one-dimensional strings. This is done by treating fixed-length fragments of rows of the two-dimensional text as atomic symbols, which we refer to as *meta-characters*. Formally, for a two-dimensional string S and a fixed width w, a meta-character is a substring of the form $S[i][j\,..\,j + w - 1]$. To effectively operate on such meta-characters, we will represent each of them by an integer from [poly(n)] that fits inside a single machine word[1], called the identifier. The identifiers of two meta-characters will be different if and only if the meta-characters themselves are different.

Compacted trie. We use the standard compacted trie, also known as a compressed trie or Patricia trie, for storing a set of strings. The strings consist of either characters or meta-characters. In either case, we assume that the symbols fit in a single machine word. A compacted trie built for a set of k strings is of size $\mathcal{O}(k)$, as we collapse maximal paths consisting of nodes with exactly one child into a single edge, thus the number of inner nodes is strictly smaller than the number of leaves. The remaining nodes are called explicit, while the dissolved nodes are called implicit.

Tools. We will need a few data structures. The first is called a deterministic dictionary.

▶ **Theorem 2** ([8]). *Given a set S of n keys, we can build a dictionary structure of size $\mathcal{O}(n)$ that allows constant-time access to any element of S (and its associated information).*

The second is a range reporting structure (we note that other trade-offs between the space and reporting time are possible, but we state only one of them to avoid clutter).

▶ **Theorem 3** ([6, Theorem 2.1]). *Given a collection of n points in $\{1, \ldots, n\}^2$, we can build a structure of size $\mathcal{O}(n)$ that, given an axis-aligned rectangle, reports all k points inside it in time $\mathcal{O}((1+k)\log^{\varepsilon} n)$, for any constant $\varepsilon > 0$.*

Finally, we need to implement *prefix search* on a compacted trie. That is, given a query string, we traverse the trie to determine whether it occurs as a prefix of any stored string. If so, we return the corresponding node (which may be implicit); otherwise, we report that no such prefix match exists.

▶ **Lemma 4.** *A compacted trie storing k strings uses $\mathcal{O}(k)$ additional space (on top of the strings themselves). Assuming constant-time read-only random access to any character, prefix search for a query string of length m takes $\mathcal{O}(m)$ time.*

Proof. To implement prefix search, we start at the root and descend following the appropriate edge. This requires storing all the edges outgoing from an explicit node in a dictionary structure, which we implement with Theorem 2. Since the total degree over all explicit nodes is $\mathcal{O}(k)$, the combined space used by all dictionaries is $\mathcal{O}(k)$. For implicit nodes, we only verify whether the subsequent characters of the query string are the same as on the edge, using constant-time read-only random access to one of the stored strings. Then, the time per each character of the query string is constant, so $\mathcal{O}(m)$ overall. ◀

3 Two-Dimensional Index Construction

In this section, we present our data structure for indexing a two-dimensional text and prove Theorem 1. The idea behind our approach is to reduce the two-dimensional pattern matching problem to a collection of one-dimensional problems. Each of these problems can then be solved efficiently by plugging in range reporting structures, as usual in the one-dimensional setting. Throughout this section, we write T to denote the $H \times W$ input text and P to denote the $h \times w$ query pattern, with $n = HW$ and $m = hw$ denoting their respective sizes.

[1] It is in fact enough that it fits in a constant number of machine words.

The data structure consists of two symmetric components, designed to handle the query depending on whether the pattern is taller or wider. Given an $h \times w$ pattern, we use one component if $h \geq w$ and the symmetric one (with rows and columns interchanged) otherwise. We describe how to handle tall patterns, i.e., those with $h \geq w$, as the other case is fully symmetric. To handle patterns of different widths, we store a separate index for each possible pattern width. More precisely, for each width $w \in \{1, \dots, W\}$, we construct a dedicated data structure that supports searching for patterns of fixed width w and any height $h \geq w$.

For a fixed width w, we reduce the two-dimensional matching problem to a one-dimensional problem. Specifically, we interpret the pattern as a one-dimensional string of length h, where the i-th symbol is a *meta-character* corresponding to the i-th row $P[i]$ of the pattern, that is, a contiguous block of w characters from that row. As explained earlier, we will assign identifiers to these meta-characters, so that we can treat them as integers from $[\text{poly}(n)]$. We first explain how to ensure that we can access the identifier of any meta-character in the pattern and the text in constant time, after $\mathcal{O}(n \log n)$ space preprocessing of the text, and then define the one-dimensional problem.

Encoding the text. The construction follows the classical Karp-Miller-Rosenberg (KMR) approach [17], and is based on assigning integer identifiers to all fragments of the form $T[i][j \mathinner{.\,.} j+2^k-1]$, for all k such that $2^k \leq W$. For each such fragment, we define its identifier as the lexicographic rank of its substring among all distinct substrings of the same length, and store this value. Then, the identifier of $T[i][j \mathinner{.\,.} j+w-1]$ consists of the identifiers of two overlapping fragments of length $2^{\lfloor \log w \rfloor}$ that together cover the whole $T[i][j \mathinner{.\,.} j+w-1]$. Since there are $\mathcal{O}(n)$ fragments per length and $\mathcal{O}(\log W)$ relevant lengths, the total space required is $\mathcal{O}(n \log n)$.

Encoding the pattern. To encode a pattern row $P[i]$ of width w, we represent the corresponding meta-character by the pair of identifiers of its prefix and suffix of length 2^k, where $k = \lfloor \log w \rfloor$. To support this, during text preprocessing, we collect all distinct substrings $T[i][j \mathinner{.\,.} j+2^k-1]$ and store them in a compacted trie, where each leaf is labeled with the lexicographic rank among all substrings of T of length 2^k. We build a separate compacted trie for each relevant value of k; since each trie requires $\mathcal{O}(n)$ space and there are $\mathcal{O}(\log W)$ values of k, the total space used by all the compacted tries is $\mathcal{O}(n \log n)$. At query time, we extract the 2^k-length prefix and suffix of $P[i]$ and search for them in the corresponding compacted trie to obtain their identifiers. If either substring is not found, we report that the pattern does not occur in the text. Otherwise, in $\mathcal{O}(m)$ time we obtain the sought identifier for each row of the pattern.

One-dimensional problems. To complete the reduction, we must identify all occurrences of the one-dimensional meta-character pattern within the two-dimensional text. To that end, we consider all windows of w consecutive columns. Each such w-column strip defines a one-dimensional text of length H, where the j-th character is a meta-character corresponding to the block of w consecutive characters in the j-th row of the strip. Formally, to define the one-dimensional texts corresponding to each w-column strip, we fix a horizontal offset $i \in \{1, \dots, W-w+1\}$ and define a one-dimensional text of length H consisting of the meta-characters $T[j][i \mathinner{.\,.} i+w-1]$ for all positions $j \in \{1, \dots, H\}$. This gives a collection of $W-w+1$ one-dimensional texts over the same meta-character alphabet. Each occurrence of the meta-character pattern in one of these one-dimensional texts corresponds one-to-one to a two-dimensional occurrence of the original pattern in the text.

In each of these one-dimensional instances, the derived meta-character pattern has length $h \geq w$. We exploit this assumption when designing a (simple) indexing structure in Section 4.

▶ **Lemma 5.** *For a parameter w and a collection of one-dimensional texts of total length n, there is a data structure requiring $\mathcal{O}(n/w)$ additional space (on top of the strings) that, given a one-dimensional pattern of length $h \geq w$, reports all k occurrences of the pattern in the texts in time $\mathcal{O}(hw + (w + k)\log^{\varepsilon} n)$, for any constant $\varepsilon > 0$.*

We could apply Lemma 5 for each width $w \in \{1, \ldots, W\}$ to handle patterns of width w and height $h \geq w$. The total additional space is bounded by $\sum_{w=1}^{W} \mathcal{O}(n/w) = \mathcal{O}(n \log n)$, matching the requirements of Theorem 1. However, when h is smaller than $\log^{\varepsilon} n$, it holds that $w \log^{\varepsilon} n$ in the query time is larger than m. We thus handle the case of small w separately as follows.

▶ **Lemma 6.** *For a collection of one-dimensional texts of total length n, there is an $\mathcal{O}(n)$-space data structure that, given a one-dimensional pattern of any length h, reports all k occurrences of the pattern in the texts in time $\mathcal{O}(h + k)$.*

Proof. We store all suffixes of all input texts, each terminated with a distinct delimiter, in a compacted trie implemented with Lemma 4. The total number of suffixes is $\mathcal{O}(n)$, so the total required space is $\mathcal{O}(n)$. Querying for a pattern of length h is implemented by performing a prefix search for the pattern in the trie and reporting all leaves in the corresponding subtree. ◀

We apply Lemma 6 for every width $w \leq \log n$, and otherwise use Lemma 5. The total space is still $\mathcal{O}(n \log n)$. To bound the query time, we observe that whenever we use Lemma 6 the query time is $\mathcal{O}(h + k) = \mathcal{O}(m + k)$, and whenever we use Lemma 5 we can assume $w > \log n$ so the query time is $\mathcal{O}(hw + (w + k)\log^{\epsilon} n) = \mathcal{O}(m + k \log^{\epsilon} n)$.

4 One-Dimensional Index for Long Patterns

In this section, we prove Lemma 5 by designing an indexing structure for the one-dimensional setting that takes advantage of the assumption that the pattern is sufficiently long. We briefly comment that such a setup can be seen as related to locally consistent anchors, and such an assumption has been used in the literature [3]. In our case, a simple and self-contained approach based on defining some compacted tries and storing a range reporting structure is enough, though. The idea of using range searching to solve indexing is, of course, quite common in the literature [19].

Setup. We assume that the input texts allow constant-time read-only random access to the individual characters, and each of them fits in a single machine word. Throughout this section, we fix a parameter w, corresponding to the lower bound on the length of the pattern. Texts shorter than w can be discarded, as they cannot contain an occurrence of a pattern of length $h \geq w$.

Preprocessing. For each text T, we define a collection of *cuts*: positions between characters spaced at regular intervals of length w. Formally, we insert a cut at every position i such that $i \equiv 0 \pmod{w}$, including $i = 0$. Each cut partitions T into a prefix $T_1 = T[\,..\,i]$ and a suffix $T_2 = T[i+1\,..\,]$, where either may be empty. We associate the cut with the pair (T_1, T_2), which serves as its representation. We organize the collection of cuts using a two-dimensional range reporting structure. Let $\mathcal{T}_1$ denote the set of all prefixes T_1 reversed, and let $\mathcal{T}_2$ denote the set of all suffixes T_2 extracted from the cuts. We build two compacted tries:

- $\mathcal{S}_1$, storing the strings in $\mathcal{T}_1$,
- $\mathcal{S}_2$, storing the strings in $\mathcal{T}_2$.

To ensure that each string from $\mathcal{T}_1$ and from $\mathcal{T}_2$ corresponds to a unique leaf in the respective trie, we conceptually prepend and append a distinct delimiter character to each prefix and suffix, respectively.

Next, we assign a pre-order number to each leaf of $\mathcal{S}_1$ and $\mathcal{S}_2$ via a depth-first traversal. Each cut then defines a point $(x, y) \in \mathbb{Z}^2$, where:

- x is the pre-order number of the leaf in $\mathcal{S}_1$ corresponding to the reversal of T_1,
- y is the pre-order number of the leaf in $\mathcal{S}_2$ corresponding to T_2.

This yields a collection of $\mathcal{O}(n/w)$ such points, one per cut. We store these points in an instance of Theorem 3.

Answering a query. Given a pattern P of length $h \geq w$, our goal is to report all of its occurrences in the input texts. Since the pattern has length at least w, any occurrence must span at least one cut position from the collection defined for the texts. We iterate over all positions at which such a cut could intersect an occurrence of the pattern. We refer to each such position within the pattern as an *anchor*, and consider only the first w possible anchors, to ensure that every occurrence of P is reported exactly once.

Specifically, we consider all positions $j \in \{0, 1, \dots, w-1\}$ where the pattern may be anchored. Each such position induces a partition of P into a prefix $P_1 = P[\,.\,.\,j]$ and a suffix $P_2 = P[j+1\,.\,.\,]$. The task is to find and report all cuts in the texts that partition some text into (T_1, T_2) such that P_1 is a suffix of T_1 and P_2 is a prefix of T_2. This is implemented as follows. First, we extract the corresponding prefix-suffix pair (P_1, P_2) from the pattern. We then locate the (possibly implicit) node v_1 in $\mathcal{S}_1$ corresponding to the reversal of P_1, and the node v_2 in $\mathcal{S}_2$ corresponding to P_2. If either node does not exist, the current position cannot yield any occurrences and is skipped. Otherwise, let $[\ell_1, r_1]$ and $[\ell_2, r_2]$ be the ranges of pre-order numbers of leaves in the subtrees of v_1 and v_2, respectively.

The problem now reduces to a two-dimensional orthogonal range reporting query: report all stored points (x, y) that lie within the rectangle $[\ell_1, r_1] \times [\ell_2, r_2]$. Each reported point corresponds to a valid cut that certifies an occurrence of the pattern. Querying the instance of Theorem 3 yields the query bound claimed in Lemma 5.

References

1 Amihood Amir, Gary Benson, and Martin Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM J. Comput.*, 23(2):313–323, 1994. doi:10.1137/S0097539792226321.

2 Amihood Amir and Martin Farach. Two-dimensional dictionary matching. *Inf. Process. Lett.*, 44(5):233–239, 1992. doi:10.1016/0020-0190(92)90206-B.

3 Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns: Anchors are all you need. *Proc. VLDB Endow.*, 16(9):2117–2131, 2023. doi:10.14778/3598581.3598586.

4 Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comput.*, 7(4):533–541, 1978. doi:10.1137/0207043.

5 Richard S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977. doi:10.1016/0020-0190(77)90017-5.

6 Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *SCG*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.

7 Ying Choi and Tak Wah Lam. Dynamic suffix tree and two-dimensional texts management. *Inf. Process. Lett.*, 61(4):213–220, 1997. doi:10.1016/S0020-0190(97)00018-5.

8 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

9 Kimmo Fredriksson, Gonzalo Navarro, and Esko Ukkonen. An index for two dimensional string matching allowing rotations. In *IFIP TCS*, volume 1872 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2000. doi:10.1007/3-540-44929-9_5.

10 Zvi Galil and Kunsoo Park. Alphabet-independent two-dimensional witness computation. *SIAM J. Comput.*, 25(5):907–935, 1996. doi:10.1137/S0097539792241941.

11 Raffaele Giancarlo. An index data structure for matrices, with applications to fast two-dimensional pattern matching. In *WADS*, volume 709 of *Lecture Notes in Computer Science*, pages 337–348. Springer, 1993. doi:10.1007/3-540-57155-8_260.

12 Raffaele Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM J. Comput.*, 24(3):520–562, 1995. doi:10.1137/S0097539792231982.

13 Raffaele Giancarlo and Roberto Grossi. Parallel construction and query of suffix trees for two-dimensional matrices. In *SPAA*, pages 86–97. ACM, 1993. doi:10.1145/165231.165243.

14 Raffaele Giancarlo and Roberto Grossi. On the construction of classes of suffix trees for square matrices: Algorithms and applications. In *ICALP*, volume 944 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 1995. doi:10.1007/3-540-60084-1_67.

15 Raffaele Giancarlo and Roberto Grossi. Parallel construction and query of index data structures for pattern matching on square matrices. *J. Complex.*, 15(1):30–71, 1999. doi:10.1006/JCOM.1998.0496.

16 Raffaele Giancarlo and Daniela Guaiana. On-line construction of two-dimensional suffix trees. *J. Complex.*, 15(1):72–127, 1999. doi:10.1006/JCOM.1998.0495.

17 Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *STOC*, pages 125–136. ACM, 1972. doi:10.1145/800152.804905.

18 Dong Kyue Kim, Joong Chae Na, Jeong Seop Sim, and Kunsoo Park. Linear-time construction of two-dimensional suffix trees. *Algorithmica*, 59(2):269–297, 2011. doi:10.1007/S00453-009-9350-Z.

19 Moshe Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *Lecture Notes in Computer Science*, pages 267–302. Springer, 2013. doi:10.1007/978-3-642-40273-9_18.

20 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

21 Joong Chae Na, Raffaele Giancarlo, and Kunsoo Park. On-line construction of two-dimensional suffix trees in $O(n^2 \log n)$ time. *Algorithmica*, 48(2):173–186, 2007. doi:10.1007/S00453-007-0063-X.

22 Peter Weiner. Linear pattern matching algorithms. In *SWAT*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.