# Optimal Prefix and Suffix Queries on Texts

Maxime Crochemore, Costas S. Iliopoulos, Mohammad Sohel Rahman

# Optimal Prefix and Suffix Queries on Texts

Maxime Crochemore[1,2,4], Costas S. Iliopoulos[1,3], and M. Sohel Rahman[1,3]

[1] Algorithm Design Group
Department of Computer Science
King's College London
Strand, London WC2R 2LS, England
http://www.dcs.kcl.ac.uk/adg
[2] Gaspard-Monge Institute
Université de Marne-la-Vallée, France
[3] {csi, sohel}@dcs.kcl.ac.uk
[4] Maxime.Crochemore@kcl.ac.uk

**Abstract.** In this paper, we study a restricted version of the position restricted pattern matching problem introduced and studied Mäkinen and Navarro [Position-Restricted Substring Searching, LATIN 2006]. In the problem handled in this paper, we are interested in those occurrences of the pattern that lies in a suffix or in a prefix of the given text. We achieve optimal query time for our problem against a data structure which is an extension of the classic suffix tree data structure. The time and space complexity of the data structure is dominated by that of the suffix tree. Notably, the (best) algorithm by Mäkinen and Navarro, if applied to our problem, gives sub-optimal query time and the corresponding data structure also requires more time and space.

## 1 Introduction

The classical pattern matching problem is to find all the occurrences of a given pattern $\mathcal{P} = \mathcal{P}[1..m]$ of length $m$ in a text $\mathcal{T} = \mathcal{T}[1..n]$ of length $n$, both being sequences of characters drawn from a finite character set $\Sigma$. This problem, along with its numerous variants, has been the focus of extensive research in the field of computer science. Due to the need of various practical applications, most recent works in pattern matching have considered *'inexact matching'*. Many types of differences have been defined and studied in the literature, namely, errors (Hamming distance, LCS [10, 19], edit distance [10, 20]), wild cards or don't cares [10, 11, 14, 27, 29], rotations [1, 4, 15], scaling [2, 6, 3], permutations [8] among others.

The indexing problem for pattern matching is to preprocess a given text $\mathcal{T}[1..n]$ over an alphabet $\Sigma$ as efficiently as possible to build a data structure to support the following form of online queries: Given a pattern $\mathcal{P}[1..m]$ over $\Sigma$ find the occurrences of $\mathcal{P}$ in $\mathcal{T}$. The indexing problem and its many variants have been central in pattern matching [17, 13, 5, 9, 22, 23, 26, 29, 28, 10]. Recently, Mäkinen and Navarro, in [24], considered an interesting variant of indexed pattern matching, where only the occurrences of a given pattern starting in a particular area, are of interest. In particular, in this variant, the query provides an

interval $[\ell..r], 1 \leq \ell \leq r \leq n$ along with the pattern $\mathcal{P}$ and the occurrences of $\mathcal{P}$ in $\mathcal{T}[\ell..r]$ are sought for. These queries, as is pointed out in [24], are fundamental in many text search situations where one wants to search only a part of the text. The authors in [24] presented a number of algorithms depending on different trade-offs between the time and space complexities. The best query time they achieved was $O(m+\log\log n+K)$ ($K$ is the output size) against a data structure exhibiting $O(n\log^{1+\epsilon} n)$ space and time complexity, where $0 \leq \epsilon \leq 1$.

In this paper, we study a restricted version of the problem handled in [24]. In particular, we are interested in those occurrences of the pattern that lies in a suffix or in a prefix of the given text. In other words, in our case, the query interval $[\ell..r]$ is of special form: either $\ell = 1$, i.e. prefix search, or $r = n$, i.e. suffix search. This kind of queries seem to be interesting in many contexts as well. For example, many of the queries in real life are restricted up to the table of contents of a book or in the title and abstract of a scientific document. Another possible application for this problem can be found in Biological Sequence Assembly where the question is to build a kind of Shortest Super-string Common to a given set of sequences. In the greedy strategy for sequence assembly, this is usually done by finding markers close to the ends i.e. suffixes of the strings: these markers witness possible overlaps between a suffix of a sequence and a prefix of another sequence. Sequence having large overlaps are assembled in a longer sequence and so on.

In this paper, we present an efficient data structure to handle such online queries in the prefix or suffix of a given text in optimal time. Note that, the best query time achieved in [24] (for the more general problem) is not optimal due to the additional (mild) $\log\log n$ term. As a result, if applied to our problem, their algorithm exhibits sub-optimal query time and the corresponding data structure also requires more time and space.

The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts. The main result of this paper is presented in Section 3. We conclude briefly in Section 4.

## 2 Preliminaries

A *text*, also called a *string*, is a sequence of zero or more symbols from an alphabet $\Sigma$. A text $\mathcal{T}$ of length $n$ is denoted by $\mathcal{T}[1..n] = \mathcal{T}_1\mathcal{T}_2\ldots\mathcal{T}_n$, where $\mathcal{T}_i \in \Sigma$ for $1 \leq i \leq n$. The *length* of $\mathcal{T}$ is denoted by $|\mathcal{T}| = n$. A string $w$ is a *factor* or *substring* of $\mathcal{T}$ if $\mathcal{T} = uwv$ for $u, v \in \Sigma^*$; in this case, the string $w$ occurs at position $|u| + 1$ in $\mathcal{T}$. The factor $w$ is denoted by $\mathcal{T}[|u| + 1..|u| + |w|]$. A *prefix* (*suffix*) of $\mathcal{T}$ is a factor $\mathcal{T}[x..y]$ such that $x = 1$ ($y = n$), $1 \leq y \leq n$ ($1 \leq x \leq n$). We define $i$th prefix to be the prefix ending at position $i$ i.e. $\mathcal{T}[1..i], 1 \leq i \leq n$. On the other hand, $i$th suffix is the suffix starting at position $i$ i.e. $\mathcal{T}[i..n], 1 \leq i \leq n$.

In traditional pattern matching problem, we want to find the occurrences of a given pattern $\mathcal{P}[1..m]$ in a text $\mathcal{T}[1..n]$. The pattern $\mathcal{P}$ is said to occur at position $i \in [1..n]$ of $\mathcal{T}$ if and only if $\mathcal{P} = \mathcal{T}[i..i+m-1]$. We use $Occ_{\mathcal{T}}^{\mathcal{P}}$ to denote the set of occurrences of $\mathcal{P}$ in $\mathcal{T}$.

The problem we handled in this paper can be defined formally as follows.

***Problem "PMP/S" (Pattern Matching in a Prefix/Suffix).*** *We are given a text $\mathcal{T}$ of length $n$. Preprocess $\mathcal{T}$ to answer the following form of queries.*
**Query:** *Given a pattern $\mathcal{P}$ and a query interval $[\ell..r]$, with $1 \leq \ell \leq r \leq n$, where either $\ell = 1$ (prefix query) or $r = n$ (suffix query), construct the set*

$$Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]} = \{i \mid i \in Occ^{\mathcal{P}}_{\mathcal{T}} \text{ and i} \in [\ell..\text{r}]\}.$$

It is easy to realize that Problem PMP/S is a special case of the problem handled in [24]. Apart from being interesting from pure combinatorial point if view, Problem PMP/S is motivated by practical applications as follows. In real life pattern matching applications, many of the queries are restricted to table of contents of a book or in the title and abstract of a scientific document. As a result, it is interesting to see whether the solution of [24] can be improved to optimal for this special case.

In traditional indexing problem one of the basic data structures used is the suffix tree data structure. In our indexing problem, we make use of this suffix tree data structure. A complete description of a suffix tree is beyond the scope of this paper, and can be found in [25, 31] or in any textbook on stringology (e.g. [12, 18]). However, for the sake of completeness, we define the suffix tree data structure as follows. Given a string $\mathcal{T}$ of length $n$ over an alphabet $\Sigma$, the suffix tree $ST_{\mathcal{T}}$ of $\mathcal{T}$ is the compacted trie of all suffixes of $\mathcal{T}\$$, where $\$ \notin \Sigma$. Each leaf in $ST_{\mathcal{T}}$ represents a suffix $\mathcal{T}[i..n]$ of $\mathcal{T}$ and is labeled with the index $i$. We refer to the list (in left-to-right order) of indices of the leaves of the subtree rooted at node $v$ as the leaf-list of $v$; it is denoted by $LL(v)$. Each edge in $ST_{\mathcal{T}}$ is labeled with a nonempty substring of $\mathcal{T}$ such that the path from the root to the leaf labeled with index $i$ spells the suffix $\mathcal{T}[i..n]$. For any node $v$, we let $\ell_v$ denote the string obtained by concatenating the substrings labeling the edges on the path from the root to $v$ in the order they appear. Several algorithms exist that can construct the suffix tree $ST_{\mathcal{T}}$ in $O(n \log \Sigma)$ time[5] [25, 31]. Given the suffix tree $ST_{\mathcal{T}}$ of a text $\mathcal{T}$ we define the "locus" $\mu^{\mathcal{P}}$ of a pattern $\mathcal{P}$ as the node in $ST_{\mathcal{T}}$ such that $\ell_{\mu^{\mathcal{P}}}$ has the prefix $\mathcal{P}$ and $|\ell_{\mu^{\mathcal{P}}}|$ is the smallest of all such nodes. Note that the locus of $\mathcal{P}$ does not exist, if $\mathcal{P}$ is not a substring of $\mathcal{T}$. Therefore, given $\mathcal{P}$, finding $\mu^{\mathcal{P}}$ suffices to determine whether $\mathcal{P}$ occurs in $\mathcal{T}$. Given a suffix tree of a text $\mathcal{T}$, a pattern $\mathcal{P}$, one can find its locus and hence the fact whether $\mathcal{T}$ has an occurrence of $\mathcal{P}$ in optimal $O(|\mathcal{P}|)$ time [25, 31]. In addition to that, all such occurrences can be reported in constant time per occurrence.

## 3 Problem PMP/S

In this section, we handle Problem PMP/S. Our basic idea is to build an index data structure that would solve the problem in two steps. At first, it will (implicitly) give us the set $Occ^{\mathcal{P}}_{\mathcal{T}}$. Then, the index would 'select' some of the

[5] For bounded alphabet the running time remains linear, i.e. $O(n)$.

occurrences to provide us with our desired set $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}$, where either $\ell = 1$ or $r = n$.

The idea we employ is as follows. We first construct a suffix tree $ST_{\mathcal{T}}$. According to the definition of suffix tree, each leaf in $ST_{\mathcal{T}}$ is labeled by the starting location of its suffix. We do some preprocessing on $ST_{\mathcal{T}}$ as follows. We maintain a linked list of all leaves in a left-to-right order. In other words, we realize the list $LL(\mathcal{R})$ in the form of a linked list, where $\mathcal{R}$ is the root of the suffix tree. In addition to that, we set pointers $v.left$ and $v.right$ from each tree node $v$ to its leftmost leaf $v_\ell$ and rightmost leaf $v_r$ (considering the subtree rooted at $v$) in the linked list. It is easy to realize that, with these set of pointers at our disposal, we can indicate the set of occurrences of a pattern $\mathcal{P}$ by the two leaves $\mu^{\mathcal{P}}_\ell$ and $\mu^{\mathcal{P}}_r$ because all the leaves between and including $\mu^{\mathcal{P}}_\ell$ and $\mu^{\mathcal{P}}_r$ in $LL(\mathcal{R})$ correspond to the occurrences of $\mathcal{P}$ in $\mathcal{T}$. In what follows, we define the term $\ell_T$ and $r_T$ such that $LL(\mathcal{R})[\ell_T] = \mu^{\mathcal{P}}_\ell$ and $LL(\mathcal{R})[r_T] = \mu^{\mathcal{P}}_r$, where $\mathcal{R}$ is the root of $ST_{\mathcal{T}}$. Now recall that our data structure has to be able to somehow "select" and report only those occurrences that lies in the query interval. To solve this we use a solution to the following much studied problem.

***Problem "RMIN/MAX" (Range Minima/Maxima Query Problem).*** *We are given an array $A[1..n]$ of numbers. We need to preprocess $A$ to answer the following form of queries:*
**Query:** *Given an interval $I = [i_s..i_e], 1 \leq i_s \leq i_e \leq n$, the goal is to find the index $k$ (or the value $A[k]$ itself) with minimum (maximum, in the case of Range Maxima Query) value $A[k]$ for $k \in I$.*

Problem RMIN/MAX has received much attention in the literature and the best solution can build a data structure in $O(n)$ time and space and can answer subsequent queries in $O(1)$ time per query [16, 7].

Now, to complete the construction of the data structure we simply preprocess the array data structure $LL(\mathcal{R})$ for both range minima and range maxima queries. Algorithm 1 formally states the steps to build our data structure. In the rest of this paper, we refer to this data structure as IDS_PMP/S.

**Algorithm 1** Algorithm to build IDS_PMP/S

1: Build a suffix tree $ST_{\mathcal{T}}$ of $\mathcal{T}$. Let the root of $ST_{\mathcal{T}}$ is $\mathcal{R}$.
2: Label each leaf of $ST_{\mathcal{T}}$ by the starting location of its suffix.
3: Construct a linked list $\mathcal{L}$ realizing $LL(\mathcal{R})$. Each element in $\mathcal{L}$ is the label of the corresponding leaf in $LL(\mathcal{R})$.
4: **for** each node $v$ in $ST_{\mathcal{T}}$ **do**
5: Store $v.left = i$ and $v.right = j$ such that $\mathcal{L}[i]$ and $\mathcal{L}[j]$ corresponds to, respectively, (leftmost leaf) $v_\ell$ and (rightmost leaf) $v_r$ of $v$.
6: **end for**
7: Preprocess $\mathcal{L}$ for both Range Minima and Range Maxima Queries.

### 3.1 Analysis

Let us analyze the the running time of Algorithm 1. Step 1 builds the traditional suffix tree requiring $O(n \log \Sigma)$ time. Note that, for bounded alphabet the time required is reduced to $O(n)$. Step 2 can be done easily while building the suffix tree. Step 3 and Step 4 can be done together in $O(n)$ by traversing $ST_{\mathcal{T}}$ using a breadth first or in order traversal. Finally, the preprocessing for range minima and range maxima queries require $O(n)$ time and space [16, 7]. So IDS_PMP/S can be constructed in $O(n)$ and $O(n \log \Sigma)$ time and space, respectively for bounded and general alphabet.

**Algorithm 2** Algorithm for Query Processing

1: Find $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$.
2: Set $i = \mu^{\mathcal{P}}.left$, $j = \mu^{\mathcal{P}}.right$.
3: $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]} = \epsilon$
4: **if** $\ell = 1$\{This is a prefix query\} **then**
5: $\quad FindPrefixOccurrence(\mathcal{L}, r, i, j)$\{See Algorithm 3\}
6: **else**
7: $\quad$ **if** $r = 1$\{This is a suffix query\} **then**
8: $\quad\quad FindSuffixOccurrence(\mathcal{L}, \ell, i, j)$\{See Algorithm 4\}
9: $\quad$ **end if**
10: **end if**

**Algorithm 3** Procedure $FindPrefixOccurrence(\mathcal{L}, r, i, j)$

1: $k = RangeMinimaQuery(\mathcal{L}, i, j)$
2: **if** $\mathcal{L}[k] < r$ **then**
3: $\quad$ Set $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]} = Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]} \bigcup \mathcal{L}[k]$
4: $\quad FindPrefixOccurrence(\mathcal{L}, r, i, k-1)$
5: $\quad FindPrefixOccurrence(\mathcal{L}, r, k+1, j)$
6: **end if**

**Algorithm 4** Procedure $FindSuffixOccurrence(\mathcal{L}, \ell, i, j)$

1: $k = RangeMaximaQuery(\mathcal{L}, i, j)$
2: **if** $\mathcal{L}[k] > \ell$ **then**
3: $\quad$ Set $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]} = Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]} \bigcup \mathcal{L}[k]$
4: $\quad FindSuffixOccurrence(\mathcal{L}, \ell, i, k-1)$
5: $\quad FindSuffixOccurrence(\mathcal{L}, \ell, k+1, j)$
6: **end if**

### 3.2 Query processing

Now we discuss the query processing. Suppose we are given a query pattern $\mathcal{P}$ along with a query interval $[\ell..r]$. We first find the locus $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$. Let $i = \mu^{\mathcal{P}}.left$ and $j = \mu^{\mathcal{P}}.right$. This means, we get the set $Occ^{\mathcal{P}}_{\mathcal{T}}$ in the form of $\mathcal{L}[i..j]$ spending $O(m)$ time. Now, suppose we are performing a prefix query, i.e. $\ell = 1$. So we want to compute the set $Occ^{\mathcal{P}}_{\mathcal{T}[1..r]}$. It is easy to see that

$$Occ^{\mathcal{P}}_{\mathcal{T}[1..r]} = \{\mathcal{L}[k] \mid i \leq k \leq j, \mathcal{L}[k] \leq r\}.$$

To compute $Occ^{\mathcal{P}}_{\mathcal{T}[1..r]}$, we apply a divide and conquer approach as follows. We perform a Range Minima Query on $\mathcal{L}$ on the interval $[i..j]$. Suppose the query returns the index $k$. If $\mathcal{L}[k] \leq r$ then $\mathcal{L}[k] \in Occ^{\mathcal{P}}_{\mathcal{T}[1..r]}$ and then we perform the range minima query on the intervals $[i..k-1]$ and $[k+1..j]$ and continue as before. If any of the queries returns $k$ such that $L[k] > r$ we stop. It is easy to verify that this would give us the set $Occ^{\mathcal{P}}_{\mathcal{T}[1..r]}$. Note that, in this way, for each found entry in $Occ^{\mathcal{P}}_{\mathcal{T}[1..r]}$, we have at most 2 intervals to perform range minima queries further. So, in total the time spent is $O(|Occ^{\mathcal{P}}_{\mathcal{T}[1..r]}|)$.
On the other hand, for a suffix query, i.e. when $r = n$, we want to compute:

$$Occ^{\mathcal{P}}_{\mathcal{T}[\ell..n]} = \{\mathcal{L}[k] \mid i \leq k \leq j, \mathcal{L}[k] \geq \ell\}.$$

So, in this case, in the above procedure, we just need to perform a Range Maxima (instead of Minima) Query and instead of checking whether $\mathcal{L}[k] \leq r$, we need to check whether $\mathcal{L}[k] \geq \ell$. The query steps are formally stated in Algorithm 2, 3 and 4. In light of the above discussion, it is straightforward to see that the total query time is $O(m + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$. The result of this paper is formally presented in the form of following theorem.

**Theorem 1.** *For Problem PMP/S, we can construct the IDS_PMP/S data structure in $O(n)$ time and space for bounded alphabet and $O(n \log \Sigma)$ time for general alphabet. We can then answer the relevant queries in optimal $O(m + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ time per query.*

## 4 Conclusion

In this paper, we have studied Problem PMP/S, a restricted version of the position restricted pattern matching problem (Problem PRPM) introduced and studied in [24]. In Problem PRPM, the query provides an interval $[\ell..r], 1 \leq \ell \leq r \leq n$ along with the pattern $\mathcal{P}$ and the occurrences of $\mathcal{P}$ in $\mathcal{T}[\ell..r]$ are sought for. In Problem PMP/S, on the other hand, we are interested in those occurrences of the pattern that lies in a suffix or in a prefix of the given text. In other words, in our case, the query interval $[\ell..r]$ is of special form: either $\ell = 1$, i.e. prefix search, or $r = n$, i.e. suffix search. We have presented an efficient data structure, IDS_PMP/S, which is an extension of the classic suffix tree data structure. The time and space complexity of IDS_PMP/S is dominated by that of the suffix tree

and hence is $O(n)$ for bounded alphabet and $O(n \log \Sigma)$ for the general case. The query time we achieve is $O(m + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ time per query, which is optimal. Notably, the (best) algorithm in [24], if applied to Problem PMP/S, gives sub-optimal query time and the corresponding data structure also requires more time and space. One final remark is that, we can use the suffix array instead of suffix tree as well with some standard modifications in the corresponding algorithms to solve Problem PMP/S.

## References


1. A. Amir, A. Butman, M. Crochemore, G. M. Landau, and M. Schaps. Two-dimensional pattern matching with rotations. *Theor. Comput. Sci.*, 314(1-2):173–187, 2004.
2. A. Amir, A. Butman, and M. Lewenstein. Real scaled matching. *Inf. Process. Lett.*, 70(4):185–190, 1999.
3. A. Amir and E. Chencinski. Faster two dimensional scaled matching. In Lewenstein and Valiente [21], pages 200–210.
4. A. Amir, O. Kapah, and D. Tsur. Faster two dimensional pattern matching with rotations. In Sahinalp et al. [30], pages 409–419.
5. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Text indexing and dictionary matching with one error. *J. Algorithms*, 37(2):309–325, 2000.
6. A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *J. Algorithms*, 13(1):2–32, 1992.
7. M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Latin American Theoretical INformatics (LATIN)*, pages 88–94, 2000.
8. A. Butman, R. Eres, and G. M. Landau. Scaled and permuted string matching. *Inf. Process. Lett.*, 92(6):293–297, 2004.
9. H.-L. Chan, W.-K. Hon, and T. W. Lam. Compressed index for a dynamic collection of texts. In Sahinalp et al. [30], pages 445–456.
10. R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In L. Babai, editor, *STOC*, pages 91–100. ACM, 2004.
11. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *STOC*, pages 592–601, 2002.
12. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.
13. P. Ferragina and R. Grossi. Fast incremental text editing. In *SODA*, pages 531–540, 1995.
14. M. Fischer and M. Paterson. String matching and other products. *in Complexity of Computation, R.M. Karp (editor), SIAM AMS Proceedings*, 7:113–125, 1974.
15. K. Fredriksson, G. Navarro, and E. Ukkonen. Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In A. Apostolico and M. Takeda, editors, *CPM*, volume 2373 of *Lecture Notes in Computer Science*, pages 235–248. Springer, 2002.
16. H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Symposium on the Theory of Computing (STOC)*, pages 135–143, 1984.
17. M. Gu, M. Farach, and R. Beigel. An efficient algorithm for dynamic text indexing. In *SODA*, pages 697–704, 1994.

18. D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.
19. D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.
20. V. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
21. M. Lewenstein and G. Valiente, editors. *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*. Springer, 2006.
22. M. G. Maaß and J. Nowak. Text indexing with errors. In A. Apostolico, M. Crochemore, and K. Park, editors, *CPM*, volume 3537 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2005.
23. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In Lewenstein and Valiente [21], pages 306–317.
24. V. Mäkinen and G. Navarro. Position-restricted substring searching. In J. R. Correa, A. Hevia, and M. A. Kiwi, editors, *LATIN*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer, 2006.
25. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
26. G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In R. Giancarlo and D. Sankoff, editors, *CPM*, volume 1848 of *Lecture Notes in Computer Science*, pages 350–363. Springer, 2000.
27. R. Pinter. Efficient string matching with dont care patterns. *In A. Apostolico and Z. Galil (Eds.). Combinatorial algorithms on words, NATO Advanced Science Institute Series F: Computer and System Sciences*, 12:11–29, 1985.
28. M. S. Rahman and C. S. Iliopoulos. Indexing factors with gaps. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, F. Plasil, and M. Bieliková, editors, *SOFSEM*, volume 4362 of *Lecture Notes in Computer Science*, pages 465–474. Springer, 2007.
29. M. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth. Finding patterns with variable length gaps or don't cares. In D. Z. Chen and D. T. Lee, editors, *COCOON*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2006.
30. S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, editors. *Combinatorial Pattern Matching, 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5-7, 2004, Proceedings*, volume 3109 of *Lecture Notes in Computer Science*. Springer, 2004.
31. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.