Indexing with Gaps

Moshe Lewenstein[⋆]

Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel
moshe@cs.biu.ac.il

Abstract. In *Indexing with Gaps* one seeks to index a text to allow pattern queries that allow gaps within the pattern query. Formally a *gapped-pattern* over alphabet Σ is a pattern of the form $p = p_1 g_1 p_2 g_2 \cdots g_\ell p_{\ell+1}$, where $\forall i, p_i \in \Sigma^*$ and each g_i is a gap length $\in N$. Often one considers these patterns with some bound constraints, for example, all gaps are bounded by a gap-bound G.

Near-optimal solutions have, lately, been proposed for the case of one gap only with a predetermined size. More specifically, an indexing solution for patterns of the form $p_1 \cdot g \cdot p_2$, where g is known apriori. In this case the solutions mentioned are preprocessed in $O(n \log^\epsilon n)$ time and $O(n)$ space, where the pattern queries are answered in $O(|p_1| + |p_2|)$, for constant sized alphabets. For the more general case when there is a bound G these results can be easily adapted with a multiplicative factor of $O(G)$ for the preprocessing, i.e. $O(n \log^\epsilon nG)$ preprocessing time and $O(nG)$ preprocessing space. Alas, these solutions do not lend to more than one gap.

In this paper we propose a solution for k gaps one with preprocessing time $O(nG^{2k} \log^k n \log\log n)$ and space of $O(nG^{2k} \log^k n)$ and query time $O(m + 2^k \log\log n)$, where $m = \sum_{i=1} |p_i|$.

1 Introduction

Indexing refers to the preprocessing of data, in our case text, in order to answer subsequent pattern queries. Suffix trees and suffix arrays are two classical data structures that index text. We denote the text $T = t_1 t_2 \cdots t_n$. The queries are then answered online quickly. Query patterns are of the form $p = p_1 p_2 \cdots p_m$.

Pattern matching with wildcards is the problem of finding all appearances of a pattern in a text where the text and pattern are over alphabet $\Sigma \cup \{\phi\}$, where ϕ denotes the wildcard; where a wildcard matches all other characters. Pattern matching with wildcards was introduced and solved efficiently using convolutional methods in [10]. Slightly tighter solutions have been presented in [5,7,13,14].

Naturally the question was whether there were efficient solutions to indexing with wildcards. Initially it seemed that even solving indexing with one mismatch or wildcard is difficult. In [1] an efficient solution was given. A similar result was also proposed in [9]. Both solutions convert the problem to geometric representations and use fast and effective geometric data structures. Lately, a new result [3]

⋆ This research was supported by the Israel Science Foundation (grant no. 1848/04).

was proposed with optimal query times. These were achieved by reducing to a data structure with tighter results. These solutions work very well for the case of one wildcard or mismatch, but do not carry over, efficiently, to a larger number of wildcards or mismatches. In [6] a solution for a larger number of wildcards was devised.

Pattern matching with gaps, a natural problem, has been considered extensively, see [4,8] for an example of one earlier paper and another recent one. The problem is important in Computational Biology and Computational Musicology. It is also an extension of pattern matching with wildcards, i.e. wildcards with a length (or an interval of lengths).

Indexing with gaps was considered in [15]. They present an interesting and detailed algorithm. However, their running time depends on parameters of the number of matches between the blocks of the non-gaps. In [18] the same problem is considered, only here only one gap in the pattern is assumed (and some other constraints). The problem was also considered in [3,12]. In all three one gap of a predetermined length was considered, i.e. the pattern is of the form $p = p_1\phi^g p_2$, where g is an input to the preprocessing. In the first paper [18] also the lengths of P_1 and P_2 were predetermined. While, the results of [18] were very initial the results in [12,3] were more sophisticated and followed along the lines of the one mismatch/wildcard solutions. Each reduces the problem to geometric data structural problems in the preprocessing and then, in the query stage, performs an appropriate query in the geometric data structure. Actually the one mismatch problem is more difficult than one wildcard (because with a mismatch we do not know where mismatch occurs). The one wildcard extends naturally to one g-length gap since the construction is basically the same.

The reduction in all these problems is to two dimensional range queries. The idea is to construct separate suffix trees one for the text T, denoted S_T, and one for it's reverse T^R, denoted S_{T^R}. The central idea is the same in all solutions. Consider the case of one wildcard then one can view any match (in the text) as having a pivot at the location i where the wildcard occurs. Therefore, if the pattern is $P_1\phi P_2$ then we can match P_2 in S_T and P_1^R in S_{T^R}. P_1^R starts at one before the pivot and P_2 starts one after the pivot. So, we construct on a 2D range grid a point for each (potential) pivot i. Specifically, we have a lexicographic ordering L_1 of the suffixes (the ordering of the suffixes at the bottom of the suffix tree = ordering of the suffix array) of T and a lexicographic ordering L_2 of the suffixes of T^R (which are prefixes of T). The point of pivot i is (x, y), where x is the location of suffix $i+1$ in L_1 and y is the location in L_2 of the prefix of T ending at $i-1$. Now, for query $P_1\phi P_2$ one walks down S_T with P_2 and down S_{T^R} and down S_{T^R} with P_1^R. In each we reach a node which defines a range of leaves. We need to find all the pivots which have a point in the pair of ranges. This translates into a two dimensional range query where all points in a rectangle need to be found. There are different data structures for this problem used in the different solutions. For g-length gaps all that is changed is that the pivot point is now constructed for a pair $i-1$ and $i+g$. It is easy to see that the solution does not carry over to more than one wildcard (or gap).

1.1 Our Results

In this paper we are interested in generalizing indexing with gaps in two senses. First we remove the constraint that the length of the gap g is known a-priori and rather assume that there is a bound G for which the gaps are no larger than it. Second, we allow more than one gap. Let us give the most general definition of pattern matching with gaps. We will later add some constraints. This definition follows along the lines of [4].

Given integers a and b, $0 \le a \le b$, a *variable length gap* $g\{a,b\}$ is an arbitrary string over alphabet Σ of length between a and b, both inclusive. A *variable length gap pattern* (for short VLG pattern) P is the concatenation of a sequence of strings and variable length gaps, that is, P is of the form

$$p = p_1 \cdot g\{a_1, b_1\} \cdot p_2 \cdot g\{a_2, b_2\} \cdots g\{a_k, b_k\} \cdot p_{k+1}.$$

A VLG pattern P matches a substring S of T iff $S = P_1 \cdot g_1 \cdots \cdot g_k \cdot P_{k+1}$, where g_i is any string of length between a_i and b_i, $i = 1, \cdots, k$. We say that P appears at location i of T if there is a substring S of T starting at i which P matches. We will classify VLG patterns into 3 classes; general VLG patterns, right-end VLG patterns, i.e. where all $a_i = 1$, and (simple) gapped-patterns, i.e. non-variable or $a_i = b_i$. We also say that a VLG pattern is G-bounded if $b_i \le G$ for all i. We say that a VLG pattern is k-gap-count-bounded if it contains at most k gaps.

For the sake of simplicity we will focus only on gapped-patterns throughout this paper and leave the handling of general VLG patterns to the journal version.

The *Indexing with Gaps problem* is defined as follows.

Preprocessing Input: Text $T = t_1 t_2 \cdots t_n$, G, and k.
Preprocessing Output: Data structure supporting G-bounded, k-gap-count-bounded gapped-pattern queries.
Query Input: A G-bounded, k-gap-count-bounded gapped-pattern P.
Query Output: All locations i in T where P appears.

To solve the first problem of removing the a-priori constraint of length g, we will use the results of [3] and update them accordingly. An $O(G)$ factor in the preprocessing seems unavoidable. On the other hand, an $O(G)$ factor is sufficient for extending the problem to G-bounded queries for the case of gapped-patterns. For the more general case of G-bounded general VLG pattern queries, where only one gap is allowed one can solve the problem with $O(nG^2)$ space with the same time bounds or with $O(nG \log G)$ overhead and $O(\log G)$ query time.

If we want to allow more than one gap we will need a different data structure than those that reduce to geometric data structures that we have seen before. We will follow along similar lines to the result presented in [6]. However, things get complicated when extending wildcards to G-length gaps. Unlike, the geometric solutions the transfer is not trivial. We will need to produce new updates.

The rest of our paper is organized as follows: in Sect. 2, we give some preliminaries and problem definitions. In Sect. 3 we show our methods for allowing multiple gaps in a few phases.

2 Problem Definitions and Preliminaries

2.1 Preliminary Definitions and Notations

Given a string S, $|S|$ is the length of S. Throughout this paper we denote $n = |S|$. An integer i is a *location* or a *position* in S if $i = 1, \ldots, |S|$. The substring $S[i\,..\,j]$ of S, for any two positions $i \leq j$, is the substring of S that begins at index i and ends at index j. Concatenation is denoted by juxtaposition. The *suffix* S_i of S is the substring $S[i\,..\,n]$.

The *suffix tree* [16,17,20,21] of a string S, denoted $\mathrm{ST}(S)$, is a compact trie of all the suffixes of $S\$$ (i.e. S concatenated with a delimiter symbol $\$ \notin \Sigma$). Each of its edges is labeled with a substring of S (actually, a representation of it, e.g. the start location and its length). The "compact" property is achieved by contracting nodes having a single child. The children of every node are sorted in the lexicographical order of the substrings on the edges leading to them. Consequently, each leaf of the suffix tree represents a suffix of S, and the leaves are sorted from left to right in the lexicographical order of the suffixes that they represent.

$\mathrm{ST}(S)$ requires $O(n)$ space. Algorithms for the construction of a suffix tree enable $O(n)$ preprocessing time when $|\Sigma|$ is constant (where Σ is the alphabet set), and $O(n \log \min(n, |\Sigma|))$ time when $|\Sigma|$ is not. In fact, the suffix tree can be constructed in linear time even for alphabets drawn from a polynomially-sized range, see [16].

LCA queries. An LCA query $lca(u, v)$ is given two nodes u, v in a tree T and reports the lowest common ancestor w of u and v in T. Once w is known, its height in the tree can also be determined. Often, data structures for constant time LCA queries are used with suffix trees, as will be the case here as well. Data structures for answering LCA queries in $O(1)$ time can be built in linear time [11,19]. These data structures also allow the reporting in $O(1)$ time of the edges exiting w on the paths to u and v. In addition, they yield the length of the longest common prefix of the suffixes s_u and s_v associated with u and v, again in $O(1)$ time.

Measured ancestor structure. We will also need a data structure for answering the following query on an n-node compressed trie in $O(\log \log n)$ time: Given a leaf u and a distance h, report the location at which the prefix of s_u of length $|s_u| - h$ ends, i.e. the location distance h above u, where edges are deemed to have length equal to their labels. We call this the measured ancestor structure. Again, such data structures can be built in linear time [2].

Centroid path decomposition. Our construction uses *centroid paths* and *centroid path decompositions*. For our setting, we define the *centroid path* of a tree T to be the path starting at T's root, which at each node v on the path branches to

v's "largest" child, with ties broken arbitrarily; the size of a node is simply the number of leaves in the subtree rooted at that node. In a centroid path decomposition, we decompose each off-path subtree of the centroid path recursively.

The weight of a node on a centroid path is defined to be the number of leaves in its off-path subtrees. In our applications, for node v on a centroid path with off-path child u, it is convenient to include edge (v, u) in the off-path subtree T_u incorporating node u. We will also say that T_u *hangs from* node v.

The following property of a centroid path decomposition of a tree is well known.

Property 1. Let T be an n-node tree with a centroid path decomposition. Let v be a node of T. The path from the root of T to v traverses at most $\log n$ centroid paths.

3 G-Bounded Queries with k Gaps

Given the text T of size $O(n)$ and a suffix tree $S(T)$, a brute-force search for a gapped-pattern $p = p_1 \cdot g_1 \cdot p_2 \cdot g_2 \cdots g_k \cdot p_{k+1}$, can proceed as follows: descend from the root of $S(T)$, find the path that exactly matches p_1. We refer to this path as the *first tier* path. Now, the search can allow for a g_1 gap by searching all the paths hanging off the first tier path at that point in the path (these are the *second tier* paths), taking the g_1 gap at the top of each of these paths. Recall that $|\Sigma|$ edges may hang off a single node of the suffix tree and, therefore, $|\Sigma|^{g_1}$ paths of length g_1 may hang off of v; this implies that we can search for a pattern containing a single gap (of length g_1) in $O(|p_1| + |\Sigma|^{g_1}|p_2|)$ time. It is not difficult to see that a simple extension of this algorithm can handle multiple gaps (for $k = 2$ we must search the *third tier* paths hanging off the second tier paths, etc.), yielding a run time of $O(|\Sigma|^{g_1+\cdots+g_k} m)$, where $m = \sum_{i=1}^{\ell} |p_i|$. This is accomplished without any modifications to the trie, and therefore only $O(n)$ space is used.

3.1 Speeding Up the Search

A first step towards improving the costly run time of the brute force method involves a tradeoff between the inefficient run time of the algorithm and the optimal space requirement of $O(n)$. Specifically, we remove the $|\Sigma|^{g_1+\cdots+g_k}$ term from the runtime, but require $\Theta(n^{k+1}G^k)$ space.

At each node of the trie, we wish to anticipate taking a single gap during the search. To this end, before having knowledge of the gapped-pattern p we preprocess the trie. The gap to be taken can be of any length between 1 and G. So, at every node we create G *gap subtrees*, one for each possible gap length, which will be searched if a gap is taken starting at that node. Note, that the traversal in a tier may end in the middle of an edge. However, in this case it is sufficient to "slide" to the node under it and adapt, g, the length of the gap, i.e. if there are x characters on the edge from the location the traversal reached on the edge until the node under it, we move to the node under it and continue the search with a gap of $g - x$.

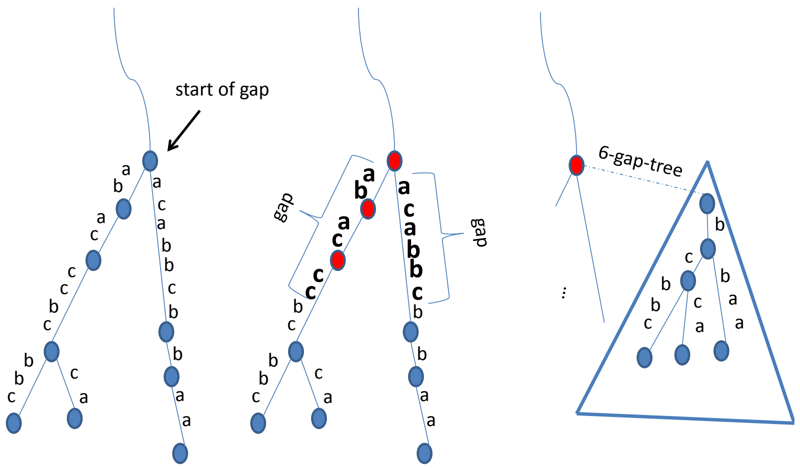

Fig. 1. (a) node v with two suffixes in subtree, (b) 6-length gaps marked, (c) 6-gap-tree created

A gap subtree for length r, $1 \leq r \leq G$, which we call a *gap-r-tree*, at node v contains a merge of all the subtrees at the end of the r-length paths starting at v, see Figure 1. Another way to envision the gap-r-tree is to think of all the leaves of the suffix tree in the subtree of v. Each corresponds to a suffix, say S_i. If we truncate the h-length prefix of S_i we have the suffix S_{i+h}. So, if the set of leaves in the subtree of v corresponds to suffixes $\{S_{i_1}, \cdots, S_{i_q}\}$ and the string associated with node v (the locus of the root-to-v node) is of length d then the gap-r-tree at node v is a compressed trie of the suffixes $\{S_{i_1+d+r}, \cdots, S_{i_q+d+r}\}$ (for those that satisfy $i_j + d + r \leq n$). To create the gap-r-tree at node v one traverses the suffixes and updates their index and uses the lexicographic ordering to resort them. The lexicographic ordering is obtainable from the suffix tree of the original tree (or one may use any other suffix sorting algorithm that one desires). Once they are resorted one uses standard techniques to construct the compressed trie over these updated suffixes giving the desired. The resorting can be done in $O(q \log\log n)$ time, where q is the number of elements in the subtree of v. The sorting is done with a van-Emde Boas tree or with a y-fast-trie. This can be done because all the "new" suffixes are really just suffixes from the original suffix tree and they have been enumerated from $1, \cdots, n$ according to their lexicographic ordering. Hence, the "universe" of the q elements that are sorted is of size n.

The size of the modified overall trie is $\Theta(n^2 G)$. This is because in the original suffix tree, every leaf u representing a suffix, may have $O(n)$ ancestors and each have G gap trees hanging off the node containing u.

A search for a gapped-pattern with one gap, i.e. $p = p_1 g_1 p_2$, on the new trie descends from the root, and as before finds the path for p_1. This is the first tier path. Now, the search will continue in the g_1-gap tree with the pattern p_2. This search takes time $O(|p_1| + |p_2|)$. It is not difficult to see that a simple extension of this algorithm can handle multiple gaps – for $k = 2$ we must create secondary gap subtrees for each node of each primary gap tree, etc. – yielding a run time of $O(m)$, where $m = \sum_{i=1}^{\ell} |p_i|$ with space $O(n^{k+1}G^k)$.

3.2 Better Tradeoffs

Although the aforementioned technique improves the run time significantly, the space requirement is now problematic. However, a variant of this technique yields an $O(nG^{2k}\log^k n)$ structure that supports gapped-pattern queries in $O(2^k m)$ time.

As before, before having knowledge of p we preprocess the trie and create a gap subtree for each node. However, there is a slight twist here. We will consider the centroid partition of the suffix tree. The gap subtrees created at a node v will not contain all (truncated) suffixes in the subtree of v as before, but rather will contain only those whose v-to-suffix path leave the centroid path within the gap size, see Figure 2.

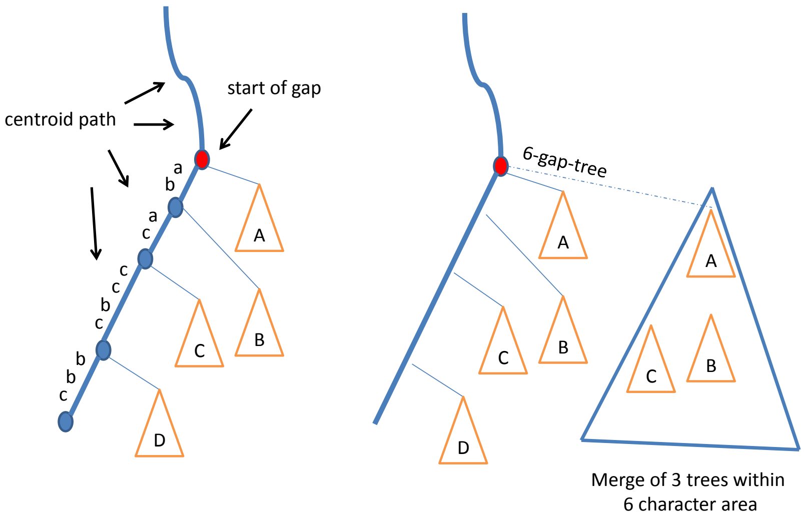

Fig. 2. (a) 1 centroid path with suffix tree subtrees hanging off, (b) merge of appropriate subtrees (off centroid-path trees) for 6-gap-tree

If during the execution of a search a gap is taken at the node, then both the gap tree and the subtree where the gap is along the centroid path must be searched. That is, taking a gap will spawn two searches – better than the $|\Sigma|^g$ searches spawned in the brute-force algorithm, but less efficient than the single search required in the search speedup.

The benefit of this tradeoff lies in reducing the size of the structure, which is now $O(nG^2 \log n)$. This bound follows easily when one considers each leaf in the original trie: Each gap tree that contains this leaf is associated with a distinct ancestor of this leaf, and further each such ancestor lies on a centroid path from which the subtree containing the leaf diverges within at most G levels down. As a leaf may have at most $\lfloor \log_2 n \rfloor$ centroid paths on the path from root to leaf, a leaf may be found in the gap subtrees of at most $G\lfloor \log_2 n \rfloor$ ancestors in the modified trie. Each such ancestor has at most G such subtrees.

It is not difficult to see that, as before, a simple extension of this algorithm can handle multiple gaps – we create k-ary gap subtrees for each node of each $(k-1)$-ary gap subtree. This yields a run time for the query of $O(2^k m)$ with space $O(nG^{2k} \log^k n)$.

In the next subsection we will see how to reduce the query time to $O(2^k \log\log n + m)$.

3.3 Final Speedup of the Query

The multiplicative factor of $O(m)$ follows from the traversals on the p_i's and the 2^k and the $\log^k n$ factors are for the number of subtrees that we need to traverse within.

A circumvention to the traversal is as follows. Recall that each of these subtrees is actually a trie over a collection of suffixes of the original suffix tree. We have also assumed that these suffixes have been sorted (lexicographically) either by the suffix tree construction or by some other suffix sorting method. Therefore, each trie is actually a collection of numbers (the lexicographic ranks). Now, when we first see the gapped pattern for each i we traverse with p_i from the root of the suffix tree. When we reach the end of p_i we are at a node (or just above a node) u. The leftmost leaf and the rightmost leaf in the subtree of u represent suffixes with p_i as a prefix of this suffix. So, they, actually their lexicographic ranks, may represent p_i.

So, now when we reach a gap subtree instead of traversing it from top with p_i we will do a predecessor query with the lexicographic rank of p_i as the query and the lexicographic ranks of the suffixes of the subtree as the data. Once we have found the predecessor and successor we will be able to find the node in the subtree representing the end of the p_i search by applying an LCA query to the newly found predecessor and successor. If there is only a predecessor (or only a successor) we may use a measured ancestor query. This predecessor/successor queries and measured ancestor queries can be implemented in $O(\log\log n)$ time. This yields:

Theorem 1. *Let T be a text of size n. One can build an indexing scheme of size $O(nG^{2k} \log^k n)$ so that one can answer gapped queries bounded by gap-bound G with k gaps in time $O(m + 2^k \log\log n)$.*

References

1. Amir, A., Keselman, D., Landau, G., Lewenstein, N., Lewenstein, M., Rodeh, M.: Text indexing and dictionary matching with one error. J. of Algorithms 37(2), 309–325 (2000)
2. Amir, A., Landau, G., Lewenstein, M., Sokol, D.: Dynamic pattern, static text matching. ACM Transactions on Algorithms 3(2) (2007)

3. Bille, P., Gørtz, I.L.: Substring range reporting. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 299–308. Springer, Heidelberg (to apppear, 2011)
4. Bille, P., Li Gørtz, I., Vildhøj, H.W., Wind, D.K.: String matching with variable length gaps. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 385–394. Springer, Heidelberg (2010)
5. Clifford, P., Clifford, R.: Self-normalised distance with don't cares. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 63–70. Springer, Heidelberg (2007)
6. Cole, R., Gottlieb, L., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: Proceedings of the Symposium On Theory of Computing (STOC), pp. 91–100 (2004)
7. Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching. In: Proceedings of the Symposium On Theory of Computing (STOC), pp. 592–601 (2002)
8. Crochemore, M., Iliopoulos, C., Makris, C., Rytter, W., Tsakalidis, A., Tsichlas, K.: Approximate string matching with gaps. Nordic J. of Computing 9(1), 54–65 (2002)
9. Ferragina, P., Muthukrishnan, S., de Berg, M.: Multi-method dispatching: A geometric approach with applications to string matching problems. In: Proceedings of the Symposium on Theory of Computing (STOC), pp. 483–491 (1999)
10. Fischer, M., Paterson, M.: String matching and other products. In: Karp, R.M. (ed.) Complexity of Computation, SIAM-AMS Proceedings, vol. 7, pp. 113–125 (1974)
11. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13(2), 338–355 (1984)
12. Iliopoulos, C., Rahman, M.: Indexing factors with gaps. Algorithmica 55(1), 60–70 (2008)
13. Indyk, P.: Faster algorithms for string matching problems: Matching the convolution bound. In: Proceedings of the Symposium on Foundations of Computer Science (FOCS), pp. 166–173 (1998)
14. Kalai, A.: Efficient pattern matching with don't cares. In: Proceedings of the Symposium on Discrete Algorithms (SODA), pp. 655–656 (2002)
15. Lam, T.-W., Sung, W.-K., Tam, S.-L., Yiu, S.-M.: Space efficient indexes for string matching with don't cares. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 846–857. Springer, Heidelberg (2007)
16. Farach-Colton, S.M.M., Ferragina, P.: On the sorting-complexity of suffix tree construction. J. ACM 47(1), 987–1011 (2000)
17. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
18. Peterlongo, M.S.P., Allali, J.: Indexing gapped-factors using a tree. Int. J. Found. Comput. Sci. 19(1), 71–87 (2008)
19. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplifications and parallelization. SIAM Journal on Computing 17(6), 1253–1262 (1988)
20. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
21. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory, pp. 1–11. IEEE, Los Alamitos (1973)