The Gapped-Factor Tree

Pierre Peterlongo[1], Julien Allali[1], and Marie-France Sagot[2,3] ⋆

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, France
pierre.peterlongo@univ-mlv.fr
[2] INRIA Rhône-Alpes and Laboratoire de Biométrie et Biologie Évolutive, UMR 5558, Université Claude Bernard, Lyon, France
[3] King's College, London, UK

Abstract. We present a data structure to index a specific kind of factors, that is of substrings, called *gapped-factors*. A gapped-factor is a factor containing a gap that is ignored during the indexation. The data structure presented is based on the suffix tree and indexes all the gapped-factors of a text with a fixed size of gap, and only those. The construction of this data structure is done online in $O(n \times |\Sigma|)$ time and space, with n the length of the text and $|\Sigma|$ the size of the alphabet. Such a data structure may play an important role in some pattern matching and motif inference problems, for instance in text filtration.

Keywords: suffix tree, k-factor factor tree, string index, gapped-factor, gapped-factor tree

1 Introduction

The indexation and extraction of repeated short words (called *k-factors*[4] for words of length k) has become a widely used technique in many text algorithmic problems. One can mention their use in, for instance, FASTA [17] and BLAST [2, 3]. Indeed, many algorithms for efficiently computing string matches [10, 24, 29] or alignments [5, 4, 9, 12, 16, 18, 21] use k-factors. In particular, filtration algorithms that have been created for quickly discarding large portions of the input before applying a more expensive algorithm on the remaining data are often based on the identification of such short repeated words [6–8, 15, 25, 26, 28].

Among the exact filtration algorithms (exact in the sense that they discard only portions of the text that can not be part of the final solution sought), some consider k-factors composed of non consecutive letters [7, 8, 15, 26], or sets of k-factors [6, 25, 28]. Both present advantages for filtering purposes in comparison with single k-factors with no letters skipped as shown in [7, 14, 15].

In order to efficiently use such k-factors, one needs data structures to index them. Depending on the kind of k-factor adopted, different types of data structures may be considered. For instance, sets of k-factors may be indexed in a hash table or using a labelling technique as proposed in [13]. In this paper, we introduce a data structure designed for the indexation of sub-words composed of a k-factor, a gap of length d not taken into account during the indexation and a k'-factor. Such a sub-word is called a *gapped-factor* as it contains a unique gap.

The new data structure is an adaptation of the suffix tree [20]. More precisely, the construction we describe in this paper is an adaptation of the construction of a

⋆ Supported by the ACI Nouvelles Interfaces des Mathématiques *π-vert* project of the French Ministry of Research, the ARC *BIN* project from the INRIA, and the ANR project *REGLIS*.

[4] Another currently used term for designing k-factors is q-grams

k-factor tree [1], which itself is an extension of the Ukkonen construction of a suffix tree [31]. A k-factor tree is a tree indexing all k-factors of a text.

As indicated in Section 5, the new data structure, called a *gapped-factor tree*, allows to extract in linear time all the repeated gapped-factors of a text or of a set of texts. Furthermore, it offers the possibility to obtain in time $O(k + k')$ the list of all the positions of a gapped-factor.

The paper is organised as follows. In Section 2, we provide the context and some definitions about text and trees. In Section 3, we formally introduce gapped-factors and the gapped-factor tree. In Section 4, we present the algorithm to construct a gapped-factor tree for indexing the gapped-factors of a text after recalling the Ukkonen construction of a suffix tree and the Allali construction of a k-factor tree. We end by indicating two basic uses of gapped-factor trees.

2 Preliminaries

A *text*, also called a *string*, is a sequence of zero or more symbols from an alphabet Σ. A text t of length n is denoted by $t[0, n-1] = t_0 t_1 \dots t_{n-1}$, where $t_i \in \Sigma$ for $0 \leq i < n$. The length of t is denoted by $|t|$. A string w is a *factor* of t if $t = uwv$ for $u, v \in \Sigma^*$; in this case, the string w occurs at position $|u|$ in the string t. A k-factor denotes a factor of length k. If $t = uv$ for $u,\ v\ \in \Sigma^*$ then v is called a *suffix* of t. A suffix starting at position i in t is denoted by $t_{i\dots}$.

A tree is a data structure composed of **nodes** connected together by **edges**. Except for a special node called the **root**, each node has exactly one **father**. Nodes with no children are called the **leaves** while all other nodes are called the **internal nodes** of the tree. An internal node having at least two children is called a **branching node**.

We call the **depth** of a node $\mathcal{N}$ the sum of the lengths of the edges that need to be traversed from the root of the tree to reach $\mathcal{N}$. By definition, the depth of the root is thus 0.

Nodes and edges may be labelled. For instance, in Figure 1, edges are labelled with letters from a given alphabet.

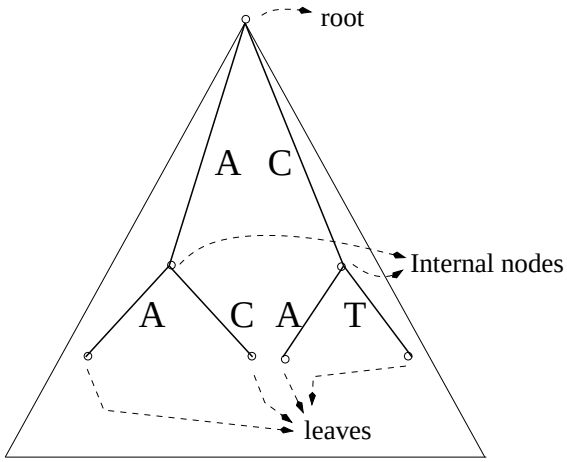

Figure 1. Example of a tree labelled with letters from a given alphabet. Reading all paths from the root to the leaves, leads to the strings AA, AC, CA and CT.

Let $\mathcal{N}$ be a node of a tree, we denote by $path(\mathcal{N})$ the text corresponding to the concatenation of the letters from a given alphabet labelling the edges from the root to $\mathcal{N}$.

For instance, if $\mathcal{N}_0$ denotes the leftmost leaf of the tree presented in Figure 1, $path(\mathcal{N}_0) = AA$.

The suffix trie of a text t is a tree with edges labelled with elements of Σ. For each factor of t, there exists a node N such that $path(\mathcal{N})$ is equal to that factor. If t has an ending symbol, all nodes $\mathcal{N}$ for which the path from the root spells a suffix of t are leaves.

The *implicit* suffix tree of t is a tree with edges labelled by non-empty elements of Σ^*. The suffix tree is a compressed version of the suffix trie. Each internal node $\mathcal{N}$ of the suffix trie that has only one child is deleted and its two adjacent edges are replaced by an edge that goes from the father of $\mathcal{N}$ to its child. The label of the new edge is equal to the concatenation of the label of the edge going from the father of $\mathcal{N}$ to $\mathcal{N}$ and of the label of the edge from $\mathcal{N}$ to its child. This tree is called implicit because not all suffixes of t lead to a leaf. The true suffix tree is obtained when a special ending symbol \$ not in Σ is added at the end of t. A suffix tree indexes all the $|t|$ suffixes of a text t.

3 Gapped-factor tree

A gapped-factor tree indexes gapped-factors that are defined as follows:

Definition 1 (Gapped-factor). *A* **gapped-factor** *is a concatenation of a factor of length k, a gap of length d and another factor of length k'. A gapped-factor occurring at position i in a text t is $t[i, i+k-1].t[i+k+d, i+k+d+k'-1]$. Such a gapped-factor is called a (k-d-k')-gapped-factor.*

An example of a (2-1-3)-gapped-factor is given in Figure 2.

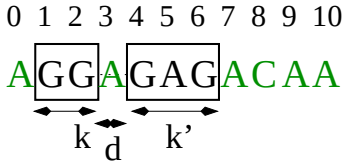

Figure 2. Example of a (2-1-3)-gapped-factor. The first factor length is $k = 2$, the gap is of length $d = 1$ and the second factor has a length $k' = 3$. It occurs at position 1 in the text. With these parameters, the content of the gapped-factor occurring at position 1 is *GGGAG* composed by *GG* and *GAG*.

We propose a new data structure, called a *gapped-factor tree*, to index all the (k-d-k')-gapped-factors of a text or of a set of texts. This is a modification of the suffix tree [20] data structure. The gapped-factor tree takes into account the gap of length d of the gapped-factors it indexes. This means that the tree contains a region up to which the k-factors are indexed as in a classical suffix tree, while below this region the second factors (of length k') of the (k-d-k')-gapped-factors starting with the same k-factor start from the same node. This region is called the *invisible region*.

An intuitive idea of such a data structure is given in Figure 3.

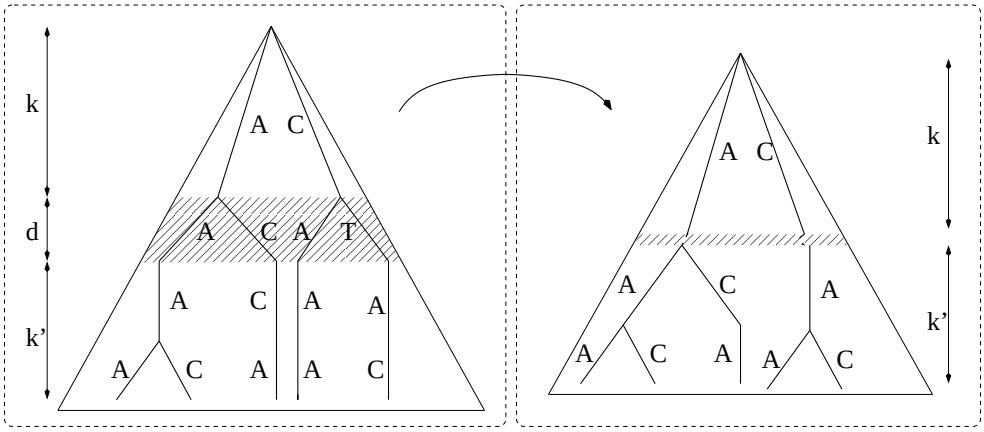

Figure 3. An intuitive view of a gapped-factor tree. Even if this is not the way the gapped-factor tree is constructed, a gapped-factor tree can be seen as a truncated suffix tree where a part has been removed, provoking merges in the lower part of the tree.

Definition 2 (Path in a Gapped-Factor Tree). *Let w be a $(k\text{-}d\text{-}k')$-gapped-factor starting at position $i < |t| - k - d - k'$ that is indexed in such a tree. Let $\mathcal{N}$ be the node at depth $z \leq k + k'$ corresponding to this $(k\text{-}d\text{-}k')$-gapped-factor. Then:*

$$path(\mathcal{N}) = \begin{cases} t[i, i+z-1] & \text{if } z \leq k \\ t[i, i+k-1].t[i+k+d, i+d+z-1] & \text{otherwise} \end{cases}$$

An example of gapped-factor tree and of a path in such a tree is presented in Figure 4.

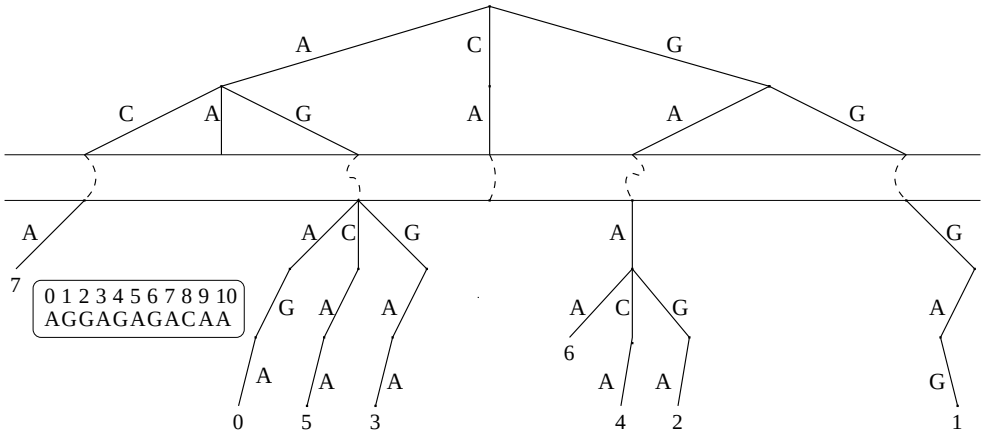

Figure 4. Example of gapped-factor tree. The input sequence is *AGGAGAGACAA*. The dashed lines correspond to the invisible region of the tree. In this case, the gapped factors indexed are (2-1-3)-gapped-factors. The information attached to one of the leaves corresponds to the starting positions of a gapped-factor in the text.

In the next section, we present the algorithm which performs the online construction of a gapped-suffix tree.

4 Construction

The algorithm for constructing a gapped-factor tree is an extension of the algorithm for constructing a k-factor tree [1], which is itself an extension of the suffix tree construction algorithm due to Ukkonen[31]. Therefore, in the following, we start by presenting the construction of a suffix tree, then the one of a k-factor tree, and finally we describe the construction of a gapped-factor tree.

4.1 Ukkonen construction of the suffix tree

To present the Ukkonen algorithm, we follow the description given in [11]. This algorithm constructs a full suffix tree of a text t in $O(|t|)$ time and space. An example of a suffix tree is given in the Figure 5.

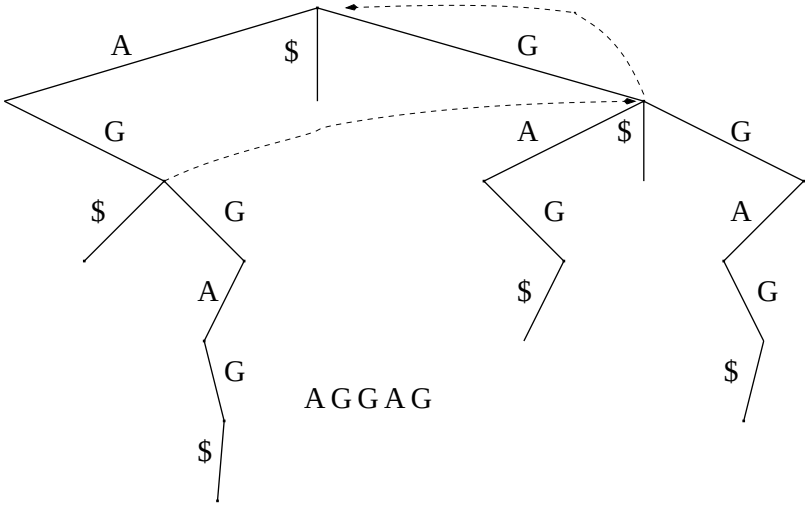

Figure 5. Example of a suffix tree for the text $AGGAG\$$. The dashed lines represent the suffix links.

The algorithm is divided into $|t|$ phases. The i^{th} phase (for $0 \leq i < |t|$) consists in the insertion of all the $i+1$ suffixes of $t[0,i]$ into the tree. The naive approach divides each phase i into $i+1$ steps, one step j ($0 \leq j < i$) consisting in the insertion of the suffix $t[j,i]$ into the tree. This naive version of the construction algorithm is presented in Algorithm 9. Clearly this algorithm is in $O(|t|^3)$.

Algorithm 9 Naive suffix tree construction algorithm

Require: A text t
Ensure: The suffix tree $ST(t)$ of t
1: **for** i from 0 to $|t|-1$ **do**
2: **for** j from 0 to i **do**
3: Add($ST(t)$,$t[j,i]$)
4: **end for**
5: **end for**

The Ukkonen algorithm uses three *tricks* in order to reduce the time complexity to $O(|t|)$.

Before we present those three tricks, we describe the encoding of a suffix tree. The suffix tree created by this algorithm does not store the text: each node $\mathcal{N}$ contains a couple of integers (s, e) corresponding to the starting and ending positions of the factor in the text that led to the creation of the node itself. In the following, we denote by $\mathcal{N}_{s,e}$ such a node. Thus, by definition, in the suffix tree of a text t, $path(\mathcal{N}_{s,e})$ is equal to $t[s, e]$.

The Ukkonen algorithm uses suffix links. A *suffix link* is an oriented link between two branching nodes of a suffix tree. Given a node $\mathcal{N}_{s,e}$, its suffix link is denoted by $S_l(\mathcal{N}_{s,e})$ and the node pointed by $S_l(\mathcal{N}_{s,e})$ is denoted by $S_n(\mathcal{N}_{s,e})$. In this case, $path\left(S_n(\mathcal{N}_{s,e})\right) = path(\mathcal{N}_{s,e})\big[1, |path(\mathcal{N}_{s,e})|\big]$. For instance, if $path(\mathcal{N}_{s,e}) = AGGT$, then, $path(S_n(\mathcal{N}_{s,e})) = GGT$.

In Figure 5, the suffix links are represented by dashed lines.

We present the three ideas leading to a linear time complexity for constructing a suffix tree for the text t.

1. Let us assume that the suffix tree is constructed for $t[0, i-1]$. During the i^{th} phase, all the leaves have to be lengthened by one in order to take the character t_i into account. In other terms, the ending integer e of each leaf has to be incremented by one. Since by definition, all leaves have the same ending integer, the latter can be coded by a global variable that is incremented by one at each phase of the Ukkonen algorithm. This global variable is equal to i during phase i. Thus, the extension of the leaves is implicit and done in constant time.
2. (a) *Fast Insertion*: during the i^{th} phase, let $\mathcal{N}_{s,e}$ be the last branching node reached during the insertion of $t[j, i]$. By construction this node contains a suffix link. In this case, $t[j, i] = path(\mathcal{N}_{s,e}).w.\sigma$ where $w \in \Sigma^*$ and $\sigma \in \Sigma$. In order to insert $t[j+1, i]$, w (which is necessarily already in the tree) is read from $S_n(\mathcal{N}_{s,e})$ and σ is added if needed.
   To avoid having to read all the letters of w from $S_n(\mathcal{N}_{s,e})$, the following trick is used. At each branching node met during the reading of w, an edge is chosen depending on the current letter in w. Once the edge is identified, the node pointed by this edge is reached and we advance in the reading of w by the number of letters in the edge. The process is repeated while w is not totally read. Thus the complexity of the reading of w is related to the number of nodes traversed and not to $|w|$.
   If σ is added, a branching node is created. The suffix link of such a node points to the last branching node met during the next insertion (it can be a created one).
   The pseudo-code of this algorithm is given in appendix in Figure 10.
   (b) During phase i, all the suffixes of $t[0, i]$ have to be inserted. Yet if during the insertion of $t[j, i]$, this word is already in the tree, then, by definition, all the words $\{t[k, i], k \in [j, i]\}$ are already in the tree as well. In this case, the i^{th} phase stops here. Similarly, the $(i + 1)^{th}$ phase can start inserting $t[j, i + 1]$: with the implicit extension of the leaves, the factors $\{t[k, i + 1], k \in [1, j - 1]\}$ are already in the tree.

A pseudo-code of this construction algorithm is given in appendix in Algorithm A.

Each phase of the algorithm is not done in constant time. However the amortised construction time is linear with respect to the input text length. The demonstration of this complexity is given in [11]. It consists in bounding the overall number of nodes traversed during all insertions.

4.2 Construction algorithm of a k-factor tree

The k-factor tree, also called truncated suffix tree, has been presented in [22] and [1]. A k-factor tree is a suffix tree cut such that each word spelt from the root to a leaf has a length bounded by k. An example of k-factor tree is given in Figure 6. This structure finds applications in various areas such as data compression [22, 23] where the indexation is made over a sliding window, or string matching and computational biology [19, 27, 30] where the length of the motifs searched for in the text is bounded.

The linear time construction algorithm we describe here is based on the Ukkonen suffix tree construction algorithm. For further details on implementation and proof of validity, the reader is referred to [1].

This algorithm is divided in two parts:

1. Build the suffix tree for $t[0, k-2]$.
2. Add in $|t| - k + 1$ phases the suffixes of $t[i-k, i]$ for i from $k-1$ to $|t|-1$.

The first part is achieved using the Ukkonen algorithm. During this part, the leaves created are added to a queue called $queue_{leaf}$.

In the second part, we have to modify the Ukkonen algorithm so that:

- for each phase i, we start by inserting $t[j, i]$ with j not smaller than $i - k + 1$;
- implicit leaf extensions are stopped when the length k is reached for the path of a leaf.

To do this last point, we use the queue $queue_{leaf}$.

During the whole construction, each leaf created is added at the end of $queue_{leaf}$. In the second part, for each phase i, there are two possibilities: either $queue_{leaf}$ is empty or not.

Suppose $queue_{leaf}$ contains at least one leaf. Let $\mathcal{L}_{s,e}$ denote a leaf starting position s and ending position e. We then have $queue_{leaf} = \mathcal{L}^1_{s^1,e} \dots \mathcal{L}^p_{s^p,e}$. We start by fixing the end position of $\mathcal{L}^1_{s^1,e}$ to i, that is $\mathcal{L}^1_{s^1,e}$ becomes $\mathcal{L}^1_{s^1,i}$. Indeed, we know that $path(\mathcal{L}^1_{s^1,i})$ has a length of k.

Suppose we are in phase $i = k-1$. Then $queue_{leaf}$ contains at least one leaf which corresponds to the one created during the insertion of $t[0]$ in the tree (first insertion of the first phase). This leaf is $\mathcal{L}^1_{0,e}$. In phase $i = k$, the leaf is now $\mathcal{L}^1_{0,k-1}$, so its length is equal to k. If there is another leaf in the queue, it corresponds to $\mathcal{L}^1_{1,e}$ and it is clear that its length will be equal to k at the next phase. And so on, as the leaves $\mathcal{L}_{s,e}$ are created with s incremented by one between two leaves.

Once the leaf at the beginning of $queue_{leaf}$ is fixed, we apply again the Ukkonen algorithm from the last leaf in $queue_{leaf}$ (the last created which can be the one we have just fixed). At the end of phase (*i.e.* no leaf created during the last insertion), we remove the leaf at the head of $queue_{leaf}$.

We describe now the case when there is no leaf in the queue. Suppose there were a leaf in the queue at the previous phase $i-1$. By fixing the end value of this leaf, we have fixed the leaf corresponding to $t[i-k, i-1]$. Then we started by inserting $t[i-k+1, i-1]$ in the tree. This insertion did not create a leaf ($queue_{leaf}$ is empty in phase i) and lead to a position p in the tree that corresponds to the spelling of $t[i-k+1, i-1]$. In the current phase i, since $queue_{leaf}$ is empty, we have to start by inserting $t[i-k+1, i]$ in the tree. This can be done in constant time by trying to insert $t[i]$ from the position p. If this insertion creates a leaf, its end value is directly set to i (not added in $queue_{leaf}$) and it is used to try to insert $t[i-k+1, i]$. If no leaf

is created, then we continue by trying to insert $t[i-k+1,i]$ from the leaf reached (we know that the insertion of $t[i-k,i]$ leads to a leaf since the path length of the leaves is bounded by k). If the insertion of $t[i-k+1,i]$ does not create a leaf, we use the position reached in the tree to start the next phase.

A pseudo-code of this algorithm is given in appendix in Algorithm A.

The time and space complexities of the algorithm are linear in the size of the input text (see[1] for details).

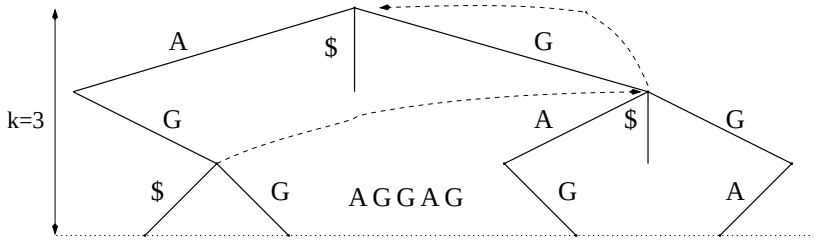

Figure 6. Example of a k-factor tree for the text $AGGAG\$$ with $k = 3$

4.3 Gapped-factor tree construction

We now present the construction algorithm of a gapped-factor tree (**gft** for short). Once again, the construction algorithm is done online. As shown in Figure 4, a gft is composed of three different regions: the upper part of depth k, the invisible region corresponding to the gap of length d, and the lower part of depth k':

1. During the construction of the gft, the first region is treated exactly as for a k-factor tree. The queue containing the leaves in extension is denoted by $queue_{leaf_up}$.
2. When a leaf reaches the depth k, it enters in the invisible region for d phases. To simulate this behaviour, a queue is created that contains the leaves in extension in the invisible region. This queue is denoted by $queue_{invisible}$. Leaves entering $queue_{invisible}$ stay inside for d phases. During those phases, leaves inside the queue are ignored. After d phases, a leaf in the queue is virtually reaching the depth $k+d$. It is then removed from the queue.
3. The construction algorithm of the lower part of the tree is again very similar to the one of a k-factor tree. All the tricks applied for the suffix tree construction are still available. Once more a queue is used to store the leaves in extension in the lower part of the tree. This queue is denoted by $queue_{leaf_low}$. The ending integer of the leaves in extension in the queue is the global variable i. The leaves stay in the queue during k' phases before they become leaves that stay fixed, and contain the positions of the gapped-factors corresponding to the path leading to them from the root.
   However, for the construction of the lower part, the use made of suffix links is slightly different than in the upper part of the tree. This is due to the following particularity of the gapped-factor tree: a node in the lower part of the tree may have up to $|\Sigma|$ suffix links. Indeed, one node in this tree may correspond to several paths. According to the first letter in the invisible region leading to a node, the suffix link to follow will not be the same. Figure 7 illustrates this observation.

The algorithm 15 given in appendix gives an overview of the whole gapped-factor tree construction algorithm.

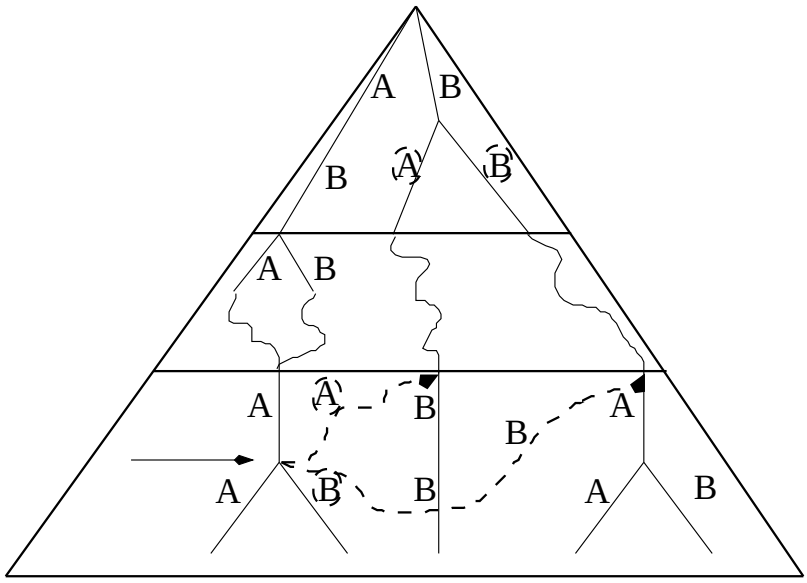

Figure 7. Example of multiple suffix links. The node pointed by an arrow has two suffix links (in dotted line). One is labelled with an A and the other is labelled with a B. The correct suffix link to follow depends on the path that leads to the node. If the node is reached reading $ABA.w$ ($w \in \Sigma^*$), the correct suffix link to follow is the one labelled with an A; it goes to a node reachable reading the text $BA.w$. Any other suffix link leaving the node would be labelled differently and would reach a node corresponding to the text $B.\sigma.w$, with $\sigma \in \Sigma$ and $\sigma \neq A$.

Complexity of the Gft construction The algorithm for constructing a gft uses all the tricks employed by Ukkonen and Allali to lead to a linear time and memory complexity. However, the multiple suffix links add a multiplicative term in $|\Sigma|$ to both complexities. Thus the total time and memory complexity for the construction of a gapped-factor tree for a text t is in $O(|t| \times |\Sigma|)$. One can notice that once the gapped-factor is constructed, the (multiple) suffix links are not useful anymore and can be removed. In this case, the memory complexity falls back to $O(|t|)$.

Generalisation to more than one text As for the suffix tree or the k-factor tree, the gft can be extended to a *generalised gapped-factor* tree and accept a set of $m > 1$ texts $t_0,\ t_1,\ \ldots,\ t_{m-1}$.

In this case, each text $i \in [0, m-1]$ ends with a special character $\$_i$ and the leaves are labelled not only with the positions of a gapped-factor but also with the sequence number in $[0, m-1]$ where the factors occur. The complexity for constructing a generalised gapped-factor tree is in $O\left(\left(\sum_{i=0}^{m-1} |t_i|\right) \times |\Sigma|\right)$.

5 Basic uses of a gapped-factor tree

To find all the positions where a (k-d-k')-gapped-factor occurs in a text given a (k-d-k')-gapped-factor tree for the text one needs to find the leaf corresponding to the given gapped-factor. This is done straightforwardly by traversing the gapped-factor tree from the root to the node as in a suffix tree. The list attached to the leaf corresponds to the positions of the occurrences of the gapped-factors.

This algorithm takes a time proportional to the number of nodes traversed, which is in the worst case $k + k'$. Thus retrieving the positions of a given (k-d-k')-gapped-factor is done in $O(k + k')$.

The gft data structure allows also to easily find all the repeated gapped-factors of a text or of a set of texts. If we are interested in finding all gapped-factors occurring at least r times in a text, for r a positive integer, we just have to visit the leaves. For each leaf, if the number of elements of the list attached to it is greater or equal to r, the corresponding gapped-factor is considered as repeated.

As the number of elements of each list may be stored in the leaves, this extraction is done in time proportional to the number of leaves. If n denotes the length of the indexed text, the number of leaves is no greater than n. The extraction is therefore done in time $O(n)$.

In the generalised case, one may want to extract all gapped-factors occurring in at least r different texts. In this case, to each leaf is attached the number of different texts in which the corresponding gapped-factor occurs. Thus extracting all gapped-factors occurring in at least r different texts is done by checking each leaf in constant time leading to a complexity in $O(\sum_{i=1}^{m} |t_i|)$.

6 Conclusion

We presented a new data structure used for indexing factors containing a gap (called the gapped-factors). This data structure is based on the suffix tree structure. Furthermore, we indicated an online construction algorithm of this data structure for a text t on an alphabet Σ in $O(|t| \times |\Sigma|)$ time and space. This algorithm is based on the Ukkonen algorithm for constructing a suffix tree.

References

[1] J. Allali and M. Sagot: *The at most k-deep factor tree*, Tech. Rep. #2004-03, Institut Gaspard Monge, Université de Marne-la-Vallée, 2004.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman: *Basic local alignment search tool*. Journal of Molecular Biology, 215(3) 1990, pp. 403–410.

[3] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman: *Gapped BLAST and PSI–BLAST: a new generation of protein database search programs*. Nucleic Acids Research, 25 1997, pp. 3389–3402.

[4] M. Brudno, M. Chapman, B. Göttgens, S. Batzoglou, and B. Morgenstern: *Fast and sensitive multiple alignment of large genomic sequences*. BMC Bioinformatics, 4 2003, p. 66.

[5] M. Brudno, C. B. Do, G. M. Cooper, M. Kim, E. Davydov, E. D. Green, A. Sidow, and S. Batzoglou: *LAGAN and Multi-LAGAN: Efficient tools for large-scale multiple alignment of genomic DNA*. Genome Research, 13 2003, pp. 721–731.

[6] S. Burkhardt, A. Crauser, P. Ferragina, H. P. Lenhof, and M. Vingron: *q-gram based database searching using a suffix array (QUASAR)*. Proceedings of the third annual international conference on Computational molecular biology (Recomb 99), 1999, pp. 77–83.

[7] S. Burkhardt and J. Kärkkäinen: *Better filtering with gapped q-grams*, in Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001), vol. 2089 of LNCS, 2001, pp. 73–85.

[8] S. Burkhardt and J. Kärkkäinen: *One-gapped q-gram filters for Levenshtein distance*. 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002), 2373 of LNCS 2002, pp. 225–234.

[9] R. C. Edgar: *MUSCLE: Multiple sequence alignment with high accuracy and high throughput*. Nucleic Acids Research, 32(5) 2004, pp. 1792–1797.

[10] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava: *Approximate string joins in a database (almost) for free*, in In Proc. of 27th Int'l Conf. on Very Large DataBases (VLDB 2001), 2001, pp. 491–500.

[11] D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[12] M. Höhl, S. Kurtz, and E. Ohlebusch: *Efficient multiple genome alignment.* ISMB (Supplement of Bioinformatics), Vol. 18 2002, pp. S312–S320.

[13] C. S. Iliopoulos, J. Mchugh, P. Peterlongo, N. Pisanti, W. Rytter, and M.-F. Sagot: *A first approach to finding common motifs with gaps.* International Journal of Foundations of Computer Science, 16(6) 2005, pp. 1145–1154.

[14] J. Kärkkäinen: *Computing the threshold for q-gram filters.* Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT 2002), 2368 of LNCS 2002, pp. 348–357.

[15] G. Kucherov, L. Noe, and M. Roytberg: *Multiseed lossless filtration.* IEEE/ACM Transactions on Computational Biology and Bioinformatics, 02 2005, pp. 51–61.

[16] M. Li and B. Ma: *PatternHunter II: Highly sensitive and fast homology search.* Genome Informatics, 14 2003, pp. 164–175.

[17] D. J. Lipman and W. R. Pearson: *Rapid and sensitive protein similarity searches.* Science, 227 1985, pp. 1435–1441.

[18] B. Ma, J. Tromp, and M. Li: *PatternHunter: Faster and more sensitive homology search.* Bioinformatics, 18(3) 2002, pp. 440–445.

[19] L. Marsan and M.-F. Sagot: *Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification*, in RECOMB, 2000, pp. 210–219.

[20] E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of the Association of Computing Machinery, 23(2) 1976, pp. 262–272.

[21] M. Michael, C. Dieterich, and M. Vingron: *Siteblast rapid and sensitive local alignment of genomic sequences employing motif anchors.* Bioinformatics, 21(9) 2005, pp. 2093–2094.

[22] J. Na, A. Apostolico, C. Iliopoulos, and K. Park: *Truncated suffix trees and their application to data compression.* Theor. Comput. Sci., 304(1-3) 2003, pp. 87–101.

[23] J. Na and K. Park: *Data compression with truncated suffix trees.* Data Compression Conference (DCC 2000), IEEE Computer Society, online edition: http://computer.org/proceedings/dcc/0592/0592toc.htm 2000, p. 565.

[24] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio: *Indexing text with approximate q-grams.* 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000), 1848 of LNCS 2000, pp. 350–363.

[25] P. Peterlongo, N. Pisanti, F. Boyer, and M.-F. Sagot: *Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array.* String Processing and Information Retrieval (SPIRE 2005), 3772 of LNCS 2005, pp. 179–190.

[26] P. Pevzner and M. Waterman: *Multiple filtration and approximate pattern matching.* Algorithmica, 13 1995, pp. 135–154.

[27] N. Pisanti, A. Carvalho, L. Marsan, and M.-F. Sagot: *RISOTTO: Fast extraction of motifs with mismatches*, in Proceedings of the 7th Latin American Theoretical Informatics Symposium, vol. 3887 of LNCS, 2006, pp. 757–768.

[28] K. R. Rasmussen, J. Stoye, and E. W. Myers: *Efficient q-gram filters for finding all ε-matches over a given length*, in 9th Annual International Conference, Research in Computational Molecular Biology (Recomb 2005), vol. 3678 of LNCS, 2005, pp. 189–203.

[29] E. Sutinen and J. Tarhio: *On using q-gram locations in approximate string matching*, in Third Annual European Symposium, (ESA 95), vol. 979 of LNCS, 1995, pp. 327–340.

[30] P. Thébault, S. DeGivry, T. Schiex, and C. Gaspin: *Combining constraint processing and pattern matching to describe and locate structured motifs in genomic sequences*, in Fifth IJCAI-05 Workshop on Modelling and Solving Problems with Constraints, Edindurgh, Scotland, 2005, pp. 330–337.

[31] E. Ukkonen: *On-line construction of suffix-trees.* Algorithmica, 14 1995, pp. 249–260.

A Pseudo-codes

Algorithm 10 Fast Insertion

Require: $\mathcal{N}$,t,$start$,end
Ensure: Insert a string $t_{start...end}$ from a node $\mathcal{N}$ assuming that the tree is already constructed for $t_{start...end-1}$ from $\mathcal{N}$
1: $endJump \leftarrow false$
2: **while** (not $endJump$) and $((end - start) \neq 0)$ **do**
3: set $child$ to the child of $\mathcal{N}$ that starts with the letter t_{start}
4: **if** $(end - start) \geq length(\mathcal{N}, child)$ **then**
5: $start \leftarrow start + length(\mathcal{N}, child)$
6: $\mathcal{N} \leftarrow child$
7: **else**
8: $endJump \leftarrow true$
9: **end if**
10: **end while**
11: **if** $(end - start) = 0$ and $\mathcal{N}$ has not a child for letter t_{end} **then**
12: add a child to $\mathcal{N}$ with edge label start equal to end
13: **end if**
14: $e \leftarrow$the label of the edge between $\mathcal{N}$ and $child$
15: **if** $e_{end-start+1} \neq s_{end}$ **then**
16: split e at position $end - start$
17: add a leaf with start position equal to end to the new node
18: **end if**

Algorithm 11 Factor Tree

Require: R, t , k , $queue_{leaf}$
Ensure: The k-factor tree of t
1: do the first $k-1$ phases using Suffix_Tree algorithm, filling $queue_{leaf}$ with each new leaf created
2: **for** i **from** k **to** $|t|$ **do**
3: **if** $queue_{leaf}$ is not empty **then**
4: set $lastLeaf$ to the leaf at the end of $queue_{leaf}$
5: **else**
6: add t_i from last position reached during the last insertion
7: **if** a leaf is created **then**
8: add this leaf at the end of $queue_{leaf}$
9: set $lastLeaf$ to this leaf
10: **else**
11: set $lastLeaf$ to the leaf reached
12: **end if**
13: **end if**
14: **Phase** (R, t , k, i, $queue_{leaf}$, $lastLeaf$)
15: remove the leaf at the head of $queue_{leaf}$ and set its end value to i
16: **end for**
17: **return** R

Algorithm 12 Function **Phase** (Suffix tree and k-factor tree construction)

Require: R, t, k, i, $\underline{queue_{leaf}}$, $lastLeaf$
Ensure: One phase of the construction of the suffix tree and of the k-factor tree. The underlined parts stand only for the k-factor tree construction.

```
1: endPhase ← false
2: repeat
3:    forward ← length(Father(lastLeaf), lastLeaf) − 1
4:    if S_l(Father(lastLeaf)) is undefined and Father(lastLeaf)! = R then
5:       forward ← forward + length(Father(Father(lastLeaf)), Father(lastLeaf))
6:       if Father(Father(lastLeaf)) is R then
7:          AddString(R,t,i − forward + 1,i)
8:       else
9:          AddString(S_l(Father(Father(lastLeaf))),t,i − forward,i)
10:      end if
11:   else
12:      if Father(lastLeaf) is R then
13:         AddString(R,t,i − forward + 1,i)
14:      else
15:         AddString(S_l(Father(lastLeaf)),t,i − forward,i)
16:      end if
17:   end if
18:   if a node was created during the previous step then
19:      set the suffix link of this node to the last node reached during the insertion
20:   end if
21:   if a leaf was created in the call to AddString then
22:      set lastLeaf to this leaf
23:      add this leaf at the end of queue_leaf
24:   end if
25:   if no node was created during the call to AddString then
26:      endPhase ← true
27:   end if
28: until not endPhase
```

Algorithm 13 Suffix Tree

Require: t
Ensure: The suffix tree of t

```
1: Add to R a leaf L with edge label t_0
2: lastLeaf ← L
3: for i from 1 to |t| − 1 do
4:    Phase (R, t , k, i, lastLeaf)
5: end for
6: return R
```

Algorithm 14 Lower_Part_Tree

Require: R, t , k , d, i, $queue_{leaf_low}$, $lastLeafLow$
Ensure: A construction phase of the lower part of the gapped-factor tree
1: $endPhaseLow \leftarrow false$
2: **repeat**
3: $forward \leftarrow length(Father(lastLeafLow), lastLeafLow) - 1$
4: **if** $S_l(t, Father(lastLeafLow))$ is defined but not labeled with the good character **then**
5: Point the $Father(lastLeafLow)$ node as the node created during the previous step
6: **end if**
7: **if** $S_l(t_\alpha, Father(lastLeafLow))$ is undefined and $Father(lastLeafLow)! = R$ **then**
8: $forward \leftarrow forward + length(Father(Father(lastLeafLow)), Father(lastLeafLow))$
9: **if** $Father(Father(lastLeafLow))$ is R **then**
10: $AddString(R,t,i - forward + 1,i)$
11: **else**
12: AddString$(S_l(t_\alpha, F(F(lastLeafLow)))t,i - forward,i)$
13: **end if**
14: **else**
15: **if** $Father(lastLeafLow)$ is R **then**
16: $AddString(R,t,i - forward + 1,i)$
17: **else**
18: $AddString(S_l(t_\alpha, F(lastLeafLow),t,i - forward,i)$
19: **end if**
20: **end if**
21: **if** a node was created during the previous step **then**
22: set the suffix link labeled t_α of this node to the last node reached during the insertion
23: **end if**
24: **if** a leaf was created during the call to AddString **then**
25: set $lastLeafLow$ to this leaf
26: add this leaf at the end of $queue_{leaf_low}$
27: **end if**
28: **if** no node was created in the call to AddString **then**
29: $endPhaseLow \leftarrow true$
30: **end if**
31: **if** the width of the last position reached during the fast insertion was $\leq k + d$ **then**
32: $endPhaseLow \leftarrow true$
33: **end if**
34: **until** not $endPhaseLow$
NOTE : α is the first character in the invisible region on the $lastLeafLow$ path

Algorithm 15 GappedFactor_Tree

Require: R, t , k , d, $queue_{leaf_up}$, $queue_{leaf_low}$, $queue_{invisible}$
Ensure: Complete construction algorithm of a gapped factor tee

```
do the first k phases using Suffix_Tree algorithm, filling queue_leaf_up with each new leaf created
for i from k to |t| do
  if index of node at the head of queue_invisible = k + 1 then
    create a new edge from this node labeled t_i
    add the new leaf at the end of queue_leaf_low
    remove the node at head of queue_invisible
  end if
  if queue_leaf_up is not empty then
    set lastLeafUp to the leaf at the end of queue_leaf_up
  else
    add t_i from last position reached during the last insertion on the upper part of the tree
    if a leaf is created then
      add this leaf at the end of queue_leaf_up
      set lastLeafUp to this leaf
    else
      set lastLeafUp to the leaf reached
    end if
  end if
  Phase(R, t, k, i, queue_leaf_up, lastLeafUp)
  remove the pseudo leaf at the head of queue_leaf_up
  if the pseudo leaf is new then
    set the pseudo leaf index value to 0 (invisible zone)
    add the pseudo leaf at the end of queue_invisible
  end if
  if i > k + d //the lower part of the tree is on construction then
    if queue_leaf_low is not empty then
      set lastLeafLow to the leaf at the end of queue_leaf_low
    else
      add t_i from last position reached during the last insertion on the low part of the tree
      if a leaf is created then
        add this leaf at the end of queue_leaf_low
        set lastLeafLow to this leaf
      else
        set lastLeafLow to the leaf reached
      end if
    end if
    Lower_Part_Tree(R, t, k, d, i, queue_leaf_low, lastLeafLow)
    if i ≥ k + d + k // End the extention of the tree then
      remove the leaf at the head of queue_leaf_low
      set the leaf end value to i
    end if
  end if
end for
return R
```