BuST-Bundled Suffix Trees

Luca Bortolussi[1], Francesco Fabris[2], and Alberto Policriti[1]

[1] Department of Mathematics and Informatics, University of Udine.
bortolussi|policriti AT dimi.uniud.it
[2] Department of Mathematics and Informatics, University of Trieste.
frnzfbrs AT dsm.uniuv.trieste.it

Abstract. We introduce a data structure, the *Bundled Suffix Tree* (*BuST*), that is a generalization of a Suffix Tree (*ST*). To build a *BuST* we use an alphabet Σ together with a non-transitive relation $\approx$ among its letters. Following the path of a substring β within a *BuST*, constructed over a text α of length n, not only the positions of the exact occurrences of β in α are found (as in a *ST*), but also the positions of all the substrings $\beta_1, \beta_2, \beta_3, \ldots$ that are related with β via the relation $\approx$ among the characters of Σ, for example strings at a certain "distance" from β. A *BuST* contains $O(n^{1+\delta})$ additional nodes ($\delta < 1$) in probability, and is constructed in $O(n^{1+\delta})$ steps. In the worst case it contains $O(n^2)$ nodes.

1 Introduction

A Suffix Tree is a data structure computable in linear time and associated with a finite text $\alpha = \alpha[1], \alpha[2], \ldots, \alpha[n] = \alpha[1...n]$, where $\alpha[i] \in \Sigma$ and $\Sigma = \{a_1, a_2 \ldots, a_K\}$ is the alphabet (that is $|\Sigma| = K$). In the following we suppose the existence of an ordering among alphabet letters and we assume to append a character $\# \notin \Sigma$ at the end of our text, as is customary when working with *ST*'s. A *ST* allows to check in $O(m)$ time if an assigned string β, $|\beta| = m$, is a substring of α; moreover, at the same time it gives the exact positions $j_1, j_2, \ldots, j_r$ of all the r occurrences of β into α in $O(r)$ additional time. Therefore, a *ST* solves the *Exact String Matching Problem* (*ESM*) in linear time with respect to the length n of the searched string. A *ST* solves in linear time also the *Longest Repeated Exact Substring Problem* (*LRES*) of an assigned text α. A complete and detailed treatment of these results can be found in [6].

Even if very efficient in solving the *ESM* and the *LRES* problem, the *ST* data structure suffers of an important drawback when one has to solve an *Approximate String Matching Problem* (*ASM*), or to solve the harder *Longest Repeated Approximate Substring Problem* (*LRAS*). In these cases, one needs to search for strings $\beta_1, \beta_2, \beta_3, \ldots$ substrings of α, such that $\mathbf{d}(\beta, \beta_j) \leq D$, where $\mathbf{d}(\cdot, \cdot)$ is a suitable *distance* (most frequently Hamming or Levenshtein distance) and D is constant or proportional to the length of β. This happens because the structure of a *ST* is not adequate to handle distance in a natural way. This

forces one to take into account errors by using unnatural and complicated strategies, that inevitably lead to cumbersome algorithms. In general, many different indexing structures other than ST are used to tackle approximate matching problems [9, 8, 5], but all these approaches use an exact index for the text together with some searching strategy to find all (approximate) occurrences of the pattern β in the text α. Among those structures, STs play a prominent role, not only for approximate matching, but also in pattern discovery algorithms, like in [7], and for statistical analysis of approximate occurrences [3], where it is important to have knowledge about the inner structure of the processed text.

In this work we present a generalization of a Suffix Tree, the *Bundled Suffix Tree* ($BuST$), which contains information about an approximate relation between strings as a *structural* property of the tree. This allows us perform some kind of approximated string matching with a $BuST$ in the same manner in which we perform exact string matching with a ST. In particular, $BuST$ are better suited for *LRES* and all the problems that require some form of exploitation of the inner (approximate) structure of a string. The matching criterion we use can be very general, in fact we only require to be given a (not necessarily transitive) relation among letters of the alphabet Σ. For example, the notion of Hamming distance induces a very natural non-transitive relation on Σ when each letter $a \in \Sigma$ is in fact a t-tuple over a sub-alphabet Σ_1 (for example $\Sigma_1 = \{A, C, G, T\}$): the relation between two Σ-characters $a_i, a_j \in \Sigma$ holds if and only if $\mathbf{d}_H(a_i, a_j) \leq D$, where, $\mathbf{d}_H(\cdot, \cdot)$ is the Hamming distance and D is a constant. Other notions of distance can be used as well.

Bundled Suffix Trees encode in a compact way the relational structure existing between the substrings of the processed text α. In fact, the relation among the letters of the alphabet can be easily extended to strings (two strings are in relation if so are all their constituting characters), and then we can consider all the relations intercurring between the substrings of α. This information is added to the Suffix Tree by marking some positions in the tree (that can be both in the middle of the edges or over its nodes) with labels corresponding to suffixes, in such a way that the existence of a label j after a certain point implies that the string labeling the path from the root to that point is in relation with a prefix of suffix j. In other words, while constructing a $BuST$, we are resurrecting some nodes of the underlying suffix trie, and attaching to them an additional information in terms of labels. The nodes are added only in the lowest position satisfying the property stated above, to avoid the insertion of redundant information (see def. 2). A detailed analysis of the dimension of $BuST$ shows that, though the worst case size is $O(|\alpha|^2)$, the average size is subquadratic (but superlinear), see Section 3.

Observe that the information we add to a ST is internal to the processed string α, in the sense that we do not add any information about the relation of substrings of α with external strings. For this reason, $BuST$ can be useful for all those applications exploiting this internal information (as *LRAS*) and not necessarily, for example, to search for the approximate occurrences of an external pattern in the text α. A suitable application for $BuST$ is presented in

this paper and concerns the calculation of the approximate frequency of appearance of a given subword (with the relative calculation of associated measures of surprise), cf. Section 5. An advantage is that the above mentioned information can be extracted from the *BuST* in the same way this extraction is done with Suffix Trees in the exact case.

The notion of relation between letters of an alphabet is a general concept, susceptible of encoding different properties connected with the specific application domain, e.g. Hamming-like distances or scoring schemes. Moreover, the particular relation used is completely orthogonal with respect to the definition, the construction and the analysis of the data structure. In this presentation we will deal with a restricted type of relation, constructed over an alphabet of macrocharacters, by means of a threshold criterion relative to a selected distance (mainly Hamming distance). The macroletters can have fixed or variable length; this is not a problem as long as they form a prefix-free code. On the other hand, the introduction of macrocharacters brings some rigidity in the type of approximate information that can be encapsulated. For instance, the Hamming-like relation introduced above puts in correspondence two strings if their distance is less than a threshold proportional to their length, and if the errors are distributed among the tuples. Moreover, only strings of length proportional to the macroletters' length can be compared. This rigidity, however, is the price to pay to "localize" the approximate information we are looking for: with the Hamming-like relation, we "localize" a global distance between two strings by splitting it evenly between their tuples.

The paper is organized as follows. In Section 2 we give the definition of the structure and a naive algorithm for its construction. In Section 3 we analyze the dimension of the data structure in the worst and in the average case. In Section 4 we give some hints to an optimal construction algorithm, while Section 5 contains an application for computing approximate surprise indexes. Finally, in Section 6 we draw some conclusions. The interested reader can find complete proofs, details on the optimal construction and further information in [4].

2 Naive construction of a *BuST*

A *ST* is not suitable to handle approximate search in a natural way essentially because of its rigidity in matching characters: they either match and the (unique) path proceeds, or the characters are different and a branching point is necessary. Conversely, in a *BuST* we accept the idea that a path is good not only when characters match, but also when they are in relation.

Let $\Sigma = \{a_1, \ldots, a_k\}$ be an alphabet, and $\approx$ be a symmetric and reflexive binary relation on Σ, encapsulating some form of approximate information.

Definition 1. *Given a string $\beta = \beta[1, \ldots, m]$, we say that $\gamma = \gamma[1, \ldots, m]$ is a* variant *of β if and only if $\beta[i] \approx \gamma[i]$, $\forall i = 1, \ldots, m$, and we write $\beta \approx \gamma$. We denote with $\approx(\beta) = \{\gamma \mid \beta \approx \gamma\}$.*

The case in which $\approx$ is an equivalence relation trivializes the approach. Hence, we assume that, in general, $\approx$ is not transitive. Other non equivalence relations could be considered as well.

Given a ST for α, the key idea for constructing the associated $BuST$ is that of marking in the ST (all) the paths corresponding to (prefixes of) $\approx$-variants of each substring $\alpha[j \dots n]$, for $1 \leq j \leq n$. This is achieved by inserting nodes over these position and labeling such nodes with the index of the starting position of the suffix of which they are $\approx$-variants (see Figure 1). Intuitively, we are *bundling* several paths over the skeleton of the ST.

In order to distinguish these newly inserted nodes, we refer to them as *red* nodes, while we call *black* the nodes of the original ST. Notice that, according to the previous characterization, a node can be both black *and* red. In addition, red nodes can have a set of labels associated to them. Moreover, red nodes that end up in between a ST edge are not branching and are simply *splitting the edge*—i.e. they are nodes of the underlying *Suffix Trie*.

To (naively) construct the $BuST$ of a text α, we can enter each suffix $\alpha[j \dots n]$ in the associated ST and find all possible paths that correspond to a (maximum length) prefix of one of its $\approx$-variants. This is done by successively comparing and ($\approx$-)matching characters of α and $\alpha[j \dots n]$. When the first letter of $\alpha[j \dots n]$, say $\alpha[p]$, not in relation with the processed letter of the current path in the ST is found, a red node with label j is inserted (if not already present) in the position just before $\alpha[p]$. If a red node is already present at that position, label j is added to its label set.

Turning back to the comparison phase, two different situations can occur. Either we are in the middle of an edge or on a branching node. In the former case we simply compare the current text character with the current suffix character $\alpha[i]$. If the character is in relation with $\alpha[i]$ we continue, otherwise we insert the red node. In the latter case we have to consider the first letter of *any* branching path from the current node. Following the alphabet ordering and always keeping operative as many paths as are the letters in relation with $\alpha[i]$, new matching paths can be generated. If no letters are found that are in relation with $\alpha[i]$, then the new red node is superimposed over the existing black branching one.

The $BuST$ for the text $\alpha = bcabbabc$ is depicted in Figure 1, in which $\Sigma = \{a, b, c\}$ and $\approx$ is defined by $a \approx b$, $b \approx c$, and $a \not\approx c$.

Below we give a formal definition of Bundled Suffix Tree.

Definition 2. *A* Bundled Suffix Tree *(*BuST*) $\mathcal{B}$ for a text $\alpha[1 \dots n]$, is a Suffix Tree $\mathcal{S}$ for α (the black skeleton) plus a set of internal (red) nodes with associated (multiple) labels, such that:*

(Main) the path label from the root to a red node labeled j is an $\approx$-variant of a prefix of $\alpha[j \dots n]$.

(Uniqueness) in every path from the root to a black leaf labeled j, there can be at most one red node with label $h \neq j$.

(Maximality) if $\alpha[h \dots h+i]$ is the string labeling the path from the root to a red node labeled j, then $\alpha[h \dots h+i] \approx \alpha[j \dots j+i]$ but $\alpha[h+i+1] \not\approx \alpha[j+i+1]$.

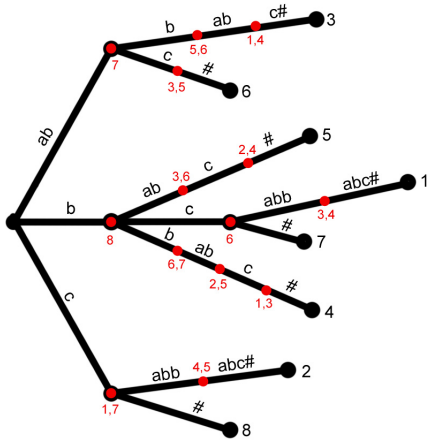

Fig. 1. Bundled Suffix Tree for the sequence $\alpha = bcabbabc\#$, with $a \approx b \approx c$.

The Main property accounts for the most important function of a *BuST*, that is to encode all $\approx$-variants of a substring of α. The Uniqueness property states that once a red node labeled h is inserted, the subtree rooted at this node cannot contain other red nodes with the same label. Maximality and uniqueness together assure that we insert at most one red node at the *deepest* possible position.

Remark 1. If β is the path label of the ST for α, the starting positions of substrings γ of α that are variants of β are found by reading all the labels rooted at the end of β.

Remark 2. The *BuST* is a data structure which is, in some sense, *in the middle* between a Suffix Tree and a Suffix Trie. We recall that a Suffix Trie is similar in shape to a *ST*, but every edge contains as many nodes as the length of its label. While constructing a *BuST*, we insert nodes splitting edges, hence the set of nodes of a *BuST* contains that of a *ST* and resembles to that of the corresponding Suffix Trie. The analogy stops here, as red nodes may have multiple labels and are added using relation $\approx$ as matching primitive.

In order to simplify the following computations, we assume that the relation $\approx$ enjoys the *hypercube-like property* over Σ: for each $a \in \Sigma$, there is a constant number V of $b \in \Sigma$, such that $a \approx b$. When elements of Σ are tuples built over a sub-alphabet Σ_1, we will put $a \approx b$ if and only if $\mathbf{d}(a, b) \leq D$, where $\mathbf{d}(a, b)$ is a suitable distance between tuples and D is a constant. In such cases we will also assume that the constant D is proportional to the length of Σ_1 t-tuples constituting elements of Σ. If we work with the Hamming distance, then the macro-characters b such that $a \approx b$ are all the elements of the Hamming sphere of radius d and centered in a. In such a case the constant V is the *volume* of this Hamming sphere.

3 Structural properties of a BuST

In order to study the structure of a *BuST*, we have to compute, for each assigned suffix $\alpha[j \dots n]$, the number $R(j)$ of red nodes inserted; then the total number of red nodes inserted[1] is $R = \sum_{j=1}^{n} R(j)$. We will perform first the average case analysis, leaving the worst case one at the end. Note that $R(j)$ corresponds to the number of substrings in α that are (maximum length) prefixes of $\approx$-variants of $\alpha[j \dots n]$. Remember, also, that for any red node with label j, the label of the path starting from the root and leading to it, is a $\approx$-variant of the suffix $\alpha[j \dots n]$. In order to find the paths with this property, we reason on the execution of the naive construction presented in the previous section.

While processing suffix j, we have to follow $\alpha[j \dots n]$ on the black skeleton as long as the two letters we are comparing are in relation. When we find the first letter in $\alpha[j \dots n]$ that is not in relation with the current letter of the ST-path (or to any letter that immediately follows a black branching node), we insert a red node with label j (or we add label j to a preexisting red node). In particular, at every branching node of the ST we have to visit only the edges starting with a character in relation with the corresponding one in the suffix. Suppose the ST has height h, then it is contained in a complete K-ary tree of height h, $K = |\Sigma|$. In the hypothesis made at the end of the previous section, we know that only V out of K characters are in relation with one letter, hence at every internal node only V out of K edges will survive during the construction. In this way, we can bound the number of survived paths at depth h, and thus $R(j)$, by V^h (at each level, the number of active paths is multiplied by a factor V). A more reasonable bound of $R(j)$ can be obtained by replacing h with the average depth d.

Therefore, the value of (an upper bound on) $R(j)$ is strictly connected with the average structure of the ST. In particular, we are interested in the average behaviour of the height and of the average depth of a path from the root to a leaf. These quantities have been analyzed in [10, 11], under the hypothesis of the text being generated by a stationary and memoryless source $\mathcal{S}$. If $\mathbb{X} = \{X_1, \dots, X_n, \dots\}$ is the sequence of random variables generated by the source, we indicate with H_n the height of the Suffix Tree built from $\{X_1, \dots, X_n\}$ and with Z_n the average depth. From [10] we have that the average value of the height, $h_n = E[H_n]$, asymptotically converges (in probability) to $\log(n)/\log(1/p^+)$, while the asymptotic behaviour of $z_n = E[Z_n]$ approaches $\log(n)/H(\mathcal{S})$. Here p^+ is the maximum value of the probability distribution on Σ that defines $\mathcal{S}$, while $H(\mathcal{S})$ is the Shannon entropy of $\mathcal{S}$.

The results stated above allow us to compute probabilistic upper bounds (denoted by $\lesssim$) for the quantity $R(j)$:

$$R(j) \lesssim V^{h_n} \backsimeq V^{\log_{1/p^+} n} = n^{\delta},$$

[1] We are correctly counting the size of the sets of labels inserted, not the number of red nodes.

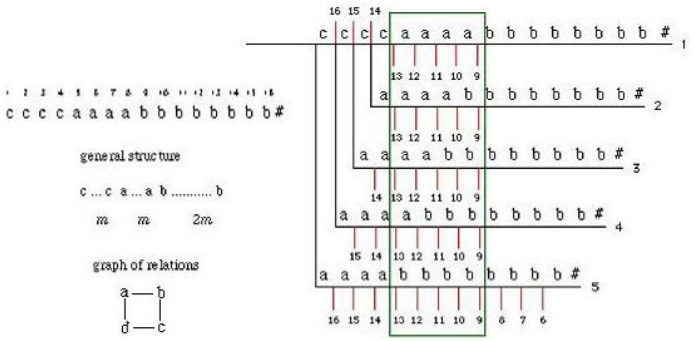

Fig. 2. A worst case BuST for the sequence $\alpha = aaccbbbb\#$

with $\delta = \log V / \log(1/p^+)$. A better estimate of $R(j)$ can be obtained by replacing h_n with z_n, obtaining $R(j) \lesssim n^{\delta'}$, with $\delta' = \log V / \log H(\mathcal{S})$.

Therefore, the total number of red nodes inserted, denoted by R, is bounded on average by:

$$R = \sum_{j=1}^{n} R(j) \lesssim \sum_{j=1}^{n} n^{\delta} = n^{1+\delta}.$$

The value of δ depends on the probability distribution of the source and on the relation between the letters of the alphabet. For instance, for an Hamming-like relation with macrocharacters of length 4 and error rate 25% built over DNA alphabet, and the maximum probability of a DNA letter varying from 0.25 to 0.5, the value of δ remains between 0.46 to 0.92, hence the size of the structure is bound by a subquadratic function.

Observe that the bound we give is coarse, in fact δ can be greater than one, while the size of the data structure cannot be more than quadratic in the length of the processed text. In fact, the number $R(j)$ of red nodes inserted while processing suffix j can be at most one per each path of the Suffix Tree, or equivalently, at most one for each suffix of the text, hence $R(j) \leq n$. Therefore $R \leq n^2$. This theoretical bound can be reached for particular texts, as shown in the following example.

Example 1. Consider a sequence of the form $\alpha = a^n c^n b^{2n}$, over the alphabet $\Sigma = \{a, b, c, d\}$, with $a \approx b \approx c \approx d \approx a$ as relation (it is hypercube like). The lengths of the runs of a,b,c in α are in proportion of $1 : 1 : 2$. Note from Figure 2 that, if the length of the text is $4n$, then the rectangular area delimited by the dashed line contains $n(n-1)$ red nodes (the ones with label from $2n+1$ to $4n-1$, repeated n times).

4 Optimal Construction

We briefly outline here an algorithm for constructing *BuST*s which is optimal, in the sense that its complexity is of the same order of magnitude of its output (i.e., essentially $O(R)$, the number of red nodes inserted). First of all, let is put forward some useful notation. Given two strings β and γ, we write $\gamma \prec \beta$ if γ is a prefix of β. $\gamma \sqsubset \beta$ means that γ is a substring of β, while $\gamma \precsim \beta$ means that γ is in $\approx$-relation with a prefix of β. Negations of these expressions are indicated by $\gamma \not\prec \beta$, $\gamma \not\sqsubset \beta$ and $\gamma \not\precsim \beta$, respectively.

Consider now a red node r_i with label i,[2] such that its path label $\ell(r_i)$ equals some string $x\gamma$, $x \in \Sigma$, $\gamma \in \Sigma^+$. Hence $x\gamma \precsim \alpha[i \dots n]$, but, $\forall y \in \Sigma$ such that $x\gamma y \sqsubset \alpha$, $x\gamma y \not\precsim \alpha[i \dots n]$. All this information implies that $\gamma \precsim \alpha[i{+}1 \dots n]$, but we cannot conclude that there must be a $(i+1)$–red node r_{i+1} after γ. In fact, there can be edges departing from γ with label γz such that $\gamma z \precsim \alpha[i+1 \dots n]$, which derive from paths labeled with $y\gamma z$, $y \neq x$ (and maybe $y \not\approx \alpha[i]$). In other words, if we cross a suffix link (*SL* from now on) from a node r_i and we find ourselves in a black node p, we may need to visit the whole subtree rooted at p to complete the insertion of $(i+1)$–red nodes.

Indeed, the situation is even worse. There can be paths in the *ST*, where a $(i{+}1)$–red nodes must be inserted, which can never be reached, neither directly traversing a *SL* from a (i)–red node, nor visiting subtrees at the end of a *SL*. These paths correspond to positions that can be reached only from *SL*s that depart from nodes with path label $z\gamma$ and $z \not\approx \alpha[i]$.

Therefore, the frontier of a *BuST* is much more complex than that of a *ST*, and it cannot be controlled easily using *SL*. In some sense, the (main) problem is that, while inserting $(i+1)$–red nodes, we need access to zones of the *ST* that are forbidden to suffix i, because their path label begins with a character which is not equivalent to $\alpha[i]$.

Now, suppose we have a $(i{+}1)$–red node r_{i+1} with path label $\ell(r_{i+1}) = \gamma$. It follows that $\gamma \precsim \alpha[i{+}1 \dots m]$ and $\forall y \in \Sigma$ such that $\gamma y \sqsubset \alpha$, $\gamma y \not\precsim \alpha[i{+}1 \dots n]$. Thus we can consider all the positions in the tree identified by the labels $x\gamma$, where $x\gamma \sqsubset \alpha$ and $x \approx \alpha[i]$, claiming that in all these points we find a (i)–red node. In fact, $\forall y \in \Sigma$ such that $x\gamma y \sqsubset \alpha$, it holds that $x\gamma y \not\precsim \alpha[i \dots n]$, otherwise we have that $\gamma y \precsim \alpha[i+1 \dots n]$, which is a contradiction.

Therefore, if we have a way to reach from a position γ all the positions $x\gamma$ in the tree, we may be able to insert all (i)–red nodes from $(i+1)$–red nodes without matching any character of $\alpha[i \dots n]$ along any path. The operation of going from γ to $x\gamma$ is, in some sense, like crossing in the inverse direction a *SL*. If we are disposed to pay a price in terms of space used, we can define a collection of pointers, called *inverse suffix links* (ISL), that do this job. Specifically for each node p of the *ST*, with path label β, and for each letter $x \in \Sigma$, there is an inverse suffix link $\mathrm{ISL}(p, x)$ that points to the position of the tree with path label $x\beta$, if any. Note that this position may well be in the middle of an edge.

[2] From now on we refer to such a node as an (i)–red node

Equipped with ISL, and with some extra care to keep correctly into account the maximality property of *BuST*, we can define an algorithm that builds the *BuST* for α starting from its *ST* and processing the text backwards, from the last suffix to the first. Red nodes for suffix i are generated from red nodes for suffix $i+1$, essentially by visiting their parent nodes (in the *ST*) and checking the positions of the tree pointed by ISL departing from there (only for the letters in relation with $\alpha[i]$). Each red node r_{i+1} can be processed in constant time (actually in $O(K^2)$, with $K = |\Sigma|$), thus giving rise to an algorithm (called ISL_BUST) with complexity $O(R)$. The interested reader can find all the details in [4], where the following theorem is proved.

Theorem 1. *ISL_BUST constructs a* BuST *for a text α in time $O(R)$.*

5 An Application of *BuST*: detecting approximate unusual words

In this section we present an application of *BuST*s, related to the detection of unusually overrepresented words in a text α. Specifically, we admit as occurrences β' of a word β also strings that are "close" to β, where the concept of closeness means that β' is a variant of β.

Before entering into the details related to the use of *BuST*s, we give a brief overview of a method presented by Apostolico et al. in [2, 1]. The problem tackled is the identification, in a reliable and computationally efficient way, of a subset of strings of a text α that have a particularly high (or low) score of "surprise". Particular care is given in finding a suitable data structure that can represent this set of strings in a space-efficient way, i.e. in a size linear w.r.t. the length of the processed text.

The class of measures of surprise considered is the so called z-score, defined for a substring β of α as $\delta(\beta) = (f(\beta) - E(\beta))/N(\beta)$. Here $f(\beta)$ is the observed frequency of β, $E(\beta)$ is a function that can be interpreted as a kind of expected frequency for β and $N(\beta)$ is a normalization factor. Intuitively, we are computing the (normalized) difference between the expected value for the frequency of β and the observed one. If this score is high, it means that β appears more often than expected, while if it is very low (and negative), than β is underrepresented in α. Conditions on E and N are given to guarantee that, whenever $f(\beta) = f(\beta\gamma)$, then $\delta(\beta) \leq \delta(\beta\gamma)$. In other words, while looking for overrepresented words, we do not have to examine all the $O(n^2)$ substrings of a text α, but we can focus on the longest strings sharing the same occurrences, as they are those having the higher z-score. It is easy to see that those strings correspond exactly to the labels of the (inner) nodes of the suffix tree for α, so we must compute the z-score only for these strings. Their frequency can be computed easily by a traversal of the tree in overall linear time. On the other hand, the computation of E and N can be far from trivial, and its complexity is deeply related to the choice of the probabilistic model adopted for the source. E

is usually taken as the expectation of the frequency of a word, while N is usually chosen as the variance or as its first order approximation $\sqrt{E(\beta)\text{Prob}(\beta)}$. If the probabilistic model of the source is stationary and memoryless, computation of E can be carried out in constant time after a linear preprocessing.

We stress that Suffix Trees not only give rise to an efficient algorithm for computing overrepresented words, but they also allow a compact representation of them. In addition, they allow to reply efficiently to a query of the type: "is a substring β of α overrepresented?". The answer to such a question is, in general, not binary. It can be the case, in fact, that β terminates in the middle of an edge of the Suffix Tree, so there exists a superstring $\beta\gamma$ of β with the same set of occurrences of β, but with an higher z-score. Therefore an answer to the above query can be this superstring, which is maximal w.r.t. the δ measure.

In order to improve the above approach, however, we can consider the case in which we are willing to admit as occurrences of a string β also strings which differ from it, but are "close enough". In [3], an approach is presented to look for overrepresented strings of length m with at most k errors, in the sense that for each β substring of α, we count the number of substrings of α of length n with distance at most k from β, we calculate the expected frequency of approximate occurrence of β, and finally we compute the z-score w.r.t. such parameters. The overall algorithm has complexity $O(kn^2)$, where $n = |\alpha|$.

The approach we present here is a straightforward adaption of the algorithm for the exact case, casted in the realm of *BuST*s. In this setting we have at our disposal a simple and powerful tool for defining a concept of "closeness" between two strings, i.e. the relation on the alphabet of macrocharacters. For instance, we can use an Hamming-like relation (cf. Introduction) on macrocharacters of length m, putting in relation two of them if their Hamming distance is D or less. In this case, we put in relation strings of length multiple of m, which can differ in D/m of their positions, with errors evenly distributed. Thus, we can search for surprising strings of variable length, by counting all the substrings at distance proportional to their length (with some rigidity induced by the usage of macroletters).

We can proceed as follows: given a text α, if we use macrocharacters of size m, we construct the m strings in this new alphabet, obtained by segmenting α starting from different positions (i.e. from position 1 to position m). Then we build the generalized *BuST* for those m strings, and we visit it to mark each internal node, both black and red, with the number of black leaves and red nodes present in the subtree rooted at it. This operation can be performed in time $O(R)$. At this point, for each substring β of α (of size multiple of m), we can read in the *BuST* its approximate frequency $f_R(\beta)$, i.e. the number of substrings of α that are variants of β. With an abuse of notation, we denote from now on by α and β also the corresponding strings in macrocharacters.

We compute the z-score under the hypothesis that α is generated by a Bernoulli process, and we indicate the probability of generating a macroletter a with p_a. However, here we have to compute, for each substring β of α the probability of finding a substring $\beta' \approx \beta$ in α. For a macroletter a we have

that $p'_a = \sum_{b \approx a} p_b$ is the probability of finding a macrocharacter in relation with a, while for a string $a_1 \dots a_k$ the probability of finding an approximate occurrence under the source model is $\text{Prob}(a_1 \dots a_k) = \prod_{i=1}^{k} p_{a_i}$. Thus the expected numbers of occurrences in α, $|\alpha| = n$ of a string in relation with β, $|\beta| = m$ is $(n - m + 1)\text{Prob}(\beta)$. Therefore we can adopt the same trick used for computing expectations for the exact case: we precompute a vector $A[i] = \text{Prob}(\alpha[1 \dots i])$ in linear time using the recursive relation $A[i] = A[i - 1]p_{\alpha[i]}$, then the expectation for $\alpha[i \dots j]$ can be computed in constant time by $\text{Prob}(\alpha[i \dots j]) = A[j]/A[i]$.
As normalization factor, we can use the expectation itself, or the first order approximation of the variance. A direct computation of the variance itself seems much more complicated, as here we cannot use anymore the method used in [2] (in essence, we should replace the concept of autocorrelation with the weaker notion of $\approx$-autocorrelation, i.e. we should look for $\approx$-periods of words; we leave this investigation for future work).

With those choices for the source model and for the normalization factor, we are able to compute the z-score δ for each string labeling an inner node (both black and red) of the *BuST* in constant time. Note that if the path in the tree labeled by β ends in the middle of an edge, its frequency is the same as that of the string $\beta\gamma$ labeling the path from the root to the fist node (black or red) below the end of β, and therefore $\delta(\beta\gamma) \geq \delta(\beta)$. So we are guaranteed, in order to find the maximal surprising strings, that we need to compute the index only for the nodes of the *BuST*. In addition the algorithm runs in a time proportional to the size of the *BuST* itself, which is subquadratic on average. Note also that the number of maximal surprising strings (modulo the approximations introduced by the relation) is of the same size of the *BuST*, so we are computing the z-score in optimal size and time.

6 Conclusions

We presented *BuST*, a new index structure for strings, which is an extension of Suffix Trees where the alphabet is enriched with a non-transitive relation, encapsulating some form of approximate information. This is the case, for instance, of a relation induced by the Hamming distance for an alphabet composed of macrocharacters on a base one. We showed that the average size of the tree is subquadratic, despite a quadratic worst case dimension, and we provided a construction algorithm linear in the size of the structure. In the final section, we discussed how *BuST* can be used for computing in a efficient way a class of measures of statistical approximate overrepresentation of substrings of a text α. We have also an implementation of the (naive) construction of the data structure in **C**, which we used to perform some tests on the size of *BuST*, showing that the bound given in Section 3 is rather pessimistic (cf. again [4]).

BuST allow to extract approximate information from a string α in a simple way, essentially in the same way exact information can be extracted from *ST*.

In addition, they are defined in an orthogonal way w.r.t. the relation and the alphabet used, hence they can be adapted in different contexts with minor efforts. Their main drawback is that the usage of a relation on the alphabet permits to encode only a localized version of approximate information, like global Hamming distance distributed evenly along strings.

Future directions include the exploration of other application domains, like using the information contained in *BuST* to build heuristics for the difficult consensus substring problem (cf. [7]).

References

1. A. Apostolico, M. E. Block, and Lonardi. Monotony of surprise and large-scale quest for unusual words. *Journal of Computational Biology*, 7(3–4):283–313, 2003.
2. A. Apostolico, M. E. Block, S. Lonardi, and X. Xu. Efficient detection of unusual words. *Journal of Computational Biology*, 7(1–2):71–94, 2000.
3. A. Apostolico and C. Pizzi. Monotone scoring of patterns with mismatches. In *Proceedings of WABI 2004*, 2004.
4. L. Bortolussi, F. Fabris, and A. Policriti. Bundled suffix trees. Technical report, Dept. of Maths and Informatics, University of Udine, 2006. `http://www.dimi.uniud.it/bortolus/techrep.htm`.
5. R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of STOC 2004*, pages 91–100, 2004.
6. D. Gusfield. *Algorithms on Trees, Strings and Sequences: Computer Science and Computational Biology*. Cambridge University Press, London, 1997.
7. L. Marsan and M. F. Sagot. Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification. In *Proceedings of RECOMB 2000*, pages 210–219, 2000.
8. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
9. G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.
10. W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Computing*, 22:1176–1198, 1993.
11. W. Szpankowski, P. Jacquet, and B. McVey. Compact suffix trees resemble patricia tries: Limiting distribution of depth. *Journal of the Iranian Statistical Society*, 3:139–148, 2004.