Dotted Suffix Trees
A Structure for Approximate Text Indexing

Luís Pedro Coelho[⋆] and Arlindo L. Oliveira

INESC-ID/IST
`{luis, aml}@algos.inesc-id.pt`

Abstract. In this work, the problem we address is text indexing for approximate matching. Given a text $\mathcal{T}$ which undergoes some preprocessing to generate an index, we can later query this index to identify the places where a string occurs up to a certain number of errors k (edition distance). The indexing structure occupies space $\mathcal{O}(n \log^k n)$ in the average case, independent of alphabet size. This structure can be used to report the existence of a match with k errors in $\mathcal{O}(3^k m^{k+1})$ and to report the occurrences in $\mathcal{O}(3^k m^{k+1} + ed)$ time, where m is the length of the pattern and *ed* and the number of matching edit scripts. The construction of the structure has time bound by $\mathcal{O}(kN|\Sigma|)$, where N is the number of nodes in the index and $|\Sigma|$ the alphabet size.

Keywords: string algorithms, suffix trees, approximate text matching, text indexing.

1 Introduction

Since their introduction [1], suffix trees have been one of the methods of choice for text indexing. However, in many real-life problems one is interested in finding places in the text where an approximate form of the pattern occurs. In 2001, Navarro et al presented a survey of existing approaches to solving this problem [2]. More recently, Maaß [3] presents both a survey of other work and his own solution, which occupies, on average, $\mathcal{O}(|\Sigma|^k n \log^k n)$[1] space for a search time $\mathcal{O}(m)$. In this work we present an approach based on an extension of suffix trees. The main advantage of this approach is that both the search and the index size are *alphabet independent* (although the indexing time is not).

The structure presented here is superficially very similar to the one presented by Chattaraj [4] as an *inexact suffix tree*, but that work has different objectives. Cole el al [5] present a structure whose initial intuition resemble ours in that it involves error trees. However, they make different time and space tradeoffs to achieve $\mathcal{O}(m + \log\log n + occ)$ searching (Hamming distance) with a size $\mathcal{O}(n\frac{\log^k n}{k!})$ index.

⋆ Supported by the Portuguese Science and Technology Foundation, project POSI/SRI/47778/2002 BioGrid.

[1] Maaß considers $|\Sigma|$ and k as constant and presents $\mathcal{O}(n \log^k n)$ as a complexity result. However, this analysis ignores the potentially large impact of alphabet size.

2 The Indexing Structure

Definition 1 (Character, string). *Given a set Σ, we say that S is a* string *over Σ if S is a (possibly empty) sequence of elements of Σ. Elements of Σ will be called* characters. *The length of the string S will be denoted by $|S|$. We shall write S_i for the i-th element of S.*

The set of all strings is denoted by Σ^* and $\Sigma^+ = \Sigma^* - \{empty\ string\}$.

For denoting characters we shall use letters from the beginning of the roman alphabet (a, b, c,...) and, for strings, we shall use letters from the end of the alphabet (w, x, ...). In what follows we assume that there are two special symbols ($ and .) which are not part of Σ.

Definition 2 (Concatenation, Prefix and Suffix). *wx or aw will denote the usual concatenation operation. If $S = wxy$, then w is a* prefix *of S, x is a* substring *of S and y is a* suffix *of S (at position $|wx|$).*

Definition 3 (Patricia tree, Suffix Tree, Suffix Link). *T is a Patricia tree if T is a rooted tree with edge labels from Σ^+. For each $a \in \Sigma$ and every node n in T, there exists at most one edge leaving n whose label starts with a. Each node in a Patricia tree has a path leading to it which forms a string. If the node n has the leading path w, we shall also refer to n as $\overline{w}$. A* compact *Patricia tree omits nodes with just one child.*

A suffix tree *for a string S is a compact Patricia tree whose leaf nodes (those without children) have paths corresponding to all suffixes of the string $S\$$. A suffix link in a suffix tree is a link from the node $\overline{aw}$ to the node $\overline{w}$. This link has the label a.*

In a suffix tree, all internal nodes have a well defined suffix link. McCreight's [6] algorithm constructs a suffix tree with suffix links in linear time.

Definition 4 (Occurrence Set, Position Set). *Given a node $\overline{w}$ in a suffix tree, we call its* occurrence set *the set of indexes in the original string where the string w occurs.*

Given a node $\overline{w}$ in a suffix tree, its position set *is the set formed by taking its occurrence set and adding the length of w to each element.*

Lemma 1 (Position set at the suffix node). *Given two nodes $\overline{aw}$ and $\overline{w}$, if one takes the position set of $\overline{w}$, subtracts one from each element, one obtains a superset of the position set of $\overline{w}$. The items shared by both sets are those positions of the string which contain an a.*

The lemma is fairly obvious given that the position set of $\overline{aw}$ contains all the positions where aw occurs which are exactly those positions where w occurs preceded by a.

Definition 5 (Aproximate Match). *We say that the string s matches the string t at position p with k errors if we can make k modifications in s to obtain s' which is a substring of t at position p. A modification is either deletion, insertion or substitution of one character.*

Definition 6 (Error Tree). *For any node $\overline{w}$, its error tree is formed by taking its position set, adding one to each element and forming the Patricia tree of the suffixes starting at those positions. If the position set includes the end of the string, that element is removed.*

The leaves are labeled by the position of the string in which their paths occur minus $|w| + 1$.

Definition 7 (1-error dotted Tree). *A* 1-error dotted tree *is the tree which is formed by adding to each node in a suffix tree, a new edge labeled by $\cdot$ which points to its error tree. The edge labeled $\cdot$ shall be called a* dot link.

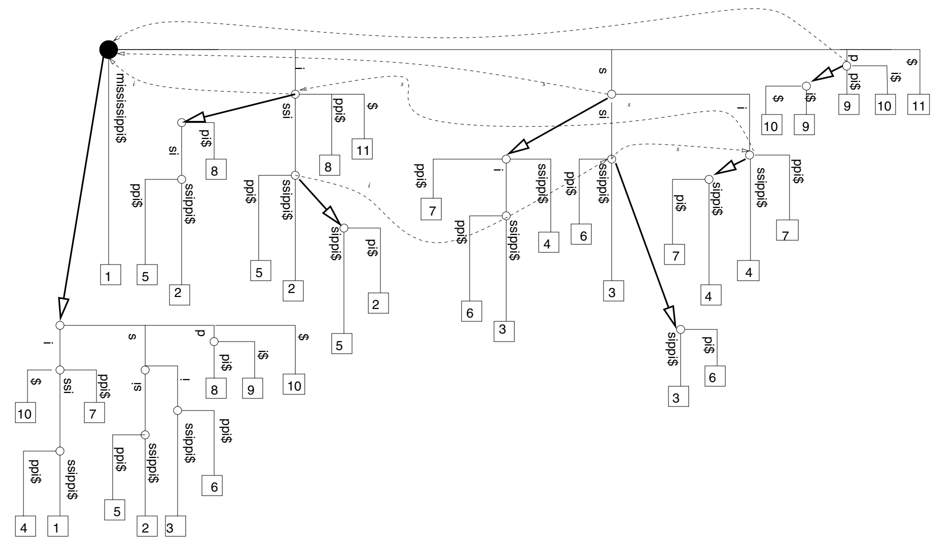


Fig. 1. 1-error dotted tree for *mississippi*

The 1-error dotted tree for *mississippi* is shown in Figure 1. The nodes are connected to their error trees by thick diagonal links. We can see some examples of the concepts above: for the node $\overline{issi}$, the occurrence set is $\{2, 5\}$ and its position set is $\{6, 9\}$. In a sense, one can say that *being* at node $\overline{issi}$ is being at positions 6 and 9 simultaneously. The error tree (at $\overline{issi}$) is formed by taking the strings starting at positions $\{7, 10\}$ (ie, *sippi\$* and *pi\$*) in a Patricia tree. In a leaf, the occurrence set is a singleton, and we label the leaf by its element.

The paths in the dotted tree are paths in the extended alphabet $\Sigma \cup \{., \$\}$. The notions of *occurrence set*, *position set* and *error tree* are valid for all nodes in a dotted tree.

Definition 8 (k-error dotted tree). *We define a k-error dotted tree as the tree obtained by adding error trees to each node in the $(k-1)$-error dotted tree which does not already contain one.*

3 Searching

Given a pattern to search for, we follow it character by character, descending the tree. We represent this walk by keeping a node and an offset from the start of its incoming link. Inside an edge, we consider that there is an implicit dot link which goes forward one character. At each point, we can take four possible actions: (1) **match**, where we descend according to the pattern (may not be possible); (2) **substitution**, where we follow the dot link (possibly implicit), moving in the pattern; (3) **insertion**, where we follow the (possibly implicit) dot link, *not moving* in the pattern; (4) **deletion**, where we advance in the pattern, while not moving in the tree. We limit ourselves to at most k non-matching operations (editions). Algorithm 1 implements the process just described.

Algorithm 1. Function findString($\overline{w}$, offset, s, k)

```
   Input: Current node $\overline{w}$
   Input: Current offset offset
   Input: String s
   Input: Maximum errors k
   Data: The tree's string treeString
1  if $k < 0$ then return *string not found*
2  if s *is empty* then *report all $\overline{w}$'s children*
3  findString($\overline{w}$,offset,s + 1,k − 1)// deletion
4  if offset = length($\overline{w}$) then
5      findString($\overline{w}$.dotLink, 0, s,k − 1)// insertion
6      findString($\overline{w}$.dotLink, 0, s + 1,k − 1)// substituition
7      child ←$\overline{w}$.getSon($s_0$)// try matching
8      if child *isn't null* then findString(child, 0, s + 1,k)
9  else
10     findString($\overline{w}$,offset +1,s,k-1)// insertion
11     if $s_0 \neq \text{treeString}_{\text{start}(\overline{w})+\text{offset}}$ then k ←k − 1
12     findString($\overline{w}$,offset +1,s + 1,k)// either match or substituition
```

There are at most $\sum_{i=1}^{k} \binom{m}{i} = \mathcal{O}(m^k)$ ways to combine k edit operations into a string of size m. Since there are 3 operations (substitution, insertion, and deletion), we have at most $\mathcal{O}(3^k m^k)$ sequences. Each sequence has at most $m + k = \mathcal{O}(m)$ elements and therefore the total time to find matches is $\mathcal{O}(3^k m^{k+1})$. Once a match has been found in the tree, reporting the leaves below the node takes time proportional to the number of leaves, ie. to the number of edit scripts which can be used to match the pattern to a substring of the text (which can be greater than the number of occurrences).[2] The total search time is $\mathcal{O}(3^k m^{k+1} + ed)$.

[2] As often happens, strings of the form a^m serve as examples of pathological behaviour as they can match any position of a string of form a^n in a large number of ways.

4 Constructing the Dotted Tree

We start with a suffix tree and show first how to construct a one-error dotted tree. We construct the error tree for the root which is almost a copy of the entire tree, except for two properties: (1) it does not have the leaf labeled 1 in the original tree and; (2) for any other leaf $\overline{w\$}$ occurring at position p in the string, we have a new leaf $\overline{.w\$}$ which occurs at position $p-1$ in the string. For any other node $\overline{aw}$, the error tree is a copy of the error tree at node $\overline{w}$ (the node pointed to by node $\overline{aw}$'s suffix link) with the following changes: (1) the leaf labeled 1 in the original error tree is not included; (2) leaves in the copy have a label which is the original value minus one; (3) a leaf labeled p is included only if $s_{p-1} = a$.

Algorithm 2. Copying a sub tree

```
Input: A node in a suffix tree $\overline{w}$
Input: An optional character $a$ (not given when copying the root)
Data: The original string string
1  copy ←make-copy($\overline{w}$)
2  if $\overline{w}$ is a leaf then
3      $p$ ←$\overline{w}$.label
4      if $p = 1$ then return null
5      if $a$ was not given or string$_{p-1} = a$ then
6          copy.label ←copy.label - 1
7  foreach $n \in \overline{w}$.sons do
8      copy of son ←copySubtree($n$,$a$)
9      if copy of son isn't null then
10         copy.sons ←copy.sons ∪ copy of son
11 if copy.sons is empty then return null
12 if copy.sons has only one element then
13     merge copy.sons into copy and return that
14 return copy
```

These conditions are an expression of Lemma 1 and an extension of the conditions for the root. Both are implemented by Algorithm 2. The only point to note is line 12. Since we filter some leaves, we can create nodes with only one child. These are removed by merging a child with its (single) parent. Since a typical suffix tree implementation just stores, at each node, indices to the start and end of the subtring labeling its incoming edge, merging is achieved by adjusting the start index. The construction of the tree using either Ukonnen's or McCreight's algorithm assures that this operation is correct.

Copying a tree takes time proportional to the number of nodes it contains. The error tree at the root is a straightforward copy of the whole tree. Every other error tree is a copy of an existing one. Since each node can have at most $|\Sigma|$ incoming suffix links, each error tree is transversed at most $|\Sigma|$ times. The sum of all these operations is therefore bounded by $|\Sigma|N$. Therefore, if the number of nodes in the final tree is N, construction is done in time $\mathcal{O}(N|\Sigma|)$.

The above algorithm can be used to construct trees with any number of errors by iterating it. To construct the $(k+1)$-error tree from the k-error tree, make an adjusted copy of the tree as above (adjusting leaves and filtering the leaves with label 1) and make this the new root error tree. Then, for every other node, remove the current error tree. Finally, for every node except the root, construct its error tree as above.

Let N_k be the number of nodes of the k-error dotted tree. We will use N for N_k if k is known from context. The analysis above remains valid and we now have that the time cost is $\mathcal{O}(N_1|\Sigma| + \ldots + N_k|\Sigma|) = \mathcal{O}(kN|\Sigma|)$.

5 Space Considerations

Let l be the maximum string depth of any node in the tree.[3] We show $N_k = \mathcal{O}(nl^k)$ by induction. It is known that $N_0 = \mathcal{O}(n)$. The algorithm for turning a k-error into a $(k+1)$-error dotted tree, can be looked at the following way[4]. First it constructs the error tree at the root and clears all the other error trees. Then it proceeds in stages, making a (possibly incomplete) copy of this tree spread amongst the nodes at string-depth 1. It processes the other nodes in increasing string-depths. At each string depth, the number of nodes is increased by a maximum of N_k. Therefore, we start with N_k nodes, make an almost full copy, and copy that at most l times, $N_{k+1} = \mathcal{O}(N_k(l+1))$. Assuming $N_k = \mathcal{O}(nl^k)$ by induction we conclude $N_{k+1} = \mathcal{O}(nl^{k+1})$.

So far, we have achieved little since in the worst case $l = n - 1$ (consider $aaaa\ldots$). However, under very general assumptions (which natural language textes and DNA experimentally verify), the expected case is $l = \mathcal{O}(\log n)$ [7] and we have $N_k = \mathcal{O}(n \log^k n)$.

6 Experimental Results

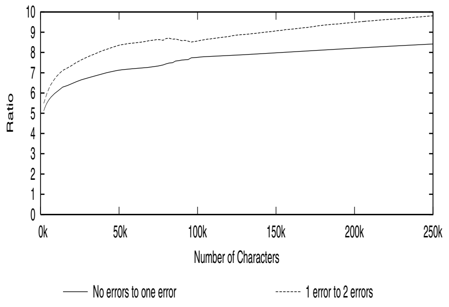
Three data sets were used: English text, the DNA of yeast, and randomly generated text. Results on all sets are qualitatively similar.

To experimentally verify the average case prediction, we show in Figure 2 the ratios between the k-error and the $(k+1)$-error dotted trees regarding the number of nodes in the trees. We can easily see that the experimental values do resemble a logarithm as predicted.

Searches were then performed on top of previously indexed text. We only report whether the string exists in the text (and not all occurrences). Therefore, the number of occurrences has no influence on the search time. Figure 3 shows

[3] For a node $\overline{w}$, its string depth is $|w|$.

[4] Having the node processed in this order is, in fact, difficult to code for. However, as an analysis tool, it is a valid assumption.



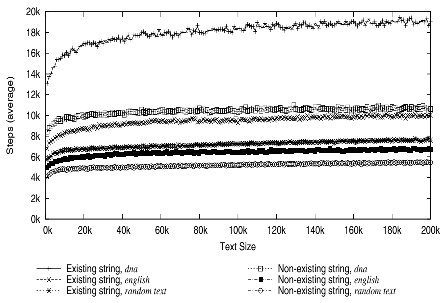


Fig. 2. Size ratio on English text

Fig. 3. Searching with 2 errors

the results of searching for 15 character long patterns with 2 errors, while varying the text size. After an initial small growth explainable by the increasing density of the tree, the search time is roughly constant.

7 Conclusions

We presented an indexing structure for approximate text matching which takes, on average, $\mathcal{O}(n \log^k n)$ space. This complexity was predicted theoretically and observed experimentally. This structure reports the existence of a match in $\mathcal{O}(3^k m^{k+1})$ and reports the positions where the matches occur in $\mathcal{O}(3^k m^{k+1} + ed)$ time. It can be constructed in $\mathcal{O}(kN|\Sigma|)$ time, N being the actual number of nodes. The structure and the algorithms to construct it are simple and easy to implement. The fact that the structure uses $\mathcal{O}(ed)$ time (instead of $\mathcal{O}(occ)$) to report the occurrences of a pattern may be a disadvantage in some applications. In other applications (eg, searching in DNA strings for degenerated occurrences of long strings), this will not be a problem since each occurrence will, in general, correspond to only one edit script.

The amount of space the index takes might limit its applicability. One direction for tackling this problem is the following remark: in the example for the string *mississippi*, presented in Figure 1, one can see that the tree below *s.i* and *ssi* are exactly the same. Whether such occurrences are the basis for a significant space saving and how to exploit them is an open question. Going further, the definition of error trees might be extended to structures such as the suffix-DAG presented by Gusfield [8, § 7.7].

Another limitation that should be addressed in future work is related with the fact that the complexity for reporting occurrences depends on the number of edit scripts, and not on the number of occurrences.

Acknowledgments. We thank L. Russo and S. Madeira for several productive discussions.

References

1. Weiner, P.: Linear pattern matching algorithms. In: FOCS, IEEE (1973) 1–11
2. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys **33** (2001)
3. Maaß, M.G., Nowak, J.: Text indexing with erros. In: Proc. 16th Annual Symp. on Combinatorial Pattern Matching (CPM). Volume 3537 of LNCS., Springer (2005) 21–32
4. Chattaraj, A., Parida, L.: An inexact-suffix-tree-based algorithm for detecting extensible patterns. Theor. Comput. Sci. **335** (2005) 3–14
5. Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC. (2004) 91–100
6. McCreight, E.: A space-economical suffix tree construction algorithm. J. ACM **23** (1976) 262–272
7. Apostolico, A., Szpankowski, W.: Self-alignments in words and their applications. J. Algorithms **13** (1992) 446–467
8. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, USA (1997)