A hard-disk based suffix tree implementation

Tristan Snowsill Florent Nicart

Abstract

Suffix trees are incredibly useful structures for computational genomics and combinatorial pattern matching. Due to the small alphabet sizes used in computational genomics, specialised hard-disk based suffix trees have been designed, but the problem of creating an efficient hard-disk based suffix tree for large and unbounded alphabet sizes remains essentially unsolved.

We have designed a hard-disk based *hybrid suffix tree*, residing on hard-disk and in RAM, which takes advantage of memory mapping, a method for treating data on a hard-disk transparently as though it was in memory. Memory mapping is provided by many modern operating systems. Through the use of memory mapping the implementation only loads a small amount of the suffix tree into working memory, which allows it to load faster and maintains a fairly efficient query speed.

The implementation is based on Ukkonen's suffix tree construction algorithm.

1 Introduction

Suffix trees are used widely in computational genomics and combinatorial pattern matching. They have remarkable properties which enable very efficient indexing and searching of sequences. For example, a large text document can be preprocessed in linear time and space, after which simple queries can be answered very efficiently, which makes suffix trees very useful for repeated analysis of a static sequence.

Suffix trees also offer competitive algorithms for solving problems such as the *longest common substring* and *longest common subsequence* problems. Solving these problems is often useful in genomic analysis.

As suffix trees are often used to analyse multiple entire genomes, they can exceed the amount of Random Access Memory (RAM) available on the computer. This problem is exacerbated by suffix trees being somewhat "bloated"

as they can require over ten times as much space as the original sequence. Research has shown that in-memory suffix tree implementations have poor memory locality. Poor memory locality is not a problem in the RAM model, where the cost of retrieving a block of data is independent of the address of that block in memory, but on hard-disks the cost of such retrievals are prohibitive. When working memory is exceeded, operating systems typically begin to use *paging* on the hard-disk, which can introduce a substantial drop in the performance of algorithms constructing or operating on the suffix tree.

The typical solution adopted by hard-disk based implementations for genomic data is to use an algorithm which is asymptotically less efficient but which in practice is much faster due to better memory locality. These implementations are typically only suitable for processing genomic data which is static, *i.e.*, additional data cannot be added to the suffix tree at a later time. For genomic data this is usually not a practical concern, but for suffix trees to be used in more diverse fields this presents a very real problem.

In this report we describe our implementation which is designed to be persistent, constructed online, efficient and to work with an unbounded alphabet size.

Outline In Section 2 we introduce formal definitions of suffix trees and describe Ukkonen's construction algorithm. In Section 3 we describe our hard-disk based implementation. In Section 4 we evaluate how well we meet the requirements laid out in Section 3.2.

2 Suffix trees and Ukkonen's construction algorithm

A *suffix tree* is a data structure that indexes all the substrings of a string; a *generalised suffix tree* is a similar data structure that indexes all the substrings of a set of strings. They are extremely efficient data structures, as they can be constructed using linear time and space, and their availability allows a number of apparently complex operations to be carried out with remarkable efficiency [10, 5].

2.1 Suffix trees

Let Σ be an alphabet and $|\Sigma|$ be the number of symbols in this alphabet. Let $S = s_0 s_1 \dots s_{n-1}$ be a string (or a sequence of symbols) on this alphabet. In the rest of the article, we will use the following notations: $|S| = n$ for the

length of S; $S[i..j] = s_i s_{i+1} \ldots s_{j-1} s_j$ with $0 \leq i \leq j < n$ for the substring of S starting at position i and ending at position j; $S[i]$ with $0 \leq i < n$ denotes the i^{th} symbol of S and $S_i = S[i..n-1]$ with $0 \leq i < n$ is the i^{th} suffix of S.

The *suffix trie* of an input string S is contains all the suffixes of $S\$$, where $\$ \notin \Sigma$ is a so-called *terminal symbol*. In particular, this trie is an edge-labelled tree where each of the edges is labelled with one symbol from $\Sigma \cup \$$, each node has at most one child node per alphabet symbol, and for each suffix there is a path starting at the root along which the suffix can be read. Note that $\$ \notin \Sigma$ implies that none of the suffixes can be a prefix of another suffix. Hence, each suffix ends in its own leaf node.

A suffix tree is stored more efficiently by using the following improvements. Firstly, edges can be labelled with sequences of characters instead of single characters. Then every internal node having only one child can be merged with its child after concatenating their labels (this turns the trie in a Patricia tree). Secondly, since each edge-label is a substring of $S\$$, we can further compress the suffix tree by storing the start and end positions of each edge label in $S\$$, rather than the actual substring.

Since each substring of a given string is a prefix of a suffix, all substrings of the string S can then be read off along the paths starting at the root, and ending at a certain point in the tree. It is in this way that a suffix tree indexes all substrings of the string it represents.

Suffix trees may also store a so-called *suffix link* for each node in the tree. If the path from the root to a node u reads ax (where a is a single character and x is a string) and the path from the root to a node v reads simply x then there is a suffix link from u to v.

Suffix trees have the following properties:

- every internal node has at least two children,
- the label of the edges leaving the same node start with a different letter,
- the tree has exactly $|S|$ leaves, each corresponding to one suffix of S: the path from the root to the i^{th} leaf is labelled by $S[i..n-1]$.

The latter property, together with the fact that the amount of space required for each node is constant, directly implies the linear space requirement of a suffix tree.

Figure 1 shows the suffix tree of the input string *BANANA*.

2.2 Generalised suffix trees

Generalised suffix trees are a simple generalisation of suffix trees, designed to store the suffixes of a set of input strings. In this case, leaves are labelled

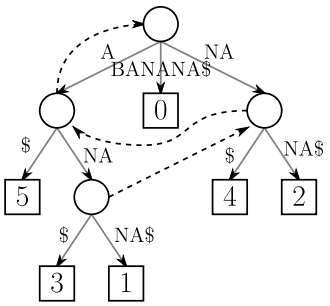

Figure 1: The suffix tree of the string *BANANA*\$. Dashed edges represent suffix links.

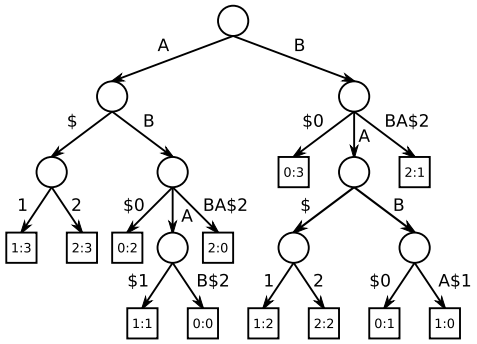

Figure 2: A generalised suffix tree made of the suffixes of the strings *ABAB*\$0, *BABA*\$1 and *ABBA*\$2. The first number on the label of a leaf is the sequence number and the second number is the starting position of the matching suffix.

with a pair composed of the start position of the matching suffix and the identifier of the input sequence, as shown in Figure 2. Generalised suffix trees can be built as a suffix tree by concatenating all strings, separated from each other by different terminal symbols.

2.3 Ukkonen's algorithm

Ukkonen's algorithm [9] is an efficient online suffix tree construction method. It processes one symbol of the input text at a time.

Let S be an input string of length $n = |S|$ and $\mathcal{T}$ be a generalised suffix tree. In order to add the n suffixes of S into $\mathcal{T}$, the Ukkonen algorithm will execute n phases during which it will perform none or several extensions. An *extension* j corresponds to the addition of the suffix j of S to the tree (*i.e.*,

$S[j..n-1]$). Let i be the current phase and j the current extension, then after browsing the tree with the sequence $S[j..i]$, three cases can arise:

1. the symbol $S[i]$ is matched from the current position; nothing is done in this phase and the extension j will be run again in the phase $i+1$,
2. there is no match for $S[i]$ and we are on a node; the current node will be extended,
3. there is no match for $S[i]$ and we are in the middle of an edge; the edge is split immediately before the non matching position and the new node will be extended

After cases 2 and 3, a new leaf labelled with the sequence number and the suffix position j is created and an edge is added from the current node to the new leaf with the label $S[i..n-1]$. Extensions are repeated until case 1 arises or $j = i$. Then the next phase starts. In the end, only n extensions are performed. Note that the algorithm is using the following optimisations to obtain linear time complexity:

- instead of browsing the tree from the root at the beginning of each extension, the algorithm uses the suffix link: it walks up from the position of the last extension to the first node above, follows the suffix link and browses down the target to the equivalent position;
- labels are compressed: instead of labelling every edge with a substring of S, every edge is labelled with a pair of integers (k, l) indicating the substring $S[k..l]$ of S that is the label of the edge.

3 A hard-disk based suffix tree implementation

In this section we describe our hard-disk based suffix tree implementation.

3.1 Intended use

We intend to use a suffix tree to analyse text written in English. While we could use a fixed size alphabet for this, *e.g.*, ASCII or UTF-8, we would rather have a "more natural" alphabet which is the set of English words. The reason this is more natural is that edges in the suffix tree will be labelled with complete words or phrases, rather than potentially containing sequences of letters which make little sense without further navigation of the tree.

Additionally using a large alphabet means that documents are encoding using shorter sequences.

In particular we will be analysing English text from news outlets. These outlets produce text regularly, meaning that at no point is the set of documents to be indexed fixed or finished, so our implementation must allow for additional documents to be added.

3.2 Requirements

We have a specific set of requirements which are not presently entirely met by existing available implementations.

Unbounded alphabet size The implementation must not place bounds on the size of the alphabet from which the sequences are drawn. The intended use of the suffix tree is to naturally index English text, using an alphabet of words rather than letters (were we to use an alphabet of letters the alphabet size could be reasonably constrained to 256, *e.g.*, by encoding in UTF-8). The English language itself grows, as does the set of words in common use (*i.e.*, dictionary words, names of people and organisations, misspellings), and the alphabet observed will also grow due to Heap's law.

Online construction The implementation must allow additional sequences to be added to the suffix tree. As the suffix tree will be used to process a text stream rather than a fixed corpus it is vital that the suffix tree implementation allows additional sequences to be added.

Persistency The suffix tree must be persistent, *i.e.*, it must be possible to preserve the suffix tree on hard-disk through machine shutdowns and so that backups can be made.

Scalability The suffix tree must not suffer substantial drops in performance as the size of the input is increased.

3.3 Existing implementations

Due to the usefulness of suffix trees for computational genomics much work has focussed on engineering implementations for genomic sequences, with alphabet sizes of 4 (DNA base pairs) or occasionally 20 (proteinogenic amino acids). Substantial achievements have been made in this field, but unfortunately these implementations are not practical for unbounded alphabet

Table 1: Capabilities of existing algorithms

Implementation	**Ukkonen** [9]	**Barsky (DiGeST)** [1]
Unbounded alphabet size	Yes	No
Online construction	Yes	No
Persistency	No	Yes
Scalability	No	Yes
Implementation	**Hunt** [6]	**Bedathur (TOP-Q)** [2]
Unbounded alphabet size	No	No
Online construction	No	Yes
Persistency	Yes	Yes
Scalability	Yes	Yes
Implementation	**Farach** [4]	**Clifford (DST)** [3]
Unbounded alphabet size	Yes	Yes
Online construction	No	Yes
Persistency	No	No
Scalability	No	Yes*

*Distributed Suffix Trees can scale efficiently if the cluster size can grow sufficiently

situations. In Table 1 we outline the capability of a selection of construction algorithms. We do not intend this to be a comprehensive listing, but we do not know of any implementations satisfying all of the criteria specified in Sec. 3.2.

3.4 Design

Our design acknowledges the difference between an efficient design for an in-memory suffix tree and for a hard-disk based suffix tree. We wanted to engineer an online construction algorithm, but one which could be hard-disk based both for persistency and to allow growth beyond physical memory size. We therefore used a hybrid tree, which could reside partly on persistent memory and partly in RAM. Parts of the suffix tree which are being actively constructed are formed in RAM while parts of the suffix tree which are unaffected remain untouched on a hard drive. There are three typical use cases for the suffix tree.

Creating a new suffix tree We create a new suffix tree in RAM and save it to hard disk after it is constructed. If the suffix tree grows too large for physical memory it can be saved prematurely to hard disk and then building

can continue as in "Updating a suffix tree".

Updating a suffix tree A suffix tree already resides on hard disk and we want to add more sequences to it. We build new parts of the suffix tree in RAM, as well as updating parts of the existing suffix tree by using memory mapping, which we detail in Sec. 3.5. Once the new part of the suffix tree has been built, it is integrated with the suffix tree on the hard drive and saved.

Reading and analysing a suffix tree The suffix tree resides on hard disk and we do not build any more of the suffix tree in memory. We load parts of the suffix tree into physical memory as necessary using memory mapping (see Sec. 3.5).

The best data representation for a hard drive based suffix tree can be very different to the best representation for a suffix tree in RAM. One of the first design choices when building a suffix tree implementation is whether to store outgoing edges as an array or a linked list. In the case where the alphabet size is fixed and very small, *e.g.*, 4 (for DNA base pairs), a fixed size array is a popular choice as it means constant time access to the child of a node (see Fig. 3). When the alphabet is fixed but larger a linked list implementation is often chosen, such that each node has a link to its first child as well as to its next sibling or its parent if it is the last of its siblings (see Fig. 3). With the linked list implementation there is an option to maintain the children in a sorted fashion, but this in practice offers no significant advantage [2].

When the alphabet size is unbounded, we can no longer iterate through the children of a node in constant time, so child lookup operations take asymptotically longer. For the suffix tree in memory we use the `std::set` data structure in C++ STL, which is implemented as a Red-Black tree (see Fig. 3). This allows children to be added to a node in $\mathcal{O}(\log|\Sigma|)$ time and for a children to be located in $\mathcal{O}(\log|\Sigma|)$ time.

We choose to use a variable sized array to store outgoing edges for the hard drive based suffix tree (see Fig. 3). This array is stored separately to the node, which allows each node to have a fixed size by including pointers to the start and end of the array of children in a file dedicated to storing outgoing edges for all nodes. These arrays are stored in a sorted fashion, sorting by the first character of the outgoing edge (which is unique for all outgoing edges of a node). As the arrays are sorted we can use a binary search to find a particular child of a node. Furthermore, by ordering the nodes of the tree we can also apply an order to the entire file by sorting first by node ordering and then by first character. We cannot directly add new

children to any nodes but a child can be located in $\mathcal{O}(\log |\Sigma|)$ time.

We also choose to fuse edges with their terminating nodes, as each node will only have one incoming edge (due to the tree structure). We refer to a node and its incoming edge as a "stick". There are then two types of sticks, internal sticks and leaf sticks. Each internal stick must know where its children are (these children could themselves be internal or leaf sticks).

We eventually have the following data structures: `Child`, `LeafData`, `InternalStick` and `LeafStick`, as shown in Fig. 4. These data structures are all fixed size, which means that they can be stored in a flat file and indexed in constant time. The data structures also take advantage of the fact that a block of many records can be read in almost the same time as a single record by having good memory locality – all the children of a node are contiguous in the `Child` data file.

3.5 Memory mapping

When we want to access a suffix tree on hard-disk, we often do not want to iterate through the entire suffix tree. The suffix tree must also be allowed to scale beyond the size of physical memory. This means that we often want to load a part of the suffix tree into memory, but not the whole tree. This is possible using traditional I/O methods of seeking and reading, but it is far more efficient instead to use *memory mapping*.

Memory mapping is a feature supported by modern operating systems which allows an entire file to be "virtually" loaded into memory. In the case where the file contains fixed size records the file can be treated as a simple array of records which can be accessed just as a normal array. When the file is memory mapped no memory is actually reserved, nor is any of the file actually read, but for every read/write access the *memory management unit* (MMU) will ensure that the corresponding part of the file is actually present in physical memory before the operation is performed. If a program attempts to access a part of the file which has not yet been loaded then a memory page will be allocated and the page loaded from the file. When a page is no longer needed it can be freed, or if write accesses have taken place these modifications are made to the file.

Memory mapping has some drawbacks: it is not efficient when a file is very small (which is not the case here), it is dangerous to use when the underlying file may change, it cannot change the size of the file. It is possible to set up a locking system to prevent the file being changed while it is memory mapped. As memory mapping cannot change the size of the file, if it necessary to increase the size of the file this can be achieved by traditional append file operations or (in the case that the file needs to remain sorted) the old file

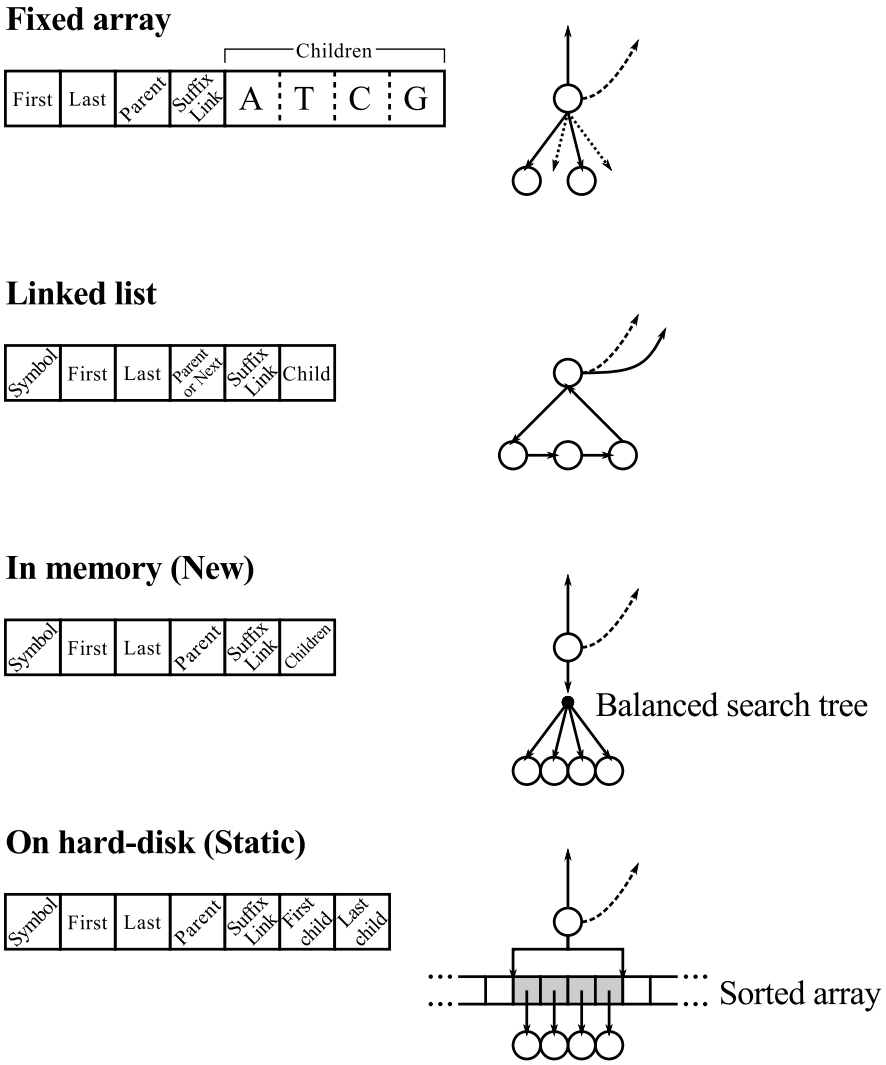

Figure 3: Different node representations (internal nodes). A fixed array is appropriate when there is a very small, fixed alphabet, such as with DNA. A linked list can be appropriate when the alphabet size is constant as a linear search for a child can be executed in constant time and insertions are also constant time operations. When the alphabet size is not fixed the optimal performance is $\mathcal{O}(\log |\Sigma|)$ time (without the use of a hashing scheme) to find a child. In memory this is achieved by using a balanced search tree, such as a Red-Black tree. On hard disk this is achieved using a sorted array, on which we can perform a binary search.

First symbol	4 bytes*
Child stick ID	4 bytes

(a) `Child`

Sequence ID	4 bytes
Suffix position	4 bytes

(b) `LeafData`

Parent stick ID	4 bytes
Sequence ID	4 bytes
Sequence start	4 bytes
Sequence end	4 bytes
Suffix link	4 bytes
Child start†	4 bytes

(c) `InternalStick`

Parent stick ID	4 bytes
Sequence ID	4 bytes
Sequence start	4 bytes
Sequence end	4 bytes
Leaf data start†	4 bytes

(d) `LeafStick`

Figure 4: Data structures representing the suffix tree. 4 bytes is generally enough for all the different components (by around three orders of magnitude), allowing up to $2^{32} = 4\,294\,967\,296$ different values for the variables. *Using fixed space does mean that the alphabet is bounded for a specific suffix tree, but this can comfortably be enlarged to 8 or 12 bytes, giving up to 7.92×10^{28} symbols. †We do not need to store the end of the range of children or leaf data because the last child will come immediately before the first child of the next `InternalStick` (similarly for leaf data).

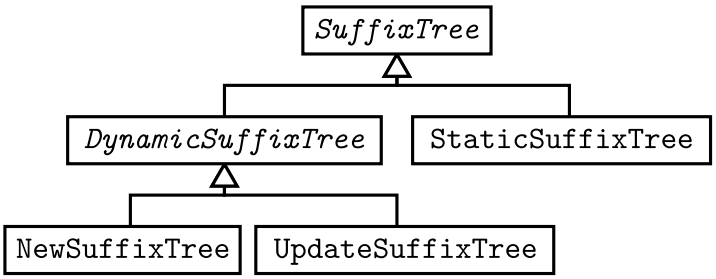

Figure 5: Suffix tree inheritance diagram. *SuffixTree* is the abstract base class and *DynamicSuffixTree* is an abstract class for suffix trees to which new sequences can be added. `StaticSuffixTree` is the class used to read a suffix tree from hard disk. `NewSuffixTree` is the class used to create a suffix tree in memory. `UpdateSuffixTree` is used to update a suffix tree on disk.

and new data can be merged and written to a new file.

3.6 Hybrid suffix tree

Because of the inherent differences between a suffix tree in memory and one on hard-disk we use a *hybrid suffix tree* when adding sequences to a suffix tree on hard-disk. Rather than loading the whole suffix tree into memory (which may not be possible) or attempting to modify the suffix tree in its hard-disk representation, instead we create an extension to the suffix tree in memory and when we are finished we save the whole suffix tree to disk. Figure 5 describes the C++ classes used in our implementation.

The `UpdateSuffixTree` is the hybrid data structure. It contains both a `StaticSuffixTree` and a `NewSuffixTree`.

Sticks are indexed by integers, with negative integers being used for leaf sticks and non-negative integers being used for internal sticks. `InternalStick` records and `LeafStick` records are stored separately such that for each, new sticks can be appended to the file without needing to move any records (as would be the case for insertions). `Child` and `LeafData` records need to be stored sorted to allow for binary searches, so when the `UpdateSuffixTree` is saved to disk these files are rewritten. Adding a sequence of length n (or a number of sequences of total length n) to a suffix tree indexing sequences of total N and saving to disk takes $\mathcal{O}(N + n \log |\Sigma|)$ time but the coefficient of $\mathcal{O}(N)$ is small as it involves simply iterating through data structures with good memory locality and writing to a file.

4 Evaluation

In this section we briefly evaluate whether our suffix tree meets our requirements as detailed in 3.2.

Unbounded alphabet size Our implementation achieves optimal performance given an unbounded alphabet, which is that edges can be found in $\mathcal{O}(\log |\Sigma|)$ time. While for a specific implementation the alphabet size will be bounded by the space allocated for storing each symbol, there is no reason why a suffix tree with 4-byte symbol allocation could not be converted to a suffix tree with 8-byte symbol allocation.

Online construction Our implementation is based on Ukkonen's algorithm, which is itself an online suffix tree construction algorithm [9]. New sequences can be added using the *hybrid suffix tree* design described in Section 3.6. When it is necessary to save to persistent memory, this takes time proportional to the size of the whole tree (rather than the size of the extension in memory) but these are fast operations.

Persistency Our implementation allows a suffix tree to be saved completely to hard-disk. It can be backed up using traditional methods, such as the Unix `tar` command.

Scalability The suffix tree can comfortably grow bigger than physical memory. Once the size of the in-memory suffix tree grows too large it can be saved to disk, freeing up memory entirely, and can then continue to grow.

5 Conclusions

In this report we have presented our implementation of a hard-disk based suffix tree for indexing text from a large unbounded alphabet in an online fashion. We have been using the suffix tree for a number of years, including, but not limited to, in research detailed in [8, 7].

References

[1] Marina Barsky, Ulrike Stege, Alex Thomo, and Chris Upton. A new method for indexing genomes using on-disk suffix trees. In *CIKM 2008*. ACM New York, NY, USA, 2008.

[2] S.J. Bedathur and J.R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proceedings of the 20th International Conference on Data Engineering (ICDE04)*, volume 1063, 2004.

[3] R. Clifford. Distributed suffix trees. *Journal of Discrete Algorithms*, 3(2-4):176–197, 2005.

[4] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science, FOCS*, volume 97. Citeseer, 1996.

[5] D Gusfield. *Algorithms on strings, trees and sequences*. Cambridge University Press, 1997.

[6] E Hunt, M.P. Atkinson, and R.W. Irving. A database index to large biological sequences. In *In VLDB*, volume 7, pages 139–148, 2001.

[7] Tristan Snowsill, Ilias Flaounas, Tijl De Bie, and Nello Cristianini. Detecting Events in a Million New York Times Articles. In José Balcázar, Francesco Bonchi, Aristides Gionis, and Michèle Sebag, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 6323 of *Lecture Notes in Computer Science*, pages 615–618. Springer Berlin / Heidelberg, 2010.

[8] Tristan Snowsill, Florent Nicart, Marco Stefani, Tijl De Bie, and Nello Cristianini. Finding surprising patterns in textual data streams. In *CIP2010: 2010 IAPR Workshop on Cognitive Information Processing*, pages 405–410, 2010.

[9] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[10] P. Weiner. Linear pattern matching algorithms. In *IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory, 1973. SWAT'08*, pages 1–11, 1973.