Vol. 00 no. 00 2006
Pages 1–2

Compressed Suffix Tree — A Basis for Genome-scale Sequence Analysis

Niko Välimäki [a], Wolfgang Gerlach [b], Kashyap Dixit [c], and Veli Mäkinen [a]

[a]Department of Computer Science,
P.O. Box 68 (Gustaf Hällströmin katu 2b), FI-00014 University of Helsinki, Finland.
[b]Technische Fakultät, Universität Bielefeld, Germany.
[c]Indian Institute of Technology, Kanpur, India.

ABSTRACT
Summary Suffix tree is one of the most fundamental data structures in string algorithms and biological sequence analysis. Unfortunately, when it comes to implementing those algorithms and applying them to real genomic sequences, often the main memory size becomes the bottleneck. This is easily explained by the fact that while a DNA sequence of length n from alphabet $\Sigma = \{A, C, G, T\}$ can be stored in $n \log |\Sigma| = 2n$ bits, its suffix tree occupies $O(n \log n)$ bits. In practice, the size difference easily reaches factor 50.

We provide an implementation of the *compressed suffix tree* very recently proposed by Sadakane (*Theory of Computing Systems*, in press). The compressed suffix tree occupies space proportional to the text size, i.e. $O(n \log |\Sigma|)$ bits, and supports all typical suffix tree operations with at most $\log n$ factor slowdown. Our experiments show that, e.g. on a 10 MB DNA sequence, the compressed suffix tree takes 10% of the space of normal suffix tree. Typical operations are slowed down by factor 60.
Availability: The C++ implementation under GNU license is available at `http://www.cs.helsinki.fi/group/suds/cst/`. An example program implementing a typical pattern discovery task is included. Experimental results in this note correspond to version 0.95.
Contact: vmakinen@cs.helsinki.fi

1 TEXT

Myriad non-trivial combinatorial questions concerning strings turn out to have efficient solutions via extensive use of *suffix trees*. This is no surprise, since suffix trees summarize the whole substring content of a *text* string in an economic way; suffix trees contain a root to leaf path for each suffix of the text such that each substring of the text can be read as a prefix of some path. Edges of the tree are labeled with text substrings, and can be represented by pointers to the text. The tree has n leaves and at most $n - 1$ internal nodes, and hence pointers in the tree occupy overall $O(n)$ computer words, n being the text length. This linear space dependency has made suffix trees attractive for many applications.

Bioinformatics is a field where suffix trees would seem to have the strongest potential; unlike the natural language texts formed by words and delimiters, biological sequences are streams of symbols without any predefined word boundaries, but rather containing yet more hidden motifs and structures. Suffix trees treat any substring equally, regardless of it being a word or not. This perfect synergy has created a vast literature describing suffix tree -based algorithms for sequence analysis problems. Several implementations exist as well (see `http://en.wikipedia.org/wiki/Suffix_tree`).

Unfortunately, the theoretically attractive properties of suffix trees do not always meet the practical realm. The main reason why suffix trees have remained mainly as theoretical tools is their immense space consumption. Even for a reasonable size genomic sequence of 100 MB, its suffix tree may require $5GB$ of main memory. This phenomenon is not just a consequence of constant factors in the implementation of the structure, but rather an asymptotic effect. When examined more carefully, one notices that a sequence of length n from an alphabet Σ requires only $n \log |\Sigma|$ *bits* of space, whereas its suffix tree requires $O(n \log n)$ bits. Hence, the space requirement is by no means linear when measured in bits.

The size bottleneck of suffix trees has made the research turn into looking for more space-economic variants of suffix trees. One popular alternative is the *suffix array*. It basically removes the constant factor of suffix trees to 1, as what remains from suffix trees is a lexicographically ordered array of starting positions of suffixes in the text. That occupies $n \log n$ bits. Many tasks on suffix trees can be simulated by $\log n$ factor slowdown using suffix arrays.

A recent twist in the area of text indexes is the extensive use of *abstract data structures*; the operations supported by a data structure are identified and the best possible implementation is sought for that supports those operations. This line of development has led to *compressed suffix arrays*. These data structures take, in essence, $n \log |\Sigma|(1 + o(1))$ bits of space. More importantly, they simulate suffix array operations with logarithmic slowdowns, and support some operations (like pattern search) even faster than plain suffix arrays or suffix trees. These structures are also called *self-indexes* as they do not need the text to function; the text is represented compressed within the index.

Very recently Sadakane (*Theory of Computing Systems*, in press) extended the abstract data structure concept to cover suffix trees, identifying typical operations suffix trees are assumed to possess. Some of these operations, like navigating in a tree, were already extensively studied by other people. In addition to these navigational operations, suffix trees have several other useful operations such as suffix links, constant time lowest common ancestor (lca) queries, possibility to attach additional information to each node/edge, and pattern search capabilities. Sadakane developed a fully functional suffix tree structure by combining compressed suffix arrays with several other non-trivial new structures. Each operation was supported by at most $\log n$ slowdown, often the slowdown being only constant. The space requirement was shown to be still asymptotically optimal, more accurately, $|CSA| + 6n + o(n)$ bits, where $|CSA|$ is the size of the compressed suffix array used.

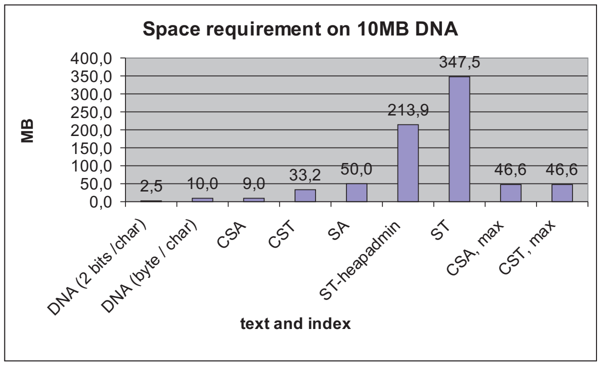


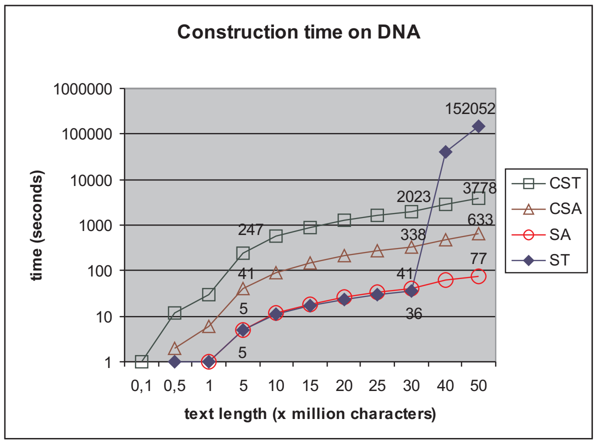
Fig. 1. Comparison of space requirements. We have added the text size to the SA and ST sizes, as they need the text to function as indexes, whereas CSA and CST work without. Here `ST-heapadmin` is the space used by suffix tree without the heap administration overhead; this large overhead is caused due to the allocation of many small memory fragments. For other indexes, this overhead is negligible. Two last values on the right report the maximum space usage during the construction; maximum is reached already during the construction of the compressed suffix array.

We implemented the structure following closely the original proposal. The time-requirement of our construction algorithm is $O(n \log n \log |\Sigma|)$. The construction uses the same asymptotic space as the final structure. The final structure supports all the mentioned suffix tree operations, each with at most $O(\log n \log |\Sigma|)$ slowdown. The implementation details and references to earlier work can be found in the technical report that can downloaded from the same page as the software.

We report experimental results on a 50 MB DNA sequence (`http://pizzachili.dcc.uchile.cl/texts/dna/dna.50MB.gz`).We used a version of the compressed suffix tree CST whose theoretical space requirement is $nH_0 + 10n + o(n \log |\Sigma|)$ bits ($H_0 \leq \log |\Sigma|$ being the zeroth order entropy of the sequence); other variants are possible by adjusting the space/time tradeoff parameters. Here $n(H_0 + 1)(1 + o(1)) + 3n$ comes from the compressed suffix array CSA, and $6n + o(n)$ from the other structures. The maximum slowdown on suffix tree operations is $O(\log n)$ under this tradeoff.

We compared the space usage against classical text indexes: a standard pointer-based implementation of suffix trees ST, and a standard suffix array SA were used. Figure 1 reports the space requirements on a 10 MB prefix of the sequence. One can see that the achieved space-requirement is attractive; CST takes less space than a plain suffix array, even when the extra space used during construction is added.

For the time requirement comparison, we measured both the construction time and the usage time (see Fig. 2). For the latter, we implemented a well-known solution to the *longest common substring (LCSS)* problem using both the classical suffix tree and the compressed suffix tree. The LCSS problem asks to find the longest substring C shared by two given input strings A and B. One can solve it by constructing the suffix tree of the concatenation $A\$B$, searching for the node whose string depth is largest and its subtree contains both a suffix from A and from B. For sanity check, we also implemented an $O(n^3)$ ($O(n^2)$ expected case) brute-force algorithm.



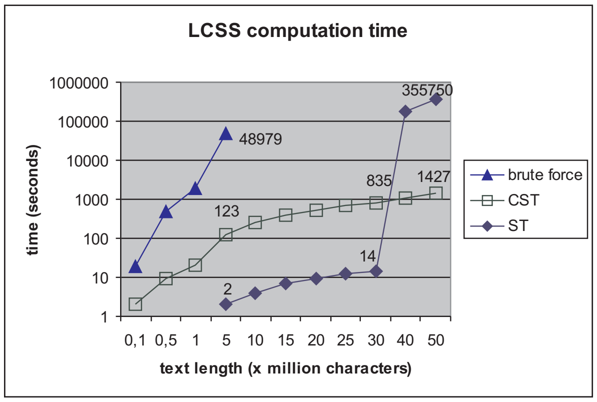


Fig. 2. Comparison of time requirements. For LCSS computation, we treated the first half of the sequence as A, the second as B. The irregular sampling of text lengths up to 5 MB is chosen in order to visualize the behavior of the brute-force algorithm.

The dramatic change in the behavior of suffix tree on sequences larger than 30 MB is due to running out of main memory; this confirms the evident fact that suffix trees in external memory are way slower than compressed suffix trees in main memory.

The experiments show that even though the compressed suffix tree is significantly slower than a classical suffix tree when both fit in main memory, it has an important application domain on genome-scale analysis tasks; when memory is the bottleneck for using classical suffix trees and brute-force solutions too slow, compressed suffix trees can provide a new opportunity to solve the problem at hand without running out of space or time.

ACKNOWLEDGMENT

The software builds on a compressed suffix array implementation by Veli Mäkinen and Rodrigo González and a balanced parentheses implementation by Gonzalo Navarro. We wish to thank them for providing us a basis. The discussions with Kunihiko Sadakane, Wing-Kai Hon, and Juha Kärkkäinen gave us better insights how to design the implementation.

The project was funded by the Academy of Finland under grant 108219, and by the From Data to Knowledge -research unit.