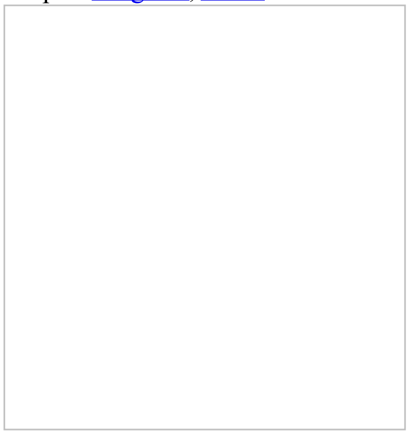
Suffix tree

From Wikipedia, the free encyclopedia
Jump to: navigation, search


Suffix tree for the text `BANANA`. Each substring is terminated with special character `$`. The six paths from the root to the leaves (shown as boxes) correspond to the six suffixes `A$`, `NA$`, `ANA$`, `NANA$`, `ANANA$` and `BANANA$`. The numbers in the leaves give the start position of the corresponding suffix. Suffix links, drawn dashed, are used during construction.

In computer science, a **suffix tree** (also called **PAT tree** or, in an earlier form, **position tree**) is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. Suffix trees allow particularly fast implementations of many important string operations.

The construction of such a tree for the string takes time and space linear in the length of . Once constructed, several operations can be performed quickly, for instance locating a substring in , locating a substring if a certain number of mistakes are allowed, locating matches for a regular expression pattern etc. Suffix trees also provide one of the first linear-time solutions for the longest common substring problem. These speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself.

Contents

History[edit]

The concept was first introduced by Weiner (1973), which Donald Knuth subsequently characterized as "Algorithm of the Year 1973". The construction was greatly simplified by McCreight (1976) , and also by Ukkonen (1995).[1] Ukkonen provided the first online-construction of suffix trees, now known as Ukkonen's algorithm, with running time that matched the then fastest algorithms. These algorithms are all linear-time for a constant-size alphabet, and have worst-case running time of in general.

Farach (1997) gave the first suffix tree construction algorithm that is optimal for all alphabets. In particular, this is the first linear-time algorithm for strings drawn from an alphabet of integers in a polynomial range. Farach's algorithm has become the basis for new algorithms for constructing both suffix trees and suffix arrays, for example, in external memory, compressed, succinct, etc.

Definition[edit]

The suffix tree for the string of length is defined as a tree such that:[2]

- The tree has exactly n leaves numbered from 1 to n.
- Except for the root, every internal node has at least two children.
- Each edge is labeled with a non-empty substring of S.
- No two edges starting out of a node can have string-labels beginning with the same character.
- The string obtained by concatenating all the string-labels found on the path from the root to leaf i spells out suffix S[i..n], for i from 1 to n.

Since such a tree does not exist for all strings, is padded with a terminal symbol not seen in the string (usually denoted `$`). This ensures that no suffix is a prefix of another, and that there will be leaf nodes, one for each of the suffixes of . Since all internal non-root nodes are branching, there can be at most $n-1$ such nodes, and $n + (n-1) + 1 = 2n$ nodes in total (n leaves, $n-1$ internal non-root nodes, 1 root).

Suffix links are a key feature for older linear-time construction algorithms, although most newer algorithms, which are based on Farach's algorithm, dispense with suffix links. In a complete suffix tree, all internal non-root nodes have a suffix link to another internal node. If the path from the root to a node spells the string , where is a single character and is a string (possibly empty), it has a suffix link to the internal node representing . See for example the suffix link from the node for `ANA` to the node for `NA` in the figure above. Suffix links are also used in some algorithms running on the tree.

Generalized suffix tree[edit]

A generalized suffix tree is a suffix tree made for a set of words instead of only for a single word. It represents all suffixes from this set of words. Each word must be terminated by a different termination symbol or word.

Functionality[edit]

A suffix tree for a string of length can be built in time, if the letters come from an alphabet of integers in a polynomial range (in particular, this is true for constant-sized alphabets).[3] For larger alphabets, the running time is dominated by first sorting the letters to bring them into a range of size ; in general, this takes time. The costs below are given under the assumption that the alphabet is constant.

Assume that a suffix tree has been built for the string of length , or that a generalised suffix tree has been built for the set of strings of total length . You can:

- Search for strings:
  - Check if a string of length is a substring in time.[4]
  - Find the first occurrence of the patterns of total length as substrings in time.
  - Find all occurrences of the patterns of total length as substrings in time.[5]
  - Search for a regular expression *P* in time expected sublinear in .[6]
  - Find for each suffix of a pattern , the length of the longest match between a prefix of and a substring in in time.[7] This is termed the **matching statistics** for .
- Find properties of the strings:
  - Find the longest common substrings of the string and in time.[8]
  - Find all maximal pairs, maximal repeats or supermaximal repeats in time.[9]
  - Find the Lempel–Ziv decomposition in time.[10]
  - Find the longest repeated substrings in time.
  - Find the most frequently occurring substrings of a minimum length in time.
  - Find the shortest strings from that do not occur in , in time, if there are such strings.
  - Find the shortest substrings occurring only once in time.
  - Find, for each , the shortest substrings of not occurring elsewhere in in time.

The suffix tree can be prepared for constant time lowest common ancestor retrieval between nodes in time.[11] One can then also:

- Find the longest common prefix between the suffixes and in .[12]
- Search for a pattern *P* of length *m* with at most *k* mismatches in time, where *z* is the number of hits.[13]
- Find all maximal palindromes in ,[14] or time if gaps of length are allowed, or if mismatches are allowed.[15]
- Find all tandem repeats in , and *k*-mismatch tandem repeats in .[16]
- Find the longest common substrings to at least strings in for in time.[17]
- Find the longest palindromic substring of a given string (using the generalized suffix tree of the string and its reverse) in linear time.[18]

Applications[edit]

Suffix trees can be used to solve a large number of string problems that occur in text-editing, free-text search, computational biology and other application areas.[19] Primary applications include:[19]

- String search, in *O(m)* complexity, where *m* is the length of the sub-string (but with initial *O(n)* time required to build the suffix tree for the string)
- Finding the longest repeated substring
- Finding the longest common substring
- Finding the longest palindrome in a string

Suffix trees are often used in bioinformatics applications, searching for patterns in DNA or protein sequences (which can be viewed as long strings of characters). The ability to search efficiently with mismatches might be considered their greatest strength. Suffix trees are also used in data compression; they can be used to find repeated data, and can be used for the sorting stage of the Burrows–Wheeler transform. Variants of the LZW compression schemes use suffix trees (LZSS). A suffix tree is also used in suffix tree clustering, a data clustering algorithm used in some search engines.[20]

Implementation[edit]

If each node and edge can be represented in space, the entire tree can be represented in space. The total length of all the strings on all of the edges in the tree is , but each edge can be stored as the position and length of a substring of *S*, giving a total space usage of computer words. The worst-case space usage of a suffix tree is seen with a fibonacci word, giving the full nodes.

An important choice when making a suffix tree implementation is the parent-child relationships between nodes. The most common is using linked lists called **sibling lists**. Each node has a pointer to its first child, and to the next node in the child list it is a part of. Other implementations with efficient running time properties use hash maps, sorted or unsorted arrays (with array doubling), or balanced search trees. We are interested in:

- The cost of finding the child on a given character.
- The cost of inserting a child.
- The cost of enlisting all children of a node (divided by the number of children in the table below).

Let be the size of the alphabet. Then you have the following costs:

	Lookup	Insertion	Traversal
Sibling lists / unsorted arrays			
Bitwise sibling trees			
Hash maps			
Balanced search tree			
Sorted arrays			
Hash maps + sibling lists			

Note that the insertion cost is amortised, and that the costs for hashing are given for perfect hashing.

The large amount of information in each edge and node makes the suffix tree very expensive, consuming about 10 to 20 times the memory size of the source text in good implementations. The suffix array reduces this requirement to a factor of 8 (for array including LCP values built within 32-bit address space and 8-bit characters.) This factor depends on the properties and may reach 2 with usage of 4-byte wide characters (needed to contain any symbol in some UNIX-like systems, see wchar t) on 32-bit systems. Researchers have continued to find smaller indexing structures.

External construction[edit]

Though linear, the memory usage of a suffix tree is significantly higher than the actual size of the sequence collection. For a large text, construction may require external memory approaches.

There are theoretical results for constructing suffix trees in external memory. The algorithm by Farach-Colton, Ferragina & Muthukrishnan (2000) is theoretically optimal, with an I/O complexity equal to that of sorting. However the overall intricacy of this algorithm has prevented, so far, its practical implementation.[21]

On the other hand, there have been practical works for constructing disk-based suffix trees which scale to (few) GB/hours. The state of the art methods are TDD,[22] TRELLIS,[23] DiGeST,[24] and B^{2}ST.[25]

TDD and TRELLIS scale up to the entire human genome – approximately 3GB – resulting in a disk-based suffix tree of a size in the tens of gigabytes.[22][23] However, these methods cannot handle efficiently collections of sequences exceeding 3GB.[24] DiGeST performs significantly better and is able to handle collections of sequences in the order of 6GB in about 6 hours.[24] . All these methods can efficiently build suffix trees for the case when the tree does not fit in main memory, but the input does. The most recent method, B^2ST,[25] scales to handle inputs that do not fit in main memory. ERA is a recent parallel suffix tree construction method that is significantly faster. ERA can index the entire human genome in 19 minutes on an 8-core desktop computer with 16GB RAM. On a simple Linux cluster with 16 nodes (4GB RAM per node), ERA can index the entire human genome in less than 9 minutes.[26]

See also[edit]

Wikimedia Commons has media related to ***Suffix tree***.

- Suffix array
- Generalised suffix tree
- Trie

Notes[edit]

1. ^ Giegerich & Kurtz (1997).
2. ^ http://www.cs.uoi.gr/~kblekas/courses/bioinformatics/Suffix_Trees1.pdf
3. ^ Farach (1997).
4. ^ Gusfield (1999), p.92.
5. ^ Gusfield (1999), p.123.
6. ^ Baeza-Yates & Gonnet (1996).
7. ^ Gusfield (1999), p.132.
8. ^ Gusfield (1999), p.125.
9. ^ Gusfield (1999), p.144.
10. ^ Gusfield (1999), p.166.
11. ^ Gusfield (1999), Chapter 8.
12. ^ Gusfield (1999), p.196.
13. ^ Gusfield (1999), p.200.
14. ^ Gusfield (1999), p.198.
15. ^ Gusfield (1999), p.201.
16. ^ Gusfield (1999), p.204.
17. ^ Gusfield (1999), p.205.
18. ^ Gusfield (1999), pp.197–199.
19. ^ *a* *b* Allison, L. "Suffix Trees". Retrieved 2008-10-14.
20. ^ First introduced by Zamir & Etzioni (1998).
21. ^ Smyth (2003).
22. ^ *a* *b* Tata, Hankins & Patel (2003).
23. ^ *a* *b* Phoophakdee & Zaki (2007).
24. ^ *a* *b* *c* Barsky et al. (2008).
25. ^ *a* *b* Barsky et al. (2009).
26. ^ Mansour et al. (2011).

References[edit]

- Baeza-Yates, Ricardo A.; Gonnet, Gaston H. (1996), "Fast text searching for regular expressions or automaton searching on tries", *Journal of the ACM* **43** (6): 915–936, doi:10.1145/235809.235810.
- Barsky, Marina; Stege, Ulrike; Thomo, Alex; Upton, Chris (2008), "A new method for indexing genomes using on-disk suffix trees", *CIKM '08: Proceedings of the 17th ACM Conference on Information and Knowledge Management*, New York, NY, USA: ACM, pp. 649–658.
- Barsky, Marina; Stege, Ulrike; Thomo, Alex; Upton, Chris (2009), "Suffix trees for very large genomic sequences", *CIKM '09: Proceedings of the 18th ACM Conference on Information and Knowledge Management*, New York, NY, USA: ACM.
- Farach, Martin (1997), "Optimal Suffix Tree Construction with Large Alphabets" (PDF), *38th IEEE Symposium on Foundations of Computer Science (FOCS '97)*, pp. 137–143.
- Farach-Colton, Martin; Ferragina, Paolo; Muthukrishnan, S. (2000), "On the sorting-complexity of suffix tree construction.", *Journal of the ACM* **47** (6): 987–1011, doi:10.1145/355541.355547.
- Giegerich, R.; Kurtz, S. (1997), "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction" (PDF), *Algorithmica* **19** (3): 331–353, doi:10.1007/PL00009177.
- Gusfield, Dan (1999), *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, ISBN 0-521-58519-8.
- Mansour, Essam; Allam, Amin; Skiadopoulos, Spiros; Kalnis, Panos (2011), "ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings" (PDF), *PVLDB* **5** (1): 49–60.

- McCreight, Edward M. (1976), "A Space-Economical Suffix Tree Construction Algorithm", *Journal of the ACM* **23** (2): 262–272, doi:10.1145/321941.321946, CiteSeerX: 10.1.1.130.8022.
- Phoophakdee, Benjarath; Zaki, Mohammed J. (2007), "Genome-scale disk-based suffix tree indexing", *SIGMOD '07: Proceedings of the ACM SIGMOD International Conference on Management of Data*, New York, NY, USA: ACM, pp. 833–844.
- Smyth, William (2003), *Computing Patterns in Strings*, Addison-Wesley.
- Tata, Sandeep; Hankins, Richard A.; Patel, Jignesh M. (2003), "Practical Suffix Tree Construction", *VLDB '03: Proceedings of the 30th International Conference on Very Large Data Bases*, Morgan Kaufmann, pp. 36–47.
- Ukkonen, E. (1995), "On-line construction of suffix trees" (PDF), *Algorithmica* **14** (3): 249–260, doi:10.1007/BF01206331.
- Weiner, P. (1973), "Linear pattern matching algorithms" (PDF), *14th Annual IEEE Symposium on Switching and Automata Theory*, pp. 1–11, doi:10.1109/SWAT.1973.13.
- Zamir, Oren; Etzioni, Oren (1998), "Web document clustering: a feasibility demonstration", *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, New York, NY, USA: ACM, pp. 46–54.

External links[edit]

This article's **use of external links may not follow Wikipedia's policies or guidelines**. Please improve this article by removing excessive or inappropriate external links, and converting useful links where appropriate into footnote references. *(August 2010)*

- Suffix Trees by Sartaj Sahni
- Suffix Trees by Lloyd Allison
- NIST's Dictionary of Algorithms and Data Structures: Suffix Tree
- suffix_tree ANSI C implementation of a Suffix Tree
- libstree, a generic suffix tree library written in C
- SDSL, various generic classical and compressed suffix tree implementations in C++
- Tree::Suffix, a Perl binding to libstree
- Strmat a faster generic suffix tree library written in C (uses arrays instead of linked lists)
- SuffixTree a Python binding to Strmat
- Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice, application of suffix trees in the BWT
- Theory and Practice of Succinct Data Structures, C++ implementation of a compressed suffix tree
- Practical Algorithm Template Library, a C++ library with suffix tree implementation on PATRICIA trie, by Roman S. Klyujkov
- A Java implementation
- A Java implementation of Concurrent Suffix Tree
- Text-Indexing project (linear-time construction of suffix trees, suffix arrays, LCP array and Burrows-Wheeler Transform)
- Ukkonen's Suffix Tree Implementation in C Part 1 Part 2 Part 3 Part 4 Part 5 Part 6

Tree data structures

Search trees (dynamic sets/associative arrays)

- 2–3
- 2–3–4
- AA
- (a,b)
- AVL
- B
- B+
- B*
- B^x
- (Optimal) Binary search
- Dancing
- HTree
- Interval
- Order statistic
- (Left-leaning) Red-black
- Scapegoat
- Splay
- T
- Treap
- UB
- Weight-balanced

Heaps	Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas
Tries	Hash · Radix · **Suffix** · Ternary search · X-fast · Y-fast
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · *k*-d (implicit *k*-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X
Other trees	Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary ·