Suffix links survival kit

Djamal Belazzougui, Fabio Cunial

1 First day

1.1 Introduction

Suffix trees not used in practice because:

- too big: approx. 20 bytes per character [13];
- poor locality (pointer-based).

Still, suffix trees are a useful theoretical construct for algorithm design.
Gusfield: “Not just a data structure, an organizing principle” $\Rightarrow$ More of a transform than of a data structure.
Related to other “data structures”:

- suffix trie $\rightarrow$ suffix tree (left-equivalence classes under right-extension);
- suffix trie $\rightarrow$ DAWG (right-equivalence classes under right-extension);
- DAWG $\rightarrow$ CDAWG (maximal repeats under right-extension);
- suffix tree $\rightarrow$ CDAWG.

We focus on suffix links.
They enable efficient ST construction [18, 14, 17].
Many applications to string analysis [6, 7, 3, 5, 12].
They can be explored using small-space data structures [7, 1, 5].

1.2 Basics

1. Explicit and implicit Weiner links.
2. Suffix-link tree.
3. If a node of ST has exactly one Weiner link, it must be explicit or a prefix of T. Proof: trivial.

This lecture was part of the 1st Summer School on Bioinformatics Data Structures, funded by BIRDS project (www.birdsproject.eu)
This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 690941

4. If $w = \texttt{suffixLink}(v)$, then $w = \texttt{lca}(\texttt{suffixLink}(\texttt{firstLeaf}(v)), \texttt{suffixLink}(\texttt{lastLeaf}(v)))$. Proof: trivial.

5. The number of implicit Weiner links is upper-bounded by $3n-3$. Proof [12]: type 1 nodes of ST have exactly one implicit Weiner link. There are at most $n-1$ of them (only internal nodes, since each leaf has just one explicit Weiner link). Type 2 nodes have at least 2 implicit Weiner links. Thus they are left-maximal substrings of T, thus they are nodes of the ST of the reverse, thus implicit Weiner links can be charged to the at most $n+(n-1)-1$ edges of the ST of the reverse.

6. $2n-2$ bound. Proof [12]: by Point 3, every implicit Weiner link can be charged to a distinct edge of the ST of the reverse, even implicit Weiner links from type-1 nodes of ST.

7. n bound: for clarity shown after the mapping SLT $\leftrightarrow$ ST of reverse.

8. Under suffix links, a path of ST might become longer. Under Weiner links, a path might become shorter.

9. Thus, a walk that performs only Weiner link and parent operations has the following property: the tree depth of the nodes in the walk, taken in reverse order, is a δ-monotone sequence[1] in which every value is at least the previous minus one.

1.3 Maximal repeats and their subgraph of ST

- Maximal repeat: node of ST with at least two Weiner links. Closed under prefix operation $\Rightarrow$ Induced subgraph of ST.

- Supermaximal: just implicit Weiner links to leaf edges, just leaves as children in ST.

- Near-supermaximal: leaf child in ST with text position i, implicit Weiner link to leaf edge with text position $i-1$;

- All nodes of ST with more than one Weiner link are located inside the subgraph induced by maximal repeats. All nodes of ST with exactly one (explicit) Weiner link are outside this subgraph.

[1]Let $a = a_0a_1\dots a_n$ and $\delta = \delta_1\delta_2\dots\delta_n$ be two sequences of nonnegative integers. Sequence a is said to be δ-monotone if $a_i - a_{i-1} \geq -\delta_i$ for all $i \in [1..n]$. Examples of δ-monotone sequences: MS, DS, PLCP array, longest previous factor array in LZ77 [2].

- *Rightmost* maximal repeat: leaf of the max. repeat subgraph.

- The max. repeat subgraph is also a spanning tree of the CDAWG. The trie refinement of the max. repeat subgraph is a spanning tree of the DAWG.

- Number of runs in the BWT is at most equal to the number of right-extensions of maximal repeats that do not lead to maximal repeats. Proof [4]: the subtree rooted at every destination node is a block of the BWT, and since the node is not left-maximal the block must contain a single character.

- Number of runs in the BWT is at least equal to the sum of the left-degree of all rightmost max. repeats, minus the number of rightmost max. repeats, plus one. Proof [4]: the subtree rooted at each rightmost max. repeat corresponds to a disjoint interval in the BWT, which contains a number of distinct characters equal to the left-degree of the node. At most one such character is identical to the one of the suffix that immediately precedes it in lex. order (such suffix might not be prefixed by a rightmost max. repeat).

- Number of LZ factors is at most equal to the number of right-extensions of max. repeats. Proof [4]: take a factor (which is right-maximal) and extend it to the left in the text until it becomes left-maximal as well. Charge the factor to the right-extension of such max. repeat by the character you see on the right of the factor in the text. If another factor is later charged to the same right-extension, then such factor should have been longer, a contradiction.

- Number of right extensions of max. repeats is $\Omega(\log n)$. Proof: the CDAWG recognizes all the n distinct suffixes of the text. Every such distinct suffix corresponds to a distinct path in the CDAWG. Consider a bitvector of length equal to the number of arcs in the CDAWG, in which a bit is set to one iff the corresponding arc is taken by a suffix path. A suffix path in the CDAWG corresponds to a bitstring. There must be at least $\log n$ bits in the bitvector, otherwise it could not represent n distinct bitstrings.

- Max. repeats form a tree by longest border (supermaximal repeats cannot be internal nodes). Left-extensions by one character of max. repeats form a tree by longest border (their longest border is either a single character, or it has the same property).

1.4 SLT and max. repeat subgraph of the reverse

- Draw ST, SLT, reverse of both, overlap [11].
- Every leaf of SLT has more than one implicit Weiner link (thus it is a max. repeat), or it is a prefix of T (called also *nested prefix* [11]). Proof: same as Point 3.
- n bound on number of implicit Weiner links. Proof [12]: every implicit Weiner link of ST can be mapped to an edge of the ST of the reverse that leads to a distinct leaf of the ST of the reverse.

2 Second day

2.1 Distance from parent

- Consider a suffix link walk. Let $f(v)$ be the distance of a node in the walk from its parent in ST. Then, $f(v)$ is monotonically nonincreasing. Parents are organized into blocks with identical value of f. Decreasing $f \Rightarrow$ Parent is a maximal repeat. Proof: the new node that becomes the parent has an implicit Weiner link, so for Point 3, it must be left-maximal.
- Note that the number of max. repeats in the parents is at least the number of max. repeats in the walk, and it is at least the number of drops in the value of f.

2.2 Implicit nodes [8, 15]

Called also "nested suffixes" [11].
Used in online ST construction algorithms, since they are affected by right-extension of T.

- Definition: a node in ST with exactly two children, one of which being the dollar.
- Two explicit nodes of ST can be separated by at most one implicit node. Proof: by contradiction, assume (u,v), (v,w), (w,x) to be arcs of ST, with v,w implicit. Since v is also a suffix of w, and w is a prefix of x, and x is right-maximal in S without dollar, then the ST must branch below v after at most a number of characters equal to the length of x minus the length of w. But it doesn't.

- Type 1 implicit node: no two suffixes of S that start at occurrences of the string can be disambiguated by a character in Σ. I.e. they can only be disambiguated by the dollar, i.e. by length. Such nodes would belong to a leaf edge if the dollar were removed.

- Type 2: at least two occurrences can be disambiguated by a character in Σ. Such nodes would belong to an edge between two internal nodes if the dollar were removed.

- Take suffixes of S (without dollar). Then, you find them in the ST in the following order: unique suffixes (last string in a leaf edge); implicit nodes of type 1; type 2; explicit. Proof: type 2 can only go to type 2 or explicit by suffix. Explicit can only go to explicit.

- Thus, all implicit nodes, if any, have consecutive lengths, and they form a "tail" in the suffix-link tree (i.e. a chain that does not necessarily end at the root). Proof: a suffix W might have a Weiner link to aW and a Weiner link to bW, but since all occurrences of W have the same character to the right, bW cannot be right-maximal and the Weiner link cannot be explicit.

- Assume that the longest implicit node, called also "active suffix" of S (without dollar), is of type one. Let it correspond to a string W with period $p \leq |W|/2$. Then W occurs exactly twice in S, it is a max. repeat, and it is the longest border of its longest leaf (the deepest node in the subtree of the node laballed by W). So that longest leaf is periodic, with period equal to the length of the leaf minus the length of W.

- If the two occurrences of W do not overlap (or the distance between their starting positions is at least $p+1$), then after taking p suffix links from the deepest implicit node we must necessarily reach a node that is not of type one.

- All implicit nodes associated with strings of length greater than p and lying on the same root-to-leaf path as the longest implicit node, are found only after p suffix links, and multiples (they otherwise belong to $p-1$ other distinct root-to-leaf paths). All such implicit nodes correspond to strings whose lengths differ by multiples of p. Proof: use the periodicity lemma [10].

- If the suffix V of W of length $|W| - p\lfloor |W|/p \rfloor$ has period p' (where clearly $p' < p$), then the suffix of W of length $|V| - p'$ cannot be an implicit node of type one.

- Let W be the string associated with the deepest implicit node of type one and assume its period is $p \geq 2$. Consider the i-th deepest implicit node of type one with $i \leq p$ and denote its associated string by V. Then V occurs exactly twice in S, is a suffix of W, and has p as a period.

- The deepest implicit node is also an upper bound for function f of all implicit nodes. Function f has the same value for all implicit nodes of type one, except possibly for the last one (the one associated with the shortest suffix) at which f might have a smaller value.

- When taking a suffix link from a type one implicit node to another type one implicit node, the frequency (number of leaves in the subtree of a given node) increases by at most one. Specifically, it increases by one iff the suffix link leads to a border of W.

2.3 Suffix links construction

We show how to build the suffix links for internal nodes in a suffix tree in linear time given a suffix tree for which we only know the suffix links from the leaves (which are easy to compute). We assume that the tree has been built on a text of length n over the integer alphabet $[1..\sigma]$ (and further assume that $\sigma \leq n$) and denote the maximal string depth (path length) of the internal nodes in the tree by d (d can typically be much smaller than n). We recall that the path of a node v is the concatenation of all edge labels from the root to v and is denoted by $\ell(v)$. The algorithm we present will only use arrays of size σ and d. The numbers manipulated by the algorithm are in the ranges $[1..\sigma]$ and $[1..d]$. Everything else will be lists and pointers. The main property of the suffix tree we will use is the following:

Lemma 1 *An internal node v in the suffix tree will have a Weiner link labelled by character c if and only if at least one of its children has a Weiner link labelled by c and it will have an explicit Weiner link labelled by c if and only if it has at least two children with Weiner links labelled by c.*

Proof Recall that an internal node v is labelled by right maximal string P and has an implicit Weiner link with character c if cP occurs in the text. This is only possible if cPa appears in the text for some character a, which means that a child of v labelled by a has a Weiner link labelled by c. This proves the first case of the claim. For the second case, an explicit Weiner link with character c will exist if and only if cP appears in the text and is right maximal, implying the existence of a node v' labelled by string cP. This implies that the text contains the substrings cPa and cPb for two distinct characters a and b which can only happen if two edges from v have labels starting with a and b respectively and their destination nodes have Weiner link labelled by c.

We now give an overview of the algorithm before entering into details. We recall that an explicit Weiner link is the reverse of a suffix link. Our algorithm will produce explicit Weiner links and then reverse them to induce suffix links. The algorithm derives from two observations. The first observation is that the nodes at string depth $i+1$ will have suffix links to nodes at string depth i. Suppose that we order the nodes with the same string depth in left-to-right order (equivalently in lexicographic order of their paths). Our second observation is that suffix links from nodes labelled by paths prefixed with the same character will preserve the order.

This suggests the following algorithm: first build for each string depth the ordered list of nodes with that string depth, and build for every node the list of characters that label its explicit Weiner links. The latter is easily done by making use of lemma proved above. Second, build the suffix links that start from nodes of string depth $i+1$, by traversing the lists of ordered nodes of string depth $i+1$ and i. This is done by first clustering the first list according to first characters of the paths (building a sublist c for each character c) and then traversing the second list, and for each node, determining the target of its explicit Weiner links by looking at the list of their labels: the target of the explicit Weiner link labelled by c is determined as the first element in sublist c and is then removed from the sublist. We now describe the details of the algorithm.

2.3.1 First Phase

In this phase the suffix tree is traversed in top-down left-to-right order. It is assumed that the string depth (path lengths) of traversed nodes are known during the traversal. If they are not stored in the suffix tree structure, we can compute them on the fly by keeping the string depth in the stack (of depth at most d) used to traverse the suffix tree. That is the string depth of a node v can be induced (during the traversal) from the string depth of its parent v' by adding the length of the label of the edge that connects v to v' with the string depth of v'.

The output of this phase consists in two elements. First element is an array $D[1..d]$ of d lists, where d is the maximal string depth. An item in each list is a pair $(\texttt{node}, \texttt{char})$. The second element of the output is an association of a list of characters labelling the explicit Weiner links starting from each node of the tree (the details of this association is given below). We now describe the construction of these outputs. While traversing the tree, when a node v has a path of length i and (the path) is prefixed by character c (the node is in the subtree of the root's child whose label start by character c), simply add the pair (v, c) at end of list $D[i]$. The list of characters that label explicit Weiner links from each node is built easily by merging the lists of characters that label all Weiner links from its children. The construction algorithm maintains two lists of Weiner link characters for every node (one list for explicit Weiner links and the other for all Weiner links). The merging is done for each node when the node is traversed the second time. It also uses two temporary structures, a stack that stores a set of distinct characters and an array $C[1..\sigma]$ of counters in $[1..\sigma]$ that associates a count to each character. Initially the stack is empty and all counters in C are set to zero. The lists for leaves are easily computed: a leaf representing a suffix preceded by character c will have its two lists containing the single element c. For an internal node, the two lists of (resp. explicit and all) Weiner link characters are built from the lists of (all) Weiner link characters of its children as follows: for each child traverse the list of its Weiner link characters and for each character c increment $C[c]$ and check if $C[c] = 1$ ($C[c]$ was equal to zero before incrementation). If that is the case, add c to the stack. At the end the list of (all) Weiner link characters is the list of characters contained in the stack and the list of explicit Weiner link characters is the list of characters in the stack for which $C[c] > 1$. At the end the stack is restored to the empty state and counters in C are reset to zero by traversing the list of Weiner link characters and resetting $C[c] = 0$ for each traversed character c. The correctness of the computation follows from Lemma 1.

2.3.2 Second Phase

The second phase traverses the array D level by level. At step i, we take the list $L = D[i+1]$ (which is a list of nodes at depth $i+1$ sorted in lexicographic order of their paths) and split its elements into α arrays, where α is the number of distinct characters that appear in the list L (as second component of the pairs). That is, produce a sparse array of sublists $L'[1..\sigma]$ from $D[i+1]$, where only α positions point to sublists of nodes at depth

$i+1$ sorted in lexicographic order (α is the number of distinct characters that appear in first positions of these nodes path's). The sublists are produced as follows: initially all sublists are empty (all positions in L' are set to null pointer). Then the list L is traversed, and for each pair (v, c), the node v is added to the sublist $L'[c]$. Now traverse the list $L = D[i]$ (which is also sorted in lexicographic order), and for each node (first component of a pair) in the list determine the explicit Weiner link as follows: if the explicit Weiner link has label c, simply take its target as the node v, the head of sublist $L'[c]$ and then remove v from the sublist. At the end of the traversal of the list $L = D[i]$ all positions in L' will contain null pointers (since all sublists $L'[c]$ are now emptied) so that the array L' is ready to be reused for building explicit Weiner links from node of string depth $i-1$ to nodes of string depth i. We continue the execution of the algorithm until we have build the explicit Weinerl inks from the root (the only node at string depth 0), at which point we will have built all explicit Weiner links for internal nodes of the tree. Finally the suffix links can be induced by just reversing the directions of the produced explicit Weiner links.

2.3.3 Analysis

The correctness of the second phase of the algorithm follows from the fact that the target nodes of the suffix links will have the same order as the source nodes as long as the path labels of the latter start with the same character. The linear running time of the algorithm follows from the fact that the number of Weiner links is linear.

3 Related topics that have not been covered

3.1 Eulerian directed multigraph 1 [9]

- The directed graph that consists of arcs of ST and suffix links can contain directed cycles: a string V is an ancestor of a string W in both ST and SLT iff V is a border of W. Proof: trivial. Note that the border of a right-maximal string is right-maximal.

- For every node v of the ST, assign to its suffix link a mass equal to the number of children of v in the ST that are leaves. Then, the total mass leaving the subtree rooted at node W is not smaller than the total mass coming into the subtree rooted at node W. Proof: the total mass leaving the subtree rooted at W is the number of occurrences of

W in the text. The total mass incoming in the subtree rooted at W consists of a set of distinct leaves, i.e. of a set of distinct suffixes of the text. Consider an incoming suffix link (aWV, WV), and assign child leaf $aWVb$ of source node aWV, starting at position i in the text, to occurrence $i+1$ of W. Since all i values are distinct, all $i+1$ values are also distinct, and they are occurrences of W.

- Let $c(v)$ be the total imbalance of the mass leaving the subtree of ST rooted at v, and the mass coming into the subtree of ST rooted at v. Assign multiplicity $c(v)$ to arc $(\texttt{parent}(v), v)$. The resulting directed multigraph is Eulerian, i.e. it contains a cycle that uses every arc exactly once. Proof: the number of in-neighbors and out-neighbors of every vertex is the same.

3.2 Eulerian directed multigraph 2 [15, 16]

Also called "suffix tour graph".
Uses arcs of ST and a new type of arc, rather than suffix links.
Used for reconstructing S given the topology of ST, the suffix links from internal nodes only, and the first character of every arc of ST.
Note that, if we knew the suffix links from the leaves, we would also know S.

- A new arc from leaf y points to the node x whose subtree must contain the destination of the suffix link from the leaf. I.e. to the node x such that $\texttt{parent}(x) = \texttt{suffixLink}(\texttt{parent}(y))$ and the label of $(\texttt{parent}(y), y)$ equals the label of $(\texttt{parent}(x), x)$.

- Let $h(x)$ be the difference between the number of leaves in the subtree rooted at x, and the number of incoming arcs to nodes in the subtree rooted at x. Note that $h(x) \geq 0$. So, if every arc (y, v) from a leaf y has multiplicity one, and every arc in ST $(\texttt{parent}(v), v)$ has multiplicity $h(v)$, the resulting directed multigraph is Eulerian.

- Why is $h(x) \geq 0$? For the same reason as before. Start from y: it corresponds to a suffix $S[i..n]$. Let $S[i..j]$ be the longest prefix of $S[i..n]$ that repeats. We then go to $S[i+1..j]$ and finally to $X = S[i+1..k]$ with $k \geq j$. Thus we assign y to occurrence $i+1$ of X. No two leaves that point to the subtree of x can be assigned to the same occurrence of X, since such leaves are distinct suffixes.

References

[1] Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 148–193. ACM, 2014.

[2] Djamal Belazzougui and Fabio Cunial. Indexed matching statistics and shortest unique substrings. In *String Processing and Information Retrieval*, pages 179–190. Springer, 2014.

[3] Djamal Belazzougui and Fabio Cunial. A framework for space-efficient string kernels. In *Annual Symposium on Combinatorial Pattern Matching*, pages 13–25, 2015.

[4] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Annual Symposium on Combinatorial Pattern Matching*, pages 26–39. Springer, 2015.

[5] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In *Algorithms–ESA 2013*, pages 133–144. Springer, 2013.

[6] Timo Beller, Katharina Berger, and Enno Ohlebusch. Space-efficient computation of maximal and supermaximal repeats in genome sequences. In *International Symposium on String Processing and Information Retrieval*, pages 99–110. Springer, 2012.

[7] Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the burrows–wheeler transform. *Journal of Discrete Algorithms*, 18:22–31, 2013.

[8] Dany Breslauer and Giuseppe F Italiano. On suffix extensions in suffix trees. In *International Symposium on String Processing and Information Retrieval*, pages 301–312. Springer, 2011.

[9] Bastien Cazaux and Eric Rivals. Reverse engineering of compact suffix trees and links: A novel algorithm. *Journal of Discrete Algorithms*, 28:9–22, 2014.

[10] Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[11] Robert Giegerich and Stefan Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[12] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I Tomescu. *Genome-Scale Algorithm Design.* Cambridge University Press, 2015.

[13] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[14] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[15] Tatiana Starikovskaya and Hjalte Wedel Vildhøj. A suffix tree or not a suffix tree? *Journal of Discrete Algorithms*, 32:14–23, 2015.

[16] I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Inferring strings from suffix trees and links on a binary alphabet. *Discrete Applied Mathematics*, 163:316–325, 2014.

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

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