TUCS

Special course in Computer Science: Advanced Text Algorithms

Lecture 5: Suffix trees and their applications

Eugen Czeizler
Department of IT, Abo Akademi
http://combio.abo.fi/teaching/textalg/

(slides originally by I. Petre, E. Czeizler, V. Rogojin)

Suffix trees

- Suffix tree = data structure used for storing all the suffixes of a text.

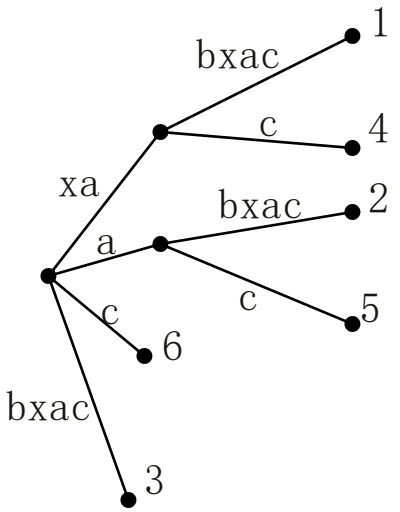
- A suffix tree T for a string S[1..m] of length m is a directed rooted tree with:
  - exactly m leaves numbered 1,..,m
  - at least two children for each internal node (with the root as a possible exception)
  - each edge labeled by a nonempty factor of S
  - no two edges out of a node beginning with the same character.

Suffix trees

- Let L(v) denote the label of a node v, i.e., the concatenation, in order, of the strings labelling the edges on the path from the root to the node v.

- Key feature of suffix trees: L(i)=S[i..m] for each leaf i.

- Example: For the string xabxac we can associate the following suffix tree: L(5)=ac, i.e., the suffix starting on position 5.



Ukkonen's algorithm

- Ukkonen's algorithm constructs a suffix tree for the string S[1..m] in linear time (O(m)).

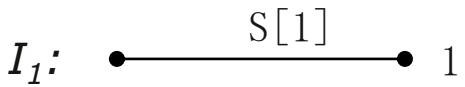
- The method builds, as intermediate results, *implicit suffix trees* for each *prefix* S[1]; S[1..2];..; S[1..m]

- Denote by I_j the implicit suffix tree of the prefix S[1..j]

- I_j contains each suffix S[1..j]; S[2..j]; ... ; S[j] of S[1..j] as a label of some path, but it is possible that this path ends in the middle of an edge (contrary to the case of suffix trees).

Ukkonen's Algorithm on a High Level

- Given a string S[1..m]$, we start with $T := I_1$



- Then update T to trees $I_2,...,I_{m+1}$ in m phases.
- Phase i+1 updates T from I_i (containing all suffixes of S[1..i]) to I_{i+1} (containing all suffixes of S[1..i+1])
- Each phase i+1 consists of extensions j = 1,..,i+1;
- Extension j ensures that suffix S[j..i+1] is in I_{i+1}

Extension j of Phase i + 1

- Phase i+1 starts with $T = I_{i,}$ containing all suffixes of S[1..i]

- For each $1 \leq j \leq i+1$, the algorithm finds the end of the path S[j..i] and extends it (if necessary) to ensure that the suffix S[j..i+1] is in the tree.

- Question: How do we extend the path?

- After locating the end of the path S[j..i], we have tree possible cases, and we have 1 extension rule for each case:

Extension rules

- Rule 1: If in the current tree the path S[j..i] ends at a leaf. Then we concatenate S[i+1] to its edge label.

- Rule 2: If no path from the end of S[j..i] starts with character S[i+1], but there is at least one labeled path from the end of S[j..i]. Then, we create a new leaf labeled j, and an edge starting from the end of S[j..i] to the new leaf labeled by S[i+1]. If the path ended at the middle of an edge, we split the edge and insert a new node as a parent of leaf j.

- Rule 3: Some path from the end of S[j..i] starts with S[i+1]. Then, we do nothing since the suffix S[j..i+1] is already in the tree.

Algorithm complexity

- The key issue in implementing Ukkonen's algorithm is locating the end of all the i+1 suffixes of S[1..i].

- Once the end of such a suffix is located, the execution of any of the extension rules takes constant time.
  - So, applying them once in each extension takes time $\Theta(i)$

- Locating the ends of the paths S[1..i];..; S[i+1..i] by traversing them explicitly takes time $\Theta(\Sigma_{l=0}^{i} l) = \Theta(i^2)$

- So, the total time for all phases i=2;..;m + 1 is

$$\Theta(\Sigma_{l=2}^{m+1} i^2) = \Theta(m^3)$$

Reducing the Complexity

- How to reduce this complexity?

- We showed in Lecture 4 that by avoiding or speeding up path traversals, we can get the total time down to $\Theta(m^2)$.

- How do we obtain $\Theta(m^2)$?

Final Improvements

- Some extensions can be found unnecessary to be computed explicitly.

- Observation 1: Rule 3 is a "show-stopper":
  - If path S[j..i+1] is already in the tree, so are paths S[j+1..i+1];...; S[i+1], too

- Trick 2: Phase i+1 can be finished at the first extension j that applies Rule 3

Final Improvements (2)

- Observation 2: A node created as a leaf remains a leaf thereafter.

- There is no mechanism (i.e., no extension rules) to add children to a leaf.

- If extension j created a leaf (numbered j), extension j of any later phase i+1 will apply Rule 1 (concatenating the next character S[i+1] to the edge label of j)

- We want to eliminate some explicit applications of the extension Rule 1.

Compressed edge representation

- First, we use a compressed edge representation:
  - on each edge we write indices (p,q) instead of the substring S[p..q].

- We also use a global value, e, that stands for "the current end position"
  - In phase i+1, when a leaf edge is first created (e.g., for the string S[j..i+1]) we write the indexes (j,e) on the edge, instead of (j,i+1)
  - The value of the index e is set to i+1 once in each phase

Eliminating Extensions

- Leaf 1 is created in phase 1.
- In any phase i, there is an initial sequence of consecutive extensions using rules 1 or 2.
- Denote by j_i the last extension (i.e., application of rule 1 or 2) of this sequence.
  - leaves 1,..,j_i have been created by the end of phase i
- Extensions 1,.. , j_i of any subsequent phase all apply Rule 1 (see Observation 2)
- So, $j_{i+1} \geq j_i$ i.e., the initial sequence of extensions using rules 1 or 2, can only get longer in successive phases.

Eliminating Extensions

- Trick 3: In phase i+1, the algorithm knows already that rule 1 will be applied in the extensions 1,.., j_i. So, it only has to:
  - Increment the global variable e (instead of explicitly applying rule 1 for the first j_i extensions): this takes constant time.
  - Execute explicitly the extensions j_i+1, j_i+2,...

- Using tricks 2 and 3, in phase i+1 we will execute explicitly extensions j_i+1, j_i+2, ... until the first execution of rule 3.

Single Phase Algorithm

- Algorithm for phase i+1 with unnecessary extensions eliminated

1. Set e := i + 1;
   - by Trick 3, this step correctly implements all implicit extensions 1,..,j_i
2. Explicitly compute extensions j_i+1, j_i+2, .., j^* until j^*>i+1 or Rule 3 was applied in extension j^*;
   - by Trick 2, this step correctly implements all additional implicit extensions j^*+1,..,i+1
3. Set j_{i+1}:= j^* -1; (for the next phase)

- All these tricks together can be shown to lead to linear time

Complexity of Ukkonen's algorithm

- Theorem: Given a string S[1..m], by using suffix links and tricks 1,2, and 3, Ukkonen's algorithm builds implicit suffix trees $I_1,..,I_m$ in time O(m).

Proof:

- The implicit extensions take constant time in each phase, so they take O(m) time over the entire algorithm.

- Let j'=1; .. ;m denote the index of the current explicit extension.

- Over all phases 2; .. ;m index j' never decreases, but it does remain the same between two consecutive phases.

- So, at most 2m extensions are computed explicitly since:
  - There are m phases
  - j'≤m

Complexity of Ukkonen's algorithm

- The execution of an explicit extension takes:
  - constant time
  - plus some time proportional to the node skips it does during the down-walks.

- We consider next how the current node-depth changes during successive extensions:
  - In each explicit extension the node-depth is first decreased by at most 2. So, the current node depth can be decremented at most 4m times, over the entire algorithm.
  - On the other hand, the current node depth cannot exceed m. So, the node-depth is incremented (by following downward edges) at most 5m times over the entire algorithm.

- Using the skip/count trick, the time per down-edge traversal is constant. Thus, the total time of the algorithm is O(m).

Creating the true suffix tree

- The final implicit suffix tree, I_m, is constructed in O(m) time.

- Add $ at the end of S[1..m] and continue Ukkonen's algorithm for S[1..m]$:
  - No suffix is a prefix of another suffix
  - Each suffix ends at a leaf

- Finally, I_{m+1} can be converted to the true suffix tree of S[1..m] by replacing all occurrences of the "current end position" marker e on edge labels by m:
  - with a simple tree traversal, in time O(m)

Ukkonen's algorithm

- Theorem: Ukkonen's algorithm builds a true suffix tree for the string S[1..m], along with all its suffix links in O(m) time.

- Remark: Ukkonen's method is "on-line", by processing S left-to-right and having a suffix tree ready for the scanned part.

Applications of suffix trees

- Suffix trees have numerous applications, often providing linear-time solutions to challenging string problems, e.g.,:
  - Exact pattern matching
  - The longest common substring
  - The longest repeated substring inside a given text
  - Frequent common substrings of a set of strings, i.e., that occur in a large number of distinct strings
  - Suffix arrays
  - Genome-scale projects

Generalized Suffix Trees

- We begin with a useful generalization of suffix trees for indexing multiple strings $\{S_1,...,S_k\}$ (instead of a single one).

- A generalized suffix tree can be used to represent all suffixes of a set of strings $\{S_1,...,S_k\}$.

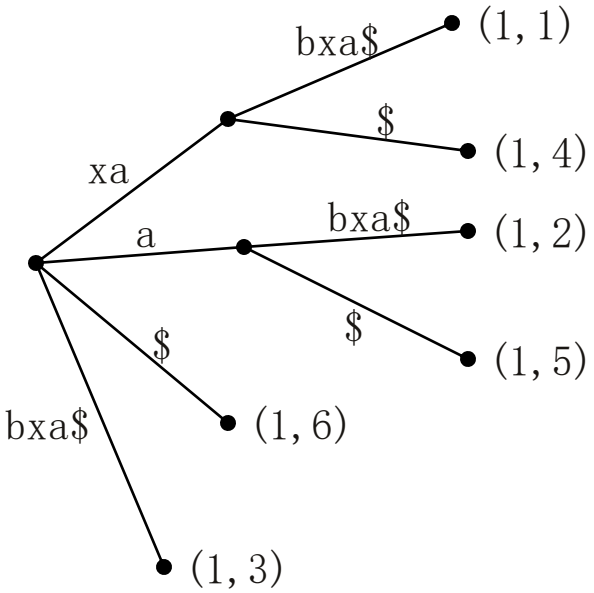
- How to build generalized suffix trees?

Incremental Construction of Generalized Suffix Trees

- Build a suffix tree T for S_1\$, keeping the suffix links.
- Traverse the longest path from the root of T that matches a prefix of S_2, let it be $S_2[1..i]$.
- Now T encodes all suffixes of S_1, and all suffixes of $S_2[1..i]$ as if the first i phases had been computed for S_2.
- Continue from Phase i+1 for S_2\$.
- After entering S_2\$, continue similarly with S_3\$,..., S_k\$.
- Total time is linear wrt the total length of the strings.

Example of Generalized Suffix-Tree Construction

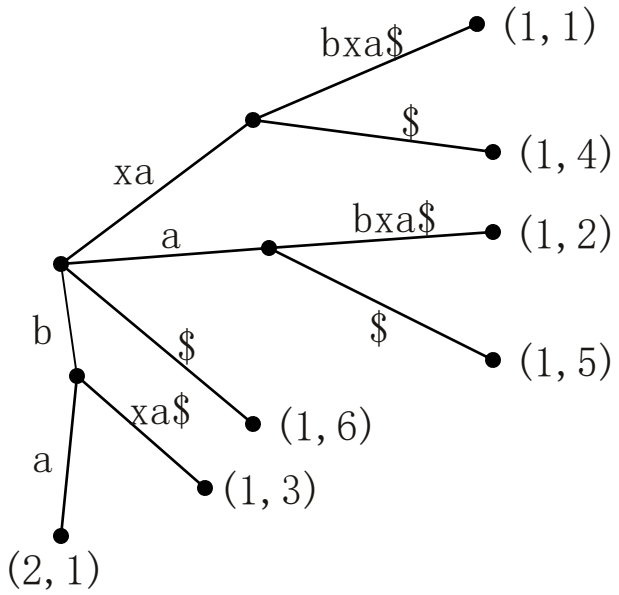
- Consider adding S_2=babxba to the generalized suffix tree T of S_1=xabxa



- The longest prefix of S_2 already in T is $S_2[1]$ =b
- Continue Ukkonen's algorithm from Phase 2, to enter suffixes of $S_2[1..2]$=ba in the tree

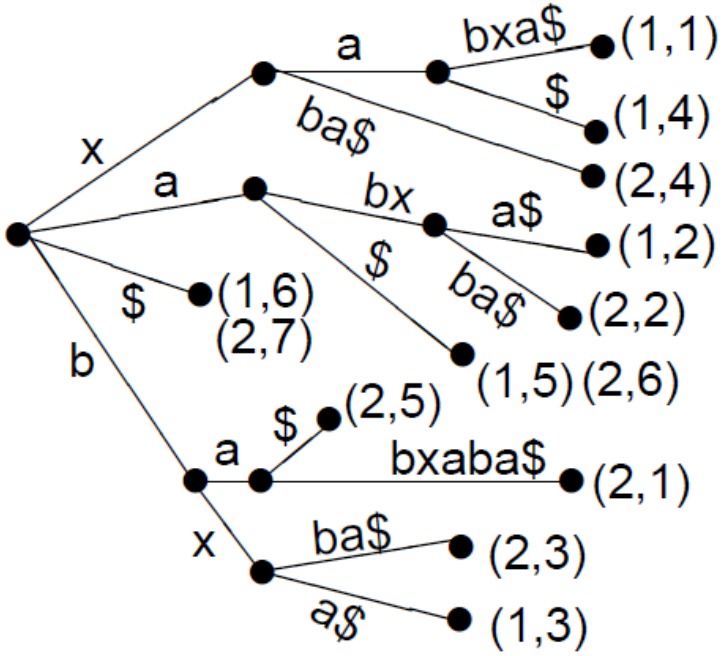
Example of Construction (2)

- The tree after adding suffixes of $S_2[1..2]$=ba



Example of Construction (3)

- Generalized suffix tree for S_1=xabxa and S_2=babxba (after completing all phases for $S_2$$):



Two Implementation Details

1. Edge-labels of a generalized suffix tree may come from more than one string
   - An edge-label $S_i[p..q]$ needs to be represented in the compressed form by a triple (i,p,q)

2. Some suffixes may occur in multiple strings (if they have a common end marker)
   - Store in each leaf v all pairs (i,j) such that L(v) is a suffix of string S_i starting at position j

Application 1:String Matching

1. Construct the suffix tree T for the text[1..n]: O(n) time complexity.

2. Match the characters of pat along the unique path from the root:
   - The pattern pat occurs in the text at position j if and only if pat is a prefix of text[j..n]
   - That is, pat labels an initial part of the path from the root of the graph to the leaf labelled by j
   - The matching path is unique since the edges leaving from a node are labelled by strings beginning with different characters.
   - The time to match pat to a path is in O(m), with m=|pat|

String Matching

a) If pat can be fully matched, let k be the number of leaves below the path labelled by pat. Each of these leaves indicates the start position of an occurrence of pat, and they can be collected in time O(k).
   - Traverse the tree starting at the end of the matching path using, e.g., a depth-first approach, and gather all the encountered leaves.

b) If pat doesn't match completely, then pat does not occur in the text

- The total time complexity of the algorithm is O(n+m+k)

String Matching

- Linear-time exact matching methods (KMP, BM) solve the problem using
  - O(m) time for preprocessing (m = |pat|), and
  - O(n) time for searching (n =|text|)

- Suffix trees give the same asymptotic total time, with
  - O(n) time for preprocessing, and
  - O(m + k) time for searching, where k≤n is the number of occurrences of pat

Exact Matching on Static Texts

- Suffix trees are not practical for single searches.

- If one has a series of patterns that need to be matched in a text, it is more suitable to use suffix trees
  - After O(|text|) pre-processing, the search for occurrences of any pat takes time O(|pat|+k) only (as opposed to O(|text|);
  - Often |pat|, k << |text|

- This may be especially useful with a generalized suffix tree built for a database of sequences
  - However, the required space, even if O(m), can be large in practice.

Application 2: Longest Common Substring

- A classic problem: How to find the longest substring common to two given strings?

- Example: The longest common substring of "DELIVERABILITIES" and "OUTLIVERS" is "LIVER"

- The longest common substring problem is easy to solve in linear time using suffix trees.

- Before the suffix trees, in 1970, D. Knuth conjectured this to be impossible!

Finding Longest Common Substring

- A maximal substring common of strings S_1 and S_2 can be found as follows:

1. Build a generalized suffix tree for S_1 and S_2
2. Add to each internal node v label 1 (or resp. 2) if the subtree below v contains a leaf corresponding to a suffix of S_1 (or resp. S_2)
   - We can do this in a bottom-up traversal of the tree
3. Traverse the tree to find nodes marked by both 1 and 2, and choose any u of them with a maximal string-depth. Then, L(u) is a maximal common substring.

Finding Longest Common Substrings (2)

- Theorem: A longest common substring of two strings can be found in linear time.

- The method can be easily extended for the case when we want to find the longest common substring of k strings.

Application 3: DNA Contamination

- Various laboratory processes cause unwanted DNA to become inserted into the string of interest, e.g.:
  - fragments of a *vector (DNA string) used to incorporate the desired DNA in* a host organism
  - DNA of the host organism (e.g., bacteria or yeast)

- The DNA contamination is a serious problem, which should be recognized to avoid unnecessary work and erroneous conclusions
  - Some years ago, there was an announcement that DNA had been successfully extracted from dinosaur bone. After sequencing it, they discovered that their sequence was closer to human DNA than to bird or reptilian DNA. This suggested that much of the DNA sequence was coming from human contamination and not from the dinosaur

DNA Contamination as a String Problem

- DNA sequences of many possible contaminants (e.g., cloning vectors, PCR primers, genome of the host organism) are already known.

- This suggests the following DNA contamination problem:

- Given a string S_1 (e.g., a newly sequenced DNA) and a set of strings S (e.g., the sources of possible contamination), find all substrings of S that occur in S_1 and are longer than some given length l

- In other application areas this problem could be called *plagiarism detection*

Solving the DNA Contamination Problem

- The problem can be solved as an extension of the previous one:

1. Build a generalized suffix tree T for $S \cup S_1$
2. Mark all internal nodes of T whose subtree contains leaves for both S_1 and some string of S
3. Traverse T. For any marked node v with string-depth at least l report L(v) as a suspicious substring

- Total time is $\Theta(m)$, with m=the total length of the strings

Genome-Scale Projects

- Let us take now three examples of genome-scale projects that apply suffix trees (with variations):

- Arabidopsis thaliana project at the Michigan State Univ. and the Univ. of Minnesota

- Analysis of *Saccharomyces cerevisae (brewer's yeast)* at the Max-Plank Institute

- Borrelia burgdorferi project *at the Brookhaven National* Laboratory.

Arabidopsis thaliana

- Arabidopsis thaliana is a small flowering plant native to Europe, Asia, and northwestern Africa.
- Genome about 100 million base-pairs
- Goal: an EST map roughly listing of relative locations for known fragments of genes
- Generalized suffix trees applied to:
  - check DNA contamination by known vector sequences
  - find duplicates and similarities with previously sequenced fragments
  - find biologically significant patterns (represented as regular expressions)

Saccharomyces cerevisae (brewer's yeast)

- *Saccharomyces cerevisiae* is a species of budding yeast, used since ancient times in baking and brewing.

- Antibodies against *S. cerevisiae* are found in 60–70% of patients with Crohn's disease and 10–15% of patients with ulcerative colitis.

- The genome is composed of about 12 million base pairs.

- Suffix trees were used for finding substrings in sequence databases.

Borrelia Burgdorferi

- The bacterium causing the Lyme disease.
- Genome about 1 million base-pairs long.
- Suffix trees and arrays were used in fragment assembly, i.e., deducing the underlying long sequence from its fragments.
  - A major problem here is to detect overlaps.
  - In this project 4,612 fragments of total length of 2 million bases were handled in quarter of an hour using suffix trees and arrays.
  - The suffix tree approach gave a 1000 times speedup over the (slightly) more accurate dynamic programming approach, and it found 99% of the significant overlaps found by the dynamic programming approach.