FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

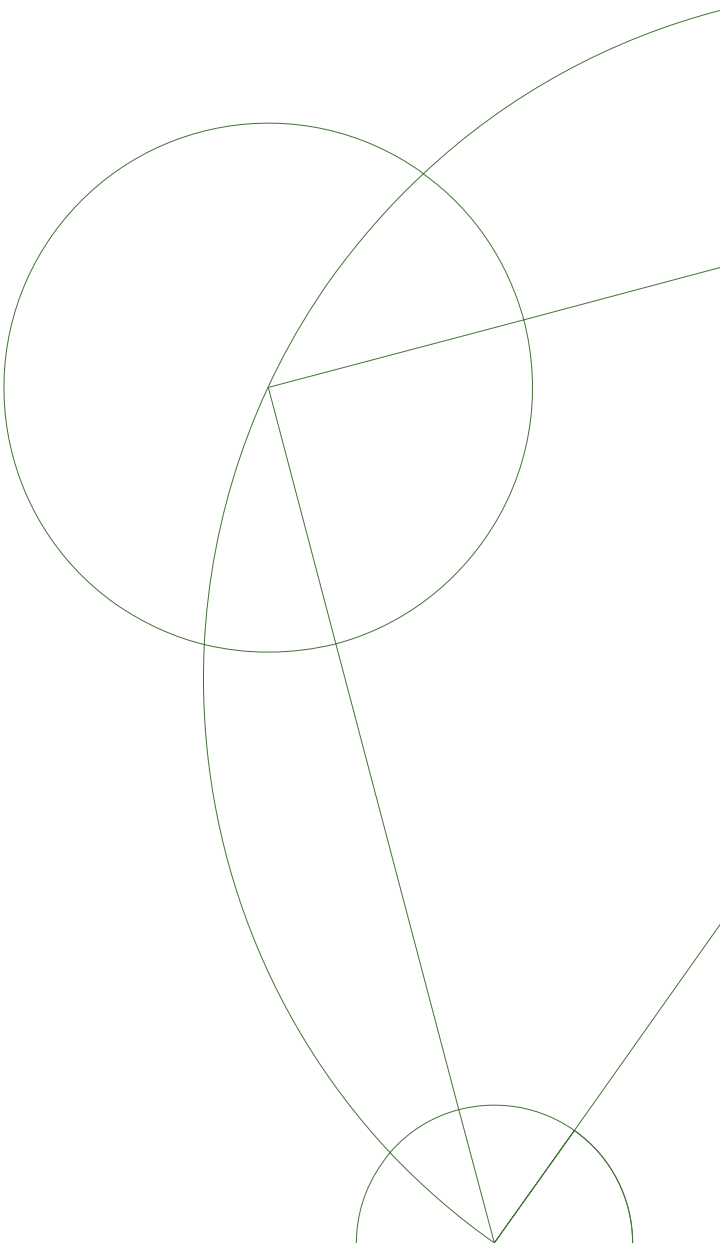
DEPARTMENT OF COMPUTER SCIENCE

Bachelor

Suffix Arrays In Intrusion Detection

April 2017

Mark Roland Larsen <fvg932@alumni.ku.dk>



Supervisors
Michae Thomsenl <m.kirkedal@di.ku.dk>
Troels Larsen <gvl981@di.ku.dk>

Contents

1 Abstract

2 Description

3 Preface

4 Limitations

I følgende opgave arbejdes der på binære træer med typen

5 Introduction

The string matching problem is found in various fields of study **?**. In biology, string matching algorithms significantly aid biologists in retrieving and comparing DNA strings, reconstructing DNA strings from overlapping string fragments and looking for new or presented patterns occurring in a DNA**?**. Text-editing applications also adopt string matching algorithms, whenever the application has to acquire an unambiguous occurrences of a user-given pattern, such as a word in some document**??**. String matching is used in music equipment, AI (artificial intelligence) and in addition, various software applications like virus scanners (anti-virus) or intrusion detection systems, frequently adopt string matching algorithms as a practical tool, to secure data security over the internet **?**. Fundamentally, string matching is a method to find some pattern $P = \{p_1, p_2, \ldots, p_n\}$ in a given text $T = \{t_1, t_2, \ldots, t_m\}$, over some finite alphabet Σ as illustrated in **?? ?**.

6 String Matching

String matching is both an algorithmic problem and data structure problem. The static data structure consist of preprocessing some predefined large text $T = \{t_1, t_2, \ldots, t_m\}$, and query some smaller pattern $P = \{p_1, p_2, ..., p_n\}$ **?**. The objective is to preprocess text T and query pattern P in text T in linear time, $O(m), m \in |T|$ [1] and $O(n), n \in |P|$, respectively **?**.
Following the tradition of Gusfield et al. this discussion begins with the naive methoed of the exact string matching paradigm.

Problem:

> Given a pattern P and a long text T, the problem consist of finding all occurrences of pattern P, if any, in text T **?**.

The occurrences of pattern $P = \{ana\}$ in text $T = \{banana\}$ are found at $T[1, 3]$ and $T[3, 5]$, as illustrated in Figure **??**. Note that pattern P may overlap.

[1] See **??** for a description of algorithmic time analysis

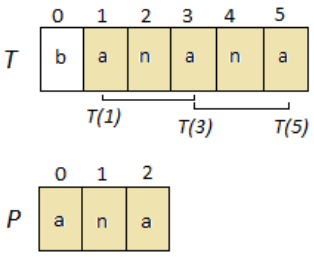


Figure 1: The text T={banana} and pattern P={ana} over the alphabet Σ={abn}. The pattern P occours in T in, at position $T[1]$ and $T[3]$. Notice that occurrences of P may overlap.

6.1 Suffix trees

The classic application for suffix tree is the substring problem **??**, which is both a data structure -and an algorithmic problem **?**. That is, given a long text T over some alphabet Σ, and some pattern P, the substring problem consist of preprocessing T in linear time $O(m)$, and hereafter T should be able to take any unknown pattern P, and in linear time $O(n)$ determine occurrences of P, if any, in T **?**. The preprocessing time is here proportional to the length of text T, and the query is proportional to the length of pattern P **?**. This paper adopts the approach of Gusfield et al., by not applying the denotation of pattern P and text T, in respect to describing suffix trees. By using the general description and denotation of suffix trees, there will be less confusion, since input string can take different roles and vary for application to application **?**.

Conceptually a suffix tree is a compressed trie **?**.

> **Definition** A trie contains all suffixes of string S, where each edge is labeled with a character from some alphabet Σ. Each path from root to leaf represent a suffix, and every suffix is represented with some path from root to leaf **??**.

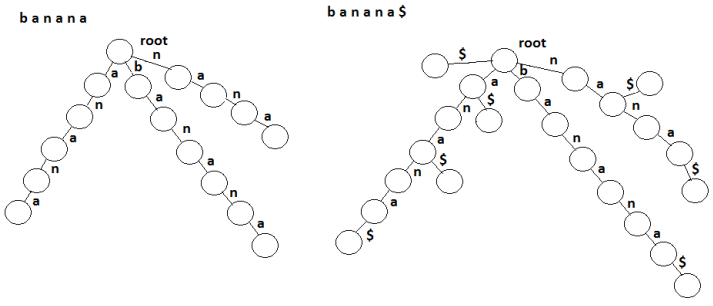


Figure 2: Left is a trie of the string *banana* and the right is a trie of the string *banana*\$.

Figure **??** illustrates two tries, left of the string banana and the right over the string *banana*\$. Note that right trie has the termination character \$ appended to the end. This is due to the fact that the definition of a trie dictates that every suffix is represented with some path from root to leaf. Suffix *ana* in left trie does not have a path from root to leaf,

but appending a termination character to S that exists nowhere else in the string, will eliminate the problem.
Creating a compressed trie, one takes each non-branching nodes and compress them, such that edge-labels from non-branching nodes concatenates into a new edge-label, as illustrated in Figure **??**. Here node 1 is a non-branching node, one then concatenate a to n, to form a new edge-label na, deleting the non-branching node **?**. The number of non-branching nodes in a trie is at most the number of leaves. By compressing, we know have that the number of internal nodes is at most the number of leaves, having O(k) nodes total.

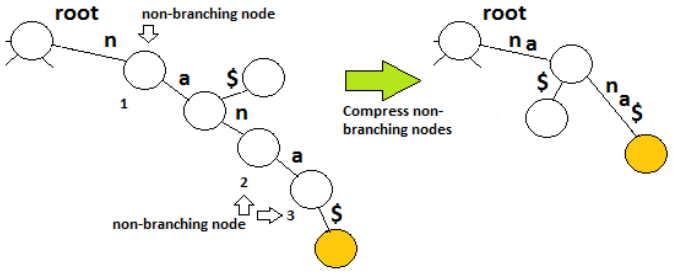


Figure 3: Compressing a trie.

Definition A Suffix tree, T, is a m-character string S concatenated with a termination character \$, that is represented as a directed rooted tree with exactly m leaves, numbered 1 to m. Except the root, each internal node contains at least two children, with each edge labeled with a nonempty substring of S. No two edges exiting a node can have labels beginning with the same character. The concatenation of edge-labels on the path from the root to leaf i, unerringly spells out the suffix of S that starts at position i, such that it spells out $S[i..m]$. The termination character \$ is assumed to appear nowhere else in S, such that no suffix of the consequential string can be a prefix of any other suffix**?**.

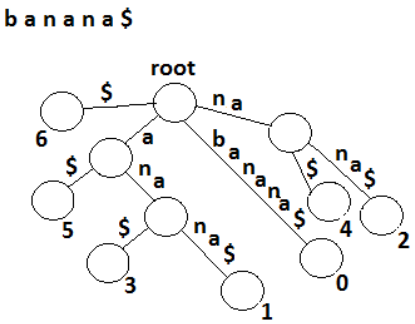


Figure 4: A suffix tree T for string *banana*\$.

The suffix tree for the string *banana*\$, in lexicographical order, is illustrated **??**. Each path from the root to a leaf i, unerringly spells out a suffix of S, starting at position i

in S. As an example, leaf numbered 2 spells out $nana\$$, starting at position 2 in the S, such that $S[2..6] = nana\$$. Each node has at least two children, and no two edges exiting a node begins with the same character.
To dive into the substring problem using linear preprocessing time, $O(m)$, and linear search time, $O(n)$ we follow the tradition, and starts with a naive and straightforward algorithm to building suffix trees before verturing into the linear time preprocessing approach **?**.

6.2 Operations on suffix trees

Bla bla bla...

6.2.1 Insertion & Deletion

6.2.2 Lowest Common Ancestor

An interesting application of suffix trees is the *lca* (Lowest Common Ancestor) problem, that is, finding the lowest common ancestor of node i and j in tree T. Lowest common ancestor was first obtained by Harel and Tarjan (1984, published online 2006 **?**) and later on simplified by Schieber and Vishkin (1988, published online in 2006 **?**)**?**.
Lowest common ancestor is an interesting application given that it is used in application as exact matching with wild cards and the k-mismatch problem, amongst others **?**. More interesting is the fact that *lca* of leaves i and j identifies the longest common prefix of suffixes i and j, which will be discussed later on.
By consuming linear time amount of preprocessing a suffix tree, that is a rooted tree, any two nodes can be identified and their *lca* can be found in constant time, $O(1)$ **??**. This paper will not dwell into the different linear time preprocessing algorithms for the *lca* predicament, but delivers an overview and clarification of the problem by introducing a simpler but slower algorithm. (maybe linear in the appendix?).

> **Definition** In a rooted tree T a node u is an ancestor of node v, if u is an unique path from the root to v **?**.

> **Definition** In a rooted tree T, the lowest common ancestor of two node u and v, is the deepest node in tree T that is an ancestor of both u and v **?**.

Let's suppose for simplification that an application is allowed preprocessing time of an upper bound of $\theta(nlgn)$, which is an acceptable bound for most applications **?**. Then, in the preprocessing state of tree T, perform a deept-first traversal of tree T and create a list L of nodes in order as they are visited. Then locating the *lca* of node 2 and 8, $lca[2, 8]$, in **??**, one only have to find any occurrences of 2 and 8 in L. Then take the lowest value in interval between $L[1] = 2$ and $L[12] = 8$. This value is the lowest common ancestor for node 2 and 8 in T, $lca[2, 8] = 1$.

6.2.3 Longest Common Prefix

What are the usages

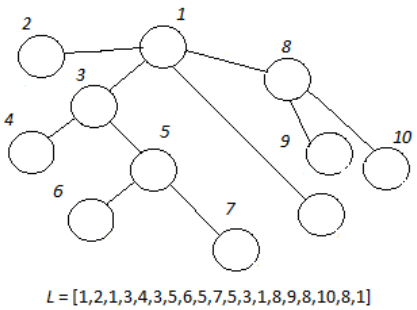


Figure 5: Rooted tree - deept-first travesal with $L = [1, 2, 1, 3, 4, 3, 5, 6, 5, 7, 5, 3, 1, 8, 9, 8, 10, 8, 1]$

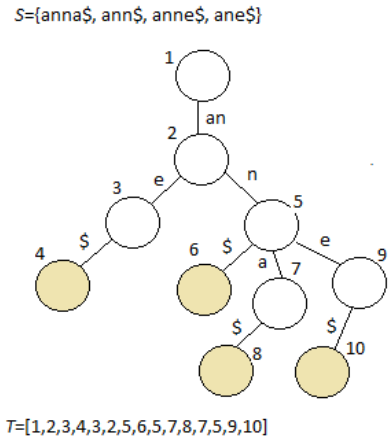


Figure 6: Rooted tree - deept-first travesal with $L = [1, 2, 1, 3, 4, 3, 5, 6, 5, 7, 5, 3, 1, 8, 9, 8, 10, 8, 1]$

6.2.4 Predecessor & Successor Amongst Strings

1 side

6.2.5 Lowest Common Extension

1 side

6.3 Space & Time Complexity

6.4 Suffix Trees To Suffix Arrays In Linear Time

6.5 Suffix Arrays

[0] two efficient algorithms

The performance of linear time suffix sorting algorithms
A new approach for suffix array construction

Suffix array are space efficient alternatives to suffix trees[0,1]. Before Manber and Meyers in 1990 introduced the first direct suffix array construction algorithm – SACA, suffix arrays were constructed using lexicographical-order traversal of suffix trees [0,1,2,3]. Manber and Meyers made suffix trees obsolete in respect to constructing suffix arrays, and their approach is known as a doubling algorithm, where with each sorting pass, doubles the depth to which each suffix are sorted. This means that suffixes are sorted in logarithmic number of passes, providing a worst case bound of $O(nlogn)$ and $O(n)$ expected, assuming linear sort, reminiscent of Radix Sort[2] and queries can be answered in $O(P + logn)$ with use of Binary Search[1].
With the discovery of four different SACAs requiring only $O - (n)$ time worst case in 2003, the situation drastically changed. SACAs have since been the focus of intense research[2,3]. In 2005 Joong Chae Na introduced more linear time SACAs, where two stood out, the Ko-Aluru (KA) algorithm for supplying good performance in practice and the Kärkkäinen-Sanders algorithm for its elegance [3].
According to a survey paper, SACAs have to fulfill three important requirements:

1. The algorithm should run in asymptotic minimal worst case time, where linear is an optimal way[3].

2. The algorithm should run fast in practice [3].

3. The algorithm should consume as less extra space in addition to the text and suffix array as possible, where constant amount is optimal[2].

Although no current SACAs fulfill the requirements in an optimal way, research into faster and more space reducing SACAs continued[3]. Later on, in 2009, Nong et al. introduced two new linear time construction algorithms, one which outperformed most known and existing SACAs, called Suffix Array Induced Sorting SA-IS algorithm, guaranteeing asymptotic linear time and almost optimal space requirements[3].

6.6 Suffix Array Induced Sorting Algorithm (SA-IS)

6.7 SA-IS - Correctness & Completeness

5-8 sider

6.8 SA-IS - Linear Time Preprocessing

2 sider

6.9 Compressed Suffix Arrays - Burrows-Wheeler Transform

6.10 Block-Size Compression

6.11 Operations on suffix arrays

2-3 sider

A One

Etiam pede massa, dapibus vitae, rhoncus in, placerat posuere, odio. Vestibulum luctus commodo lacus. Morbi lacus dui, tempor sed, euismod eget, condimentum at, tortor. Phasellus aliquet odio ac lacus tempor faucibus. Praesent sed sem. Praesent iaculis. Cras rhoncus tellus sed justo ullamcorper sagittis. Donec quis orci. Sed ut tortor quis tellus euismod tincidunt. Suspendisse congue nisl eu elit. Aliquam tortor diam, tempus id, tristique eget, sodales vel, nulla. Praesent tellus mi, condimentum sed, viverra at, consectetuer quis, lectus. In auctor vehicula orci. Sed pede sapien, euismod in, suscipit in, pharetra placerat, metus. Vivamus commodo dui non odio. Donec et felis.