58093 String Processing Algorithms

Lectures, Autumn 2015, period II

Juha Kärkkäinen

Contents

0. Introduction

Strings and sequences are one of the simplest, most natural and most used forms of storing information.

- natural language, biosequences, programming source code, XML, music, any data stored in a file
- Many algorithmic techniques are first developed for strings and later generalized for more complex types of data such as graphs.

The area of algorithm research focusing on strings is sometimes known as stringology. Characteristic features include

- Huge data sets (document databases, biosequence databases, web crawls, etc.) require efficiency. Linear time and space complexity is the norm.
- Strings come with no explicit structure, but many algorithms discover implicit structures that they can utilize.

About this course

On this course we will cover a few cornerstone problems in stringology. We will describe several algorithms for the same problems:

- the best algorithms in theory and/or in practice
- algorithms using a variety of different techniques

The goal is to learn a toolbox of basic algorithms and techniques.

On the lectures, we will focus on the clean, basic problem. Exercises may include some variations and extensions. We will mostly ignore any application specific issues.

Strings

An alphabet is the set of symbols or characters that may occur in a string. We will usually denote an alphabet with the symbol Σ and its size with σ.

We consider three types of alphabets:

- General alphabet: An arbitrary set $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$. We assume that the alphabet is *ordered*: $c_1 < c_2 < \cdots < c_\sigma$.
- Integer alphabet: $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$.
- Constant alphabet: A general alphabet for a (small) constant σ.

The alphabet types are really used for classifying and analysing algorithms rather than alphabets:

- Algorithms for ordered alphabet use only *character comparisons*.
- Algorithms for integer alphabet can use more powerful operations such as using a *symbol as an address* to a table or *arithmetic operations* to compute a *hash function*.
- Algorithms for constant alphabet can perform almost *any operation* on characters and even sets of characters in *constant time*.

The assumption of a constant alphabet in the analysis of an algorithm often indicates one of two things:

- The effect of the alphabet on the complexity is complicated and the constant alphabet assumption is used to *simplify the analysis*.
- The time or space complexity of the algorithm is heavily (e.g., linearly) dependent on the alphabet size and the algorithm is effectively *unusable for large alphabets*.

An algorithm is called alphabet-independent if its complexity has no dependence on the alphabet size.

Example 0.1: Suppose we want to count how many distinct characters occur in a given string. We can do this by maintaining a *set of distinct characters*. Each character of the string is added to the set if not already there. The type of alphabet determines the choice of the set data structure:

- For general alphabet, we can use a *balanced binary search tree*.
- For integer alphabet, we can use an *array* of size σ, which is much faster.
- If σ is large, an array of size σ might be too big. We could use a *hash table* instead.
- For constant alphabet, we could even use an *unordered list* without affecting the time complexity.

An alternative solution is to sort the characters, which groups identical characters together. The choice of the sorting algorithm depends on the alphabet.

A string is a sequence of symbols. The set of all strings over an alphabet Σ is

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

where

$$\begin{aligned}\Sigma^k &= \overbrace{\Sigma \times \Sigma \times \cdots \times \Sigma}^{k} \\ &= \{a_1 a_2 \ldots a_k \mid a_i \in \Sigma \text{ for } 1 \le i \le k\} \\ &= \{(a_1, a_2, \ldots, a_k) \mid a_i \in \Sigma \text{ for } 1 \le i \le k\}\end{aligned}$$

is the set of strings of length k. In particular, $\Sigma^0 = \{\varepsilon\}$, where ε is the empty string.

We will usually write a string using the notation $a_1 a_2 \ldots a_k$, but sometimes using $(a_1, a_2, \ldots, a_k)$ may avoid confusion.

There are many notations for strings.

When describing algorithms, we will typically use the array notation to emphasize that the string is stored in an array:

$$S = S[1..n] = S[1]S[2] \dots S[n]$$
$$T = T[0..n) = T[0]T[1] \dots T[n-1]$$

Note the half-open range notation $[0..n)$ which is often convenient.

In an abstract context, we often use other notations, for example:

- $\alpha, \beta \in \Sigma^*$
- $x = a_1 a_2 \dots a_k$ where $a_i \in \Sigma$ for all i
- $w = uv$, $u, v \in \Sigma^*$ (w is the concatenation of u and v)

We will use $|w|$ to denote the length of a string w.

Individual characters or their positions usually do not matter. The significant entities are the substrings or factors.

Definition 0.2: Let $w = xyz$ for any $x, y, z \in \Sigma^*$. Then x is a prefix, y is a factor (substring), and z is a suffix of w.
If x is both a prefix and a suffix of w, then x is a border of w.

Example 0.3: Let $w = \mathtt{bonobo}$. Then

- $\varepsilon, \mathtt{b}, \mathtt{bo}, \mathtt{bon}, \mathtt{bono}, \mathtt{bonob}, \mathtt{bonobo}$ are the prefixes of w
- $\varepsilon, \mathtt{o}, \mathtt{bo}, \mathtt{obo}, \mathtt{nobo}, \mathtt{onobo}, \mathtt{bonobo}$ are the suffixes of w
- $\varepsilon, \mathtt{bo}, \mathtt{bonobo}$ are the borders of w
- $\varepsilon, \mathtt{b}, \mathtt{o}, \mathtt{n}, \mathtt{bo}, \mathtt{on}, \mathtt{no}, \mathtt{ob}, \mathtt{bon}, \mathtt{ono}, \mathtt{nob}, \mathtt{obo}, \mathtt{bono}, \mathtt{onob}, \mathtt{nobo}, \mathtt{bonob}, \mathtt{onobo}, \mathtt{bonobo}$ are the factors of w.

Note that ε and w are always suffixes, prefixes, and borders of w.
A suffix/prefix/border of w is proper if it is not w, and nontrivial if it is not ε or w.

Some Interesting Strings

The Fibonacci strings are defined by the recurrence:

$$F_0 = \varepsilon$$
$$F_1 = \mathtt{b}$$
$$F_2 = \mathtt{a}$$
$$F_i = F_{i-1}F_{i-2} \text{ for } i > 2$$

Example 0.4:

$F_3 = \mathtt{ab}$
$F_4 = \mathtt{aba}$
$F_5 = \mathtt{abaab}$
$F_6 = \mathtt{abaababa}$
$F_7 = \mathtt{abaababaabaab}$
$F_8 = \mathtt{abaababaabaababaababa}$

The infinite Fibonacci string is the limit F_∞.
For all $i > 1$, F_i is a prefix of F_∞.

Fibonacci strings have many interesting properties:

- $|F_i| = f_i$, where f_i is the ith Fibonacci *number*.
- F_∞ has exactly $k + 1$ distinct factors of length k.
- For all $i > 1$, we can obtain F_i from F_{i-1} by applying the substitutions $\mathtt{a} \mapsto \mathtt{ab}$ and $\mathtt{b} \mapsto \mathtt{a}$ to every character.

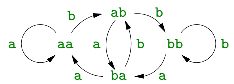
A De Bruijn sequence B_k of order k for an alphabet Σ of size σ is a cyclic string of length σ^k that contains every string of length k over the alphabet Σ as a factor exactly once. The cycle can be opened into a string of length $\sigma^k + k - 1$ with the same property.

Example 0.5: De Bruijn sequences for the alphabet $\{\mathtt{a}, \mathtt{b}\}$:

$$B_2 = \mathtt{aabb}(\mathtt{a})$$
$$B_3 = \mathtt{aaababbb}(\mathtt{aa})$$
$$B_4 = \mathtt{aaaabaabbababbbb}(\mathtt{aaa})$$

De Bruijn sequences are not unique. They can be constructed by finding Eulerian or Hamiltonian cycles in a De Bruijn graph.

Example 0.6: De Bruijn graph for the alphabet $\{\mathtt{a}, \mathtt{b}\}$ that can be used for constructing B_2 (Hamiltonian cycle) or B_3 (Eulerian cycle).



1. Sets of Strings

Basic operations on a set of objects include:

Insert: Add an object to the set

Delete: Remove an object from the set.

Lookup: Find if a given object is in the set, and if it is, possibly return some data associated with the object.

There can also be more complex queries:

Range query: Find all objects in a given range of values.

There are many other operations too but we will concentrate on these here.

An efficient execution of the operations requires that the set is stored as a suitable data structure.

- A (balanced) binary search tree supports the basic operations in $\mathcal{O}(\log n)$ time and range searching in $\mathcal{O}(\log n + r)$ time, where n is the size of the set and r is the size of the result.
- An ordered array supports lookup and range searching in the same time as binary search trees. It is simpler, faster and more space efficient in practice, but does not support insertions and deletions.
- A hash table supports the basic operations in constant time (usually randomized) but does not support range queries.

A data structure is called dynamic if it supports insertions and deletions (tree, hash table) and static if not (array). Static data structures are constructed once for the whole set of objects. In the case of an ordered array, this involves another important operation, sorting. Sorting can be done in $\mathcal{O}(n \log n)$ time using comparisons and even faster for integers.

The above time complexities assume that basic operations on the objects including comparisons can be performed in constant time. When the objects are strings, this is no more true:

- The worst case time for a string comparison is the length of the shorter string. Even the average case time for a random set of n strings is $\mathcal{O}(\log_\sigma n)$ in many cases, including for basic operations in a balanced binary search tree. We will show an even stronger result for sorting later. And sets of strings are rarely fully random.
- Computing a hash function is slower too. A good hash function depends on all characters and cannot be computed faster than the length of the string.

For a string set $\mathcal{R}$, there are also new types of queries:

Prefix query: Find all strings in $\mathcal{R}$ that have the query string S as a prefix. This is a special type of range query.

Lcp (longest common prefix) query: What is the length of the longest prefix of the query string S that is also a prefix of some string in $\mathcal{R}$.

Thus we need special set data structures and algorithms for strings.

Trie

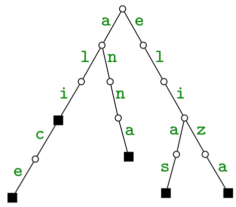
A simple but powerful data structure for a set of strings is the trie. It is a rooted tree with the following properties:

- Edges are labelled with symbols from an alphabet Σ.
- For every node v, the edges from v to its children have different labels.

Each node represents the string obtained by concatenating the symbols on the path from the root to that node.

The trie for a string set $\mathcal{R}$, denoted by $trie(\mathcal{R})$, is the smallest trie that has nodes representing all the strings in $\mathcal{R}$. The nodes representing strings in $\mathcal{R}$ may be marked.

Example 1.1: $trie(\mathcal{R})$ for $\mathcal{R} = \{\mathtt{ali}, \mathtt{alice}, \mathtt{anna}, \mathtt{elias}, \mathtt{eliza}\}$.



The trie is conceptually simple but it is not simple to implement efficiently. The time and space complexity of a trie depends on the implementation of the child function:

> For a node v and a symbol $c \in \Sigma$, *child*(v, c) is u if u is a child of v and the edge (v, u) is labelled with c, and *child*$(v, c) = \perp$ (null) if v has no such child.

As an example, here is the insertion algorithm:

Algorithm 1.2: Insertion into trie
Input: $trie(\mathcal{R})$ and a string $S[0..m) \notin \mathcal{R}$
Output: $trie(\mathcal{R} \cup \{S\})$
(1) $v \leftarrow root$; $j \leftarrow 0$
(2) while *child*$(v, S[j]) \neq \perp$ do
(3) $\quad v \leftarrow$ *child*$(v, S[j])$; $j \leftarrow j + 1$
(4) while $j < m$ do
(5) $\quad$ Create new node u (initializes *child*(u, c) to $\perp$ for all $c \in \Sigma$)
(6) $\quad$ *child*$(v, S[j]) \leftarrow u$
(7) $\quad v \leftarrow u$; $j \leftarrow j + 1$
(8) Mark v as representative of S

There are many implementation options for the child function including:

Array: Each node stores an array of size σ. The space complexity is $\mathcal{O}(\sigma N)$, where N is the number of nodes in $trie(\mathcal{R})$. The time complexity of the child operation is $\mathcal{O}(1)$. Requires an integer alphabet.

Binary tree: Replace the array with a binary tree. The space complexity is $\mathcal{O}(N)$ and the time complexity $\mathcal{O}(\log \sigma)$. Works for a general alphabet.

Hash table: One hash table for the whole trie, storing the values *child*$(v, c) \neq \perp$. Space complexity $\mathcal{O}(N)$, time complexity $\mathcal{O}(1)$. Requires an integer alphabet.

A common simplification in the analysis of tries is to assume a constant alphabet. Then the implementation does not matter: Insertion, deletion, lookup and lcp query for a string S take $\mathcal{O}(|S|)$ time.

Note that a trie is a complete representation of the strings. There is no need to store the strings separately.

Prefix free string sets

Many data structures and algorithms for a string set $\mathcal{R}$ become simpler if $\mathcal{R}$ is prefix free.

Definition 1.3: A string set $\mathcal{R}$ is prefix free if no string in $\mathcal{R}$ is a prefix of another string in $\mathcal{R}$.

There is a simple way to make any string set prefix free:

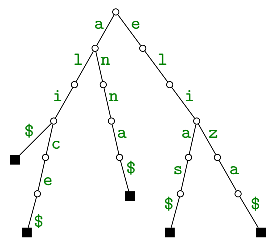
- Let $\$ \notin \Sigma$ be an extra symbol satisfying $\$ < c$ for all $c \in \Sigma$.
- Append $ to the end of every string in $\mathcal{R}$.

This has little or no effect on most operations on the set. The length of each string increases by one only, and the additional symbol could be there only virtually.

Example 1.4: The set {ali, alice, anna, elias, eliza} is not prefix free because ali is a prefix of alice, but {ali$, alice$, anna$, elias$, eliza$} is prefix free.

If $\mathcal{R}$ is prefix free, the leaves of $trie(\mathcal{R})$ represent exactly $\mathcal{R}$. This simplifies the implementation of the trie.

Example 1.5: The trie for {ali$, alice$, anna$, elias$, eliza$}.



Compact Trie

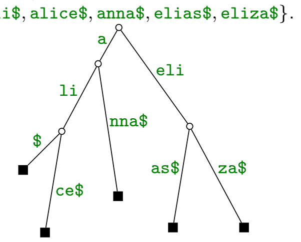
Tries suffer from a large number of nodes, close to $||\mathcal{R}||$ in the worst case.

- For a string set $\mathcal{R}$, we use $|\mathcal{R}|$ to denote the number of strings in $\mathcal{R}$ and $||\mathcal{R}||$ to denote the total length of the strings in $\mathcal{R}$.

The space requirement can be problematic, since typically each node needs much more space than a single symbol.

Compact tries reduce the number of nodes by replacing branchless path segments with a single edge.

Example 1.6: Compact trie for {ali$, alice$, anna$, elias$, eliza$}.



The space complexity of a compact trie is $\mathcal{O}(|\mathcal{R}|)$ (in addition to the strings):

- In a compact trie, every internal node (except possibly the root) has at least two children. In such a tree, there is always at least as many leaves as internal nodes. Thus the number of nodes is at most $2|\mathcal{R}|$.
- The egde labels are factors of the input strings. If the input strings are stored separately, the edge labels can be represented in constant space using pointers to the strings.

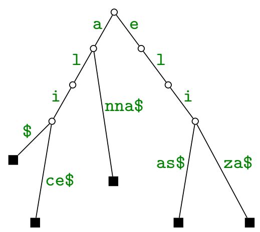
The time complexities are the same or better than for tries:

- An insertion adds and a deletion removes at most two nodes.
- Lookups may execute fewer calls to the child operation, though the worst case complexity is the same.
- Prefix and range queries are faster even on a constant alphabet (exercise).

There is also an intermediate form of trie called leaf-path-compacted trie, where branchless path segments are compacted only when they end in a leaf.

- Typically (though not in the worst case) this achieves most of the advantages of a compact trie.
- For trie algorithms, this means stopping the normal search, when only one string is remaining in the subtree.

Example 1.7: Leaf-path-compacted trie for {ali$, alice$, anna$, elias$, eliza$}.



Ternary Trie

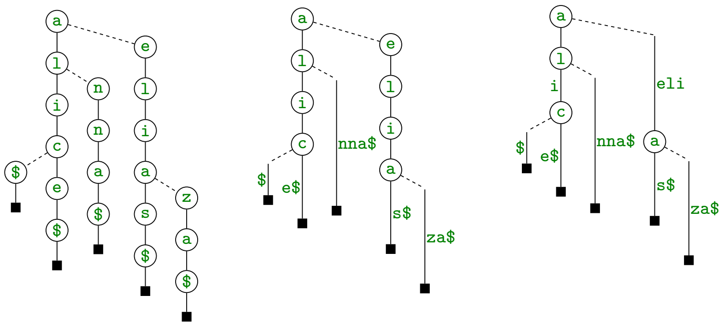
Tries can be implemented for ordered alphabets but a bit awkwardly using a comparison-based child function. Ternary trie is a simpler data structure based on symbol comparisons.

Ternary trie is like a binary search tree except:

- Each internal node has three children: smaller, equal and larger.
- The branching is based on a single symbol at a given position as in a trie. The position is zero (first symbol) at the root and increases along the middle branches but not along side branches.

There are also compact ternary tries and leaf-path-compated ternary tries based on compacting branchless path segments.

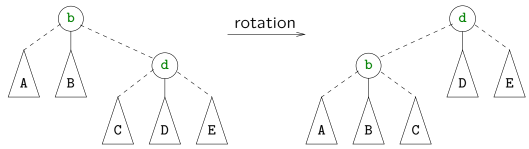
Example 1.8: Ternary tries for $\{$ali\$, alice\$, anna\$, elias\$, eliza\$$\}$.



Ternary tries have the same asymptotic size as the corresponding (σ-ary) tries.

A ternary trie is balanced if each left and right subtree contains at most half of the strings in its parent tree.

- The balance can be maintained by rotations similarly to binary search trees.



- We can also get reasonably close to a balance by inserting the strings in the tree in a random order.

Note that there is no restriction on the size of the middle subtree.

In a balanced ternary trie each step down either

- moves the position forward (middle branch), or
- halves the number of strings remaining in the subtree (side branch).

Thus, in a balanced ternary trie storing n strings, any downward traversal following a string S passes at most $|S|$ middle edges and at most $\log n$ side edges.

Thus the time complexity of insertion, deletion, lookup and lcp query is $\mathcal{O}(|S| + \log n)$.

In comparison based tries, where the *child* function is implemented using binary search trees, the time complexities could be $\mathcal{O}(|S| \log \sigma)$, a multiplicative factor $\mathcal{O}(\log \sigma)$ instead of an additive factor $\mathcal{O}(\log n)$.

Prefix and range queries behave similarly (exercise).