58093 String Processing Algorithms

Lectures, Autumn 2012, 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

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:

- *Ordered alphabet* $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$, where $c_1 < c_2 < \cdots < c_\sigma$.
- *Integer alphabet* $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$.
- *Constant alphabet* An ordered alphabet for a (small) constant σ.

The alphabet types are really used for *classifying 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.

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

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

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:

$$\begin{aligned} S &= S[1..n] = S[1]S[2]\ldots S[n] \\ T &= T[0..n) = T[0]T[1]\ldots T[n-1] \end{aligned}$$

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

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

- $\alpha, \beta \in \Sigma^*$
- $x = a_1 a_2 \ldots a_k$ 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 0.1: 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 0.2: 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.

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 *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 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 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

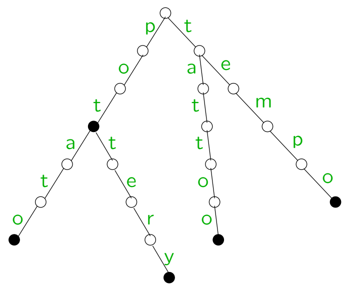
A simple but powerful data structure for a set of strings is the *trie*. It is a *rooted tree* with the following properties:

- 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 strings 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 $\mathcal{R} = \{\texttt{pot}, \texttt{potato}, \texttt{pottery}, \texttt{tattoo}, \texttt{tempo}\}$.



The trie is not only a practical data structure but a useful tool for thinking about a set of strings in a more abstract level. We will illustrate many basic concepts on string sets by relating them to the trie, starting with:

Definition 1.2: The *prefix closure* of a string set $\mathcal{R}$ is the set of all prefixes of all strings in $\mathcal{R}$:

$$prefix\_closure(\mathcal{R}) = \{u \in \Sigma^* \mid u \text{ is a prefix of } v \text{ for some } v \in \mathcal{R}\} .$$

A string set $\mathcal{R}$ is *prefix closed* if $\mathcal{R} = prefix\_closure(\mathcal{R})$.

The nodes of $trie(\mathcal{R})$ represent exactly $prefix\_closure(\mathcal{R})$. Thus the number of nodes in $trie(\mathcal{R})$ is $|prefix\_closure(\mathcal{R})|$. The number of edges is always one smaller than the number of nodes.

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*$(v, c) = \perp$ if v has no such child.

As an example, here is the insertion algorithm:

Algorithm 1.3: Insertion into trie
Input: $trie(\mathcal{R})$ and a string $S[0..m) \notin \mathcal{R}$
Output: $trie(\mathcal{R} \cup \{\mathcal{S}\})$

```
v ← root; j ← 0
while child(v, S[j]) ≠ ⊥ do
    v ← child(v, S[j]); j ← j + 1
while j < m do
    Create new node u (initializes child(u, c) to ⊥ for all c ∈ Σ)
    child(v, S[j]) ← u
    v ← u; j ← j + 1
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 $\mathcal{O}(1)$.

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

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

Array and hash table implementations require an integer alphabet; the binary tree implementation works for an ordered 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 elsewhere.

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

Definition 1.4: A string set $\mathcal{R}$ is *prefix free* if no string in $\mathcal{R}$ is a prefix of another string in $\mathcal{R}$.

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

- Only internal nodes need the child data structure.
- Only leaves need the representation markers.

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. The length of each string increases by one only, and the additional symbol could be there only virtually.