58093 String Processing Algorithms

Lectures, Fall 2010, period II

Juha Kärkkäinen

Who is this course for?

- Master's level course in Computer Science, 4 cr
- Subprogram of Algorithms and Machine Learning
  - Together with Project in String Processing Algorithms (in the next teaching period) one of the three special course combinations, one of which must be included in the Master's degree.
- Suitable addition to Master's degree program for Bioinformatics, particularly for those interested in biological sequence analysis
- Prerequisites: basic knowledge on algorithms, data structures, finite automata, algorithm analysis.
  - Recommended prerequisite courses: Data Structures,Models of Computation, Designs and Analysis of Algorithms

Course components

Lectures: Tue and Thu 12–14 in B222

- Lecture notes will be available on the course home page. Attending lectures is not obligatory but can be useful. There will be additional examples and explanations.
- The lectures do not follow any book, but the books mentioned on the home page can provide useful additional material.

Exercises: Thu 14-16 in B119 and 16-18 in BK107 (starting 11.11.)

- Exercise problems will be available on the course home page a week before the exercise session. Model solutions will be available on the home page after the exercise session.
- You should solve the problems at home and be prepared to present your solutions at the exercise session. You do not need to solve all problems but additional points are awarded according to how many problems you have solved (see later).

Course components (continued)

Exam: Thu 16.12. at 16–19 (check time and place later)

- The exam consists of problems similar to those in the exercises. No notes or other material is allowed in the exam.
- If you fail, there is a renewal exam (exercise points count) and separate exams (no exercise points).

In the next period, there is a follow up course Project in String Processing Algorithms (2 cr).

- Consist of implementing some algorithms on this course, experimenting with them, and presenting the results.

Grading

Exercises 1–10 points

- Scoring is based on the number of exercises solved.
- About 20% is *required* and gives 1 point.
- About 80% gives the maximum of 10 points.

Exam 25–50 points

- 25 points is *required*

Total 30–60 points

- 30 points is required to pass and gives the lowest grade 1.
- 50 points gives the highest grade 5.

Contents

0. Introduction
1. Exact string matching
   - How to find a pattern (string) in a text (string)
2. Approximate string matching
   - How to find in the text something that is similar to the pattern
3. Sorting and searching
   - How to compare strings efficiently in the context of sorting and search trees
4. Suffix tree and array
   - How to preprocess a long text for fast string matching and all kinds of other tasks

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 problem:

- 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 two types of alphabets:

- *Ordered alphabet* $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$, where $c_1 < c_2 < \cdots < c_\sigma$.
- *Integer alphabet* $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$.

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

- Algorithms for ordered alphabets use only character comparisons.
- Algorithms for integer alphabets can use operations such as using a symbol as an address to a table.

Algorithms for integer alphabets are more restricted in their applicability.

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:

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

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)

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

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 $w = \texttt{bonobo}$. Then

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

Let $T = T[0..n)$ be the *text* and $P = P[0..m)$ the *pattern*. We say that P *occurs* in T at position j if $T[j..j+m) = P$.

Example: $P =$ `aine` occurs at position 6 in $T =$ `karjalainen`.

In this part, we will describe algorithms that solve the following problem.

Problem 1.1: Given text $T[0..n)$ and pattern $P[0..m)$, report the first position in T where P occurs, or n if P does not occur in T.

The algorithms can be easily modified to solve the following problems too.

- Is P a factor of T?
- Count the number of occurrences of P in T.
- Report all occurrences of P in T.

The naive, brute force algorithm compares P against $T[0..m)$, then against $T[1..m+1)$, then against $T[2..m+2)$ etc. until an occurrence is found or the end of the text is reached.

Algorithm 1.2: Brute force
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T
(1) $i \leftarrow 0; j \leftarrow 0$
(2) *while* $i < m$ *and* $j < n$ *do*
(3) *if* $P[i] = T[j]$ *then* $i \leftarrow i + 1; j \leftarrow j + 1$
(4) *else* $j \leftarrow j - i + 1; i \leftarrow 0$
(5) *if* $i = m$ *then* output $j - m$ *else* output n

The worst case time complexity is $\mathcal{O}(mn)$. This happens, for example, when $P = \mathtt{a}^{m-1}\mathtt{b} = \mathtt{aaa..ab}$ and $T = \mathtt{a}^n = \mathtt{aaaaaa..aa}$.

Knuth–Morris–Pratt

The Brute force algorithm forgets everything when it moves to the next text position.

The Morris–Pratt (MP) algorithm remembers matches. It never goes back to a text character that already matched.

The Knuth–Morris–Pratt (KMP) algorithm remembers mismatches too.

Example 1.3:

Brute force	Morris–Pratt	Knuth–Morris–Pratt
ainaisesti-ainainen	ainaisesti-ainainen	ainaisesti-ainainen
ainai~~n~~en (6 comp.)	ainai~~n~~en (6)	ainai~~n~~en (6)
~~a~~inainen (1)	ai~~n~~ainen (1)	~~a~~inainen (1)
~~a~~inainen (1)	~~a~~inainen (1)	
ai~~n~~ainen (3)		
~~a~~inainen (1)		
~~a~~inainen (1)		

MP and KMP algorithms never go backwards in the text. When they encounter a mismatch, they find another pattern position to compare against the same text position. If the mismatch occurs at pattern position i, then $fail[i]$ is the next pattern position to compare.

The only difference between MP and KMP is how they compute the *failure function* $fail$.

Algorithm 1.4: Knuth–Morris–Pratt / Morris–Pratt
Input: text $T = T[0 \ldots n)$, pattern $P = P[0 \ldots m)$
Output: position of the first occurrence of P in T
(1) compute $fail[0..m]$
(2) $i \leftarrow 0; j \leftarrow 0$
(3) *while* $i < m$ *and* $j < n$ *do*
(4) *if* $i = -1$ *or* $P[i] = T[j]$ *then* $i \leftarrow i + 1; j \leftarrow j + 1$
(5) *else* $i \leftarrow fail[i]$
(6) *if* $i = m$ *then* output $j - m$ *else* output n

- $fail[i] = -1$ means that there is no more pattern positions to compare against this text positions and we should move to the next text position.
- $fail[m]$ is never needed here, but if we wanted to find all occurrences, it would tell how to continue after a full match.

We will describe the MP failure function here. The KMP failure function is left for the exercises.

- When the algorithm finds a mismatch between $P[i]$ and $T[j]$, we know that $P[0..i) = T[j-i..j)$.
- Now we want to find a new $i' < i$ such that $P[0..i') = T[j-i'..j)$. Specifically, we want the largest such i'.
- This means that $P[0..i') = T[j-i'..j) = P[i-i'..i)$. In other words, $P[0..i')$ is the *longest proper border* of $P[0..i)$.

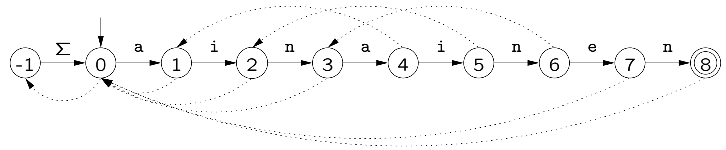
Example: `ai` is the longest proper border of `ainai`.

- Thus $fail[i]$ is the length of the longest proper border of $P[0..i)$.
- $P[0..0) = \varepsilon$ has no proper border. We set $fail[0] = -1$.

Example 1.5: Let $P =$ `ainainen`.

i	$P[0..i)$	border	$fail[i]$
0	ε	–	-1
1	`a`	ε	0
2	`ai`	ε	0
3	`ain`	ε	0
4	`aina`	`a`	1
5	`ainai`	`ai`	2
6	`ainain`	`ain`	3
7	`ainaine`	ε	0
8	`ainainen`	ε	0

The (K)MP algorithm operates like an automaton, since it never moves backwards in the text. Indeed, it can be described by an automaton that has a special *failure transition*, which is an ε-transition that can be taken only when there is no other transition to take.



An efficient algorithm for computing the failure function is very similar to the search algorithm itself!

- In the MP algorithm, when we find a match $P[i] = T[j]$, we know that $P[0..i] = T[j-i..j]$. More specifically, $P[0..i]$ is the longest prefix of P that matches a suffix of $T[0..j]$.
- Suppose $T = \#P[1..m)$, where $\#$ is a symbol that does not occur in P. Finding a match $P[i] = T[j]$, we know that $P[0..i]$ is the longest prefix of P that is a proper suffix of $P[0..j]$. Thus $fail[j+1] = i+1$.

Algorithm 1.6: Morris–Pratt failure function computation
Input: pattern $P = P[0 \dots m)$
Output: array $fail[0..m]$ for P
(1) $i \leftarrow -1; j \leftarrow 0;\ fail[j] = i$
(2) *while* $j < m$ *do*
(3) *if* $i = -1$ *or* $P[i] = P[j]$ *then* $i \leftarrow i+1; j \leftarrow j+1;\ fail[j] = i$
(4) *else* $i \leftarrow fail[i]$
(5) output $fail$

- When the algorithm reads $fail[i]$ on line 4, $fail[i]$ has already been computed.

Theorem 1.7: Algorithms MP and KMP preprocess a pattern in time $\mathcal{O}(m)$ and then search the text in time $\mathcal{O}(n)$.

Proof. It is sufficient to count the number of comparisons that the algorithms make. After each comparison $P[i] = T[j]$ (or $P[i] = P[j]$), one of the two conditional branches is executed:

then Here j is incremented. Since j never decreases, this branch can be taken at most $n+1$ ($m+1$) times.

else Here i decreases since $fail[i] < i$. Since i only increases in the then-branch, this branch cannot be taken more often than the then-branch.

□