String Matching

Finding all occurrences of a pattern in a text is a problem that arises frequently in various contexts:

text-editing typically the text is a document being edited, and the pattern searched for is a particular word supplied by the user.

DNA mapping in this case we are interested in finding a particular pattern in a (long) DNA sequence.

WWW searching the text this time is the union of all web-pages in the internet (...)

Problem definition.

Given an alphabet $\mathcal{A}$, a text T (an array of n characters in $\mathcal{A}$) and a pattern P (another array of $m \leq n$ characters in $\mathcal{A}$), we say that P *occurs with shift* s *in* T (or P *occurs beginning at position* $s+1$ *in* T) if $0 \leq s \leq n-m$ and $T[s+j] = P[j]$ for $1 \leq j \leq m$. A shift is *valid* if P occurs with shift s in T and *invalid* otherwise. The string-matching problem is the problem of finding all valid shifts for a given choice of P and T.

Example

$$T \equiv \mathsf{tadadattaetadadadafa}$$

$$P \equiv \mathsf{dada}$$

Valid shifts are two, twelve and fourteen.

A Short History

The development of the algorithms that we'll be examining has an interesting history.

First came the obvious brute-force algorithm (still in widespread use).

Its worst-case running time is $O(nm)$, but the strings which arise in practice lead to a running time $O(n+m)$.

Furthermore, it is well suited to good architectural features on most computer systems, so an optimised version provides a "standard" which is difficult to beat with a clever algorithm.

In 1970, Steve Cook (Cook's theorem, do you remember?) proved a theoretical result about a particular type of abstract computer implying that an algorithm exists which solves the pattern-matching problem in time $O(n+m)$ (worst case). His theorem unfortunately did not explicitly provide an effective way to implement such an algorithm.

Donald Knuth and V. R. Pratt laboriously followed through the construction Cook used to prove his theorem to get an actual algorithm which they were then able to refine to be a relatively simple practical algorithm.

This seemed a rare and satisfying example of a theoretical result having immediate (and unexpected) practical applicability. But it turned out that J. H. Morris had discovered virtually the same algorithm as a solution to an annoying practical problem that confronted him when implementing a text editor (he didn't want to ever "back up" in the text string).

Knuth, Morris, and Pratt didn't get around to publishing their algorithm until 1976, and in the meantime R. S. Boyer and J. S. Moore (and, independently, R. W. Gosper) discovered an algorithm which is much faster in many applications, since it often examines only a fraction of the characters in the text string. Many text editors use this algorithm to achieve a noticeable decrease in response time for string searches.

Notations (1)

s, $\boxed{s}$ Strings will be denoted either by single letters or, occasionally, by boxed letters.

$|s|$ if s is a string, denotes the *length* of s, i.e. the number of characters in the string.

$\mathcal{A}^*$ ("A-star") the set of all finite-length strings formed using character from the alphabet $\mathcal{A}$.

ε the *empty string*, the unique string of length 0.

st is the *concatenation* of strings s and t, obtained by appending the characters of t after those of s. Clearly $|st| = |s| + |t|$.

Notations (2)

- A string w is a *prefix* of x, if $x \equiv wy$ for some string $y \in \mathcal{A}^*$. Of course the length of w cannot be larger than that of x.
- A string w is a *suffix* of x, if $x = yw$ for some string $y \in \mathcal{A}^*$. We have $|w| \leq |x|$.
- If x, y are both suffix of z then only three cases arise: if $|x| \leq |y|$ then x is also a suffix of y, else y is a suffix of x. If particular $|x| = |y|$ implies $x = y$.

Notations (3)

- We shall denote the k character prefix of a pattern, say, P, by P_k. Thus $P_0 = \varepsilon$, $P_m = P$, and the pattern matching problem is that of finding all shifts s such that P is a suffix of T_{s+m}.
- We will assume that checking equality between two strings takes time proportional to the length of the shortest of the two.

Brute-Force

The obvious method that immediately comes to mind is just to check, for each possible position in the text whether the pattern does in fact match the text.

```
BRUTE-MATCHING (T, P)
    n ← length(T)
    m ← length(P)
    for s ← 0 to n − m
        if P = T[s + 1..s + m]
            print "pattern occurs with shift s"
```

How good is BRUTE-MATCHING

The algorithm finds all valid shifts in time $\Theta((n-m+1)m)$.

It is easy to verify the upper bound by inspecting the code.

If $T = aa\ldots a$ (n times) and P is a substring of length m of T, the algorithm BRUTE-MATCHING will actually spend $O(m)$ time for each of the possible $n-m+1$ positions.

Examples

Consider the following text:

0101010101010100101010101010101010100101010101

and the pattern

010100

The algorithm is often very good in practice. Find the word joy in the following text:

> A very popular definition of Argumentation is the one given in an important Handbook, which in a way testifies the current period of good fortune which Argumentation Theory is enjoying. The definition characterises the process of argumentation as a "verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener, by putting forward a constellation of proposition intended to justify (or refute) the standpoint before a rational judge". Having described many of the characteristics of argumentation theory in the previous chapter, here we will concentrate on an important aspect of the arguing activity, that is that it is a process involving two entities: a listener and a speaker. Dialogue, therefore, is paramount to argumentation. The present chapter will examine the dialogic component of the argumentation activities, by providing a survey of literature in this subject, and concluding with our own approach to the treatment of this aspect.

The text contains only 4 substrings matching j, and just a single substring matching both jo and joy:

> A very popular definition of Argumentation is the one given in an important Handbook, which in a way testifies the current period of good fortune which Argumentation Theory is en j oying. The definition characterises the process of argumentation as a "verbal and social activity of reason aimed at increasing (or decreasing) the acceptability of a controversial standpoint for the listener, by putting forward a constellation of proposition intended to j ustify (or refute) the standpoint before a rational j udge". Having described many of the characteristics of argumentation theory in the previous chapter, here we will concentrate on an important aspect of the arguing activity, that is that it is a process involving two entities: a listener and a speaker. Dialogue, therefore, is paramount to argumentation. The present chapter will examine the dialogic component of the argumentation activities, by providing a survey of literature in this sub j ect, and concluding with our own approach to the treatment of this aspect.

What is wrong with BRUTE-MATCHING?

The inefficiency comes from not using properly the partial matches.

If a pattern like: 1010010100110111 has been matched to 10100101001... (and then a miss-match occurs) we may argue that we don't need to reset the pointer to the T string to consider 1 0100101001... .

The partial match (and our knowledge of the pattern) tells us that, at least, we may restart our comparison from 10100 101001... . The subsequent quest for a $O(n+m)$ algorithm focused on efficient ways of using this information.

String matching with finite automata

An alternative approach that, nearly, overcomes the inefficiencies of BRUTE-MATCHING is based on the idea of avoiding any backward movements on the text T by using a finite amount of information about portions of T that partially match P.

The most natural framework to describe such information is provided by the Theory of Finite State machines (or Automata).

The key idea is, given P, to first construct a finite-state automaton[a] and then "hack" the way in which the finite-state machine "recognises" T to perform the matching of P as well.

[a] i.e. its transition table.

Finite State Machines

A (deterministic) *finite state machine* or *automaton* M is defined by

$\mathcal{A}$ a finite alphabet;

Q, a finite set of *states*: $Q = \{q_0, q_1, q_2, \ldots, q_k\}$;

$S \in Q$, the *initial* state;

$F \subseteq Q$, the set of *final* states;

$\delta : Q \times \mathcal{A} \to Q$, the *state-transition function.*

Automata spend their lives crunching strings.

Given string in $\mathcal{A}^*$, the automaton will start in state S. It will then read one character at the time and while doing this it may decide to move from one state to another according to the state-transition function.

Finite State Machines: real-life examples

A CD player controller.

A lift.

A graphical user interface menu structure.

Few more examples from COMP209 lecture slides ... and later on this week.

Automata as acceptors

A *language* L is ... just a set of strings in $\mathcal{A}^*$.

An automaton *accepts* L if for each string w, M, started in the initial state on the leftmost character of w, after scanning all characters of w, enters a final state if and only if $w \in L$.

Example

$\mathcal{A} = \{0, 1, \ldots, 9\}$

$Q = \{$BEGIN, EVEN, ODD$\}$

$S =$ BEGIN

$F = \{$EVEN$\}$

Here's the definition of the function δ.

	$s \in \mathcal{A}$									
q	0	1	2	3	4	5	6	7	8	9
BEGIN		ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD
EVEN	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD
ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD	EVEN	ODD

A quick simulation

On reading 125, the automaton will start in the state BEGIN, reading the leftmost digit (that's "1"). $\delta($BEGIN,$1) =$ ODD, so the automaton moves to state "ODD", before reading the next digit. Now it reads "2", since $\delta($ODD$, 2) =$ EVEN, the automaton moves to state "EVEN". Then reads "5" and, since $\delta($EVEN$, 5) =$ ODD, the automaton moves to state "ODD". Since the number is finished "ODD" is the final answer of the automaton!

Exercises. Write an automaton that recognises multiples of 3.

String-matching automaton.

There will be an automaton for each pattern P. First an important definition:

> Let σ_P be a function that, for any string x, returns the length of the longest prefix of P that is a suffix of x.

Examples. Let $P \equiv$abc. Then $\sigma_P(\mathsf{caba}) = 1$, $\sigma_P(\mathsf{cabab}) = 2$, and $\sigma_P(\mathsf{cababc}) = 3$.

Here's the definition of the automaton!

- $Q = \{0, 1, \ldots, m\}$.
- $q_0 = 0$
- The only accepting state is m.
- The input alphabet is Σ.
- The transition function is defined by the following equation:

$$\delta(q, a) = \sigma(P_q a)$$

How is this to be used?

```
FINITE-AUTOMATON-MATCHING (T, δ, m)
    n ← length(T)
    for i ← 1 to n
        q ← δ(q, T[i])
        if q = m
            print "pattern occurs with shift i − m"
```