WORKSHEET, WEEK 13: KMP STRING SEARCH

Objectives:

1. Concepts of strings
2. Deterministic Finite Acceptor (DFA).
3. Prefix function
4. KMP algorithm

13.1 Concepts of Strings

String Search (Matching) Algorithms:

Text Editors: One text, one pattern

Detecting Plagiarism, DLP Many training documents, one testable document. (DLP - Data Leak prevention.) May be interested in pattern search, "longest commmon substring", finding matches with sliding window, detecting approximate matches (Copy-Paste and edit a little).

Document Retrieval Very many documents can be indexed and preprocessed. Pattern search should be fast (linear in pattern size, not in the total size of Internet).

13.1.1 String Matching Problem

1. Let us have a *finite alphabet* Σ (may be very large; think of Unicode with its 65536 symbols).
2. Let us have a text of length n – $T = T[0], \ldots, T[n-1]$.
3. Let us have a pattern of length m – $P = P[0], \ldots, P[m-1]$

Question: Is there a pattern P in the text T. (Usually we want to have the first occurrence, the first c occurrences or all occurrences.) One occurrence is named a *shift* (offset of the pattern in the text).

13.1.2 Substring not same as Subsequence

Definition: Given a string $T[0], T[1], \ldots, T[n-1]$, its *substring* is a string $T[i], T[i+1], \ldots, T[i+m-1]$ – it has some nonnegative length $m \leq n$. An empty string (ε) is a substring of every string.

Definition: Given a string $T[0], T[1], \ldots, T[n-1]$, its *subsequence* is a string $T[i_0], T[i_1], \ldots, T[i_{m-1}]$ of some symbols from the original string in increasing order ($i_0 < i_1 < \ldots < i_{m-1}$), but not necessarily a contiguous piece of the original string.

Compare these two algorithms:

1. Longest common substring.
2. Longest common subsequence.

Example: Consider the string of five letters – `APPLE`. It has many substrings:

$$\varepsilon = \text{""}, \text{"A"}, \text{"E"}, \text{"L"}, \text{"P"}, \text{"AP"}, \text{"LE"}, \text{"PL"}, \text{"PP"}, \text{"APP"}, \text{"PLE"}, \text{"PPL"}, \text{"APPL"}, \text{"PPLE"}, \text{"APPLE"}.$$

There are also 24 subsequences of `APPLE`. (In general, their number does not exceed 2^m, if all characters are different.)

13.1.3 Longest Common Substring - Matrix Algorithm

Consider two strings $A = A[0] \ldots A[m-1]$ and $B = B[0] \ldots B[n-1]$, of lengths m and n respectively.

Define the function LONGESTCOMMONSUFFIX(i, j) to be the longest common suffix of $A[i:]$ and $B[j:]$ respectively.

Define $(m+1) \times (n+1)$ size matrix $M[i, j]$ recurrently:

$$\begin{cases} M[0,0] = 0 \\ M[i,0] = 0, \ 1 \leq i \leq m \\ M[0,j] = 0, \ 1 \leq j \leq n \\ M[i,j] = M[i-1, j-i] + 1 \text{ if } A[i] = B[j] \\ M[i,j] = 0 \text{ otherwise} \end{cases}$$

		A	B	C	X	Y	Z	A	Y
	0	0	0	0	0	0	0	0	0
X	0	0	0	0	1	0	0	0	0
Y	0	0	0	0	0	2	0	0	1
Z	0	0	0	0	0	0	3	0	0
A	0	1	0	0	0	0	0	4	0
B	0	0	2	0	0	0	0	0	0
C	0	0	0	3	0	0	0	0	0
B	0	0	1	0	0	0	0	0	0

13.1.4 Levenstein Distance

Consider two strings $A = A[0] \dots A[m-1]$ and $B = B[0] \dots B[n-1]$, of lengths m and n respectively. These operations (*edits*) are permitted:

- Replacing one symbol with another (cost is 1).
- Erasing any symbol (cost is 1).
- Inserting symbol in any location (cost is 1).

Variants:

- What if only erasing and inserting are allowed (and replacing a symbol needs two steps).
- What happens, if you need to transform a string into a subsequence/substring of another?
- What if edit costs depend on the symbol?
- What if replacing multiple symbols at the same location is cheaper than multiple single-symbol edits?

Motivation: This problem can become *approximate sequence alignment*; also in biology – how many mutations are needed to transform one sequence into another one.

13.1.5 Algorithm for Levenstein's Distance

Given two strings $A = A[0] \dots A[m-1]$ and $B = B[0] \dots B[n-1]$, define $(m+1) \times (n+1)$ size matrix $M[i,j]$ with recurrences:

$$
\begin{aligned}
&M[0,0] = 0 \\
&M[i,0] = i,\ 1 \le i \le m \\
&M[0,j] = j,\ 1 \le j \le n \\
&M[i,j] = \min \begin{cases} M[i-1,j-1] + 0, & \text{if } A[i] = B[j] \\ M[i-1,j-1] + 1, & \text{(replacing letter)} \\ M[i,j-1] + 1, & \text{(inserting letter)} \\ M[i-1,j] + 1, & \text{(erasing letter)} \end{cases}
\end{aligned}
$$

Prove by induction that for all $i \in [1,m]$ and $j \in [1,n]$, the bottom-right element $M[i,j]$ is the Levenstein's distance between $A[0:i]$ and $B[0:j]$.

See Levenshtein Distanz.

Example:

	–	a	t	c	a	c	a
–	0	1	2	3	4	5	6
t	1	1	1	2	3	4	5
c	2	2	2	1	3	3	4
a	3	2	3	2	1	4	3
t	4	3	2	3	2	2	3

Find the edits (from the bottom right corner):

$$\texttt{tcat} \to \texttt{atcat} \to \texttt{atcac} \to \texttt{atcaca}.$$

13.2 Introducing Knuth-Morris-Pratt

Let us recap:

String Matching Problem: Is there a pattern P in the text T. (Usually we want to have the first occurrence, the first c occurrences or all occurrences.) One occurrence is named a *shift* (offset of the pattern in the text).

Rewrite the naive algorithm (unlike the previous lecture, we show detailed steps as j goes through the pattern:

NAIVESTRINGMATCHING(T, P):
    $n = len(T)$
    $m = len(P)$
    for $i = 0$ **to** RANGE$(0, n - m + 1)$:
        $j := 0$
        while $j < m - 1$ **and** $P[j]$!= $T[i + j]$:
            $j := j + 1$
        if j == m:
            output "Pattern found at offset", i

Initial idea: What happens, if we use the naive algorithm and the current shift is i, but comparison in the pattern has been done up to the position j.

If we have that $T[i] = P[0], \ldots, T[i + j - 1] = P[j - 1]$, but $T[i + j] \neq P[j]$, then we choose (i^*, j^*) optimally (and we want to read every character in the text T just once).

We do not **need** to choose $(i^*, j^*) = (i + 1, 0)$ as in the naive algorithm.

13.2.1 Deterministic Finite Acceptor (DFA)

Consider a directed graph, every node has exactly $|\Sigma|$ outgoing edges marked with the symbols from the alphabet Σ.

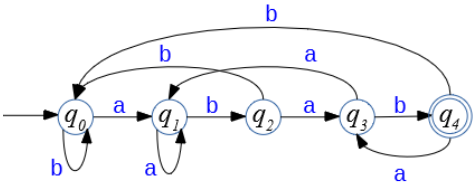
1. One state is the initial state; some states (in our case – just one state) are accepting states.
2. At every stage read symbol $c = T[i]$ from the text T
3. Use an edge labeled with symbol c to find the next state.
4. If we reach an accepting state, output the location (in fact, the shift is $s = i - (m - 1)$).

This is also named *finite state machine* or *deterministic finite automaton* (also DFA). (Acceptors are a subclass of such automata – they only react with changing their state; real automata may also produce output or do something useful.)

Denote the states of this DFA by $q_0, q_1, \ldots, q_m$.

Guideline for the acceptor: We want to be in the state q_i iff the last i characters of the text T matched the initial i characters from the pattern P.

Example: Here is an acceptor to search for $P =$ `abab`:



Note: After we find $P =$ `abab`, upon the symbol "a" go to q_3 (not q_1).

Time complexity for the DFA automaton:

After the DFA is built, we only read text once, so it adds $O(n)$ – one operation per letter.

Pattern preprocessing: An automaton needs to know the next state q' for each combination of the current state q and the input character.

We get $O(n + m \cdot |\Sigma|)$, where Σ is the alphabet being used. Preprocessing here is unpleasantly large (and KMP algorithm will fix that). We can typically assume that $n >> m$, but $m \cdot |\Sigma|$ may be large.

13.3 Prefix Function

- To fix these issues, we define a *prefix function* π in a table.
- Prefix function only depends on the pattern $P = P[0] \dots P[m-1]$. (It does not need the text).
- This function includes knowledge how the pattern P partially overlaps with itself.

In this case we do not need a full automaton with an arrow for every possible input symbol $c \in \Sigma$.

The preprocessing time (to compute the prefix function P will take just the time $O(m)$).

Definition: For each $j = 1, \dots, m$ find the maximum k ($k < j$) which satisfies:

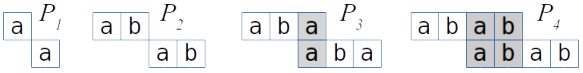
$$\begin{cases} P[0] = P[j-k] \\ P[1] = P[j-k+1] \\ \dots \\ P[k-1] = P[j-1] \end{cases}$$

Prefix function takes value $\pi[j] = k$. If there is no such k ($k < j$), then $\pi[j] = 0$.

Alternative definition: With $P[0 : s]$ denote the prefix of sequence P of length s. Then $\pi(j) = k$ equals to the longest suffix of $P[0 : j]$ that is shorter than the j itself.

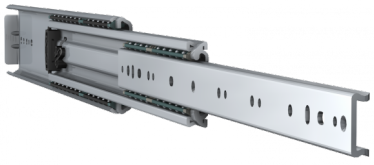
$$\pi(j) = \max \{k \ : \ k < j \text{ and } P[0 : k] \text{ is the suffix of } P[0 : j]\} .$$

Example 1: Find the prefix function for a pattern $P =$ `abab`.

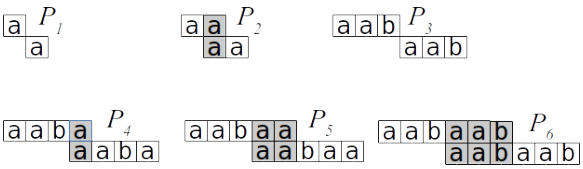


j	1	2	3	4
$\pi(j)$	0	0	1	2

Maximum Telescopic Sliding:



Example 2: Find the prefix function for a pattern $P =$ `aabaab`.



j	1	2	3	4	5	6
$\pi(j)$	0	1	0	1	2	3

13.4 KMP Algorithm

KMPMATCHER(T, P):
    $n = len(T)$
    $m = len(T)$
    $\pi =$ COMPUTEPREFIXFUNCTION(P)
    $k = 0$
    for i **in** RANGE$(0, n)$:
        while $k > 0$ **and** $P[k] \neq T[i]$:
            $k = \pi(k)$
        if $P[k]$ == $T[i]$
            $k = k + 1$
        if k == m
            output "Pattern found at offset", $i - m$
            $k = \pi(k)$

Correctness of KMP:

Let the *shift* be some number $(i - k) \in \{0, \ldots, n - m - 1\}$: We hoped to find the pattern P in T starting with the shift/offset s.

But it turned out that the current symbol in T ($T[i]$) does not match $P[k]$ (where $k \in \{0, \ldots, m - 1\}$). Then we must have these equalities:

$$\begin{cases} T[i] & = P[j-k] & = P[0] \\ T[i+1] & = P[j-k+1] & = P[1] \\ \ldots & \ldots & \ldots \\ T[i+k+1] & = P[j-1] & = P[k-1] \end{cases}$$

The next position in T where the pattern P can start – it is starting at the last k letters from the T fragment already received.

Example of KMP Execution:

Search for pattern $P =$ `ababaca` in the text $T =$ `ababaababaca`.

Solution:

Find the prefix function for $P =$ `ababaca`:

j	1	2	3	4	5	6	7
$\pi(j)$	0	0	1	2	3	0	1

i	0	1	2	3	4	5	6	7	8	9	10	11	
	a	b	a	b	a	a	b	a	b	a	c	a	$k = 0$
	a	b	a	b	a	c	a						$k = 1, 2, 3, 4, 5$
	a	b	a	b	a	c	a						$k = \pi(5) = 3$
			a	b	a	b	a	c	a				$k = \pi(3) = 1$
					a	b	a	b	a	c	a		$k = \pi(1) = 0$
						a	b	a	b	a	c	a	$k = 1, 2, 3, 4, 5, 6, 7$

See also http://whocouldthat.be/visualizing-string-matching/ containing a visualization (it uses a different variant of the prefix function and finds only the first match).

Time Complexity of KMP String Matching:

Assume that $\pi(j)$ is already given. Note that for every comparison of P with the text T one of the following two statements hold:

- if $P[k] == T[i]$, then increment i, but do not change $i - k$.
- if $P[k] \neq T[i]$, then increment $i - k$, but do not change i.

Since both i and $i - k$ are integers that are initially 0, but cannot exceed n, then there should be no more than $2n$ comparisons. Therefore, the speed of KMP is $O(n)$.

13.4.1 Pseudocode for the Prefix Function

COMPUTEPREFIXFUNCTION(P)

$m := len(P)$
Initialize the table $\pi(1) \ldots \pi(m)$
$\pi(1) = 0$
$k = 0$
for $q = 2$ **to** m
    while $k > 0$ **and** $P[k] \neq P[q-1]$
        $k := \pi(k)$
    if $P[k] == P[q-1]$
        $k := k + 1$
    $\pi(q) = k$
return π

Example: Find the prefix function for aa pattern $P =$ `ababaca`.

	a	b	a	b	a	c	a
i	1	2	3	4	5	6	7
$q = 1$	0						
$q = 2$	0	0					
$q = 3$	0	0	1				
$q = 4$	0	0	1	2			
$q = 5$	0	0	1	2	3		
$q = 6$	0	0	1	2	3	$3 \downarrow 1$	
$q = 6$	0	0	1	2	3	$1 \downarrow 0$	
$q = 6$	0	0	1	2	3	0	
$q = 7$	0	0	1	2	3	0	1

Time Complexity of the Prefix Function:

- The outer loop runs $m - 1$ times.
- In every iteration of the inner loop the value $\pi[i + 1]$ is reduced.
- This value can be incremented by 1 as the outer loop runs; so it cannot exceed m.
- As it never becomes negative, it can only be reduced m times.

So, the prefix function algorithm is $O(m)$.

13.5 Problems

Question 1: Use the dynamic programming algorithm (matrix) to find the longest common substring of two strings: $T_1 =$ `banana` and `cabana`.

Question 2: Create a deterministic finite acceptor to create the string `aabab` in the text input.

Question 3: In the string $T =$ `947892879487` find the pattern $P =$ `9487`. Find the KMP prefix function $\pi(i)$, and draw the shifts of the pattern which are compared. (These shifts and looking at specific symbols can be shown in a table: columns are the positions in text; rows are the possible shifts. Write in the pattern at the shifts that were ever active and circle those symbols in "P" that were compared with the text "T".

Question 4:

(A) Build the KMP data structures if we need to match the pattern `abababc`.

(B) Show how this works on the following text: $T = abcababacababababc$.

(C) Draw a finite state acceptor (FSA) for the pattern.

Question 5: Find the prefix function for the pattern $P =$ `abcbcab`. Demonstrate how it works on the text $T =$ `abcabbcabcbcababababcbcab`.