CS 240 — Lecture 18

Bartosz Antczak *Instructor: Eric Schost* *March 14, 2017*

18.1 KMP Algorithm

Just like the brute-force method, this compares the pattern to the text from *left-to-right*; however, it shifts the pattern more intelligently than the brute-force counterpart.

18.1.1 Implementation

Suppose we have a match up to position T[i-1] = P[j-1], but not at the next position. We define F[j-1] as the index we will have to check in P, after we bring the pattern to its next possible position. We shift the pattern by F[j-1] unites.

Example 18.1.1. *Constructing F[j] using a pattern P = abacaba*

j	P[0 ... j]	F[j]
0	a	0
1	ab	0
2	aba	1
3	abac	0
4	abaca	1
5	abacab	2
6	abacaba	3

$F[0] = 0$

$F[j]$, for $j > 0$, is the length of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$

Consider $P =$ abacaba

j	$P[1..j]$	P	$F[j]$
0	—	abacaba	0
1	b	abacaba	0
2	ba	abacaba	1
3	bac	abacaba	0
4	baca	abacaba	1
5	bacab	abacaba	2
6	bacaba	abacaba	3

18.1.2 KMP Analysis

When constructing the **failure array**, there are no more than $2m$ iterations of the while loop, so it takes a time of $\Theta(m)$, where m is the length of the pattern string.

When running the KMP algorithm, there are no more than $2n$ iterations of the while loop, so the running time is $\Theta(n)$, where n is the length of the text for which we're finding a pattern in.

18.2 Boyer-Moore Algorithm

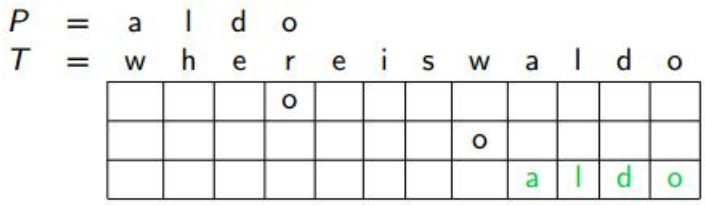
It's based on three key ideas

- **Reverse-order searching** compare P with a subsequence of T moving *backwards*
- **Bad character jumps** when a mismatch occurs at T[i] = c
  - If P contains c, we can shift P to align the last occurrence of c in P with T[i]
  - Otherwise, we can shift P to align P[0] with T[i+1]
- **Good suffix jumps** if we have already matched a suffix of P, then get a mismatch, we can shift P forward to align with the previous occurrence of that suffix (with a mismatch from the suffix we read). If non exists, look for the longest prefix of P that is a suffix of what we read (this is similar to failure array in KMP)

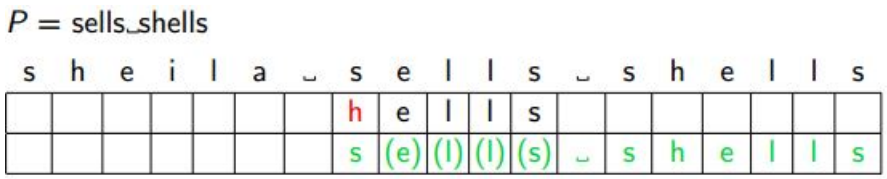
Using these ideas, we can skip large parts of T.

18.2.1 Bad Character Examples

We check the last letter of the pattern P with the text T. We skip parts of T if we encounter a letter that isn't in the pattern P, because we know for a fact that the pattern won't be in the substring since that particular character exists.



18.2.2 Good Suffix Examples



18.2.3 Last-Occurrence Function

In order to utilize the previous two examples, we must *preprocess* the pattern P and the alphabet Σ. We will build the *last-occurrence function* L mapping Σ to integers.
$L(c)$ is defined as:

- The largest index i such that $P[i] = c$ or

- -1 if no such index exists

Example 18.2.1. *Creating a Last-Occurrence function as shown in the course slides*

$$\Sigma = \{a, b, c, d\}, P = abacab$$

c	a	b	c	d
$L(c)$	4	5	3	-1

The runtime to build this function is $O(|\Sigma| + m)$ (we build an array of size $|\Sigma|$ and then we check every character in P by doing $L[P[i]] = i$.

18.2.4 Good Suffix Array

Again, we *preprocess* P to build a table. We create a **suffix skip array** S of size m: for $0 \leq i < m, S[i]$ is the largest index j such that $P[i+1...m-1] = P[j+1...j+m-1-i]$ **and** $P[j] \neq P[i]$. This is computed

$$P = \text{bonobobo}$$

i	0	1	2	3	4	5	6	7
$P[i]$	b	o	n	o	b	o	b	o
$S[i]$	-6	-5	-4	-3	2	-1	2	6

in $\Theta(m)$ time.