TWO NEW APPROACHES TO STRING DELIMITATIONS

DOMINIC VAN DER ZYPEN

1. Problem setting

Where does a string end? This is addressed using three different methods:

(1) Number of characters in front of string:
`12:hello world!`
(2) Quoting, and escaping quotes within string:
`"hello, \"world\""`.
(3) Delimiter such as EOF. Leads to problems when that delimiter appears in original string.

This note used a modified version of (3) by constructing for every string an *individual* delimiter $d(s)$ that does not appear in s. Then we can represent s as

$$d(s) : s : d(s)$$

So, for instance, if $s =$ `hello world` we might have $d(s) =$ `!` and the representation of s will be

```
!:hello world:!
```

In the following we discuss some ways of constructing $d(s)$ with these goals in mind:

(1) $d(s)$ must not be a substring of s,
(2) $d(s)$ should be built in $O(n)$ time, where n is the length of the input string, and
(3) the length of $d(s)$ should be kept as small as possible.

2. Two approaches

2.1. **A simple counting algorithm.** Fix a character such as `7` and assign to any string s the non-negative integer $M(s)$ which is defined in the following way:

> Let $M(s)$ be the smallest number k of consecutive instances of `7` such that
>
> $$\underbrace{77\ldots 7}_{k \text{ times}}$$

does **not** appear in s.

Then we let $d(s) = \underbrace{\texttt{77}\ldots\texttt{7}}_{M(s)\text{ times}}$.

Example. Let $s =$ `hello 7 world 77`. Then $M(s) = 3$, since 3 is the smallest number n such that `7` does not appear n consecutive times in the string s. So, $d(s) =$ `777`, and the representation of s is

```
777:hello 7 world 77:777
```

It is easy to find a linear-time counting algorithm to determine $M(s)$ stepping through s once.

The *disadvantage* of this representation is that $d(s)$ can grow linearly with respect to s in the worst case. The next method we present will guarantee a length of the delimiter of $O(\log(n))$ where n is the length of the string. Also, we will show that this is the best worst-case length you can get.

2.2. **Constant length moving window.** The overall goal of this paragraph will be to construct a delimiter consisting of the characters `0` and `1` such that

(1) the length of the delimiter is $O(\log(n))$ where n is the length of the input string, and
(2) finding the delimiter happens in $O(n)$ time.

First, we will observe that it is not possible to "go below $O(\log(n))$ length".

Observation 2.1. *Consider the following* `0,1`*-string of length 8:*

$$s = \texttt{01110000}$$

We cannot find a `0,1`*-delimiter of length 2, as all of the* `0,1`*-strings of length 2 (i.e.,* `00, 01, 10, 11`*) appear in* s.

Next we prove that $k = \lceil \log(n) \rceil + 1$ is a sufficient substring length.

Proposition 2.2. *If* s *is a* `0,1`*-string of length* n *and* $k := \lceil \log(n) \rceil + 1$ *there is a* `0,1`*-string of length* k *that is* not *a substring of* s.

Proof. There are $n - k$ substrings in s of length k, and there are 2^k `0,1`-strings of length k in total. Since $n - k \leq 2^k - k < 2^k$, there must be a `0,1`-string of length k that is not a k-length-substring of s. □

So we are armed to give a solution assuming that s consists of characters `0` and `1` only, and then show how we can reduce general strings to `0,1`-strings.

2.2.1. *Solution for* `0,1`*-strings.* Let s be a given string consisting of characters `0` and `1` only. Let $n \in \mathbb{N}$ be the length of s and let $k := \lceil \log(n) \rceil + 1$. Note that by $\log(\cdot)$ we denote the logarithm of base 2, and for any $x \in \mathbb{R}$ we define

$$\lceil x \rceil = \inf\{z \in \mathbb{Z} : z \geq x\}.$$

We proceed along the following steps:

(1) Initialise a *bit-field* B of length 2^k with constant value 0.
(2) Use a k-window to step through s from beginning to end, as depicted below:

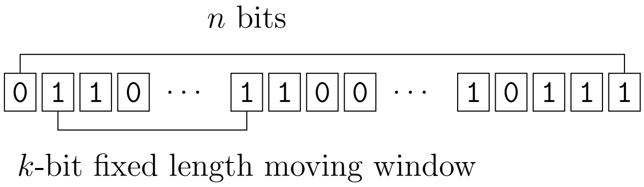

FIGURE 1. The k-bit window moves through the string, and it is interpreted as a k-bit binary number.

(3) For every k-substring, interpret the string as integer x with $0 \leq x \leq 2^k - 1$ and in bit-field B, set bit number x to 1. Actually, this binary number will be stored in a `uint64` variable `w`, and the window rolling procedure can be quickly done in `w`, see Appendix.
(4) When this is completed, look for first bit-position set to 0. This gives a `0,1`-string not contained in original string!

2.2.2. *Reducing general strings to* `0,1`*-strings.* This is the old parity trick: Step through every character (bit) of the given string s and for every bit `b` only save its parity using bit-wise AND (`&`):

```
b = b & 1; // get 0 or 1 as bit
```

So we get a `0,1`-string $s_{0,1}$ out of s, and we proceed as above. It is easy to see that the delimiter $d(s_{0,1})$ that the algorithm of 2.2.1 produces does not appear in s.