A Linear-Time Algorithm to Find the First Overlap in a Binary Word

Thomas H. Park
Router Software Team, Router Technology Department
Electronics and Telecommunications Research Institute
thomas@etri.re.kr

Abstract

First, we give a linear-time algorithm to find the first overlap in an arbitrary binary word. Second, we implement the algorithm in the C language and show that the number of comparisons in this algorithm is less than $31n$, where $n \geq 3$ is the length of the input word.

1 Introduction

Kfoury developed a linear-time algorithm to decide whether a binary word is overlap-free [2]; his algorithm can be adapted to find the first overlap in an arbitrary binary word in linear time. Here we take a new look at the latter problem and develop a linear-time algorithm for it, and then show that this algorithm runs with a relatively small increase in the proportionality constant.

Concerning notation and terminology, late Greek letters (π, ρ and τ) will denote variables ranging over the set of possible words, and late Roman letters (x, y and z) will stand for individual symbols from finite alphabets.

2 The Algorithm

Formally, a word ρ contains an *overlap* if it contains a finite sub-word of the form $\rho\tau\rho'$ such that $\rho\tau = \tau\rho'$, where ρ, τ, and ρ' are non-empty[1]. The sub-word τ is called the overlap. The input to the algorithm is an arbitrary $input \in \{0, 1\}^+$. Since the algorithm finds the first overlap in *input*, if there is any, it is easy to show that the length of the overlap is always 1. At the nth iteration, $n \geq 1$, it carries out the following steps:

step 1. If $input_n \in \{0, 1, 00, 11\}$, there is no overlap.

step 2. Decompose $input_n$ as $\pi_n\rho_n\pi'_n$ with π_n, $\pi'_n \in \{\lambda, 0, 1, 00, 11\}$ and $\rho_n \in \{01, 10\}^+$. If this is not possible, find a overlap using FIND_OVERLAP. If $input_n$ has more than one such decomposition, take π_n as short as possible.

step 3. If $\pi_n = xx$, with $x \in \{0, 1\}$, and xxx or $xx\bar{x}xx\bar{x}x$ is a prefix of $\pi_n\rho_n$, find a overlap using FIND_OVERLAP.

step 4. If $\pi'_n = yy$, with $y \in \{0, 1\}$, and yyy or $y\bar{y}yy\bar{y}yy$ is a suffix of $\rho_n\pi'_n$, find a overlap using FIND_OVERLAP.

step 5. If ($\pi_n = x$ and xx) and ($\pi'_n = y$ or yy) and ($x\rho_n y = \tau\tau$ for some $\tau \in \{0, 1\}^*$) then go to step 6 else go to step 7.

step 6. Define $input_{n+1}$ from MAPPING2(π_n, ρ_n, π'_n) by

mapping consecutive occurrences of 01 and 10 into 0 and 1, respectively, and go to the $(n + 1)$st iteration.

step 7. Define $input_{n+1}$ from MAPPING1(π_n, ρ_n, π'_n) by mapping consecutive occurrences of 01 and 10 into 0 and 1, respectively, and go to the $(n + 1)$st iteration.

As shown in above steps, functions such as FIND_OVERLAP and MAPPING1 & MAPPING2 are defined in more detail in the following section. It will explain how BACKTRACKING functions are being called up, and how the MAPPING functions are used to reduce the binary string.

3 Some functions used in the Algorithm

A function DECOMPOSE decomposes a binary string array *input* as $\pi_j\rho_j\pi'_j$ with π_j, $\pi'_j \in \{\lambda, 0, 1, 00, 11\}$ and $\rho_j \in \{01, 10\}^+$. $p[j]$ and $q[j]$ are pointers to π_j and π'_j, respectively, at the jth iteration. If the binary input has more than one such decomposition, take π_j as short as possible.

A function SELECTION determines whether the binary string array *input* can be divided as follows: $x\rho_j y = \tau\tau$ for some $\tau \in \{0, 1\}^*$, where $\pi_j = x$ or xx and $\pi'_j = y$ or yy. SELECTION returns *True*, if it is possible and *False*, otherwise.

A function PATTERN finds a overlap which occurs at the jth iteration. It returns the length and the starting index of the overlap at the jth iteration.

A function FIND_OVERLAP finds a overlap which occurs in a binary string input by calling BACKTRACKING recursively. Since the array *flag* indicates which mapping function was used at the jth iteration, FIND_OVERLAP calls either BACKTRACKING1 or BACKTRACKING2 according to the array *flag*.

The functions BACKTRACKING1 and BACKTRACKING2 update *length* and *start_index* variables, which are the length and the beginning index of the first overlap at every iteration, according to the following conditions:

- BACKTRACKING1

(i) If *start_index* = 0 and *length* is equal to the length of the input string at the jth iteration, then

$$\begin{cases} length \leftarrow length * 2 - 2, & if\ \pi = x\ and\ \pi' = y; \\ length \leftarrow length * 2 - 1, & otherwise. \end{cases}$$

(ii) If *start_index* = 0 and *length* is shorter than the length of the input string at the jth iteration, then

$$\begin{cases} length \leftarrow length * 2 - 1. \\ start\_index \leftarrow start\_index + 1, & if\ \pi = xx. \end{cases}$$

(iii) If *start_index* ≠ 0, then

$$\begin{cases} length \leftarrow length * 2 - 1. \\ start\_index \leftarrow start\_index * 2 - 1, & if\ \pi = x; \\ start\_index \leftarrow start\_index * 2, & otherwise. \end{cases}$$

- BACKTRACKING2

(i) If *start index* = 0 and *length* is equal to the length of the input string at the jth iteration, then

$$\begin{cases} length \leftarrow length * 2 - 1. \\ start\_index \leftarrow start\_index + 1, \\ \quad if\ \pi = x\ and\ \pi' = y\ or\ yy; \\ start\_index \leftarrow start\_index + 2, \\ \quad if\ \pi = xx\ and\ \pi' = y\ or\ yy. \end{cases}$$

(ii) If *start index* = 0 and *length* is shorter than the length of the input string at the jth iteration,

then

$$\begin{cases} length \leftarrow length^* 2 - 1. \\ start\_inde\ x \leftarrow start\_inde\ x + 1, \\ \quad if\ \pi = xx\ and\ \pi' = \lambda; \\ start\_inde\ x \leftarrow start\_inde\ x + 1, \\ \quad if \pi = x\ and\ \pi' = y\ or\ yy; \\ start\_inde\ x \leftarrow start\_inde\ x + 2, \\ \quad if\ \pi = xx\ and\ \pi' = y\ or\ yy. \end{cases}$$

(iii) If $start\_index \neq 0$, then

$$\begin{cases} length \leftarrow length * 2 - 1. \\ start\_index \leftarrow start\_index * 2 - 1, \\ \quad if = x\ and\ \pi' = \lambda; \\ start\_index \leftarrow start\ index * 2, \\ \quad if\ \pi = \lambda\ or\ xx\ and\ \pi' = \lambda; \\ start\_index \leftarrow start\ index * 2 + 2, \\ \quad if\ \pi = xx\ and\ \pi' = y\ or\ yy; \\ start\_index \leftarrow start\ index * 2 + 1, \\ \quad if\ \pi = x\ and\ \pi' = y\ or\ yy; \\ start\_index \leftarrow start\ index * 2, \\ \quad if\ \pi = \lambda\ and\ \pi' = y\ or\ yy; \end{cases}$$

We define the functions MAPPING1 and MAPPING2 from $\{0, 1\}^+$ to $\{0, 1\}^+$. They are defined for all words of the form $\pi\rho\pi'$, where $\pi, \pi' \in \{\lambda, 0, 1, 00, 11\}$ and $\rho \in \{01, 10\}^+$, and by:

- MAPPING1

$$(\pi\rho\pi') = \begin{cases} \rho, & \text{if } \pi = \pi' = \lambda; \\ xx\rho, & \text{if } \pi = x\ or\ xx \text{ and } \pi' = \lambda; \\ \rho yy, & \text{if } \pi = \lambda \text{ and } \pi' = y\ or\ yy; \\ xx\rho yy, & \text{if } \pi = x\ or\ xx, \text{ and } \pi' = y\ or\ yy; \end{cases}$$

- MAPPING2

$$(\pi\rho\pi') = \begin{cases} \rho, & \text{if } \pi = \pi' = \lambda; \\ xx\rho, & \text{if } \pi = x\ or\ xx \text{ and } \pi' = \lambda; \\ \rho yy, & \text{if } \pi = \lambda \text{ or } x\ or\ xx \text{ and } \pi' = y\ or\ yy;; \end{cases}$$

4 Analysis of Algorithm

We can get an estimate for the proportionality constant in the time complexity of the algorithm as follows. Suppose n is the length of the input string. Each function performs the following number of comparisons:

- number of comparisons for DECOMPOSE is at most $6n$ (each iteration of the **while** loop has at most 4 comparisons and there are $n/2$ iterations and the entire **while** loop has a possibility to be performed at most three times).
- number of comparisons for MAPPING1 (or MAPPING2) is at most $n/2$ (only one of three **while** loops can be performed and each iteration of the **while** loop has one comparison and there are $n/2$ iterations).
- number of comparisons for SELECTION is at most $n/2$.
- number of comparisons for PATTERN is at most $n + 6$.
- number of comparisons for FIND OVERLAP is 9 log n (each iteration of the **while** loop has at most 9 comparisons and there are at most log n iterations).
- number of comparisons for BACKTRACKING1 (or BACKTRACKING2) is at most 7.

Since $n + (n/2) + (n/4) + \ldots\ldots \leq 2n$ and the function PATTERN and FIND_OVERLAP are performed at most once, the running time of the algorithm is less than $(6 + 1/2 + 1/2) * 2n + n + 9\log n + c = 15n + 9\log n + c$, where c is the cost incurred in each step which is less than $19 \log n + 6$ (2 comparisons for each of step 1 and 2, 4 comparisons for each of step 3 and 4, and 7 comparisons for step 5, and multiplied by the maximun number of iterations). Therefore, the total number of comparisons in this algorithm is $15n + 28 \log n + 6 \leq 31n$, where $n \geq 3$.

References

[1] T.H. Cormen & C.E. Leiserson & R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[2] A. J. Kfoury. *A Linear Algorithm to Decide whether a Binary Word Contains an Overlap*. Theoretical Informatics and Applications, 1988.

[3] A. Salomaa. *Jewels of Formal Language Theory*, Computer Science Press, 1981.

xxx̄xxx̄x