String Processing
(Based on [Manber 1989])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

Data Compression

Problem

Given a text (a sequence of characters), find an encoding for the characters that satisfies the prefix constraint and that minimizes the total number of bits needed to encode the text.

The *prefix constraint* states that the prefixes of an encoding of one character must not be equal to a complete encoding of another character.

Denote the characters by $c_1, c_2, \cdots, c_n$ and their frequencies by $f_1, f_2, \cdots, f_n$. Given an encoding E in which a bit string s_i represents c_i, the length (number of bits) of the text encoded by using E is $\sum_{i=1}^{n} |s_i| \cdot f_i$.

A Code Tree

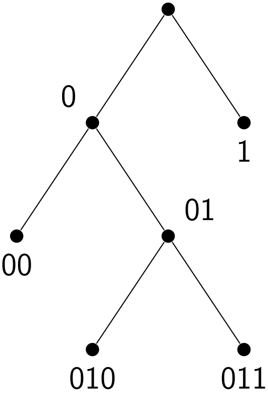

Figure: The tree representation of encoding.

Source: redrawn from [Manber 1989, Figure 6.17].

A Huffman Tree

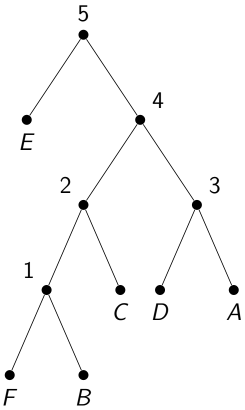

Figure: The Huffman tree for a text with frequencies of A: 5, B: 2, C: 3, D: 4, E: 10, F:1. The code of B, for example, is 1001. The numbers labeling the internal nodes indicate the order in which the corresponding subtrees are formed.

Source: redrawn from [Manber 1989, Figure 6.19].

Huffman Encoding

IM NTU

Algorithm Huffman_Encoding (S, f);
    insert all characters into a heap H
        according to their frequencies;
    while H not empty **do**
        if H *contains only one character* X **then**
            make X *the root of* T
        else
            delete X *and* Y *with lowest frequencies*;
                from H;
            create Z *with a frequency equal to the*
                sum of the frequencies of X *and* Y;
            insert Z *into* H;
            make X *and* Y *children of* Z *in* T

Huffman Encoding

Algorithm Huffman_Encoding (S, f);
    insert all characters into a heap H
        according to their frequencies;
    while H not empty **do**
        if *H contains only one character X* **then**
            make X the root of T
        else
            delete X and Y with lowest frequencies;
                from H;
            create Z with a frequency equal to the
                sum of the frequencies of X and Y;
            insert Z into H;
            make X and Y children of Z in T

What is its time complexity?

Huffman Encoding

Algorithm Huffman_Encoding (S, f);
  insert all characters into a heap H
    according to their frequencies;
  while H not empty **do**
    if H *contains only one character* X **then**
      make X *the root of* T
    else
      delete X *and* Y *with lowest frequencies*;
        from H;
      create Z *with a frequency equal to the*
        sum of the frequencies of X *and* Y;
      insert Z *into* H;
      make X *and* Y *children of* Z *in* T

What is its time complexity? $O(n \log n)$

String Matching

IM NTU

Problem

Given two strings A $(= a_1 a_2 \cdots a_n)$ and B $(= b_1 b_2 \cdots b_m)$, find the first occurrence (if any) of B in A. In other words, find the smallest k such that, for all i, $1 \leq i \leq m$, we have $a_{k-1+i} = b_i$.

A (non-empty) *substring* of a string A is a consecutive sequence of characters $a_i a_{i+1} \cdots a_j$ $(i \leq j)$ from A.

Straightforward String Matching

$A = xyxxyxyxyyxyxyxyyxyxyxx$. $B = xyxyyxyxyxx$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
1 :	x	y	x	y	·	·	·																
2 :		x	·	·	·																		
3 :			x	y	·	·	·																
4 :				x	y	x	y	y	·	·	·												
5 :					x	·	·	·															
6 :						x	y	x	y	y	x	y	x	y	x	x							
7 :							x	·	·	·													
8 :								x	y	x	·	·	·										
9 :									x	·	·	·											
10 :										x	·	·	·										
11 :											x	y	x	y	y	·	·	·					
12 :												x	·	·	·								
13 :													x	y	x	y	y	x	y	x	y	x	x

Figure: An example of a straightforward string matching.

Source: redrawn from [Manber 1989, Figure 6.20].

Straightforward String Matching (cont.)

- What is the time complexity?

Straightforward String Matching (cont.)

- What is the time complexity?
  - B $(= b_1 b_2 \cdots b_m)$ may be compared against
    - $a_1 a_2 \cdots a_m$,
    - $a_2 a_3 \cdots a_{m+1}$,
    - $\ldots$, and
    - $a_{n-m+1} a_{n-m+2} \cdots a_n$
  - For example, $A = xxxx \ldots xxxy$ and $B = xxxy$.

Straightforward String Matching (cont.)

- What is the time complexity?
  - $B\ (= b_1 b_2 \cdots b_m)$ may be compared against
    - $a_1 a_2 \cdots a_m$,
    - $a_2 a_3 \cdots a_{m+1}$,
    - $\ldots$, and
    - $a_{n-m+1} a_{n-m+2} \cdots a_n$
  - For example, $A = xxxx \ldots xxxy$ and $B = xxxy$.
- So, the time complexity is $O(m \times n)$.

Straightforward String Matching (cont.)

- What is the time complexity?
  - B $(= b_1 b_2 \cdots b_m)$ may be compared against
    - $a_1 a_2 \cdots a_m$,
    - $a_2 a_3 \cdots a_{m+1}$,
    - $\ldots$, and
    - $a_{n-m+1} a_{n-m+2} \cdots a_n$
  - For example, $A = xxxx \ldots xxxy$ and $B = xxxy$.
- So, the time complexity is $O(m \times n)$.
- But the best possible is linear-time, with a preprocessing.

Straightforward String Matching (cont.)

- What is the time complexity?
  - B $(= b_1 b_2 \cdots b_m)$ may be compared against
    - $a_1 a_2 \cdots a_m$,
    - $a_2 a_3 \cdots a_{m+1}$,
    - ..., and
    - $a_{n-m+1} a_{n-m+2} \cdots a_n$
  - For example, $A = xxxx \ldots xxxy$ and $B = xxxy$.
- So, the time complexity is $O(m \times n)$.
- But the best possible is linear-time, with a preprocessing.
- The cause of deficiency: tries from 7 to 12 in the example are doomed to fail. Why?

Straightforward String Matching (cont.)

- What is the time complexity?
  - B $(= b_1 b_2 \cdots b_m)$ may be compared against
    - $a_1 a_2 \cdots a_m$,
    - $a_2 a_3 \cdots a_{m+1}$,
    - $\ldots$, and
    - $a_{n-m+1} a_{n-m+2} \cdots a_n$
  - For example, $A = xxxx \ldots xxxy$ and $B = xxxy$.
- So, the time complexity is $O(m \times n)$.
- But the best possible is linear-time, with a preprocessing.
- The cause of deficiency: tries from 7 to 12 in the example are doomed to fail. Why?
- How can we avoid the futile tries?

Matching the Pattern Against Itself

IM NTU

- In the example, when the ongoing matching fails at b_{11} against a_{16}, we know that $b_1 b_2 \ldots b_{10}$ equals $a_6 a_7 \ldots a_{15}$.

Matching the Pattern Against Itself

- In the example, when the ongoing matching fails at b_{11} against a_{16}, we know that $b_1 b_2 \ldots b_{10}$ equals $a_6 a_7 \ldots a_{15}$.
- The next possible substring of A that equals B must start at a_{13}, because $a_{13} a_{14} a_{15}$ is the longest suffix of $a_6 a_7 \ldots a_{15}$ that equals a prefix of $b_1 b_2 \ldots b_{10}$, namely $b_1 b_2 b_3$.

Matching the Pattern Against Itself

- In the example, when the ongoing matching fails at b_{11} against a_{16}, we know that $b_1 b_2 \ldots b_{10}$ equals $a_6 a_7 \ldots a_{15}$.
- The next possible substring of A that equals B must start at a_{13}, because $a_{13} a_{14} a_{15}$ is the longest suffix of $a_6 a_7 \ldots a_{15}$ that equals a prefix of $b_1 b_2 \ldots b_{10}$, namely $b_1 b_2 b_3$.
- We can tell this by just looking at B, as $a_{13} a_{14} a_{15}$ equals $b_8 b_9 b_{10}$.

Matching the Pattern Against Itself

- In the example, when the ongoing matching fails at b_{11} against a_{16}, we know that $b_1 b_2 \ldots b_{10}$ equals $a_6 a_7 \ldots a_{15}$.
- The next possible substring of A that equals B must start at a_{13}, because $a_{13}a_{14}a_{15}$ is the longest suffix of $a_6 a_7 \ldots a_{15}$ that equals a prefix of $b_1 b_2 \ldots b_{10}$, namely $b_1 b_2 b_3$.
- We can tell this by just looking at B, as $a_{13}a_{14}a_{15}$ equals $b_8 b_9 b_{10}$.

$B =$	x	y	x	y	y	x	y	x	y	x	x
		x	$\cdot$	$\cdot$	$\cdot$						
			x	y	x	$\cdot$	$\cdot$	$\cdot$			
				x	$\cdot$	$\cdot$	$\cdot$				
					x	$\cdot$	$\cdot$	$\cdot$			
						x	y	x	y	y	
							x	$\cdot$	$\cdot$	$\cdot$	
								x	y	x	

Figure: Matching the pattern against itself.

Source: redrawn from [Manber 1989, Figure 6.21].

The Values of *next*

$i =$	1	2	3	4	5	6	7	8	9	10	11
$B =$	x	y	x	y	y	x	y	x	y	x	x
$next =$	-1	0	0	1	2	0	1	2	3	4	3

Figure: The values of *next*.

Source: redrawn from [Manber 1989, Figure 6.22].

The value of $next[j]$ tells the length of the longest proper prefix that is equal to a suffix of $b_1 b_2 \ldots b_{j-1}$.

The Values of *next*

$i =$	1	2	3	4	5	6	7	8	9	10	11
$B =$	x	y	x	y	y	x	y	x	y	x	x
$next =$	-1	0	0	1	2	0	1	2	3	4	3

Figure: The values of *next*.

Source: redrawn from [Manber 1989, Figure 6.22].

The value of $next[j]$ tells the length of the longest proper prefix that is equal to a suffix of $b_1 b_2 \ldots b_{j-1}$.

If the ongoing matching fails at b_j against a_i, then $b_{next[j]+1}$ is the next to try against a_i.

Note: $next[1]$ is set to -1 so that this unique case is easily differentiated (see the main loop of the KMP algorithm).

The KMP Algorithm

Algorithm String_Match (A, n, B, m);
begin
    $j := 1$; $i := 1$;
    $Start := 0$;
    while $Start = 0$ and $i \le n$ **do**
        if $B[j] = A[i]$ **then**
            $j := j + 1$; $i := i + 1$
        else
            $j := next[j] + 1$;
            if $j = 0$ **then**
                $j := 1$; $i := i + 1$;
        if $j = m + 1$ **then** $Start := i - m$
end

The KMP Algorithm (cont.)

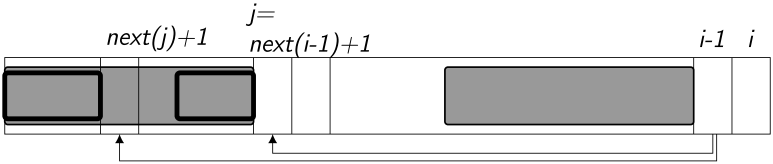

Figure: Computing $next(i)$.

Source: redrawn from [Manber 1989, Figure 6.24].

The KMP Algorithm (cont.)

Algorithm Compute_Next (B, m);
begin
    $next[1] := -1;\;\; next[2] := 0;$
    for $i := 3$ **to** m **do**
        $j := next[i-1] + 1;$
        while $B[i-1] \neq B[j]$ and $j > 0$ **do**
            $j := next[j] + 1;$
        $next[i] := j$
end

The KMP Algorithm (cont.)

- What is its time complexity?

The KMP Algorithm (cont.)

- What is its time complexity?
  - Because of backtracking, a_i may be compared against
    - b_j,
    - b_{j-1},
    - ..., and
    - b_2

The KMP Algorithm (cont.)

- What is its time complexity?
  - Because of backtracking, a_i may be compared against
    - b_j,
    - b_{j-1},
    - ..., and
    - b_2
  - However, for these to happen, each of $a_{i-j+2}, a_{i-j+3}, \ldots, a_{i-1}$ was compared against the corresponding character in $b_1 b_2 \ldots b_{j-1}$ just once.

The KMP Algorithm (cont.)

- What is its time complexity?
  - Because of backtracking, a_i may be compared against
    - b_j,
    - b_{j-1},
    - $\ldots$, and
    - b_2
  - However, for these to happen, each of $a_{i-j+2}, a_{i-j+3}, \ldots, a_{i-1}$ was compared against the corresponding character in $b_1 b_2 \ldots b_{j-1}$ just once.
  - We may re-assign the costs of comparing a_i against $b_{j-1}, b_{j-2}, \ldots, b_2$ to those of comparing $a_{i-j+2} a_{i-j+3} \ldots a_{i-1}$ against $b_1 b_2 \ldots b_{j-1}$.

The KMP Algorithm (cont.)

- What is its time complexity?
  - Because of backtracking, a_i may be compared against
    - b_j,
    - b_{j-1},
    - $\ldots$, and
    - b_2
  - However, for these to happen, each of $a_{i-j+2}, a_{i-j+3}, \ldots, a_{i-1}$ was compared against the corresponding character in $b_1 b_2 \ldots b_{j-1}$ just once.
  - We may re-assign the costs of comparing a_i against $b_{j-1}, b_{j-2}, \ldots, b_2$ to those of comparing $a_{i-j+2} a_{i-j+3} \ldots a_{i-1}$ against $b_1 b_2 \ldots b_{j-1}$.
- Every a_i is incurred the cost of at most two comparisons.
- So, the time complexity is $O(n)$.

String Editing

IM NTU

Problem

Given two strings A $(= a_1 a_2 \cdots a_n)$ and B $(= b_1 b_2 \cdots b_m)$, find the minimum number of changes required to change A character by character such that it becomes equal to B.

Three types of changes (or edit steps) allowed: (1) insert, (2) delete, and (3) replace.

String Editing (cont.)

Let $C(i,j)$ denote the minimum cost of changing $A(i)$ to $B(j)$, where $A(i) = a_1 a_2 \cdots a_i$ and $B(j) = b_1 b_2 \cdots b_j$.

For $i = 0$ or $j = 0$,

$$\begin{aligned} C(i,0) &= i \\ C(0,j) &= j \end{aligned}$$

For $i > 0$ and $j > 0$,

$$C(i,j) = \min \begin{cases} C(i-1,j) + 1 & (\text{deleting } a_i) \\ C(i,j-1) + 1 & (\text{inserting } b_j) \\ C(i-1,j-1) + 1 & (a_i \rightarrow b_j) \\ C(i-1,j-1) & (a_i = b_j) \end{cases}$$

String Editing (cont.)

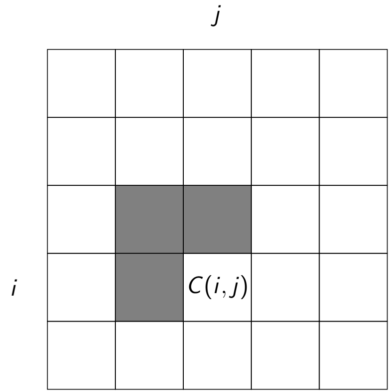

Figure: The dependencies of $C(i,j)$.

Source: redrawn from [Manber 1989, Figure 6.26].

String Editing (cont.)

Algorithm Minimum_Edit_Distance (A, n, B, m);

```
for i := 0 to n do C[i,0] := i;
for j := 1 to m do C[0,j] := j;
for i := 1 to n do
    for j := 1 to m do
        x := C[i-1,j] + 1;
        y := C[i,j-1] + 1;
        if a_i = b_j then
            z := C[i-1,j-1]
        else
            z := C[i-1,j-1] + 1;
    C[i,j] := min(x,y,z)
```

String Editing (cont.)

Algorithm Minimum_Edit_Distance (A, n, B, m);

```
for i := 0 to n do C[i, 0] := i;
for j := 1 to m do C[0, j] := j;
for i := 1 to n do
    for j := 1 to m do
        x := C[i − 1, j] + 1;
        y := C[i, j − 1] + 1;
        if a_i = b_j then
            z := C[i − 1, j − 1]
        else
            z := C[i − 1, j − 1] + 1;
    C[i, j] := min(x, y, z)
```

Its time complexity is clearly $O(mn)$.