IA167: String Algorithms **Spring 2014**

Lecture 5: March 17

Lecturer: Alex Popa *Scribes: Vit Stanislav*

5.1 Approximate pattern matching with mismatches

We search pattern p in text t as in all previous lectures. But now we can have a limited number of character mismatches.

Problem 5.1 *Given text t, pattern p and a number of mismatches k, find all occurrences of p in t with at most k mismatches.*

Example 5.2 *Input:*
t = *banana*
p = *axa*
k = *2*

Output:
p matches t at positions: 2 and 4.

5.1.1 Naive solution

At each position i of the text:

- Count the number of mismatches between t from position i and p.
- Position i is accepted if there are fewer mismatches than k.

This way at each of n steps is done up to m character comparisons, therefore time complexity of the algorithm is $\mathcal{O}(mn)$.

5.1.2 Kangaroo hopping

At first, this algorithm computes LCP (Longest Common Prefix) for concatenation of text t, symbol \$ and pattern p. Using the result we can get LCP for any two positions i and j by calling $LCP(i, n+1+j)$, where i is position in the text and j is position in the pattern. Then for each position i in the text we start by calling $LCP(i, n+1+1)$. It returns value j, which means that at position $j+1$ of the pattern is mismatch and we can call again LCP for positions i and $j+2$ (see Figure 5.1). This can be repeated up to k times. If the end of p is reached, then match at position i his reported.
Time complexity of this algorithm consists of $\mathcal{O}(m+n)$ for computing LCP and for each of n positions of the text k times $\mathcal{O}(1)$ LCP query. All together it is $\mathcal{O}(m+n+nk) = \mathcal{O}(nk)$.

```
       i
t = aaaaaaaaaxaaaaaaaayaaaay
       |LCP | |LCP   |
p =    aaaaaayaaaaaaaaxaaaaz
       1    j
```

Figure 5.1: Kangaroo hopping illustration

5.2 Abrahamson/Kasarajn Algorithm

Problem 5.3 *Given text t and pattern p find the number of mismatches of t at every position of t.*

Definition 5.4 *A character is **frequent** if it appears more than b times in the pattern.*

Lemma 5.5 *Two polynomials can be multiplied using FFT (Fast Fourier Transform) in $\mathcal{O}((n+m)\log(n+m))$*

5.2.1 Algorithm 1 - Frequent characters

For each character of the alphabet is the text and the pattern transformed to a polynomial with coefficients 1 and 0. 1 if there is the given character at the corresponding position, 0 otherwise. The two polynomials are multiplied using FFT. Coefficient of the product at position i: is the number of matches between the pattern and the text from position i.
Since there are at most $\frac{m}{b}$ frequent charaters (each occuring at least b times), complexity of the Algorithm 1 is $\mathcal{O}(\frac{m}{b} n \log m)$.

Example 5.6 *t = ababaaab*
p = bbbb

$c_{t,b}$ = 01010001
$c_{p,b}$ = 1111

$c_{t\times p,b}[1] = 1\times0 + 1\times1 + 1\times0 + 1\times1 = 2$
$c_{t\times p,b}[2] = 1\times1 + 1\times0 + 1\times1 + 1\times0 = 2$
...
$c_{t\times p,b}$ = 22111

5.2.2 Algorithm 2 - Infrequent characters

For each character c of the pattern is made a list of its positions in the pattern:
$s[c] = j_1, j_2, ...j_k$
We need an array r of length n containing zeros. Then we iterate over the positions of the text. At each position i with a character c, $s[c] = j_1, j_2, ...j_k$ tells us that the pattern and the text match if pattern starts at one of the following positions

$i - j_1 - 1$, $i - j_2 - 1$, ..., $i - j_k - 1$.
We just have to increment values in the array r at those positions (if not out of range).
After iterating over all the positions, the array r contains for each position i the number of matches between the text from position i and the pattern. The number of mismatches is length of the pattern minus the number of matches.
Overall we do n steps and at each at most b operations, therefore time complexity of the algorithm is $\mathcal{O}(nb)$.

Example 5.7 *t = mississippi*
p = isip
s[i] = 1, 3
s[s] = 2
s[p] = 4
r = 00000000000

i = 1
t[i] is 'm', there is no 'm' in s therefore no change to r
r = 00000000000

i = 2
t[i] is 'i', s['i'] is 1, 3. Incremented should be positions $i - 1 + 1 = 2$ *and* $i - 3 + 1 = 0$. *The latter is out of bounds, therefore r changes to the following*
r = 01000000000

i = 3
t[i] is 's', s['s'] is 2. Incremented should be position $i - 2 + 1 = 2$, *therefore r changes to the following*
r = 02000000000

etc.

After the whole text is processed:
r = 02202311000

5.2.3 Using Algorithm 1 and Algorithm 2 together

Abrahamson/Kasarajn algorithm uses for each character one of the two algorithms depending on whether the character is frequent, using $b = \sqrt{m \log n}$.

Overall complexity is just the sum of complexity of Algorithm 1 and Algorithm 2, i. e.

$$\mathcal{O}(\frac{m}{b} n \log m) + \mathcal{O}(nb) = \mathcal{O}(\frac{m}{\sqrt{m \log n}} n \log m + n\sqrt{m \log n}) = \mathcal{O}(n\sqrt{m \log n})$$

5.3 Approximate pattern matching with wildcards

Problem 5.8 *Given a text t and a pattern p, which contains wildcards (characters that match any single character), find out whether the pattern occurs in the text.*

Sometimes is used a modification that allows wildcards in the text as well.

Example 5.9 *Input:*
t = banana
p = n?n?

Output:
p matches t at position 3

5.3.1 Algorithm 1 ($\mathcal{O}(nm)$)

Compare the pattern with each position of the text. In the character comparison function, the wildcard equals to any character.

5.3.2 Algorithm 2 ($\mathcal{O}(|\Sigma|^2 n \log(n))$)

For each pair of characters a, b of the alphabet, convert the pattern and the text to binary vectors. The vector for t has ones at positions of a and zeros otherwise. The vector for p has ones at positions of b and zeros otherwise.

Example 5.10 *Vectors for characters a, b:*

```
t = accacca
    1001001

p = ?bc?ccb
    0100001
```

Compute the number of mismatches between these two characters using FFT.
Sum all the vectors obtained from FFT.
The result is a binary vector that contains zero at position i if and only if the text from position i matches the pattern.

5.3.3 Algorithm 3 ($\mathcal{O}(|\Sigma| n \log(n))$)

Convert pattern and text to binary vectors for each character a of the alphabet. The vector for t has ones at positions of a and zeros otherwise. The vector for p has zeros at positions of a and ones otherwise.

Example 5.11 *For character a:*

```
t = accacca
    1001001
```

```
p = cca?cca
    1100110
```

The rest is the same as in the previous algorithm.

5.3.4 Algorithm 4 ($\mathcal{O}(\log(|\Sigma|)n\log(n))$)

It uses a bit representation of characters.
At each step i:

- Vector of the text contains for each character its ith bit.
- Vector of the pattern contains for each character negation of its ith bit.
- Compute the number of mismatches in ith bit using FFT.

The rest is the same as in the previous algorithm.

5.3.5 Algorithm 5 ($\mathcal{O}(n\log(n))$)

In this algorithm we consider every character to be a positive integer (e.g. its ASCII code) and the wildcard to be zero.
We compute the following sum for each position i of the text.

$$\sum_{j=1}^{m} t[i+j-1]p[j](t[i+j-1]-p[j])^2$$

There is a match at position i if and only if the sum above is zero. It is zero only if each of it's elements is zero, because no element of the sum is negative.
An element of the sum is zero because of one of these reasons:

- $p[j]$ is zero (wildcard in the pattern).
- $t[i+j-1]-p[j]$ is zero (the two characters are the same).
- $t[i+j-1]$ is zero (wildcard in the text).

In order to achieve $\mathcal{O}(n\log(n))$ time complexity notice that $(a-b)^2 = a^2 - 2ab + b^2$ can be applied here. $(t[i+j-1]-p[j])^2$ can be split into $t[i+j-1]^2 - 2t[i+j-1]p[j] + p[j]^2$, then the sum can be split into three sums and each of them computed separately using FFT.