31251 – Data Structures and Algorithms

Week 10, Autumn 200

Xianzhi Wang

This week, in the exciting world of algorithms:

- String Searching
- A use of hashing other than storing data
- A Probabilistic Algorithm
- Another Dynamic Programming Algorithm

String Searching

Alphabet (Σ): a finite set of symbols.
$A = (a_0, a_2, \ldots, a_{n-1})$: a string over Σ, $n > 0$.
$B = (b_0, \ldots, b_{m-1})$: another string over Σ, $0 < m \leq n$.

String Matching — determining whether B is a substring of A.

That is to tell whether there is some k (≥ 0) such that
$b_0 = a_k$,
$b_1 = a_{k+1}$,
$\ldots$,
$b_{m-1} = a_{k+m-1}$.

Example

A Naïve Deterministic Solution

We can take a sliding window approach:

1. Start with B lined up with the beginning of A.
2. Match the characters one by one.
    - If it matches, we've found it.
    - Otherwise, move B along one character and start again.
3. Stop when we find it, or run out of A to check against.

There's $n - m + 1$ positions that B could start at, and we do m comparisons each time, so we get $O((n - m + 1)m) \leq O(mn)$.

Replace strings of length m with a function $f : \Sigma^m \to \mathbb{N}$ - then we only need to do one comparison at each step.

- We compute $\beta = f(b_0, \ldots, b_{m-1})$ and then

$$\begin{array}{ccc} \alpha_0 & = & f(a_0, \ldots, a_{m-1}) \\ \alpha_1 & = & f(a_1, \ldots, a_m) \\ \vdots & \vdots & \vdots \\ \alpha_{n-m} & = & f(a_{n-m}, \ldots, a_{n-1}) \end{array}$$

  and compare only the substrings when $\alpha_j = \beta$.

A Slightly Better Approach

The question is what function f to use:

- We could just take $f(x_0, x_1, \ldots, x_{m-1}) = \sum_{i=0}^{m-1} x_i$.
  - This is pretty easy to compute ▸ Example.
  - However too many strings have the same value under f - too many “collisions”.

- We can pick better f and get a better result.

We can use an algorithm that employs the idea of `hashing`.

Hashing produces a "fingerprint" for each string.

- We can pick a hash function such that any two different strings will probably produce a different hash.

Pick a large prime p and a random integer $r \in [1, p-1]$,

We can set

$$f(x_1, x_2 \ldots, x_m) = \sum_{i=\{1,2,\ldots,m\}} (x_i r^{m-i}) \bmod p$$

or

$$f(x_0, x_1 \ldots, x_{m-1}) = \sum_{i=\{0,1,\ldots,m-1\}} (x_i r^{m-1-i}) \bmod p$$

and compute this efficiently.

How Many Collisions Do We Have Now?

How often do we have collisions, i.e.,

$$f(c_0, \ldots, c_{m-1}) == f(d_0, \ldots, d_{m-1}) \ ?$$

Not too often because the above collision implies that

$$(e_0 r^{m-1} + e_1 r^{m-2} + \ldots + e_{m-2} r + e_{m-1}) \mod p == 0$$

where $e_i = c_i - d_i$.

According to Lagrange Theorem[1], there are at most $m-1$ "bad" values of r causing collisions, for each pair of m-tuples $(a_j, \ldots, a_{n-j+1}) \neq (b_0, \ldots, b_{m-1})$.

Considering all comparisons, we have at most

$$\underbrace{(m-1)}_{collision\#\ per\ pair} \quad * \quad \underbrace{(n-m+1)}_{\#\ of\ pairs}$$

collisions.

If $p >> (m-1)(n-m+1)$, collision is very unlikely.

[1] A polynomial of degree k has at most k roots.

Back to the Algorithm

- So we just choose p and r, and then compute f for all length m substrings.
- We only check if a substring $(a_j, \ldots, a_{j+m-1})$ is equal to B when $f(\alpha_j) == f(\beta)$.
- Otherwise B is not a substring of A.

Example

How do we compute f efficiently?

We can adapt the sliding window approach

- For overlapping substrings, we can reuse previous results - `Dynamic Programming`!

$$
\begin{aligned}
f(a_{j+1}, \ldots, a_{j+m}) &= a_{j+1} r^{m-1} + \ldots + a_{j+m} \\
&= r(a_{j+1} r^{m-2} + \ldots + a_{j+m-1}) + a_{j+m} \\
&= r\left(f(a_j, \ldots, a_{j+m-1}) - a_j r^{m-1}\right) + a_{j+m}
\end{aligned}
$$

As such, we can compute all the f values for A in linear time!

With high probability we only have to do m comparisons.

This is also a probabilistic algorithm!

- It is still possible to get a lot of collisions.
- But if we start with too many, we can just guess a different r and start again.

What does that all mean pratically?

- With high probability, the algorithm finds the substring in $O(n+m) = O(n)$.
    - Better than the naïve $O(nm)$ algorithm.
    - Central to this is computing the hash fingerprint quickly – reusing previous results is key.
- In the worse case, we end up with $O(nm)$.

Examples

Example

Given the alphabet $\Sigma = \{*, \&, \%\}$
and the string
$A = \& * \&\% * \% * *\& * \& * \%\% * \% * *\&\% * \underbrace{\& * *\%}_{B} \& *$

If $B = \& * *\%$ then Yes!
If $B = \% * *\%$ then No!

Back

Example

$\Sigma = \{*, \&, \%\} \quad \longrightarrow \quad \Sigma = \{0, 1, 2\}$

$$A = \& * \&\% * \% * *\& * \& * \%\% * \% * *\&\% * \& * *\%\&*$$

$$A = 101202001010220200120100210$$

$$B = \& * *\% \quad \longrightarrow \quad B = 1002$$

$$\beta = 3$$

$$\alpha_j \quad = \quad 4, 3, 5, 4, 2, 3, 1, 2, 2, 3, 5, 4,$$
$$6, 4, 2, 3, 3, 3, 4, 3, 1, 3, 3, 3$$

Back

Example

$$A = 101202001010220200120100210 \quad B = 1002$$

$n = \text{length}(A) = 27$
$m = \text{length}(B) = 4$
$n - m + 1 = 27 - 4 + 1 = 24$

Choose $p = 9973,\ r = 5347,$ Then

$$\beta = 1258$$

$$\begin{aligned} \alpha_j\big|_{j=0,1,\ldots,24} \quad = \{ \quad & 6605, 8512, 6867, 3233, 5609, 2513, \\ & 5347, 7792, 6603, 7793, 1979, 6330, \\ & 8123, 3233, 5609, 2513, 5349, 8512, \\ & 6866, 7859, 7791, \color{red}{1258}, \quad 722, \ 983 \quad \} \end{aligned}$$

Back