31251 – Data Structures and Algorithms

Week 9

Luke Mathieson

This week, in the exciting world of algorithms:

- Searching Strings
- A use of hashing outside just storing data
- (A first) Probabilistic Algorithm
- (A first) Dynamic Programming Algorithm

String Searching

Let Σ be a finite set of symbols called an alphabet.

Let $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$ be two strings over Σ (i.e. $A \in \Sigma^n$ and $B \in \Sigma^m$) with $m \leq n$.

The *String Matching Problem* is the problem of determining whether B is a substring of A (i.e. whether there is some k such that $b_1 = a_k, b_2 = a_{k+1}, \ldots, b_m = a_{k+m-1}$). Example

A Naïve Deterministic Solution

We can take a sliding window approach:

- Start with B lined up with the beginning of A.
- Match the characters one by one.
- If it matches, we've found it.
- If there's a mismatch, move B along one character and start again.
- 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 this takes about $O((n - m + 1)m)$ time.

A Slightly Better Approach

Replace strings of length m with a function $f : \Sigma^m \to \mathbb{N}$ of m variables - then we only need to do one comparison at each step. (It helps to treat Σ as a subset of $\mathbb{N}$).
We compute $\beta = f(b_1, \ldots, b_m)$ and then

$$\begin{array}{ccc} \alpha_1 & = & f(a_1, \ldots, a_m) \\ \alpha_2 & = & f(a_2, \ldots, a_{m+1}) \\ \vdots & \vdots & \vdots \\ \alpha_{n-m+1} & = & f(a_{n-m+1}, \ldots, a_n) \end{array}$$

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

A Slightly Better Approach

The question is what function f to use:

- We could just take $f(x_1, \ldots, x_m) = \sum_{i=1}^{m} x_i$.
  - 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.

Rabin-Karp Algorithm I

We can use an algorithm that employs the same basic idea as 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.

Rabin-Karp Algorithm II

Pick a large prime p and randomly select an integer $r \in [1, p-1]$.

Set

$$f(x_1, \ldots, x_m) = \sum_{i \in [m]} x_i r^{m-i} (\bmod\ p)$$

Looks complicated, but we can actually compute this efficiently.

How Many Collisions Do We Have Now?

How often do we have collisions, that is, how often

$$f(c_1, \ldots, c_m) = f(d_1, \ldots, d_m) \ ?$$

Answer: Not too often because the above collision implies that

$$e_1 r^{m-1} + e_2 r^{m-2} + \ldots + e_{m-1} r + e_m \equiv 0 \pmod{p}$$

where $e_i = c_i - d_i$.

Lagrange Theorem: A polynomial of degree k has at most k roots.

As we have a polynomial of degree $m-1$, for each pair of m-tuples $(c_1, \ldots, c_m) \neq (d_1, \ldots, d_m)$ there are at most $m-1$ "bad" values of r for which a collision is possible.

So if B is not a substring of A, there are at most $(m-1)(n-m+1)$ values of r which might give the same value for f.

So if p is much bigger than $(m-1)(n-m+1)$ and $r \in [1, p-1]$ is selected "at random", the probability of collision is very small.

Back to the Algorithm

- So we can just compute f for all length m substrings (having chosen p and r).
- If $f(B)$ is the same as any of these, we check if the corresponding substring is equal.
- Otherwise B is not a substring of A.

Example

But...

How do we compute f efficiently?
We can adapt the sliding window approach: for overlapping substrings, we can reuse previous results - this is called *Dynamic Programming* (a term to remember).

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

What We Get

So we can compute all the f values for A in *linear* time!

With high probability we only have to do m actual 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 searches the string in $O(n+m)$ time – as $m \leq n$, this is just $O(n)$.
  - Better than the naïve $O(nm)$ algorithm.
  - Central to this is computing the hash fingerprint quickly – reusing previous results as partial, overlapping solutions is key.
- If everything goes wrong though, we just end up with the naïve algorithm – $O(nm)$ time.

Example

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

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

Back

Example

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

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

$$A = 1012020010102202001201\,00210$$

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

$$\beta = 3$$

$$\begin{aligned} \alpha_j \quad = \quad & 4, 3, 5, 4, 2, 3, 1, 2, 2, 3, 5, 4, \\ & 6, 4, 2, 3, 3, 3, 4, 3, 1, 3, 3, 3 \end{aligned}$$

Back

Example

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

$m = 4, \; n = 27$
Choose $p = 9973, \; r = 5347,$ Then

$$\beta = 1258$$

$$\begin{aligned} \alpha_j \quad = \quad & 6605, 8512, 6867, 3233, 5609, 2513, \\ & 5347, 7792, 6603, 7793, 1979, 6330, \\ & 8123, 3233, 5609, 2513, 5349, 8512, \\ & 6866, 7859, 7791, \textcolor{red}{1258}, 722, 983 \end{aligned}$$

Back