9. STRING MATCHING

Raveen de Silva, r.desilva@unsw.edu.au
office: K17 202
Course Admin: Anahita Namvar, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 3, 2021

Table of Contents

String Matching algorithms

- Suppose you have an alphabet $\mathcal{S} = \{s_0, s_1, \ldots, s_{d-1}\}$ of d characters.
- You want to determine whether a string $B = b_0 b_1 \ldots b_{m-1}$ appears as a (contiguous) substring of a much longer string $A = a_0 a_1 \ldots a_{n-1}$.
- The "naive" string matching algorithm runs in $O(nm)$; we want to do better.

Table of Contents

Rabin - Karp Algorithm

- We now show how hashing can be combined with recursion to produce an efficient string matching algorithm.

- We compute a hash value for the string $B = b_0 b_1 b_2 \ldots b_{m-1}$ in the following way.

- First, map each symbol s_i to a corresponding integer i:

$$\mathcal{S} = \{s_0, s_1, s_2, \ldots, s_{d-1}\} \longrightarrow \{0, 1, 2, \ldots, d-1\},$$

  so as to identify each string with a sequence of these integers.

- Hereafter, when we refer to an integer a_i or b_i, we really mean the ID of the symbol a_i or b_i.

Rabin - Karp Algorithm

- We can therefore identify B with a sequence of IDs $\langle b_0, b_1, b_2, \ldots, b_{m-1} \rangle$, each between 0 and $d-1$ inclusive. Viewing these IDs as digits in base d, we can construct a corresponding integer

  $$h(B) = h(b_0 b_1 b_2 \ldots b_m) = d^{m-1} b_0 + d^{m-2} b_1 + \ldots + d \cdot b_{m-2} + b_{m-1}.$$

- This can be evaluated efficiently using Horner's rule:

  $$h(B) = b_{m-1} + d(b_{m-2} + d(b_{m-3} + d(b_{m-4} + \ldots + d(b_1 + d \cdot b_0) \ldots))),$$

  requiring only $m-1$ additions and $m-1$ multiplications.

- Next we choose a large prime number p and define the hash value of B as $H(B) = h(B) \bmod p$. We require that $(d+1)\,p$ fits in a register; it will be made clear later why this is important.

Rabin - Karp Algorithm

- Recall that $A = a_0 a_1 a_2 a_3 \ldots\ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots\ldots a_{n-1}$ where $n \gg m$.
- We want to find efficiently all s such that the string of length m of the form $a_s a_{s+1} \ldots a_{s+m-1}$ and string $b_0 b_1 \ldots b_{m-1}$ are equal.
- For each contiguous substring $A_s = a_s a_{s+1} \ldots a_{s+m-1}$ of string A we also compute its hash value as

  $H(A_s) = d^{m-1} a_s + d^{m-2} a_{s+1} + \ldots + d^1 a_{s+m-2} + a_{s+m-1} \bmod p.$

Rabin - Karp Algorithm

- We can now compare the hash values $H(B)$ and $H(A_s)$ and do a symbol-by-symbol matching only if $H(B) = H(A_s)$.
- Clearly, such an algorithm would be faster than the naive symbol-by-symbol comparison only if we can compute the hash values of substrings A_s faster than what it takes to compare strings B and A_s character by character.
- This is where recursion comes into play: we do not have compute the hash value $H(A_{s+1})$ of $A_{s+1} = a_{s+1}a_{s+2}\dots a_{s+m}$ "from scratch", but we can compute it efficiently from the hash value $H(A_s)$ of $A_s = a_s a_{s+1}\dots a_{s+m-1}$ as follows.

Rabin - Karp Algorithm

Since

$$H(A_s) = d^{m-1}a_s + d^{m-2}a_{s+1} + \ldots d^1 a_{s+m-2} + a_{s+m-1} \bmod p,$$

by multiplying both sides by d we obtain

$$\begin{aligned}
& d \cdot H(A_s) \\
& = d^m a_s + d^{m-1}a_{s+1} + \ldots + d^1 a_{s+m-1} \\
& = d^m a_s + (d^{m-1}a_{s+1} + \ldots + d^1 a_{s+m-1} + a_{s+m}) - a_{s+m} \\
& = d^m a_s + H(A_{s+1}) - a_{s+m} \bmod p
\end{aligned}$$

Rabin - Karp Algorithm

- Consequently,

$$H(A_{s+1}) = d \cdot H(A_s) - d^m a_s + a_{s+m} \bmod p.$$

- To find $d^m a_s \bmod p$, we use the precomputed value $d^m \bmod p$, multiply it by a_s and again take the remainder modulo p.

- Also, since $(-d^m a_s + a_{s+m}) \bmod p$ and $H(A_s)$ are each less than p, it follows that

$$d \cdot H(A_s) + (-d^m a_s + a_{s+m}) \bmod p < (d+1)\, p.$$

- Thus, since we chose p such that $(d+1)\, p$ fits in a register, all the values and the intermediate results for the above expression also fit in a single register.

Rabin - Karp Algorithm

- Thus, we first compute $H(B)$ and $H(A_0)$ using Horner's rule.
- The $O(n)$ subsequent values of $H(A_s)$ for $s > 0$ are computed in constant time using the above recursion.
- $H(A_s)$ is compared with $H(B)$, and if they are equal the strings A_s and B are compared by brute force character by character to confirm whether they are genuinely equal.

Rabin - Karp Algorithm

- Since p was chosen large, the false positives when $H(A_s) = H(B)$ but $A_s \neq B$ are very unlikely, which makes the algorithm run fast in the average case.
- However, as always when we use hashing, we cannot achieve useful bounds for the worst case performance.
- So we now look for algorithms whose worst case performance is guaranteed to be linear.

Table of Contents

String matching with finite automata

- A string matching finite automaton for a pattern B of length m has:
  - $m + 1$ many states $0, 1, \ldots, m$, which correspond to the number of characters matched thus far, and
  - a transition function $\delta(s, c)$, where $0 \leq s \leq m$ and $c \in \mathcal{S}$.
- Suppose that the last s characters of the text A match the first s characters of the pattern B, and that c is the next character in the text. Then $\delta(s, c)$ is the new state after character c is read, i.e. the largest s' so that the last s' characters of A (ending at the new character c) match the first s' characters of B.

String matching with finite automata

We first suppose that $\delta(s, c)$ is given as a pre-constructed table; we will discuss how to construct this table later. As an example, consider the string $B = xyxyxzx$. The table defining $\delta(s, c)$ would then be as follows:

s	matched	x	y	z
0		**1**	0	0
1	x	1	**2**	0
2	xy	**3**	0	0
3	xyx	1	**4**	0
4	$xyxy$	**5**	0	0
5	$xyxyx$	1	4	**6**
6	$xyxyxz$	**7**	0	0
7	$xyxyxzx$	1	2	0

String matching with finite automata

This table can be visualised as a state transition diagram. From each state s, we draw an arrow to $\delta(s, c)$ for each character c that could be encountered next. Arrows pointing to state 0 have been omitted for clarity.

$$\begin{tikzcd}
{} \arrow[r] & 0 \arrow[r, "x"] & 1 \arrow[loop above, "x"] \arrow[r, "y"] & 2 \arrow[r, "x"] & 3 \arrow[r, "y"] \arrow[ll, bend left, "x"] & 4 \arrow[r, "x"] & 5 \arrow[r, "z"] \arrow[l, bend left, "y"] \arrow[llll, bend right, "x"'] & 6 \arrow[r, "x"] & 7 \arrow[llllll, bend right, "x"'] \arrow[lllll, bend left, "y"]
\end{tikzcd}$$

$$B = xyxyxzx$$

String matching with finite automata

- How do we compute the transition function δ, i.e., how do we fill the table?
- Let $B_k = b_0 \ldots b_{k-1}$ denote a prefix of length k of the string B.
- Being at state k means that so far we have matched the prefix B_k.
- If we now see an input character a, then $\delta(k, a)$ is the largest m such that the prefix B_m of string B is a suffix of the string $B_k a$.
- In the particular case where a happens to be b_k, i.e. $B_k a = B_{k+1}$, then $m = k + 1$ and so $\delta(k, a) = k + 1$.

String matching with finite automata

- But what if $a \neq b_k$? We can't extend our match from length k to $k + 1$.

- We'd like to extend some shorter match.

- How do we find $\delta(k, a)$, the largest m such that B_m is a suffix of $B_k a$?

- We match the string against itself: we can recursively compute a function $\pi(k)$ which for each k returns the largest integer m such that the prefix B_m of B is a proper suffix of B_k.

String matching with finite automata

- Suppose we have already found that $\pi(k)$, i.e. $B_{\pi(k)} = b_0 \ldots b_{\pi(k)-1}$ is the longest prefix of B which is a proper suffix of B_k.
- To compute $\pi(k+1)$, we first check whether $b_k = b_{\pi(k)}$.
  - If true, then $\pi(k+1) = \pi(k) + 1$.
  - If false, then we cannot extend $B_{\pi(k)}$. What's the next longest prefix of B which is a proper suffix of B_k?
  - It's $B_{\pi(\pi(k))}$! So we check whether $b_k = b_{\pi(\pi(k))}$.
    - If true, then $\pi(k+1) = \pi(\pi(k)) + 1$.
    - If false, check whether $b_k = b_{\pi(\pi(\pi(k)))}$ ...

The Knuth-Morris-Pratt algorithm

```
function COMPUTE-PREFIX-FUNCTION($B$)
    $m \leftarrow |B|$
    let $\pi[1..m]$ be a new array, and $\pi[1] \leftarrow 0$
    $\ell \leftarrow 0$
    for $k = 1$ to $m - 1$ do
        while $\ell \geq 0$ do
            if $b_k = b_\ell$ then
                $\ell \leftarrow \ell + 1$
                break
            end if
            if $\ell > 0$ then
                $\ell \leftarrow \pi[\ell]$
            else
                break
            end if
        end while
        $\pi[k + 1] \leftarrow \ell$
    end for
    return $\pi$
end function
```

The Knuth-Morris-Pratt algorithm

- What is the complexity of this algorithm? There are $O(m)$ values of k, and for each we might try several values ℓ; is this $O(m^2)$?
- No! It is actually linear, i.e. $O(m)$.
- Maintain two pointers: the left pointer at $k+1-\ell$ (the start point of the match we are trying to extend) and the right pointer at k.
- After each 'step' of the algorithm (i.e. each comparison between b_k and b_ℓ), exactly one of these two pointers is moved forwards.
- Each can take up to m values, so the total number of steps is $O(m)$. This is an example of *amortisation*.

The Knuth-Morris-Pratt algorithm

- We can now do our search for string B in a longer string A.
- Suppose B_s is the longest prefix of B which is a suffix of $A_i = a_0 \dots a_{i-1}$.
- To answer the same question for A_{i+1}, we begin by checking whether $a_i = b_s$.
    - If true, then the answer for A_{i+1} is $s + 1$
    - If false, check whether $a_i = b_{\pi(s)}$ ...
- If the answer for any A_i is m, we have a match!
    - Reset to state $\pi(m)$ to detect any overlapping full matches.
- By the same two pointer argument, the time complexity is $O(n)$.

The Knuth-Morris-Pratt algorithm

```
function KMP-Matcher($A, B$)
    $n \leftarrow |A|$
    $m \leftarrow |B|$
    $\pi \leftarrow$ Compute-Prefix-Function($B$)
    $s \leftarrow 0$
    for $i = 0$ to $n - 1$ do
        while $s \geq 0$ do
            if $a_i = b_s$ then
                $s \leftarrow s + 1$
                break
            end if
```

The Knuth-Morris-Pratt algorithm

```
            **if** $s > 0$ **then**
                $s \leftarrow \pi[k]$
            **else**
                **break**
            **end if**
        **end while**
        **if** $s = m$ **then**
            **print** match found from index $i - m$
            $s \leftarrow \pi[m]$
        **end if**
    **end for**
**end function**
```

Looking for imperfect matches

Sometimes we are not interested in finding just the perfect matches, but also in matches that might have a few errors, such as a few insertions, deletions and replacements.

Problem

Instance: a very long string

$$A = a_0 a_1 a_2 a_3 \ldots\ldots a_s a_{s+1} \ldots a_{s+m-1} \ldots\ldots a_{n-1},$$

a shorter string $B = b_0 b_1 b_2 \ldots b_{m-1}$ where $m \ll n$, and an integer $k \ll m$.

Task: find all matches for B in A which have up to k errors.

Looking for imperfect matches

Solution Outline

Split B into $k + 1$ substrings of (approximately) equal length. Then any match in A with at most k errors must contain a substring which is a perfect match for a substring of B.

We look for all perfect matches in A for each of the $k + 1$ parts of B. For every match, we test by brute force whether the remaining parts of B match sufficiently with the appropriate parts of A.

Table of Contents

PUZZLE!!

On a rectangular table there are 25 round coins of equal size, and no two of these coins overlap. You observe that in current arrangement, it is not possible to add another coin without overlapping any of the existing coins and without the coin falling off the table (for a coin to stay on the table its centre must be within the table).

Show that it is possible to completely cover the table with 100 coins (allowing overlaps).

That's All, Folks!!