Assignment Project Exam Help

7. STRING MATCHING

https://tutorcs.com

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

WeChat: cstutorcs

Term 2, 2022

Table of Contents

String matching algorithms

Suppose you have an alphabet consisting of d symbols. Strings are just sequences of these symbols.

Problem

Determine whether a string $B = b_0 b_1 \ldots b_{m-1}$ (the *pattern*) appears as a (contiguous) substring of a much longer string $A = a_0 a_1 \ldots a_{n-1}$ (the *text*).

String matching problem

Attempt 1

The "naïve" algorithm for string matching is as follows: for every position i in A, check $a_i \dots a_{i+m-1}$ against B character-by-character.

Even with early exit, this runs in $O(mn)$ time in the worst case.

Question

Can we do better?

Table of Contents

String matching with hashing

- 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 $B = b_0 b_1 b_2 \dots b_{m-1}$ in the following way.

- First, map each symbol to a corresponding integer $i \in \{0, 1, 2, \dots, d-1\}$ 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.

Polynomial rolling hash

- 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 $H(B) = h(B) \bmod p$.

Polynomial rolling hash

- 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

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

Polynomial rolling hash

- We can now compare the hash values $H(B)$ and $H(A_s)$.

  - If they disagree, we know that there is no match.
  - If they agree, there might be a match. To confirm this, we'll check it character-by-character.

- There are $O(n)$ substrings A_s to check. If we compute each hash value $H(A_s)$ in $O(m)$ time, this is no better than the naïve algorithm.

- This is where recursion comes into play: we do not have compute the hash value $H(A_{s+1})$ "from scratch". Instead, we can compute it efficiently from the previous hash value $H(A_s)$.

Polynomial rolling hash

- Recall that $A_s = a_s a_{s+1} \ldots a_{s+m-1}$, so

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

- Multiplying both sides by d gives

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

since $A_{s+1} = a_{s+1} \ldots a_{s+m-1} a_{s+m}$.

Polynomial rolling hash

- Rearranging the last equation, we have

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

- Thus we can compute $H(A_{s+1})$ from $H(A_s)$ in constant time.

Note

When implementing this, we need to make sure that the intermediate values are not too large. In particular, we need to choose p such that $(d+1)p$ fits in a single register. This is in conflict with the desired to choose large primes for better hashing.

Rabin-Karp Algorithm

Algorithm

1. First compute $H(B)$ and $H(A_0)$ in $O(m)$ time using Horner's rule.

2. Then compute the $O(n)$ subsequent values of $H(A_s)$ for $s > 0$ each in constant time using the recurrence above.

3. Compare $H(A_s)$ with $H(B)$, and if they are equal then confirm the potential match by checking the strings A_s and B character-by-character.

Rabin-Karp Algorithm

- Since p was chosen large, the false positives (where $H(A_s) = H(B)$ 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 $\delta(k, a)$, where $0 \leq k \leq m$ and $a \in \mathcal{S}$.

- Suppose that the last k characters of the text A match the first k characters of the pattern B, and that a is the next character in the text. Then $\delta(k, a)$ is the new state after character a is read, i.e. the largest k' so that the last k' characters of A (ending at the new character c) match the first k' characters of B.

String matching with finite automata

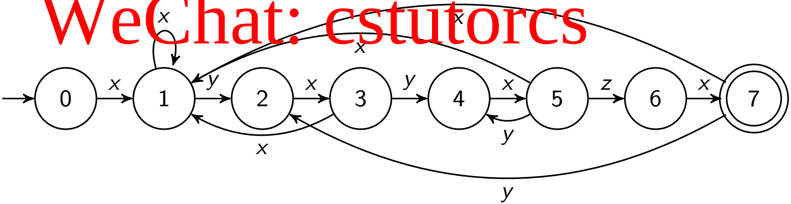
As an example, consider the string $B = xyxyxzx$. The table defining $\delta(k, a)$ would then be as follows:

k	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 k, we draw an arrow to $\delta(k, a)$ for each character a that could be encountered next. Arrows pointing to state 0 have been omitted for clarity.

$$B = xyxyxzx$$



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 ℓ such that the prefix B_ℓ of string B is a suffix of the string $B_k a$.

String matching with finite automata

Question

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

- If a happens to match with b_k, i.e. $B_k a = B_{k+1}$, then $\ell = k + 1$ and so $\delta(k, a) = k + 1$.

- But what if $a \neq b_k$? Then we can't extend our match from length k to $k + 1$.
- We'd like to extend some shorter match instead.
- Any other candidates to be extended must be both a proper suffix of B_k and a prefix of B.

The failure function

Definition

Let $\pi(k)$ be the largest integer ℓ such that the prefix B_ℓ of B is a proper suffix of B_k.

B_ℓ is therefore the longest substring which appears at both the start and the end of B_k (allowing partial but not total overlap).

We will refer to π as the *failure function*.

The failure function

The transition function $\delta(k, a)$ is closely related to this failure function.

- If $a = b_k$, then $\delta(k, a) = k + 1$.

- Otherwise, we look to extend a shorter match. The next candidate has length $\pi(k)$, so we check whether $a = b_{\pi(k)}$ and if so we have $\delta(k, a) = \pi(k) + 1$.

- Otherwise, the next candidate has length $\pi(\pi(k))$ (why?) and so on.

We will describe the algorithm and complete the analysis entirely in terms of the failure function.

The failure function

- Our first step is to solve the value of the failure function for each index of the pattern B.

- We will do this recursively using dynamic programming.

- Naturally, the subproblems are the $\pi(k)$ values for each $1 \leq k \leq m$, and the base case is $\pi(1) = 0$.

- Suppose we have solved $\pi(1), \ldots, \pi(k)$. How do we find $\pi(k+1)$?

The failure function

- Recall that $B_{\pi(k)} = b_0 \dots b_{\pi(k)-1}$ is the longest prefix of B which is a proper suffix of $B_k = b_0 \dots b_{k-1}$.
- To compute $\pi(k+1)$, we first try to extend this match, by checking 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 $b_k = b_{\pi(\pi(\pi(k)))} \dots$

The failure function

```
function Compute-Failure-Function(B)
    m ← |B|
    let π[1..m] be a new array, and π[1] ← 0
    ℓ ← 0
    for k = 1 to m − 1 do
        while ℓ ≥ 0 do
            if b_k = b_ℓ then
                ℓ ← ℓ + 1
                break
            end if
            if ℓ > 0 then
                ℓ ← π[ℓ]
            else
                break
            end if
        end while
        π[k + 1] ← ℓ
    end for
    return π
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 the pattern B in the text A.
- Suppose B_s is the longest prefix of B which is a suffix of $A_i = a_0 \ldots a_{i-1}$.
- 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 $a_i = b_{\pi(s)} \ldots$
- 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-Failure-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

```
            if s > 0 then
                s ← π[s]
            else
                break
            end if
        end while
        if s = m then
            print match found from index i − m
            s ← π[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 text

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

a pattern $B = b_0 b_1 b_2 \ldots b_{m-1}$ 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. Within any match in A with at most k errors, at least one of these $k+1$ parts of B must be matched perfectly.

For each of the $k+1$ parts of B, we use the Knuth-Morris-Pratt algorithm to find all perfect matches in A.

Then for each of these matches, 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!!