CDM

String Searching

Klaus Sutner

Carnegie Mellon University

1 **String Searching**

2 **The Prefix Problem**

3 **State Systems**

4 **The KMP Algorithm**

5 **The Aho-Corasick Algorithm**

6 **The Baeza-Yates-Gonnet Algorithm**

7 **The Rabin-Karp Algorithm**

Here is a natural algorithmic problem:

> Given a word W and a text T, check whether W occurs in T.

The search word W is also referred to as a pattern, we will talk about more general patterns in a while.

Disclaimer:

- We will only talk about exact matching, there are important variants where one is interested in approximate matches. For example, we might be uncertain about the spelling of a word.
- Later we will discuss the problem of searching for multiple words (of course, without rerunning the basic algorithm).
- There is stronger generalization where the target pattern describes a whole class of strings (pattern matching).

Notation

Let m be the length of the pattern W, and n the length of the text T.

$$W = w_1 w_2 \dots w_m \qquad T = t_1 t_2 \dots t_n$$

Write $T(i : k) = t_i t_{i+1} \dots t_{i+k-1}$ for the block of length k in T, starting at position i.

In applications W is typically short, but T is very long (a few dozen versus a million characters). We will always assume that $n \geq m$ but think $n \gg m$.

The underlying alphabet is usually either binary, or ASCII. However, other important alphabets do appear in practice, so we need a general method (think about biology).

Typical Applications

Finding information in potentially very large (or very many) text files.

For example, the OED has an electronic version containing some 50 million words.

Or you might have to search through hundreds of files in a directory.

First fully recognized in the design of `Unix`: most information is placed into text files. See for example /etc/.

There are several small tools that operate on text files and allow one to find and process the information easily.

The `grep` family of string search tools is a core element of these text tools.

Problem Variants

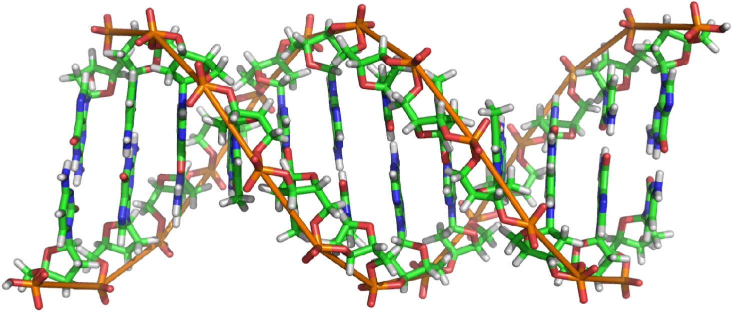
There are several natural variants of the problem:

$$\text{Find the } \left\{\begin{array}{l}\text{first}\\ \text{last}\\ \text{some}\\ \text{all}\end{array}\right\} \text{ position(s) } i \text{ such that } T(i:m) = W.$$

As we will see, in most algorithms it is easy to deal with all these and similar variants by modifying the basic approach slightly.

For example, the Unix utility `grep` has on option `-c` which causes a count of the matches (rather than the match positions) to be returned.

Killer Application: Computational Biology



DNA is just a long word over a 4-letter alphabet.

Brute Force Searching

There is an obvious brute force algorithm: just try all possible places where the search word could possibly occur in the text.

```
for i = 1 .. n - m + 1  do
     if( W == T(i:m) )
         report match at i;
```

Clearly works, but since each comparison `W == T(i:m)` is $O(m)$ the whole algorithm may take up to $O(n \cdot m)$ steps: think about searching for $00 \ldots 001$ in a text of all zeros.

This won't do unless the search string is very short.

Shift One Rule

One way of thinking about the brute-force algorithm is to picture the pattern placed underneath the text.

We scan the pattern from left to right and we scan the corresponding letters in the text. If we get all the way to the end we have a match.

If there is a mismatch, we shift the pattern one position to the right and start all over.

```
... c a b b a a c a a c a c a c c b ...
            a a c a a d
              a a c a a d
                a a c a a d
                  a a c a a d
                    a a c a a d
```

Preprocessing

Clearly there are occasions when we actually could shift by more than one place, e.g., when we have found a letter in the text that does not appear at all in the pattern.

```
... c a b b a a c a a x a c a c c b ...
            a a c a a d
                        a a c a a d
```

How about trying to figure out ahead of time how far we have to shift before another match could occur?

The preprocessing operates on the pattern alone and the computed information can be used for any text.

1. String Searching
2. **The Prefix Problem**
3. State Systems
4. The KMP Algorithm
5. The Aho-Corasick Algorithm
6. The Baeza-Yates-Gonnet Algorithm
7. The Rabin-Karp Algorithm

Detour: The Prefix Problem

Let's digress for a moment and solve a slightly bizarre problem that will turn out to be very useful for our matching problem.

The problem originally arose in the study of repetitions in strings, an issue related to string searching but somewhat different in nature.

At any rate, the Prefix Problem has some great features:

- It has an elegant (and far from obvious) dynamic programming type algorithm that runs in linear time.
- It can be used to solve the string searching problem.
- It will come in handy again later for another string searching algorithm.

Finding Maximal Prefixes

Suppose we have a string S of length n (just a single string, no pattern and text). We want to find maximal blocks in S are also prefixes of S. More precisely, we want to compute the values

$$Z_k = \max\big(\, \ell \mid S(1:\ell) = S(k:\ell) \,\big)$$

for $k = 2, \ldots, n$.

So $0 \leq Z_k < n$ where 0 indicates that the first letter already differs: $s_1 \neq s_k$.

Clearly we can do this in $O(n^2)$ steps by brute-force.

Is there any hope to this faster?

Linear Time?

Perhaps we can handle the Prefix Problem in time linear in n?

Linear time means that we cannot afford to touch the same letter more than once (OK, really $O(1)$ times and there may be $O(1)$ exceptions).

Hence we should avoid recomputation and systematically exploit already known prefixes.

For example, if we know that there is a prefix of length 25 starting at position $k = 90$: $Z_{90} = 25$.

Then to compute Z_{100} we don't need to look at S_{100}, S_{101}, ..., S_{114}: we already know what these letters are.

So we can jump to S_{115}. Sounds like a plan.

Exploiting Previous Work

Say we compute $Z_2, \dots, Z_n$ in this order.

We want to exploit knowledge of Z_i, $i < k$, for the computation of Z_k.

Let

$$r = \max\big(\, i + Z_i - 1 \mid i < k, Z_i > 0 \,\big)$$

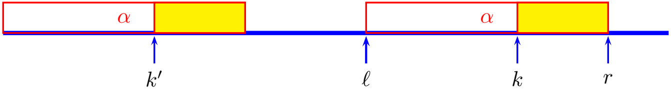
be the right end of the largest match prior to k and let $l < k$ be the corresponding index.

Key Observation

- If $k > r$ then we have to use brute-force.
- But for $k \leq r$ we can skip all comparisons up to position r.

The Critical Case

$k < r = \ell + Z_\ell - 1$



When $k < r$ we already have some partial information about Z_k available: we have already dealt with the block from k to r.

The Prefix Algorithm

The easy part: brute-force when $k > r$.

```
l = r = 0;

for( k = 1; k <= m; k++ )
{
  if( k > r )                // brute-force matching
  {
    for( j = 1; k + j <= m && S[j] == S[k+j-1]; j++ );
    Z[k] = j-1;
    r = k + Z[k] - 1;
    l = k;
  }
  else                       // interesting part, reuse information
```

The Hard Part

```
    else                           // k <= r
    {
      d = r - k + 1;               // distance to end of known match
      kk = k - l + 1;              // start of known match
      if( Z[kk] < d )              // can’t get longer match
           Z[k] = Z[kk];
      else                         // extend by brute-force
      {
        for( j = 1; r + j <= m && S[d+j] == S[r+j]; j++) ;
        Z[k] = r + j - k;
        r = r + j - 1;
        l = k;
      }
    }
}
```

Discussion

This is similar to dynamic programming, but not quite the same.

Theorem

The running time of the prefix algorithm is $O(n)$.
The extra space requirement is $\Theta(n)$ for the Z values.

Proof.

Note that r is non-decreasing and comparison of characters in S only take place in positions to the right of r.

Whenever there is a match, r is increased accordingly.

□

Linear Time String Searching

Theorem

We can find all occurrences of a pattern W in a text T in time $O(n+m)$ using $\Theta(m)$ extra space requirement.

Proof.

Run the prefix algorithm on

$$S = W\#T$$

where $\#$ is a separator symbol not in the alphabet of W and T.

Then W occurs in T at position k iff $Z_{k+m+1} = m$.

It suffices essentially to compute the first m values of Z.

□

Some Comments

There is an important concept hiding here: reduction.

Tackle one computational problem by translating it into another for which we already have a good algorithm.

If the translation is fast, we have a fast solution to the original problem.

Exercise

Provide the details for this application of the prefix algorithm.

Exercise

What if we wanted only the leftmost occurrence? How about the rightmost occurrence?

Exercise

Explain why the linear space requirement for the Prefix Algorithm is not really an issue here.

Case Closed, Next

Seems like we have slaughtered the beast: it's clearly impossible to beat linear time.

Or is it?

This looks like some kind of adversary argument:

> Suppose you claim to have a faster, sublinear algorithm. Then the algorithm cannot look at all the letters in W and T. But then we can change these letters without the algorithm ever noticing. If the algorithm says Yes, we change a few letters so the answer is No and vice versa.

A similar argument can be used to show that in an unordered array search takes at least linear time.

But is this really correct for string searching?

Shifting

Again, think of brute-force as shifting the pattern across the text. We use the shift 1 rule when a mismatch occurs.

```
... c a b b a a c a a c a c a ...
            a a c a a d
              a a c a a d
                a a c a a d
                  a a c a a d
                    a a c a a d
```

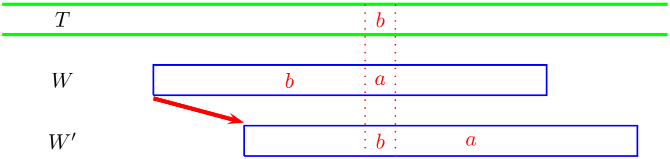
There is a mismatch in position 6 ($d \neq c$), but it is bad to shift the pattern only by one position: there cannot possibly be a match, e.g. since the second a would move to a position with a c.

We ought to be able to exploit this to shift more than one place, at least sometimes. So we should be able to obtain sublinear running time, at least sometimes.

Bad Letter Shift Rule

Here is one example of a better shifting rule.

Suppose we have a match of the first $k-1$ letters of W but there is a mismatch in position k of W. Let $b \neq a = w_k$ be the corresponding letter in T and consider the rightmost occurrence of b in $W(k)$.



Shift Away

Computing the rightmost occurrence of b in $W(k)$ is too complicated, here is a conservative approximation. Define $\pi(a)$ to be the position of the rightmost occurrence of letter a in W where a is any letter in the alphabet. If a does not appear in W let $\pi(a) = 0$.

Then we can shift the pattern at least

$$\max(1, k - \pi(b))$$

places.

Note that the crucial problem here is to make sure we don't miss potential occurrences of the pattern in T by shifting too far.

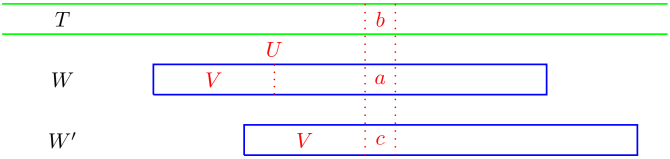
Exercise

Show that the Bad Letter Shift Rule works: no occurrence of the pattern can be missed. Discuss the usefulness of this rule.

Longest Suffix Rule

More generally, how far can we possibly shift?

We have just seen a prefix U of W in T, followed by a letter a that does not match the next letter b in T:



Seems that V should be chosen to be the longest suffix of U that is also a prefix of it.

LSR Matching Algorithm

The whole matching algorithm now looks like so. Initially place W at the begin of T.

- If the next letter in T is the same as the next letter in W we just move ahead ($a = b$ in the picture).
- If there is a mismatch, we apply the LSR to determine how far we have to move the pattern to the right and continue with the matching process. And so on till we find a match or fall off the end of the string.

We will see that it is actually useful to break up the shift operation into several steps (using a so-called failure function).

Longest Suffix lemma

Lemma

If shifting is done according to the Longest Suffix Rule then no occurrence of the pattern can be missed.

Proof.

Show by induction on $0 \leq r \leq n$ that at any time during the execution of the algorithm the part of W that is currently matched is the longest prefix of W that is a suffix of $T(r)$. In other words, we have found the largest k such that

$$W(k) = T(r - k + 1 : k)$$

Initially $r = k = 0$. The LSR guarantees that the matching property is preserved at each step.

But then we find all occurrences of all prefixes of W, and in particular all occurrences of W itself.

□

1. String Searching
2. The Prefix Problem
3. **State Systems**
4. The KMP Algorithm
5. The Aho-Corasick Algorithm
6. The Baeza-Yates-Gonnet Algorithm
7. The Rabin-Karp Algorithm

A State/Transition Model

Here is a better way of thinking about the matcher: a system that has internal states, sometimes called a finite state machine or an automaton.

It reads the text letter by letter, and each letter causes the system to make a transition into a new state.

The next state depends solely on

- the current state and
- the next letter.

If we write Q for the set of states and Σ for the possible letters, the transitions can be explained in terms of a transition function

$$\delta : Q \times \Sigma \to Q$$

Alphabet Sizes

Here are some sizes of Σ that occur in practical applications:

- 2: a binary alphabet
- 4: nucleotide bases (Adenine, Thymidine, Cytosine and Guanine)
- 16: hexadecimal numbers
- 20: amino acids (in polypeptide chains)
- 128: 7-bit ASCII
- 256: 8-bit ASCII, bytes

Other values are quite possible, too. Think about pattern matching in a compressed file.

A Detailed Example

For examples, it is better to use small toy-alphabets such as $\{a, b\}$.

Consider the pattern $W = aabaab$.

The natural states in our pattern matcher are all the prefixes of W:

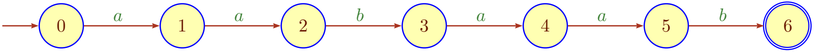
$$\varepsilon, a, aa, aab, aaba, aabaa, aabaab$$

In the actual implementation one would use integers instead: $Q = \{0, 1, \ldots, 6\}$.

So state p means: we have seen the prefix $W(p)$ of length p of the search string W.

Forward Transitions

Part of δ is easy to describe: if we are in state p and we get the “right” letter we go to state $p + 1$.



We reach state 6 whenever we have a match.

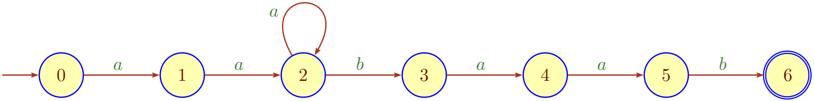
The problem is to decide where to go when the next letter we read from T is “wrong”.

Backward Transitions

For example, if we are in state 2 and we see another a (rather than the desired b), where should we go?

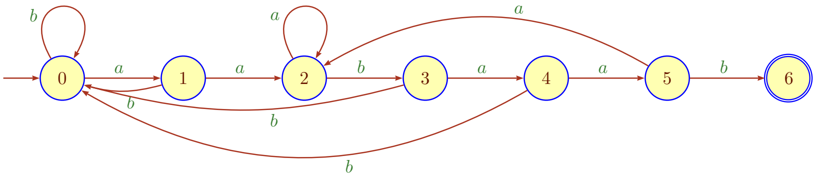
Resetting to 0 would be a mistake: we could miss a match.

We have to use the Longest Suffix Rule from above: in this case we have to stay in state 2 since we still have seen $W(2) = aa$.



All Transitions

Applying the LSR everywhere we get:



Expressed as a table this diagram looks like so:

p	0	1	2	3	4	5	6
a	1	2	2	4	5	2	–
b	0	0	3	0	0	6	–

Running the Matcher

Once we have δ represented by a lookup table like the one above it is really easy to check T for matches: we "run" the automaton on input T like so:

```
p = 0;

for i = 1 .. n do
       p = delta[ p ][ T[i] ];
       if( p == m )
            report match             // and reset p
```

Exactly how p is reset depends on what we are looking for.

Exercise

Figure out how to deal with the following queries: leftmost occurrence, all occurrences including overlapping ones and all occurrences excluding overlapping ones.

The Transition Table

So how do we compute the table given $W = w_1 w_2 \dots w_m$ and Σ?

Write $W(i)$ for the prefix $w_1 w_2 \dots w_i$.

Suppose the next letter read from T is a.

We use the following transition strategy, given current state i:

- If $a = w_{i+1}$, we move on to state $i + 1$.
- Otherwise, go to j maximal such that $W(j)$ is a suffix of $W(i)a$.

This translates easily into a brute-force algorithm.

Brute-Force

```
for i = 0,..,m do
foreach a in Sigma do
{
        j = min( m, i+1 );
        while( w(j)  is not a suffix of w(i)a )
                j--;
}
```

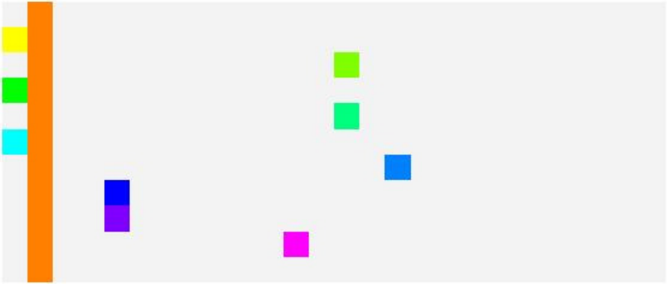
This version is $O(m^3 \cdot k)$ where k is the size of the alphabet.

Can be improved to $O(m \cdot k)$, and thus in practice linear in the size of the search string.

For the whole search process, we get $O(m \cdot k + n)$.

Storing Zeros

The last method is inefficient in the sense that we compute a large two-dimensional table (at least for 8-bit ASCII) that mostly contains trivial transitions back to state 0.



The transition matrix for search word $W = bananapeel$ over a 26-letter alphabet (lower-case ASCII). Almost all entries are trivial.

1. String Searching
2. The Prefix Problem
3. State Systems
4. **The KMP Algorithm**
5. The Aho-Corasick Algorithm
6. The Baeza-Yates-Gonnet Algorithm
7. The Rabin-Karp Algorithm

A better solution is to calculate a so-called failure function π that determines where to go whenever there is a mismatch in the next letter.

The crucial trick is that the failure function is defined only on the states (and can be stored in a linear array).

The best way to describe the failure function is to use as states the set P of all prefixes of W. Also let $P^+ = P - \{\varepsilon\}$ the set of non-empty prefixes.

$$\begin{aligned}
&\pi : P^+ \to P \\
&\pi(u) = \text{ longest proper suffix of } u \text{ in } P.
\end{aligned}$$

Missing Transitions

Given the failure function, we can compute the transitions as follows:

$$
\begin{aligned}
&\delta : P \times \Sigma \to P \\
&\delta(u,a) = \begin{cases} ua & \text{if } ua \in P, \\ \delta(\pi(u),a) & \text{if } ua \notin P, u \neq \varepsilon \\ \varepsilon & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note the recursive definition. We are actually applying the failure function repeatedly:

$$\delta(\pi^{i}(u),a)$$

until we either can use a forward transition labeled a or until we are back at ε.

Iterating

In the KMP algorithm we apply δ repeatedly, starting at state ε and reading in the text letter by letter.

$$\delta(p, a_1 a_2 \dots a_k) = \delta(\dots \delta(\delta(p, a_1), a_2), \dots, a_k)$$

For the correctness of the algorithm one has the following result:

Lemma

$\delta(\varepsilon, T(k))$ is the longest prefix of W that is a suffix of $T(k)$.

So we have a match if, and only if, $\delta(\varepsilon, T(k)) = W$, or whenever the automaton enters state m.

No Matrix

Again: there is no need to store the whole transition matrix, we just compute the values of δ on the fly.

In practice, the failure function would be implemented with integers, not prefixes:

$$\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$$
$$\pi(p) = \max\Big(i < p \mid W(i) \text{ is suffix of } W(p) \Big)$$

Example

$W = aaabbbaaaaaab$ produces

$$\pi = (0, 1, 2, 0, 0, 0, 1, 2, 3, 3, 3, 3, 4)$$

Lazy Approach

We still need to find a way to compute the failure function efficiently.

Ideal would be a pre-computation of cost $O(m)$.

One way is to use the Prefix Algorithm from above on input W and then set

$$\pi(p) = \min\Big(k \mid p = k + Z_k - 1 \Big)$$

Exercise

Explain how to compute π from the Z values in time linear in m.

Exercise

Explain why this method is correct.

A Direct Method

For a more direct approach (and slightly more efficient approach), first note that $\pi(1) = 0$ and for $p > 1$ we have:

$$\pi(p{+}1) = \begin{cases} \pi^i(p) + 1 & i \geq 1 \text{ minimal: } W_{\pi^i(p)+1} = W_{p+1}, \\ 0 & \text{no such } i \text{ exists.} \end{cases}$$

This calls for a dynamic programming approach: we compute $\pi(1), \pi(2), \ldots, \pi(m)$ in this order.

Computing the Failure Function

```
pi[1] = 0;
i = 0;

for p = 2 .. m do                       //      i = pi[p−1];

        while( i > 0 && w[i+1] != w[p] ) i = pi[i];

        if( w[i+1] == w[p] ) i++;

        pi[p] = i;
```

Note that it is not clear that the running time is linear, nor is it clear that the algorithm is correct.

Analysis

Lemma

The failure function can be computed in time linear in m using dynamic programming.

Proof.

Running time: Easy induction shows `pi[p] < p`. Hence i strictly decreases every time the while-loop is executed. But $i \geq 0$, and i is incremented at most $m - 1$ times.

Correctness: By induction on the pattern: assume it works for w, show it works for wa. Note: up to $p = m + 1$ the execution of the code is the same for wa as for w. So focus on the last execution of the loop.

Suppose $w = ub \ldots u$ where $\pi(w) = u$, $b \in \Sigma$. (the strings might overlap). Clearly $|\pi(wa)| \leq \pi(w) + 1$. If $b = a$ we have $\pi(wa) = ua = ub$ and we're done. Otherwise $\pi(wa) = \pi(ua)$: the while-loop will perform these reductions to a shorter string until we find a matching extension, or until we hit 0. □

Examples

Failure functions are rather tedious to compute by hand, make sure to implement the algorithm to see how it works.

Example

$w = aabaab$	$\pi = 0, 1, 0, 1, 2, 3$
$w = aaabaabaaa$	$\pi = 0, 1, 2, 0, 1, 2, 0, 1, 2, 3$
$w = cccccacaaacbaccbabac$	$\pi = 0, 1, 2, 3, 4, 0, 1, 0, 0, 0, 1, 0, 0, 1, 2, 0, 0, 0, 0, 1$
$w = coconut$	$\pi = 0, 0, 1, 2, 0, 0, 0$

The Matching Algorithm

Given the pi[] array, the actual search is now easy:

```
p = 0;

for i = 1 .. n do

    while( 0 < p && w[p+1] != T[i] )      // backtrack
            p = pi[p];

    if( w[p+1] == T[i] )                  // advance if possible
            p++;

    if( p == m )
            return match;
```

The KMP Theorem

Theorem

The KMP algorithm works in time $\Theta(m + n)$ and correctly finds occurrences of the search string w in text T.

Correctness and running time arguments are very similar to the argument for the failure function.

Note we failed to get sublinear performance.

As a matter of fact, KMP is mostly of interest for historical reasons.

However, the idea of using a transition system is crucial for a number of algorithms that deal with much more general patterns (finite state machine based matchers such as `grep`).

Application: 2-Dim Pattern Matching

Suppose we have a 2-dimensional text $T \in \Sigma^{n \times n'}$ and a pattern $P \in \Sigma^{m \times m'}$. We want to check whether P occurs in T.

Say, $\Sigma = \{0, 1, \ldots, k{-}1\}$ and let $\Gamma = \Sigma^m$. A super-row of T is a block of m consecutive rows in T. We can think of a super-row as a word in $\Gamma^{n'}$. Similarly, $P \in \Gamma^{m'}$. Pick a sufficiently large machine-sized prime p and fingerprint $a \in \Gamma$ as $a \bmod p$.

Starting at $\ell = 1$, apply KMP to check for P in the super-row $T[\ell : m, n']$. Use a Rabin-Karp style sliding window to compute the next super-row, and keep searching.

Provided that p is large enough to avoid many false positives, the running time is exptected to be $O(mm' + nn')$.

1 String Searching

2 The Prefix Problem

3 State Systems

4 The KMP Algorithm

5 The Aho-Corasick Algorithm

6 The Baeza-Yates-Gonnet Algorithm

7 The Rabin-Karp Algorithm

Multiple String Matching

Here is a very simple case of a more complicated pattern: instead of searching for a single word W we search for a collection of words:

$$W_1, W_2, \ldots, W_k$$

The search is supposed to report occurrences of any one of the search words (disjunctive as opposed to conjunctive).

Of course, we can handle this problem by running k separate searches for each word W_i.

Correct, but clearly inefficient. How do we speed this up?

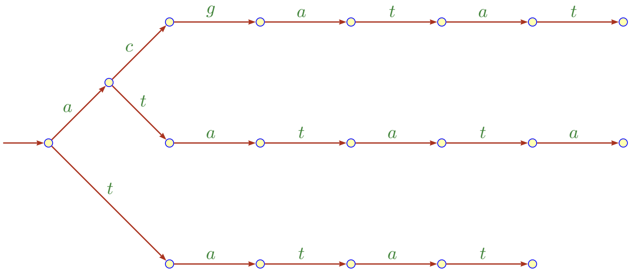
It is tempting to try to generalize the transition system approach.

Instead of dealing with all prefixes of a single word W we now have to handle the prefixes of all words W_i.

Example

Suppose the search words are $acgatat$, $atatata$, $tatat$.

Since some of prefixes are the same, we will not have three separate linear structures but a tree-like structure:



The intrepid 211 student immediately recognizes this as a trie.

The problem is that we only have taken care of the case when the letters in T match.

What do we do when there is a mismatch?

We can adapt the Longest Suffix Rule: suppose U is the current match, followed by the "wrong" letter.

Then we need to find the longest suffix V of U that is also a prefix of a search word W_i (though not necessarily the same word that U is a prefix of).

In the worst case, we back out all the way to the empty prefix ε (the root of the trie).

Failure Transitions Table

Here are the failure transitions determined be the Longest Suffix Rule for the words

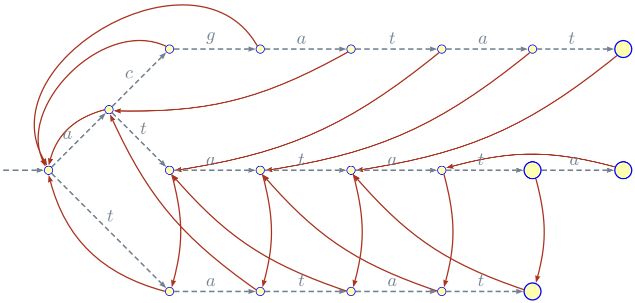
$$acgatat, atatata, tatat$$

u	$\pi(u)$	u	$\pi(u)$
a	ε	$atat$	tat
ac	ε	$atata$	$tata$
acg	ε	$atatat$	$tatat$
$acga$	a	$atatata$	$atata$
$acgat$	at	t	ε
$acgata$	ata	ta	a
$acgatat$	$atat$	tat	at
at	t	$tata$	ata
ata	ta	$tatat$	$atat$

Failure Transitions Diagram

The corresponding diagram.

The larger states indicate that a match has been found. This arrangement is for the “find all occurrences” version.



The matching algorithm is essentially the same as KMP: backtrack until a good forward transition is found or until you get back to the root.

Matching Conditions

Note that in the original trie only the leaf nodes correspond to a full match: each leaf corresponds to exactly one of the search words.

But when we add the failure transitions we have propagate the matched words backwards to the source of the transition.

In general, a single state can correspond to having reached multiple matches.

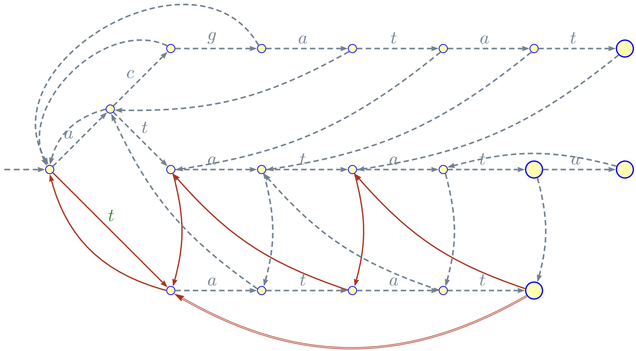
For example, consider the search words

$$aaab, aaaab, aaaaab$$

Every match of the last word is also a match for the other two (albeit with different starting positions.

Sample Step

Current state: $tatat$, next input t.



Aho-Corasick Algorithm (1975)

Is used in the famous `fgrep` utility.

It is clear how to build the the trie in time $O(M)$ where $M = \sum |W_i|$.

For large alphabets one would have to worry about efficiency if one winds up maintaining lots of null pointers (for nucleotide bases there are only 4 letters, no problem).

To describe the algorithm it is best to think of the nodes in the trie as prefixes. So let P be the set of all prefixes of $W_1, W_2, \ldots, W_k$, and $P^+ = P - \{\varepsilon\}$ the set of non-empty prefixes.

The forward transitions describing the trie are given by a partial function

$$\delta : P \times \Sigma \to P$$

Failure to Transition

This is very similar to KMP, except that the set of all prefixes comes not from one word but from several.

$$\begin{aligned} &\pi : P^+ \to P \\ &\pi(p) = \text{ longest proper suffix of } p \text{ in } P. \end{aligned}$$

Given the failure function, we can compute the transitions on the fly as follows:

$$\delta(u,a) = \begin{cases} ua & \text{if } ua \in P, \\ \delta(\pi(u),a) & \text{if } ua \notin P, u \neq \varepsilon \\ \varepsilon & \text{otherwise.} \end{cases}$$

Computing the Failure Function

How do we pre-compute π in the first place?

Exactly the way we did in KMP.

The only difference is that instead of traversing a single string left-to-right we now have to traverse a trie.

If the the traversal is in breadth-first mode, the values of π for all length-lex smaller q are already known when we compute $\pi(p)$.

Essentially the same argument as in the KMP case shows that the whole computation is linear.

Exercise

Explain in detail how to generalize the computation of the failure function from KMP to Aho-Corasick.

Aho-Corasick Theorem

Theorem

The Aho-Corasick algorithm works in time $\Theta(M + n)$ where M is the total length of all search words. It correctly finds all occurrences of all search words W_i in text T.

Proof.

Proof is essentially the same as for KMP.

□

Exercise

Prove the theorem.

1 String Searching

2 The Prefix Problem

3 State Systems

4 The KMP Algorithm

5 The Aho-Corasick Algorithm

6 **The Baeza-Yates-Gonnet Algorithm**

7 The Rabin-Karp Algorithm

Baeza-Yates-Gonnet Shifting

Let's return to single word searches.

One can avoid the comparison based approach of the previous algorithms and get excellent performance for small patterns by computing partial matches in a clever way.

As always, assume W has length m and T has length $n > m$.

First, some clever notation (due to Knuth): let φ be some assertion.

$$[\varphi] = \begin{cases} 1 & \text{if } \varphi \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

Example

$\sum_{k \leq n} [k \text{ prime}]$ is the number of primes up to n.

A Match Matrix

Define a $m+1$ by $n+1$ binary matrix M by

$$M(i,j) = [W(i) = T(j-i+1:j)]$$

Thus $M(i,j) = 1$ if the prefix $W(i)$ appears in T ending in position j.

The 1's in the bottom row of M indicate all the matches.

Of course, we cannot afford to compute all of M.

But if m is small, we can compute the columns of M quickly.

Shift and And

For any bitvector $\boldsymbol{x} = x_1, x_2, \ldots, x_r$ define

$$\text{shift}(\boldsymbol{x}) = 1, x_1, x_2, \ldots, x_{r-1}$$

For any two vectors we write

$$\boldsymbol{x} \mathbin{\&} \boldsymbol{y}$$

for the bit-wise "and" of the two vectors.

For any letter a of the alphabet let P_a be the bitvector that indicates the positions in W where a occurs.

$$P_a(i) = 1 \iff W_i = a$$

Iterative Algorithm

We can now compute the matrix M column by column:

$\boldsymbol{x}_0 = (1, 0, \ldots, 0)$ is the first column.

Given $\boldsymbol{x}_{k-1}$ define

$$\boldsymbol{x}_k = \mathrm{shift}(\boldsymbol{x}) \mathbin{\&} P_{T_k}$$

An easy induction shows that $\boldsymbol{x}_k$ is indeed the kth column of M.

Whenever the last component of $\boldsymbol{x}_k$ is 1 we have found a match.

Discussion

How useful is the Baeza-Yates-Gonnet approach?

If the bitvectors fit into a single computer word (4 or 8 bytes) then the operations “shift” and “bit-wise and” are both $O(1)$ and very fast.

Pre-computation of the vectors P_a requires $O(m^2)$ steps, but recall that $m \leq 32($ or $64)$, so the part is fast.

The actual scan is then linear in n, the length of T.

Again, this does not scale to longer patterns.

1 String Searching

2 The Prefix Problem

3 State Systems

4 The KMP Algorithm

5 The Aho-Corasick Algorithm

6 The Baeza-Yates-Gonnet Algorithm

7 **The Rabin-Karp Algorithm**

Rabin-Karp Fingerprinting

Here is a method that uses modular arithmetic instead of combinatorics.

Suppose we are dealing with 8-bit ASCII text. We can think of the search string W as a number in base $B = 256$:

$$\mathrm{val}(W) = w_1 \cdot B^{m-1} + w_2 \cdot B^{m-2} + \ldots + w_m.$$

So we can simply compute $\mathrm{val}(T(i:m))$ for all $i = 1, \ldots, n-m+1$ and compare them all to $\mathrm{val}(W)$.

Of course, computing all these values takes $\Omega(nm)$ arithmetic steps, and is too slow.

But ...

Fast Updates

But note that $\mathrm{val}(T(i+i:m))$ can be computed from $\mathrm{val}(T(i:m))$ relatively inexpensively:

$$\mathrm{val}(T(i+1:m)) = (B \cdot \mathrm{val}(T(i:m))) \bmod B^m + T_{i+m}.$$

Thus we only need $O(m+n)$ arithmetic steps to compute all the T values.

But note that $256^m = 2^{8m}$ is a rather large integer for long search words so we cannot expect to use machine-sized integers for this.

Arbitrary precision arithmetic is now commonplace, but the operations are expensive.

Cheaper Arithmetic

Here is a trick to make the arithmetic cheap:

> Compute modulo p for some suitable prime p (machine sized integer).

In other words, we generate the fingerprint $\mathrm{val}(T(i:m)) \bmod p$ and compare it to the fingerprint $\mathrm{val}(W) \bmod p$ of W.

The problem is that we may have false positives:

$$\mathrm{val}(W) = \mathrm{val}(T(i:m)) \pmod{p}$$

but still $W \neq T(i:m)$.

We have to verify a real hit at an additional cost of $O(m)$ steps using letter-by-letter comparisons.

Picking the Prime

Problem: How many false alarms are there?

Suppose s_0 is the number of correct hits, and s_1 the number of spurious hits.

Then the running time is $O(n + (s_0 + s_1)m)$.

Let's say we pick p larger than m (which is easy to do).

Assuming that taking mods works like a random function (which seems quite reasonable) we can estimate $s_1 = O(n/p)$, yielding a total running time of

$$O(n + (s_0 + n/p)m) = O(n + s_0 m).$$

Summary

- String matching can be solved in linear time for a single target string and for multiple target strings.
- Specialized methods exist for very short search strings.
- Fingerprinting can be used to reduce the number of explicit comparisons that are needed.
- There are many important generalizations (inexact matching, regular expressions, generalized regular expressions).