max planck institut
informatik

Approximate pattern matching

Algorithms on Strings

Paweł Gawrychowski

June 19, 2013

Outline

We already know how to find an exact occurrence of the pattern in the text (very efficiently). But do we really care about exact occurrences?

Pattern matching with k mismatches

Given a pattern $p[1..m]$ and a text $t[1..n]$, does p occur in t with at most k mismatches, i.e., is there i such that p and $t[i..i+m-1]$ differ at at most k positions?

Trivial solution works in $\mathcal{O}(nm)$ time, so we will see how to solve this in $\mathcal{O}(nk)$ time. For small k, this is very fast, and for large k... well, large k are not that interesting anyway.

Recall that we have the following useful tool.

Suffix array + constant time RMQ queries

Given a text w, we can construct in linear time a structure of size $\mathcal{O}(|w|)$ which allows us to answer any query of the form "what is the longest common prefix of $w[i..|w|]$ and $w[j..|w|]$?" in constant time.

We apply the tool to $w = t\$p$. Then we can compute the longest common prefix of any $t[i..n]$ and $p[j..m]$ in constant time.

We iterate over all possible starting positions in t. For a fixed starting position, we want to check if the pattern occurs there with at most k mismatches.

We iteratively compute and cut the longest common prefix of both the remaining part of the pattern and the remaining part of the corresponding part of the text. We stop when we either process the whole pattern or detect more than k mismatches.

Lemma

The above algorithm correctly solves pattern matching with k mismatches in $\mathcal{O}(nk)$ time.

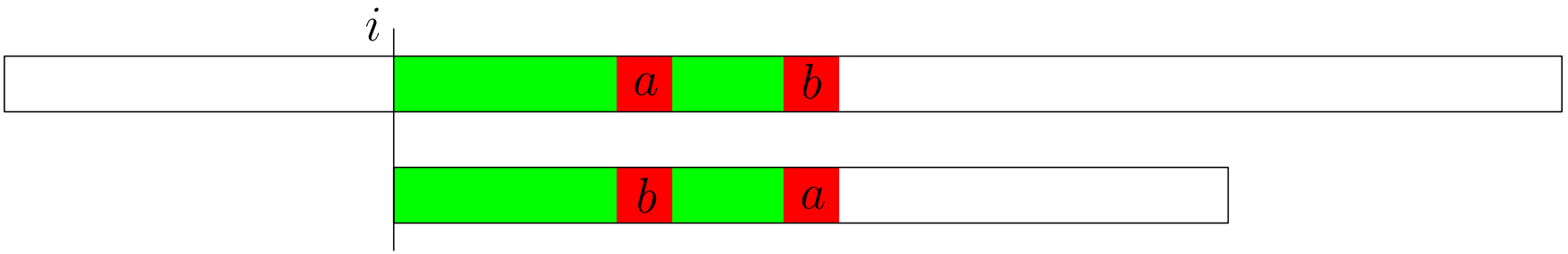
We iterate over all possible starting positions in t. For a fixed starting position, we want to check if the pattern occurs there with at most k mismatches.

We iteratively compute and cut the longest common prefix of both the remaining part of the pattern and the remaining part of the corresponding part of the text. We stop when we either process the whole pattern or detect more than k mismatches.

Lemma

The above algorithm correctly solves pattern matching with k mismatches in $\mathcal{O}(nk)$ time.

We iterate over all possible starting positions in t. For a fixed starting position, we want to check if the pattern occurs there with at most k mismatches.

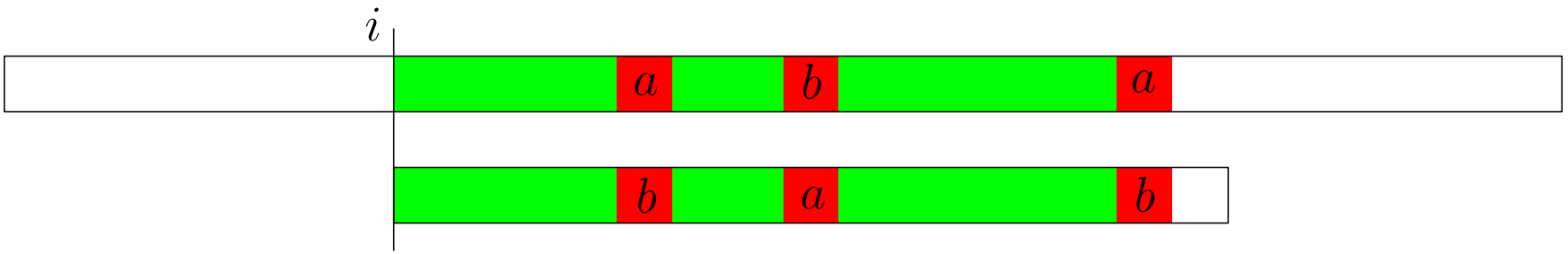


We iteratively compute and cut the longest common prefix of both the remaining part of the pattern and the remaining part of the corresponding part of the text. We stop when we either process the whole pattern or detect more than k mismatches.

Lemma

The above algorithm correctly solves pattern matching with k mismatches in $\mathcal{O}(nk)$ time.

We iterate over all possible starting positions in t. For a fixed starting position, we want to check if the pattern occurs there with at most k mismatches.

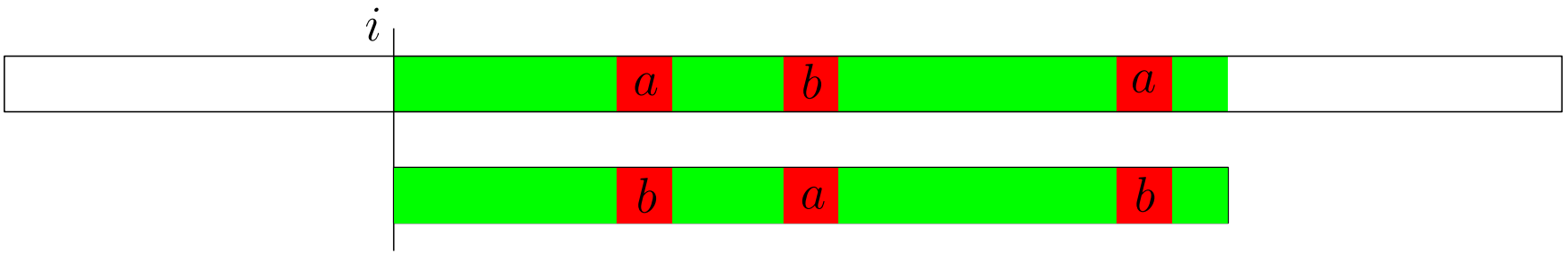


We iteratively compute and cut the longest common prefix of both the remaining part of the pattern and the remaining part of the corresponding part of the text. We stop when we either process the whole pattern or detect more than k mismatches.

Lemma

The above algorithm correctly solves pattern matching with k mismatches in $\mathcal{O}(nk)$ time.

We iterate over all possible starting positions in t. For a fixed starting position, we want to check if the pattern occurs there with at most k mismatches.

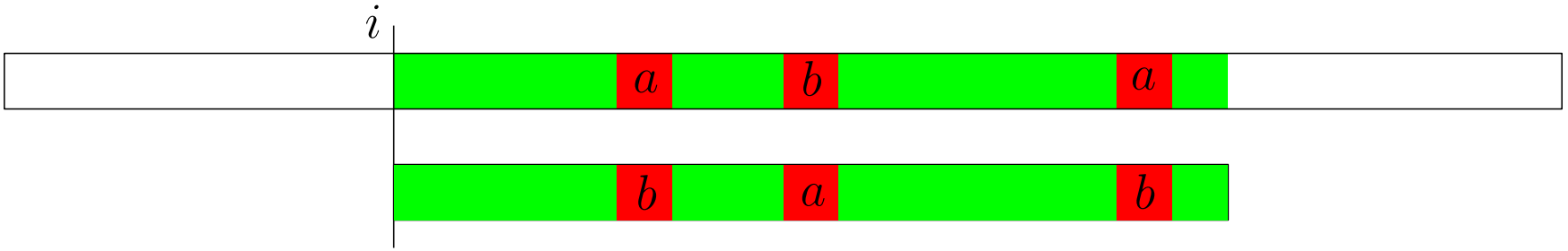


We iteratively compute and cut the longest common prefix of both the remaining part of the pattern and the remaining part of the corresponding part of the text. We stop when we either process the whole pattern or detect more than k mismatches.

Lemma

The above algorithm correctly solves pattern matching with k mismatches in $\mathcal{O}(nk)$ time.

We iterate over all possible starting positions in t. For a fixed starting position, we want to check if the pattern occurs there with at most k mismatches.



We iteratively compute and cut the longest common prefix of both the remaining part of the pattern and the remaining part of the corresponding part of the text. We stop when we either process the whole pattern or detect more than k mismatches.

Lemma

The above algorithm correctly solves pattern matching with k mismatches in $\mathcal{O}(nk)$ time.

But what if k is large? Can we do better?

Abrahamson 1987

We can compute for each i the number of mismatches between $t[i..i+m-1$ and p in $\mathcal{O}(n\sqrt{m \log m})$ total time.

We will see the idea later. It's really cute.
Another question is whether $\mathcal{O}(nk)$ is the best you can do for small values of k?

Amir, Lewenstein, Porat 2004

Pattern matching with k mismatches can be solved in $\mathcal{O}(n\sqrt{k \log k})$ time.

This is super complicated, and we are not going to talk how to achieve such complexity.

Is pattern matching with mismatches what we really really want to solve in real-life?
Maybe, but the notion is surely not perfect. For instance removing just a single character might increase the number of mismatches in the best alignment to m. So, maybe we should modify the problem?

Pattern matching with k errors

Given a pattern $p[1..m]$ and a text $t[1..n]$, does p occur in t with at most k errors, i.e., is there $i \leq j$ such that the edit distance between p and $t[i..j]$ is at most k?

Given that we are dealing with the edit distance, it's maybe not surprising that dynamic programming is the way to go. We will see how to apply it to solve the problem in $\mathcal{O}(nm)$ time.

Hmm, actually, we have seen the solution already! We compute a big table $T[1..n][1..m]$.

$$
\begin{aligned}
T[i][j] = \min(& \\
& T[i-1][j] + 1, \\
& T[i][j-1] + 1, \\
& T[i-1][j-1] + [s[i] \neq t[j]] \\
) &
\end{aligned}
$$

$T[i][j]$ is supposed to be the smallest edit distance between $p[1..j]$ and (some) suffix of $t[1..i]$. The only difference is that we initialize the table by setting $T[i][0] = 0$ for all i (not just $T[0][0] = 0$!).

How to detect an occurrence with at most k errors by looking at the table?

When k is small (and probably it is, right?), $\mathcal{O}(nm)$ is disappointing. Can we do better?

Landau and Vishkin 1989

Pattern matching with k errors can be solved in $\mathcal{O}(nk)$ time.

...actually, the $\mathcal{O}(nd)$ time algorithm for edit distance is quite similar to the above result. It does need some modifications, though, and we are not going to talk about this.
It's possible to do even better.

Cole and Hariharan 1998

Pattern matching with k errors can be solved in $\mathcal{O}(n\frac{k^4}{m} + n)$ time.

Very high-brow stuff!

Another possible definition of approximate pattern matching is that we allow our pattern to contain wildcards, usually denoted by ?. They can be replaced with any other character, so for instance *aab*?*a* matches both *aabaa* and *aabba*.

Pattern matching with don't cares

Given a pattern $p[1..m]$ containing any number of wildcards and a text $t[1..n]$, does p occur in t?

Sometimes people allow the text to contain wildcards, too. Pattern matching with wildcards is maybe less practical, but it's a beautiful example of a somewhat unexpected method. The method is based on the Fast Fourier Transform (FFT).

A polynomial of degree d is a sum $p(x) = \sum_{i=0}^{d} a_i x^i$, where $a_0, a_1, \ldots, a_d$ are its coefficients. In our case the coefficients will be always (not very big) natural numbers.

Polynomial multiplication

Given two polynomials $f(x) = \sum_{i=0}^{N} a_i x^i$ and $g(x) = \sum_{j=0}^{M} b_j x^j$ their product is a polynomial of degree $N + M$ of the form $\sum_{k=0}^{N+M} c_k x^k$, where $c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0$.

Convolution

If the coefficients are written as vectors $[a_0, a_1, \ldots, a_N]$ and $[b_0, b_1, \ldots, b_M]$, their convolution is the vector $[c_0, c_1, \ldots, c_{N+M}]$, with all c_k defined as above.

FFT

Given two vectors, their convolution can be computed in $\mathcal{O}((N + M) \log(N + M))$ time.

We will see how to apply convolution to compute for each i the number of mismatches between $t[i..i+m-1]$ and $p[1..m]$, with both t and p being allowed to contain any number of wildcards. First the simple case with $\Sigma = \{a, b\}$.

What is a mismatch?

a where we would expect b, or the other way around.

So, for each position i we should count the number of mismatches of each type. Say that we want to count j such that $t[i+j-1] = a$ but $p[j] = b$. How to apply FFT here?

So, for each position i we should count the number of mismatches of each type. Say that we want to count j such that $t[i+j-1] = a$ but $p[j] = b$. How to apply FFT here?

Our small trick

Define binary vectors with $N = n - 1$ and $a_i = 1$ iff $t[i+1] = a$, $M = m - 1$ and $b_j = 1$ iff $p[m-j] = b$. Then look at their convolution.

Lemma

c_k is exactly the number of mismatches of the first type corresponding to the starting position $i = k - m + 1$.

So, we just have to apply FFT twice, and we get $\mathcal{O}(n \log n)$ time algorithm. But what happens for larger Σ?

First algorithm

Apply FFT for each pair of $a, b \in \Sigma$, $a \neq b$. Time is $\mathcal{O}(|\Sigma|^2 n \log n)$.

Second algorithm

Apply FFT for each pair of $b \in \Sigma$ to count mismatches such that we would expect b but we get something else. Time is $\mathcal{O}(|\Sigma| n \log n)$.

But we just want to detect an occurrence, not count the mismatches for each starting position. So maybe we could do better?

If $a \neq b$, then the i-th bit of a is 1 and the i-th bit of b is 0, or the other way around, for some $i \in \{0, 1, \ldots, \log |\Sigma|\}$.

Call them i-mismatches of type 1 and 2, respectively.

Third algorithm

For each $i \in \{0, 1, \ldots, \log |\Sigma|\}$ apply FFT to detect starting positions resulting in a i-mismatch of type 1 and 2. Time is $\mathcal{O}(\log |\Sigma| n \log n)$.

That $\log |\Sigma|$ is annoying. It was known how to remove it, but the solutions were kind of complicated. Until...

Clifford and Clifford 2007

Pattern matching with don't cares can be solved in $\mathcal{O}(n \log n)$ time for any alphabet.

The idea is VERY simple.

$a \neq b$ iff $(a-b)^2$ is nonzero. Then write $(a-b)^2 = a^2 - 2ab + b^2$ and compute each part separately!

More precisely, for each i compute $\sum_{j=1}^{m} t[i+j-1]p[j](t[i+j-1]-p[j])^2$, where ? is 0, and other letters are positive numbers. Then the sum is 0 exactly when we have a match!

Something to think about: $\mathcal{O}(n \log n)$ can be actually decreased to $\mathcal{O}(n \log m)$ in a very simple manner. Do you see how?