

From coding theory to efficient pattern matching

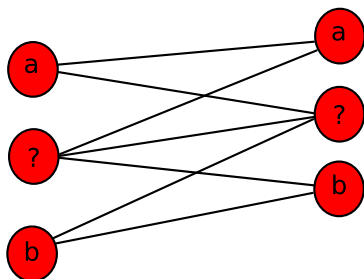
Raphaël Clifford

Department of Computer Science,
University of Bristol

Joint work with
Klim Efremenko, Ely Porat and Amir Rothschild

What is promiscuous matching?

A special symbol '?' matches any other symbol in the alphabet (i.e. a don't care or wildcard symbol)?



- ▶ Pattern $p = p_0 \cdots p_{m-1}$ and text $t = t_0 \cdots t_{n-1}$.
- ▶ To find all occurrences of p in t we can take $O(nm)$ time naively.
- ▶ *Without* don't cares we only need linear time, e.g. KMP (1977).

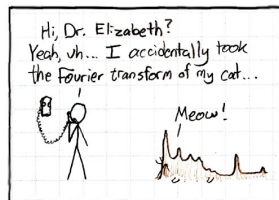
The magic of the fast Fourier transform

All the inner-products,

$$p \otimes t[i] = \sum_{j=0}^{m-1} p_j t_{i+j}, \quad 1 \leq i \leq n - m,$$

can be calculated accurately and efficiently in $O(n \log m)$ time in the RAM model.

1. For binary alphabet, set p' and t' so that $a = -1, b = 1, ? = 0$.
2. Set p'' and t'' so that $a = 1, b = 1, ? = 0$.



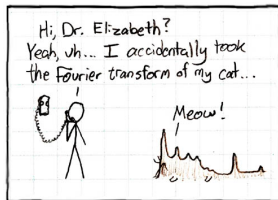
The magic of the fast Fourier transform

All the inner-products,

$$p \otimes t[i] = \sum_{j=0}^{m-1} p_j t_{i+j}, \quad 1 \leq i \leq n - m,$$

can be calculated accurately and efficiently in $O(n \log m)$ time in the RAM model.

1. For binary alphabet, set p' and t' so that $a = -1, b = 1, ? = 0$.
2. Set p'' and t'' so that $a = 1, b = 1, ? = 0$.



Example

$p = ab?ab$ and text $t = b?bbabba$

$p' = -1, 1, 0, -1, 1$ and $t' = 1, 0, 1, 1, -1, 1, 1, -1$

$p' \otimes t' = -3, 3, 0, -4$

Compare to $p'' \otimes t'' = 3, 3, 4, 4$

Recoding as integers - larger alphabets

The squared L_2 norm,

$$\sum_{j=0}^{m-1} (p_j - t_{i+j})^2 = \sum_{j=0}^{m-1} (p_j^2 - 2p_j t_{i+j} + t_{i+j}^2)$$

can be calculated for all i in $O(n \log m)$ time.

Recoding as integers - larger alphabets

The squared L_2 norm,

$$\sum_{j=0}^{m-1} (p_j - t_{i+j})^2 = \sum_{j=0}^{m-1} (p_j^2 - 2p_j t_{i+j} + t_{i+j}^2)$$

can be calculated for all i in $O(n \log m)$ time.

For exact matching with don't cares, define a new string p' with $p'_j = 0$ if $p_j = ?$, and $p'_j = 1$ otherwise. Similarly, define $t'_i = 0$ if $t_i = ?$, and 1 otherwise.

- ▶ We have an exact match with don't cares if and only if

$$\sum_{j=0}^{m-1} p'_j t'_{i+j} (p_j - t_{i+j})^2 = 0$$

Recoding as integers - larger alphabets

The squared L_2 norm,

$$\sum_{j=0}^{m-1} (p_j - t_{i+j})^2 = \sum_{j=0}^{m-1} (p_j^2 - 2p_j t_{i+j} + t_{i+j}^2)$$

can be calculated for all i in $O(n \log m)$ time.

For exact matching with don't cares, define a new string p' with $p'_j = 0$ if $p_j = ?$, and $p'_j = 1$ otherwise. Similarly, define $t'_i = 0$ if $t_i = ?$, and 1 otherwise.

- ▶ We have an exact match with don't cares if and only if

$$\sum_{j=0}^{m-1} p'_j t'_{i+j} (p_j - t_{i+j})^2 = 0$$

$O(n \log m \log |\Sigma|)$ [Fischer and Paterson (1974)] $\rightarrow O(n \log m)$
[Clifford and C. - 2007]

Matching with few errors

Perhaps the most natural measure of distance is just to count the number of mismatches.

Example

$p = abc?b$ and text $t = b?bbabba$. The array of Hamming distances is 3, 1, 2, 4

Matching with few errors

Perhaps the most natural measure of distance is just to count the number of mismatches.

Example

$p = abc?b$ and text $t = b?bbabba$. The array of Hamming distances is 3, 1, 2, 4

- ▶ Without don't cares and with a distance bound k , $O(nk)$ time algorithm based on constant time LCA queries [Landau,Vishkin:86], improved to $O(n\sqrt{k \log k})$ via a method based on filtering, LCA and cross-correlations. [Amir, Lewenstein, Porat:2000]

Matching with few errors

Perhaps the most natural measure of distance is just to count the number of mismatches.

Example

$p = abc?b$ and text $t = b?bbabba$. The array of Hamming distances is 3, 1, 2, 4

- ▶ Without don't cares and with a distance bound k , $O(nk)$ time algorithm based on constant time LCA queries [Landau, Vishkin:86], improved to $O(n\sqrt{k \log k})$ via a method based on filtering, LCA and cross-correlations. [Amir, Lewenstein, Porat:2000]
- ▶ With don't cares, randomised $\tilde{O}(nk)$ and deterministic $\tilde{O}(nk^2)$ time solutions [C, Efremenko, Porat, Rothschild:2007]. Also $\tilde{O}(nk^2)$ and $\tilde{O}(nk^3)$ time with different log factors [Linhart, Shamir:2008].

Pattern matching as a coding problem

- ▶ Rephrase problem as one that looks like decoding BCH codes.



Copyright © 1997 United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited

Pattern matching as a coding problem

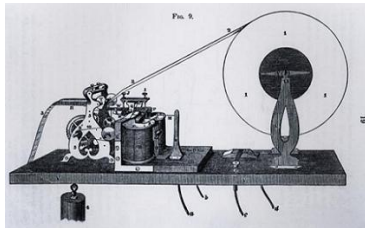
- ▶ Rephrase problem as one that looks like decoding BCH codes.
- ▶ Recode everything as elements of a finite field with small characteristic and borrow a fast polynomial factorisation tool from [Shoup:1991]



Copyright © 1997 United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited

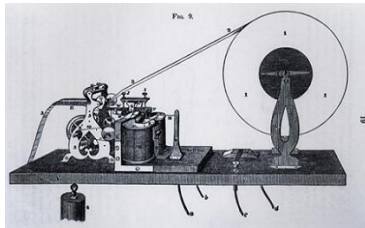
Pattern matching as a coding problem

- ▶ Consider the difference between pattern and text as code words over a noisy channel.



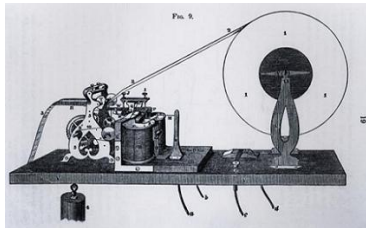
Pattern matching as a coding problem

- ▶ Consider the difference between pattern and text as code words over a noisy channel.
- ▶ Set up syndrome equations that encode the mismatch locations.



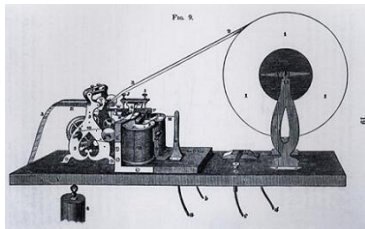
Pattern matching as a coding problem

- ▶ Consider the difference between pattern and text as code words over a noisy channel.
- ▶ Set up syndrome equations that encode the mismatch locations.
- ▶ Create a separate error locator polynomial per position.



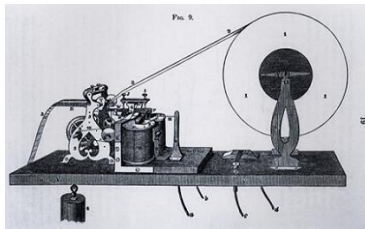
Pattern matching as a coding problem

- ▶ Consider the difference between pattern and text as code words over a noisy channel.
- ▶ Set up syndrome equations that encode the mismatch locations.
- ▶ Create a separate error locator polynomial per position.
- ▶ Find roots of the polynomial, giving us the location of the mismatches.



Pattern matching as a coding problem

- ▶ Consider the difference between pattern and text as code words over a noisy channel.
- ▶ Set up syndrome equations that encode the mismatch locations.
- ▶ Create a separate error locator polynomial per position.
- ▶ Find roots of the polynomial, giving us the location of the mismatches.
- ▶ Verify the mismatches found.



Create syndrome equations

Calculate $2k + 1$ arrays such that

$$s_{\ell,i} = \sum (p_j - t_{i+j})(i+j)^\ell p'_j t'_{i+j}$$

where ℓ is in the range $0 \leq \ell \leq 2k$. We write p_j , t_{i+j} and $i+j$ when we mean the corresponding elements in $GF(2^{O(\log m)})$.

We also create binary arrays p' and t' so that $p'_j = 0$ ($t'_j = 0$) if $p_j = ?$ ($t_j = ?$) and $p'_j = 1$ ($t'_j = 1$) otherwise.

Create syndrome equations

Calculate $2k + 1$ arrays such that

$$s_{\ell,i} = \sum (p_j - t_{i+j})(i+j)^\ell p'_j t'_{i+j}$$

where ℓ is in the range $0 \leq \ell \leq 2k$. We write p_j , t_{i+j} and $i+j$ when we mean the corresponding elements in $GF(2^{O(\log m)})$.

We also create binary arrays p' and t' so that $p'_j = 0$ ($t'_j = 0$) if $p_j = ?$ ($t_j = ?$) and $p'_j = 1$ ($t'_j = 1$) otherwise.

These take $O(nk \log m \log \log m)$ time to create using convolutions over the finite field. Small detail: we require a '3'-adic transform as '2' is not a unit in our field.

Syndrome equations

For any given alignment i ,

$$s_{\ell,i} = \sum_{j=1}^{HD(i)} r_j x_j^{\ell},$$

where r_j is the difference between the pattern and text at the j th mismatch.

Syndrome equations

For any given alignment i ,

$$s_{\ell,i} = \sum_{j=1}^{HD(i)} r_j x_j^{\ell},$$

where r_j is the difference between the pattern and text at the j th mismatch. Rewriting, we get:

$$\begin{array}{rcccccc} r_1 & + & r_2 & + & \dots & + & r_{k'} & = & s_{0,i} \\ r_1 x_1 & + & r_2 x_2 & + & \dots & + & r_{k'} x_{k'} & = & s_{1,i} \\ r_1 x_1^2 & + & r_2 x_2^2 & + & \dots & + & r_{k'} x_{k'}^2 & = & s_{2,i} \\ & & & & \vdots & & & & \vdots \\ r_1 x_1^{2k} & + & r_2 x_2^{2k} & + & \dots & + & r_{k'} x_{k'}^{2k} & = & s_{2k,i} \end{array}$$

Solving these equations gives us the positions of all the mismatches.

Solving the system of equations

We assume for the moment that $k' = HD(i) \leq k$. The two main steps taken to solve the system of equations are as follows.

1. Calculate the coefficients of the polynomial

$$P(z) = \prod_{i=1}^{k'} (zx_i - 1)$$

from the $2k + 1$ values s_ℓ .

2. Find the roots of the polynomial $P(z)$. The inverse of these roots will give us the position of the mismatches of the pattern and text.

Turning the wheel



1. Fast Berlekamp-Massey algorithm returns the coefficients of the polynomial $P(z) = \prod_{i=1}^{k'} (zx_i - 1)$ in $O(k \log^2 k)$ time.
2. Given a polynomial $P(z)$ of degree at most k , its roots can be found deterministically in $\tilde{O}(k)$ time in the field $GF(2^{O(\log m)})$ [Shoup:1991].

Checking the result

Using integers instead of elements of a finite field, compute

$$D[i] = \sum (p_j - t_{i+j})^2 p'_j t'_{i+j}.$$

We now go through the list of $O(nk)$ mismatch positions found and “correct” $D[i]$ at each location i where a mismatch is found by subtracting $(p_j - t_{i+j})^2$.

Checking the result

Using integers instead of elements of a finite field, compute

$$D[i] = \sum (p_j - t_{i+j})^2 p'_j t'_{i+j}.$$

We now go through the list of $O(nk)$ mismatch positions found and “correct” $D[i]$ at each location i where a mismatch is found by subtracting $(p_j - t_{i+j})^2$.

If all mismatches have been found then at the end $D[i] = 0$. Otherwise, it must be that the true Hamming distance was greater than k at that location.

Checking the result

Using integers instead of elements of a finite field, compute

$$D[i] = \sum (p_j - t_{i+j})^2 p'_j t'_{i+j}.$$

We now go through the list of $O(nk)$ mismatch positions found and “correct” $D[i]$ at each location i where a mismatch is found by subtracting $(p_j - t_{i+j})^2$.

If all mismatches have been found then at the end $D[i] = 0$. Otherwise, it must be that the true Hamming distance was greater than k at that location.

The time for this verification stage is $O(n \log m + nk)$ and so does not affect the overall running time.

Final result

Theorem

The k -mismatch with wildcards problem can be solved in $O(nk \log^2 m(\log^2 k + \log \log m))$ or more simply, $\tilde{O}(nk)$, time.

Final result

Theorem

The k -mismatch with wildcards problem can be solved in $O(nk \log^2 m(\log^2 k + \log \log m))$ or more simply, $\tilde{O}(nk)$, time.

1. The time to perform the encoding step is $O(nk \log m \log \log m)$.
2. The time to create the error locator polynomials for every position in the text is $O(nk \log^2 k)$ in total using fast Berlekamp-Massey.
3. The time complexity for finding all the roots of all the polynomials is $O(nk \log^2 k \log^2 m)$.
4. Finally an extra $O(n \log m + nk)$ time is needed to verify which of the locations found were in fact k -mismatches. The total time taken to solve the k -mismatch with wildcards problem is therefore $O(nk \log^2 m(\log^2 k + \log \log m))$.

Open problems

Approximate matching and coding theory seem intimately linked.
What else can be tackled using these techniques?

Open problems

Approximate matching and coding theory seem intimately linked.
What else can be tackled using these techniques?

If we merely want to count the number of mismatches, can we do it faster?

Open problems

Approximate matching and coding theory seem intimately linked.
What else can be tackled using these techniques?

If we merely want to count the number of mismatches, can we do it faster?

How fast can we make these methods in practice?

Open problems

Approximate matching and coding theory seem intimately linked.
What else can be tackled using these techniques?

If we merely want to count the number of mismatches, can we do it faster?

How fast can we make these methods in practice?

Thank you for listening!